MASTER EN SCIENCES INFORMATIQUES

Création d'un framework générique en Java et d'un outil de démonstration didactique des algorithmes génétiques

Caudron, Olivier

Award date:
2003

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique.
Année académique 2002-2003

Création d'un framework générique en Java et d'un outil de démonstration des Algorithmes Génétiques

Olivier Caudron

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique.

Résumé

Les algorithmes génétiques sont un domaine très prometteur de la science informatique moderne. Ils peuvent en effet être appliqués efficacement à des catégories de problèmes extrêmement variées. Créés par John Holland dans les années 60, ils commencent seulement depuis ces dernières années à entrer dans l'usage courant.

Les algorithmes génétiques manquent malheureusement d'implémentations généralistes et didactiques, car la plupart sont soit très spécialisées (implémentations-types du problème du voyageur de commerce par exemple), soit très arides (implémentations en C pauvrement documentées).

Ce mémoire tente de remédier à cette situation en proposant une librairie bien documentée qui s'adresse directement à ces manques, en offrant un panel complet des fonctionnalités de base, dans un modèle purement orienté objet permettant l'implémentation aisée et rapide de problèmes divers. Un outil de démonstration présentant quelques problèmes classiques vient compléter cette librairie.

Mots-clés : Algorithmes Génétiques, Algorithmes Evolutionnaires, Optimisation, Intelligence Artificielle, Programmation Orientée Objet, Java.

Abstract

Genetic Algorithms are a very promising domain of modern computer science. Indeed, they can be efficiently applied to extremely different problem categories. Created by John Holland in the sixties, they only began recently to enter common practice.

Genetic Algorithms unfortunately lack generic and didactic implementations, because applications are mainly either very specialized (like typical implementations of the Traveling Salesman Problem) or very arid (poorly documented C implementations).

This thesis tries to remedy to this by proposing a well-documented framework that addresses directly these issues, by offering a complete panel of the base Genetic Algorithm functionality, in a purely object-oriented model that allows fast and easy implementation of various problems. A demonstration tool showing some classical problems completes the framework.

Keywords : Genetic Algorithms, Evolutionary algorithms, Optimization, Artificial Intelligence, Object Oriented Programming, Java.

Avant-propos

Remerciements

Mes remerciements vont avant tout à M. Leclercq, mon promoteur, pour sa grande patience, sa disponibilité et ses conseils inestimables. Merci également à Madame d'Udekem-Gevers, conseillère à l'éducation, pour son aide et son support moral ; à MM Tsolakidis et Borremans, pour avoir bien voulu réviser ce rapport ; et à Mesdames Breny, Massart et Vereeke, pour l'aide précieuse qu'elles apportent aux étudiants avec gentillesse et compétence.

Je dédie ce mémoire à mon épouse Véronique, qui est ma force et mon inspiration ; à mes enfants Logan et Cassandre, qui je l'espère ne m'en voudront pas trop de n'avoir pas été assez disponible ces dernières années ; à mes parents enfin, André et Rose, mes grand-parents, Maurice et Rose, et mon frère, Frédéric, qui m'ont appris le goût du beau, le plaisir d'apprendre, et la joie de vivre.

Accréditations techniques

Ce rapport de mémoire a été mis en page au moyen de LaTeX, un macro langage créé par Leslie Lamport et maintenu par Frank Mittelbach pour le système de mise en page TeX créé par Donald E. Knuth. Il contient des extensions PDFTeX permettant de générer optionnellement un document PDF. Pour toute information sur TeX, LaTeX et PDFTeX, voir le site www.ctan.org.

Le texte a été rédigé principalement sur l'éditeur de texte freeware *Crimson Editor* de Ingyu Kang. Voir www.crimsoneditor.com pour plus d'informations. La correction orthographique ainsi qu'une partie des figures ont été réalisées sur *OpenOffice 1.0.3*, une suite applicative sous licence Open Source maintenue sur www.openoffice.org. Les captures d'écran ont été réalisées au moyen de *Corel Capture 7* de Corel corporation (www.corel.com), produit commercial sous licence utilisateur unique.

Le framework et l'application de démonstration ont été réalisés en *Java* (java.sun.com) avec un JDK 1.3.1 et développés initialement sur *JCreator LE v.2.0*, un produit freeware de Wendel D. de Witte (www.jcreator.com), puis sur *Eclipse*, un produit open source sous licence CPL (www.eclipse.org/legal/cpl-v10.html) initié par la société IBM et maintenu par un consortium (www.eclipse.org).

L'outil de démonstration utilise des icônes extraites de l'excellente collection d'icônes *AquaFusion v. 0.5* de Daniel Flax et Markus Mueller, distribués sous licence de type "design science" et disponible sur www.kde-look.org.

Table des matières

Chapitre 1

Introduction

Il est difficile de croire, lorsqu'on entre en contact pour la première fois avec les algorithmes génétiques, que la technique existe depuis près de quarante ans, et a été utilisée avec succès dans plusieurs applications industrielles, sans parler des expérimentations diverses du domaine de l'intelligence artificielle. Les algorithmes génétiques semblent avoir vécu depuis toujours en marge de l'algorithmique générale.

Heureusement, la tendance semble en passe de s'inverser. Le secteur est plus actif que jamais, comme le prouve la pléthore de matériel disponible sur Internet, et l'orientation commerciale que prennent certains acteurs du domaine (voir à titre d'exemple *Genocop* sur www.nutechsolutions.com). Il en est de même du domaine plus large dans lequel s'inscrivent les algorithmes génétiques, à savoir les algorithmes évolutionnaires, et d'une manière générale les techniques d'optimisation et d'intelligence artificielle.

Hélas, le domaine garde l'image d'un sujet pointu voire ardu et difficile d'accès. Le fait est que les algorithmes génétiques et évolutionnaires s'expriment souvent dans des domaines niches, et les applications réalisées dans ce domaine sont souvent spécialisées et peu accessibles. Certains efforts ont déjà été consentis, mais le fait est que les algorithmes génétiques manquent d'une implémentation permettant de les expérimenter d'une manière facile d'accès, claire et pratique, en un mot, didactique.

Or ce n'est pas du tout inimaginable. La puissance des algorithmes génétiques fait qu'un même code peut servir aisément à traiter des classes très variées de problèmes, au prix d'une adaptation très limitée. En fait, la flexibilité des algorithmes génétiques et leur adaptabilité rend la création d'un outil générique et didactique relativement aisée et efficace.

Ce mémoire propose une telle implémentation de référence, dans un but essentiellement didactique et de démonstration.

Chapitre 2

Généralités

2.1 Historique succinct

Les algorithmes génétiques sont un des éléments du groupe des *algorithmes évolutionnaires*, eux-même entrant dans la large catégorie des algorithmes utilisant des stratégies inspirées de techniques que l'on trouve dans la nature.

Mélanie Mitchell in [Mitchell 1996] fait remonter les algorithmes évolutionnaires aux années 50 et 60, avec les travaux de Rechenberg, Schwefel, et Fogel, Owens et Walsh, en ce qui concerne l'optimisation de paramètres en valeurs réelles, et Box, Friedman, Bledsoe, Bremermann, et Reed, Toombs, et Baricelli pour l'apprentissage des machines et l'optimisation.

Il est admis cependant que le domaine précis des algorithmes génétiques a été inventé par *John Holland* de l'Université du Michigan dans les années 60, et développé par lui-même et ses étudiants dans les deux décennies qui ont suivi. L'apport de Holland est double : le principe de l'utilisation d'une population (plutôt qu'un nombre réduit d'individus, 1 ou 2, comme ça avait été fait précédemment) avec l'application systématique de *crossover*, *inversion* et *mutation*, et la mise en place d'une base théorique pour expliquer l'efficacité de ces stratégies, connue sous le nom de *théorie des schémas*. Nous décrirons ces concepts en plus de détails dans les chapitres suivants.

Il semble que le but originel des travaux de Holland était l'étude formelle de l'évolution d'une population soumise à la sélection et aux opérateurs génétiques, plutôt que la résolution de problèmes au moyen de ces techniques. Ce second but a été poursuivi par la suite, en particulier par un des étudiants de Holland, *David Goldberg*, devenu depuis un personnage incontournable en matière d'algorithmes génétiques, en particulier avec son ouvrage de référence, *Genetic Algorithms in Search, Optimization and Machine Learning* [Goldberg 1989].

Le travail de référence de Goldberg fut la réalisation d'un système de contrôle de conduite de gaz utilisant un algorithme génétique pour l'optimisation, et un *système de classification* (*classifier system*) pour l'apprentissage de la machine.

Le domaine des algorithmes évolutionnaires, dont les algorithmes génétiques sont les représentants les plus éminents, a depuis sa création vécu en marge de l'algorithmique générale, avec des succès certains, mais dans des domaines niches et à faible exposition. Cela paraît être moins le cas depuis quelques années, ces technologies rencontrant un intérêt croissant, en particulier de l'industrie qui

commence -enfin- à les découvrir.

2.2 Définition

Le domaine mouvant des algorithmes évolutionnaires se prête mal aux définitions exactes et synthétiques. En effet, les domaines sont souvent connexes, avec un recouvrement parfois non négligeable, et la collaboration étroite entre diverses technologies est courante.

Par ailleurs, les algorithmes génétiques sont applicables à des catégories de problèmes très variées, allant de l'optimisation pure de fonctions à la simulation de processus génétiques réels, en passant par l'apprentissage de réseaux neuronaux. Il serait donc illusoire de vouloir définir les algorithmes génétiques en fonction des problèmes qu'ils résolvent.

Il faut donc les définir en fonction des processus dont ils se composent. Voici les processus minimaux permettant de définir un algorithme génétique :

- Création d'une population de départ de manière aléatoire ;
- Création de nouvelles générations au moyen :
  - d'une stratégie de sélection probabiliste permettant de choisir les individus reproducteurs en fonction de leur qualité par rapport au problème à résoudre (viabilité ou *fitness*) ;
  - d'opérateurs génétiques (crossover, mutation, inversion, etc.) permettant l'échange et l'évolution de caractéristiques des individus.

Tout ceci permet de faire évoluer la population, de génération en génération, vers de meilleures solutions.

On peut considérer ces caractéristiques comme un ensemble de conditions nécessaires et suffisantes permettant de définir un algorithme génétique.

2.3 La place des algorithmes génétiques

Le domaine des algorithmes évolutionnaires comprend diverses technologies et applications. Les algorithmes génétiques sont les représentants les plus courants de ce domaine. Leur usage le plus élémentaire, sur lequel ce travail sera plus particulièrement focalisé, est la *recherche d'optimum*. Il est possible de résoudre au moyen des algorithmes génétiques des problèmes de ce type difficilement traitables par d'autres méthodes.

Mais les algorithmes évolutionnaires proposent bien plus.

La *programmation génétique*, par exemple, utilise une technologie similaire à celle des algorithmes génétiques, mais dans le but particulier d'optimiser des processus et technologies. On lira avec intérêt, à ce propos, l'article de J. Koza et al. paru récemment dans *pour la science* ([Koza 2003]). Voir aussi le site Internet de l'auteur principal, sur www.genetic-programming.com.

Les systèmes d'apprentissage de machines basés sur la génétique, et en particulier les *classifier systems*, permettent d'adapter un ensemble de règles simples définissant le comportement d'une machine au moyen d'un algorithme génétique, lui permettant d'évoluer dans un environnement arbitraire. Ces systèmes sont basés sur un principe de récompense/punition remplaçant, ou plutôt implémentant, la notion de viabilité dans un environnement quelconque.

L'ensemble de règles évolue lentement en fonction des résultats de son interaction avec l'environnement, synthétisé par le gain obtenu par la balance entre récompense et punition. [Goldberg 1989] donne une introduction très complète sur les *classifier systems.*

Par ailleurs, les algorithmes génétiques peuvent être utilement combinés avec d'autres technologies du domaine de l'intelligence artificielle. Un exemple très intéressant décrit dans l'annexe D, explique comment il est possible d'utiliser un algorithme génétique pour optimiser les pondérations des noeuds intermédiaires d'un réseau neuronal sans rétroaction. Voir aussi à ce propos le travail de M. Salvino Piazza, *Algorithmes Génétiques et leurs applications aux Réseaux de Neurones*, mémoire de licence en informatique présenté cette année (2003).

Le domaine des algorithmes évolutionnaires inclut des technologies alternatives, par exemple, les *algorithmes de fourmis.* Les algorithmes de fourmis (*Ant Systems* ou *Ant Colony Optimization*) ont été créés en 1991/1992 par Marco Dorigo de l'école polytechnique de Milan. Leur principe s'inspire de stratégies utilisées par les colonies de fourmis pour régler certains problèmes ardus de recherche de trajet optimal. Le but premier poursuivi par M. Dorigo était le traitement du problème du voyageur de commerce, mais cette technique peut s'appliquer sur plusieurs problèmes d'optimisation. La source centrale d'information sur ce domaine de recherche se trouve sur iridia.ulb.ac.be/ mdorigo/ACO/ACO.html.

Enfin, le domaine des algorithmes basés sur des stratégies inspirées de la nature (mais pas nécessairement évolutionnaires) incluent notamment le *recuit simulé* (*simulated annealing*), *l'intelligence des essaims* (*swarm intelligence*), les *automates cellulaires* (*cellular automata*), et les *réseaux neuronaux* (*neural networks*).

Les algorithmes génétiques occupent indiscutablement une place centrale dans le domaine des algorithmes évolutionnaires, et sont en train de prendre une place importante, comme technologie principale ou en support d'autres technologies, dans les domaines de l'intelligence artificielle et des techniques d'optimisation au sens large du terme.

2.4 Cuvier

Georges Cuvier (1769-1832) est un célèbre naturaliste français, considéré comme un des principaux contributeurs à la fondation de l'anatomie comparée et la paléontologie des vertébrés. Ses travaux se distinguent par la clarté de la démarche scientifique et l'importance quantitative et qualitative de ses recherches dans ces domaines.

FIG. 2.1 – Georges Cuvier

Bien que lui-même ne crût pas à l'évolution, il n'est pas douteux que ses travaux ont amené une foule d'informations utiles à l'édification des théories de l'évolution, dont ont bénéficié les naturalistes et paléontologues par la suite. Son rejet de l'évolution lui vaut cependant, semble-t-il, une certaine marginalisation dans l'histoire des sciences de la nature. C'est pourquoi, bien qu'il n'ait pas de lien *stricto sensu* avec la génétique, j'ai désiré rendre hommage à ce grand homme de science en donnant son nom au framework et à l'outil de démonstration que j'ai développés.

Voir [Encyclopedia Universalis, Cuvier, 1979], ou www.ucmp.berkeley.edu/history/cuvier.html, ou encore www.infoscience.fr/histoire/biograph/biograph.php3 ?Ref=18.

Chapitre 3

Théorie des Algorithmes génétiques

3.1 Terminologie

Les algorithmes génétiques empruntent pour une large part leur terminologie à la biologie, à la "génétique naturelle". Cependant, la terminologie naturelle ne s'applique pas inconditionnellement à l'informatique. L'exactitude de la métaphore n'est d'ailleurs pas le but recherché, mais plutôt l'efficacité de la méthode. De plus, ceux qui ont manipulé ces concepts étaient des informaticiens, pas des biologistes, ce qui a pu également amener des imprécisions.

Ceci explique sans doute pourquoi une certaine confusion existe quant aux termes utilisés dans les algorithmes génétiques, confusion d'ailleurs reconnue par Goldberg lui-même :

> (...) the terminology used in the GA literature is an unholy mix of the natural and the artificial [Goldberg 1989].

Il n'est donc pas inutile de débuter ce chapitre par une mise au point sur la terminologie, d'autant que le lecteur n'est pas nécessairement familier avec le vocabulaire de la génétique.

3.1.1 La génétique : notions élémentaires

Commençons donc par résumer les mécanismes biologiques dont s'inspirent les AG. Les sources d'information de cette section sont [Rostand et Tétry 1972], [Merriam-Webster] et [Encyclopedia Universalis, Génétique, 1979].

Les êtres vivants sont créés sur base d'un "plan de construction" appelé le *génotype*, qui est un très grand ensemble de caractéristiques élémentaires propres à l'individu en devenir. Au moyen de processus chimiques complexes, et sous l'influence de l'environnement (dans le sens large du terme), un individu est créé sur base de ce plan de construction. Les caractéristiques de l'individu formé sont dérivées de celles du génotype, mais nullement identiques : certaines caractéristiques ne s'expriment pas, d'autres sont influencées par l'environnement. L'ensemble des caractéristiques de l'individu formé s'appelle le *phénotype*.

Une caractéristique élémentaire du génotype est portée par un *gène*. Le gène est l'unité d'information *signifiante* minimale et indivisible qui compose

le génotype. La distinction entre génotype et phénotype provient notamment de ce que tous les gènes que porte un individu ne s'expriment pas en des caractéristiques de cet individu (voir ci-dessous l'exemple de la dominance).

Dans la nature, l'information génétique est portée par des chaînes de *nucléotides*. Un nucléotide est un élément chimique composé d'une base purique (adénine ou guanine) ou pyrimidique (thymine, cytosine ou uracile), et d'un groupe phosphate lié à un sucre à 5 atomes de carbone (pentose). Des liaisons s'établissent entre les complexes sucre-phosphate pour constituer la chaîne sur laquelle l'information génétique (portée par la base purique ou pyrimidique) est écrite [Rostand et Tétry 1972]. Le nucléotide est l'*unité de codage élémentaire* de l'information génétique.

L'acide désoxyribonucléique (*ADN*) et l'acide ribonucléique (*ARN*) sont des chaînes polynucléotidiques. Pour mémoire, une molécule d'ADN peut comporter de l'ordre de 10^7 paires de nucléotides.

Une chaîne continue de nucléotides constitue un *chromosome*. Les êtres vivants comportent généralement plusieurs chromosomes.

Dans la nature, les êtres vivants peuvent comporter un stock simple de chromosomes, on dit alors que ce sont des organismes *haploïdes*. Dans certains cas, le stock de chromosomes est dédoublé. On parle alors d'organismes *diploïdes*. Chaque gène de chacun des éléments d'une paire de chromosomes correspondants exprime la même caractéristique, mais pas nécessairement de la même manière. Par exemple, un organisme diploïde peut avoir le gène qui code la couleur des yeux dans sa version "bleue" et dans sa version "marron". Ce sont alors les mécanismes de *dominance* qui déterminent quel gène s'exprime dans le phénotype. On voit ici un exemple tout à fait représentatif de la nécessité de distinguer le génotype (caractères en devenir) et phénotype (caractères exprimés). La pression de l'environnement peut aussi influencer le phénotype, comme dans l'exemple bien connu du sexe des crocodiles du Nil, déterminé par la température de leur incubation.

Les deux gènes correspondant à une caractéristique donnée que l'on peut trouver dans le génotype d'un organisme diploïde sont appelés les *allèles* de ce gène. Le terme *allèle* fait également référence à *toutes* les formes alternatives possibles d'un gène donné [Merriam-Webster]. Nous utiliserons exclusivement cette dernière acception dans la terminologie des algorithmes génétiques utilisée dans ce mémoire.

Enfin, on utilise le terme *locus* pour désigner l'emplacement sur un chromosome d'un gène donné (ou d'un allèle dans le cas d'organismes diploïdes).

3.1.2 Limites de la métaphore

Dans la majorité, les algorithmes génétiques utilisent des simulations d' "organismes" haploïdes. Il existe cependant des implémentations mettant en oeuvre des simulations d'organismes diploïdes, utilisant des notions de dominance (voir [Goldberg 1989], chapitre 5, à ce sujet). Il s'agit d'extensions des mécanismes élémentaires des algorithmes génétiques qui sont hors de notre propos.

De même, certaines recherches font la distinction entre génotype et phénotype, mais nous considérerons un individu comme étant entièrement déterminé par son génotype et sa viabilité (voir ci-après).

Ensuite, la séparation du génotype en plusieurs chromosomes n'est pas utile pour les AG, et un individu ne comportera généralement qu'un chromosome.

Enfin, dans la nature, le rôle d'un gène ne dépend pas de sa localisation sur le chromosome (son locus), bien que des gènes puissent occuper des positions relatives signifiantes en fonction de leur rôle [Rostand et Tétry 1972]. Dans les AG, la position est souvent un discriminant majeur des gènes.

Ces limitations sont sans doute dépassées dans l'état actuel de la recherche. Cependant, elles définissent assez bien ce que l'on peut attendre d'un algorithme génétique "simple". Ce genre d'algorithme est suffisant pour résoudre un nombre important de problèmes de catégories très différentes. Notre propos étant principalement didactique, nous nous limiterons à décrire ces AG simples.

3.1.3 Adaptation de la terminologie aux A.G.

L'unité de codage élémentaire du génotype est le *nucléotide*. Le nucléotide n'est pas signifiant en soi. Il faut en toute généralité un nombre donné de nucléotides pour obtenir une information génétique signifiante. L'équivalent informatique du nucléotide est le *bit*. La littérature des algorithmes génétiques utilise pratiquement exclusivement le terme *bit* et pas *nucléotide*. Les deux termes sont utilisés indifféremment dans ce rapport.

Un *gène* est un ensemble de nucléotides codant une caractéristique élémentaire et indivisible. Dans les algorithmes génétiques, un gène peut être extrait par programmation de l'information stockée sur la chaîne de nucléotides, ou être maintenu dans un espace mémoire séparé et synchronisé avec la chaîne de bits. Un gène codera généralement une des variables du problème traité, et contiendra souvent une valeur numérique traduite de la chaîne de bits correspondante.

Un *chromosome* est un ensemble de gènes. Si l'on s'en tient à la restriction que nous appliquons ici, le génotype d'un individu est porté par un chromosome unique, et donc un individu est entièrement représenté par un chromosome. L'ensemble des gènes portés par le chromosome représente le *génotype* de l'individu. Dans les limites d'application de ce rapport, le *phénotype* est strictement identique au *génotype*, et la sélection naturelle est établie au moyen d'une métaphore conventionnelle appelée la *stratégie de sélection* et d'une mesure conventionnelle de la capacité de l'individu à s'assurer une descendance, appelée la *viabilité* (*fitness*).

Dans la littérature des AG, le terme *locus* est généralisé par rapport à la terminologie génétique, car il est utilisé aussi bien pour désigner l'emplacement d'un bit unique (d'un nucléotide) que l'emplacement d'un gène.

De même, le terme *allèle* fait référence aux valeurs possibles d'un gène donné, mais souvent également aux valeurs possibles d'un nucléotide donné. Cet abus d'usage n'est cependant pas très utile, tout en portant à confusion, nous éviterons donc de l'utiliser.

3.2 Principes de base

3.2.1 Exemple introductif

Supposons que nous voulions optimiser (minimiser ou maximiser, peu importe) une fonction quelconque. Plusieurs possibilités s'offrent à nous, à commencer par les résolutions purement mathématiques, en particulier la méthode

du gradient. Celle-ci suppose cependant la détermination d'une dérivée de la fonction, ce qui n'est pas toujours praticable. Si l'on a une dérivée, un autre problème peut se poser. Si la fonction connaît potentiellement plusieurs optima locaux, comment s'assurer que l'optimum trouvé est bien l'optimum global, et non un optimum local ?

Une alternative à la résolution mathématique est la recherche de l'espace des solutions. Ceci suppose que l'espace d'application de la fonction soit borné et puisse être "discrétisé" raisonnablement. Deux options existent dans ce cas, la recherche exhaustive et la recherche aléatoire.

La recherche exhaustive donne l'assurance de trouver l'optimum de l'espace de recherche (et pourvu que l'échantillonnage soit assez fin pour ne pas "rater" un optimum très localisé). Cependant, en fonction de la taille et du nombre de dimensions de l'espace de recherche, cette recherche peut être très longue. Par contre, la durée de la recherche est toujours identique pour un même problème, en effet, il faut parcourir l'entièreté de l'espace pour s'assurer qu'un meilleur optimum n'existe pas.

La recherche aléatoire peut trouver un optimum plus rapidement, mais pose la question du critère d'arrêt. Quand considère-t-on que l'on a trouvé un optimum satisfaisant ? Si l'on limite l'exécution à un nombre donné d'essais, par exemple, on peut très bien obtenir un résultat très pauvre, ou gaspiller un grand nombre d'itérations.

L'algorithme génétique est un peu à la limite des deux mondes de la recherche dirigée et de la recherche aléatoire. Il part d'un ensemble de solutions déterminé aléatoirement, et améliore la qualité de l'ensemble des solutions d'itération en itération, de sorte que les solutions convergent vers la solution optimale (idéalement). Cette convergence est obtenue par l'évaluation de la qualité des solutions, c'est le rôle de la *fonction de viabilité* (*fitness*), et par l'application de la sélection et d'*opérateurs génétiques* comme le crossover et la mutation.

3.2.2 Rôle de la sélection

La *viabilité* (*fitness*) est une mesure conventionnelle simulant la capacité de l'organisme à prévaloir dans la lutte à la reproduction. Comme l'organisme n'est pas mis en situation de lutter réellement pour sa dominance (puisque l'individu et le *phénotype* ne sont pas générés), il faut arbitrairement définir une mesure basée sur les informations disponibles. En règle générale, cette mesure est exprimée par une fonction dont les paramètres sont les composantes du génotype de l'individu (contenu et/ou localisation des gènes). Cette mesure peut occasionnellement faire intervenir des paramètres extérieurs (liste de localisations de villes pour le problème du voyageur de commerce et le problème de Steiner, liste d'items pour le problème du sac de contrebandier), voire environnementaux.

L'algorithme génétique utilisera ensuite une *stratégie de sélection* permettant, sur base de la viabilité des individus, de choisir ceux qui seront amenés à se reproduire. En se reproduisant, ils génèreront une descendance qui bénéficiera des gènes des reproducteurs, avec un brassage amené par les opérateurs génétiques, dont nous parlerons ci-après. La sélection, dans le cadre des algorithmes génétiques, est une métaphore de la sélection naturelle. Comme dans la nature, un même individu peut avoir plusieurs chances de se reproduire, et certains peuvent n'avoir pas de descendance. Par contre, en règle générale, l'effectif de la population est constant.

Diverses stratégies existent pour simuler la sélection. Nous en détaillerons quelques-unes au chapitre 5, mais notons les points importants déterminant l'efficacité de la stratégie de sélection.

Le but de la stratégie de sélection est d'amener les individus de meilleure viabilité à se reproduire relativement plus souvent que ceux de faible viabilité (selon un modèle probabiliste). L'idée derrière cette approche est que les individus plus viables possèdent probablement des gènes utiles. En les amenant à se reproduire plus souvent, on espère améliorer l'ensemble de la population de génération en génération. Nous allons bientôt formaliser cette intuition.

Insistons cependant d'emblée sur les qualités attendues de la stratégie de sélection. Le point majeur est la bonne balance de la convergence.

Un algorithme génétique ne doit pas converger *trop* vite. Il est clair que si l'on sélectionne le meilleur individu de la population de départ, on aura une solution relativement bonne (tout du moins par rapport à la qualité moyenne de la population), mais on sera probablement très loin de la solution optimale. Si l'on obtient rapidement beaucoup de copies de cet individu relativement riche en termes d'écart à la moyenne de la population, mais probablement pauvre en termes de solution au problème, la population n'aura plus l'occasion de s'améliorer. Une *convergence prématurée* peut aussi causer un blocage sur un optimum local, ce qu'un maintien de la diversité de la population dans les premières générations permet généralement d'éviter. Une pression de sélection relativement faible[1] est donc généralement requise en début de traitement par algorithme génétique. C'est un des éléments les plus importants en influençant l'efficacité.

D'un autre côté, on voudra également converger *raisonnablement* vite. Si la pression de sélection est insuffisante, la population ne s'améliorera pas, sa qualité stagnera. Le terme *stagnation* est d'ailleurs utilisé pour désigner le blocage de la convergence d'un algorithme [2]. De plus, comme on le verra plus loin, l'approche de la solution par un algorithme génétique est généralement asymptotique, ce qui implique que les individus de la population, en fin d'exécution, ont des valeurs de viabilité très proches. Une pression de sélection faible n'arrivera plus à les départager, et la sélection deviendra pratiquement aléatoire. Il est donc préférable, en toute généralité, que la pression de sélection soit plus forte en fin d'exécution de l'algorithme.

Notons encore qu'un individu de moins bonne qualité pour la fonction de viabilité peut très bien se trouver à proximité d'un meilleur optimum que ceux poursuivis par les individus de meilleure viabilité. C'est encore une bonne intuition de l'importance de laisser une chance raisonnable aux individus moins viables, et donc de bien gérer la pression de sélection.

La stratégie de sélection sera donc choisie afin de préserver ces contraintes au mieux. En fonction du problème, certaines stratégies conviendront mieux que d'autres : les problèmes traités nécessitent idéalement des modèles d'évolution de la pression de sélection adaptés. Certaines stratégies gèrent très finement cet aspect, en laissant à l'utilisateur la possibilité de faire évoluer plus ou moins vite la pression de sélection (stratégie de sélection Boltzmann), ou de faire évoluer cette pression selon un modèle donné (sigma scaling).

[1] Quant à savoir "combien" cette pression doit être faible, c'est un paramètre extrêmement dépendant du problème traité.

[2] Notons que la stagnation peut indiquer l'approche de l'optimum. Il faut cependant que l'algorithme ait d'abord convergé raisonnablement pour considérer cela comme une indication utile.

La stratégie classique, décrite par Goldberg et probablement utilisée exclusivement dans les premiers temps des algorithmes génétiques, est la stratégie *proportionnelle à la viabilité*. L'idée, très simple, consiste à sélectionner les individus aléatoirement mais en fonction d'une pondération proportionnelle à leur valeur de viabilité. En d'autres termes, c'est comme si on jouait à la roulette (le jeu de casino), mais les différents numéros (métaphore pour les individus) recevraient des cases de taille variable, proportionnelle à leur viabilité, ce qui leur donnerait une probabilité de sélection plus ou moins élevée. D'où le surnom de cette stratégie, la *sélection par la roulette* (*roulette wheel selection*).

On parle bien ici d'une *probabilité de se reproduire* : chaque nouvel individu est généré par deux individus de la génération précédente, sélectionnés selon un modèle probabiliste, et chaque individu participe à chaque tirage au sort, et donc un même individu peut être sélectionné plusieurs fois (de même que les numéros à la roulette).

Cette stratégie classique est assez efficace mais pose plusieurs problèmes d'implémentation. De plus, d'autres stratégies, parfois plus simples (par exemple la stratégie du tournoi), donnent des résultats au moins aussi bons. Nous en détaillerons quelques-unes dans les chapitres suivants.

3.2.3 Création d'une nouvelle génération

Une fois les individus "reproducteurs" sélectionnés, il faut effectivement générer une nouvelle population basée sur leur patrimoine génétique. Notons que dans la plupart des cas, les algorithmes génétiques travaillent à effectif constant, c'est à dire que chaque génération compte autant d'individus que la précédente. Dans cette introduction, nous nous limiterons à cette hypothèse.

Dans la nature, la reproduction est le cadre de différents phénomènes de recombinaison et de modification des gènes. Les algorithmes génétiques utilisent des métaphores de certains de ces phénomènes élémentaires. Comme signalé précédemment, la métaphore a cependant ses limites, ce qui est assez naturel puisque la métaphore n'est pas un but en soi. De fait, les algorithmes génétiques utilisent souvent des opérateurs génétiques totalement originaux (c'est à dire qui ne s'inspirent pas de mécanismes naturels).

Les opérateurs les plus centraux en théorie des AG sont le *crossover*[3] et la *mutation*.

La plupart des opérateurs génétiques entrent *grosso modo* dans deux catégories : les opérateurs "combinatoires" qui combinent deux individus pour en générer un nouveau, et les opérateurs "mutateurs" qui opèrent une modification d'un individu donné. La création d'une nouvelle génération s'opère généralement par l'application d'un opérateur de chaque type.

Le *crossover* permet de générer un individu à partir de deux parents. L'idée générale est que le descendant sera composé d'un panachage du patrimoine génétique des parents[4]. Les algorithmes génétiques proposent des variantes de

[3] Les traductions possibles du terme "crossover" prêtent toutes à une certaine confusion, et la littérature francophone ([Rostand et Tétry 1972], [Encyclopedia Universalis, Génétique, 1979]) utilise généralement le terme tel quel, je me conforme donc à cet usage.

[4] Cet opérateur existe dans la nature et permet un genre de reproduction sexuée pour les organismes haploïdes ; il s'applique également pour des organismes diploïdes pendant la phase essentiellement haploïde de la méiose (voir [Encyclopedia Universalis, Génétique, 1979])

ce schéma de base : le crossover peut s'opérer en un point, deux points ou de multiples points. Nous décrirons les variantes que nous avons implémentées au chapitre 5. Pour l'instant, considérons que le crossover simule le croisement de deux individus avec l'échange génétique qui en résulte et forme un nouvel individu dont le génotype contient un mélange variable des caractéristiques des parents.

La *mutation* est un phénomène un peu plus marginal (tout au moins en probabilité) mais néanmoins essentiel, aussi bien dans la nature que dans les AG. Il consiste en un changement localisé et aléatoire d'un gène. La fréquence des mutations est très faible, mais elle permet une diversification occasionnelle nécessaire pour éviter les phénomènes de stagnation. Dans le domaine des algorithmes génétiques, la mutation se traduira par l'inversion aléatoire d'un bit donné sur un chromosome en fonction d'une certaine probabilité (faible).

Une nouvelle génération sera donc générée depuis le groupe d'individus sélectionnés, par combinaison au moyen du crossover d'individus choisis aléatoirement dans ce groupe, et par mutation à probabilité faible de tous les nucléotides de tous les individus.

Contrairement à ce qui se passe dans la nature, le crossover des algorithmes génétiques se produit au niveau des nucléotides et non des gènes. Comme on le verra au chapitre 5, l'extension de ce mécanisme au niveau des gènes réclame des adaptations non négligeables du fait des contraintes particulières des algorithmes génétiques.

3.2.4 Critère d'arrêt

Nous savons à présent que l'on sélectionne les individus reproducteurs, et que l'on génère une nouvelle population en les amenant à se reproduire. Cette population sera sensiblement modifiée par mutation. La question finale est : quand arrête-t-on le processus, en d'autres termes, quand considère-t-on que la solution trouvée est suffisamment proche de l'optimum ?

Une première remarque s'impose suite à cette question. Les algorithmes génétiques ne peuvent pas prétendre, en toute généralité, à la localisation de l'optimum exact. Ils ont une approche asymptotique de la solution, ce qui ne signifie pas que la solution ne puisse pas être atteinte, mais que la "vitesse" d'approche de la solution diminue au fil de l'exécution de l'algorithme, et atteindre la solution précise peut être extrêmement coûteux en temps machine (d'autant qu'il n'y a généralement pas d'indication directe que la solution est atteinte). Dans la plupart des cas, il faut être prêt à accepter une solution approchée pour garder un niveau de performance acceptable.

La nature des problèmes étudiés doit rendre cette contrainte acceptable. C'est en pratique souvent le cas. Si l'on veut optimiser la production d'un ou plusieurs produits à partir de matières premières, par exemple, on peut accepter une perte de rentabilité de quelques fractions de pourcentage par rapport à la solution optimale, puisque la production elle-même générera sans doute des pertes de matière. La précision des algorithmes génétiques est généralement largement suffisante dans ce genre de situation.

Il existe plusieurs stratégies d'arrêt pour les algorithmes génétiques. Une option est de prévoir un nombre suffisant de générations pour s'assurer que la convergence a amené le problème assez près de la solution. Cette solution n'est évidemment pas optimale puisque soit on prend le risque d'avoir une approxi-

mation très large en limitant le nombre de générations, soit on prend le risque de gaspiller un grand nombre d'itérations.

Une autre option consiste à considérer un "seuil" au-delà duquel on considère la solution comme assez bonne. Ceci implique cependant une connaissance préalable approximative de la valeur de la solution optimale. Pour certains problèmes, cette situation est réaliste. Dans le problème d'ordonnancement classique du sac de contrebandier[5], par exemple, on sait où se situe l'optimum ultime : quand le sac est rempli au maximum de sa capacité. Naturellement, c'est en toute généralité un but qu'on ne peut atteindre. Mais on peut accepter un certain pourcentage de perte, quitte à générer plusieurs exécutions pour s'approcher par petits pas d'une meilleure solution.

On peut aussi utiliser la stagnation : une stagnation qui suit une convergence active de la population pendant plusieurs générations, peut être un signe que l'optimum est atteint ou approché de très près. On peut améliorer cette technique en forçant la population à se re-diversifier, par exemple en remplaçant une proportion donnée de la population, sélectionnée aléatoirement, par des individus nouvellement générés (*kick*). En appliquant cette technique à plusieurs reprises, on améliore la probabilité que l'optimum trouvé soit bien un optimum global.

Il est clair que toutes ces techniques peuvent être utilement combinées, selon les problèmes, pour fournir un critère d'arrêt satisfaisant.

3.3 Théorie des schémas

3.3.1 Introduction

L'efficacité des algorithmes génétiques est relativement contre-intuitive. En effet, partant d'une population générée aléatoirement et la faisant évoluer avec des méthodes essentiellement stochastiques, on ne s'attendrait pas *a priori* à ce que l'on converge vers la solution. Or, c'est bien ce qui arrive, comme un grand nombre d'expérimentations différentes des algorithmes génétiques l'a déjà prouvé : non seulement les algorithmes génétiques sont capables d'approximer d'une manière très satisfaisante la solution des problèmes, mais de plus ils le font efficacement et d'une manière relativement constante[6].

Holland et Goldberg [Goldberg 1989] ont développé une théorie complète tendant à montrer comment un algorithme génétique peut améliorer sa meilleure solution de génération en génération. C'est la théorie des schémas[7]. Nous allons en donner une courte explication inspirée du compte rendu de Goldberg, un développement complet de cette théorie étant en-dehors du propos de ce mémoire.

---

[5] Qui consiste à remplir un sac de capacité donnée avec des objets d'un poids donné, de manière à maximiser la valeur des objets se trouvant dans le sac.

[6] Des disparités ponctuelles peuvent cependant apparaître, et des disparités peuvent d'ailleurs être forcées par l'algorithme lui-même, en cas de *kick* par exemple. Mais dans la plupart des cas la convergence est plutôt constante.

[7] La littérature anglo-saxonne utilise le pluriel *schemas* ou *schemata* indifféremment

3.3.2 Schémas - définitions

Considérons un chromosome comme une chaîne de bits. Nous ignorons pour l'instant le fait que les bits forment des gènes, et que les gènes forment à leur tour le chromosome[8]. Nous ignorons également toujours la distinction entre chromosome et individu (entre génotype et phénotype). Chaque bit peut prendre la valeur 1 ou 0, en d'autres termes, le chromosome est exprimé au moyen de l'alphabet $V = \{0, 1\}$.

Nous allons définir un vocabulaire nous permettant d'exprimer l'enrichissement des *patterns* au cours du temps, de génération en génération. Pour cela, nous allons considérer des nucléotides "fixés", c'est-à-dire dont les valeurs sont connues, et des nucléotides de valeur quelconque, que nous allons représenter par un astérisque. L'alphabet, constitué des valeurs fixes et de l'astérisque $V+ = \{0, 1, *\}$ permet de définir des schémas de chromosomes.

Considérons par exemple, pour un chromosome constitué de 8 bits/nucléotides, le schéma suivant H=**1**0**. Le chromosome 11111000 est représentatif de ce schéma, 00000000 pas.

Les schémas ont certaines caractéristiques particulières. Certains sont par exemple plus *spécifiques* que d'autres. 1010101* est très spécifique, 1******* très peu. Certains fixent une plus large portion du chromosome, par exemple 1*****1*, ou plus courte comme 1*0*****. Ces caractéristiques sont importantes pour le crossover, car un schéma couvrant une plus longue portion de chromosome sera plus facilement cassé par un crossover.

Goldberg définit deux propriétés des schémas permettant de définir leurs caractéristiques, l'*ordre* et la *longueur définissante*.

L'ordre (*order*) d'un schéma est simplement le nombre de positions fixes. Par exemple, O(1*11**01)=5.

La longueur définissante (*defining length*) est la différence entre l'index de la première position fixe et l'index de la dernière. Par exemple, δ(*1*1**0*)=7-2=5.

3.3.3 Les schémas interprétés comme des hyperplans

A la suite de Goldberg, il est usuel d'utiliser la lettre H pour désigner un schéma. En réalité, cette convention provient de ce qu'un schéma donné peut être vu comme un hyperplan dans un espace discret (binaire) à n dimensions, n étant la longueur du chromosome (le nombre de nucléotides ou bits).

Par exemple, considérons un chromosome à trois nucléotides, ce qui correspond donc à un espace à trois dimensions pratique à représenter. Dans cette situation, il est clair que le schéma d'ordre 0 *** représente l'ensemble des points de l'espace, les schémas d'ordre 1 (par exemple 1**, 0**, **1, *0*, etc) représentent des plans de cet espace, les schémas d'ordre 2 représentent des droites (des plans à une dimension), et les schémas d'ordre 3 sont des points entièrement définis.

[8] [Goldberg 1989] signale qu'il utilise le terme *gène* en référence aux bits qui composent un chromosome (page 28). Ceci me paraît incorrect et de nature à créer une certaine confusion sur la terminologie. Je m'en tiendrai pour ma part à considérer que les bits ($\Leftrightarrow$ nucléotides) composent les gènes qui eux-mêmes composent les chromosomes.

La figure 3.1 est inspirée de Goldberg et donne une représentation graphique de ces éléments (tous les détails ne sont pas montrés pour des raisons de lisibilité).

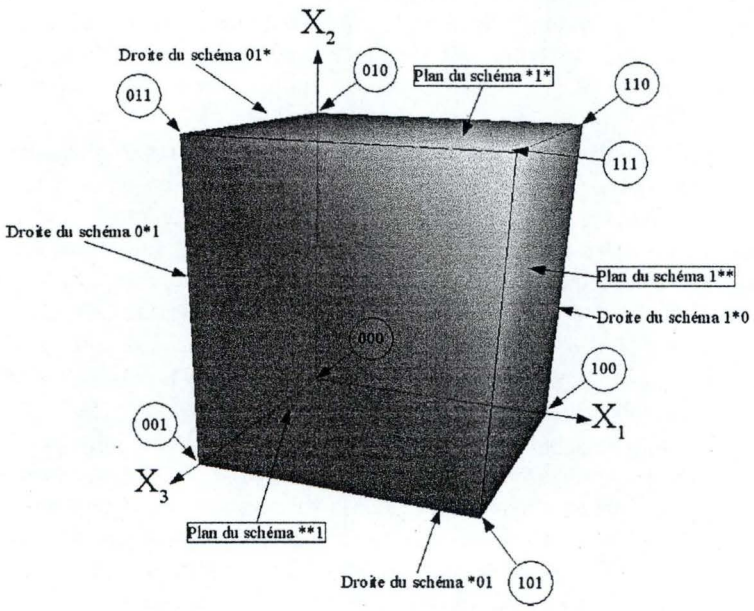


FIG. 3.1 – Les schémas vus comme des hyperplans

De même, pour un chromosome de longueur 4, les schémas d'ordre 1 représentent des hyperplans à trois dimensions, les schémas d'ordre 2 représentent des hyperplans à deux dimensions (des plans), etc.

3.3.4 Influence de la sélection sur les schémas

La démonstration de Goldberg repose sur la supposition que la probabilité d'un individu d'être sélectionné pour la reproduction est proportionnelle à sa viabilité. C'est un point de vue un peu réducteur car les stratégies de sélection peuvent s'éloigner sensiblement de ce modèle, en particulier la probabilité de sélection d'un type d'individu peut être différente d'une génération à l'autre selon le modèle de sélection choisi. Cependant, il convient de voir cette hypothèse de proportionnalité entre la viabilité et la probabilité de sélection comme une hypothèse minimale, les autres stratégies de sélection étant des améliorations (généralement heuristiques), du modèle de base.

Sur base de la notation et des notions que nous venons de définir, nous allons tenter d'exprimer l'influence de la sélection seule (en ignorant pour l'instant les opérateurs génétiques) sur la population, dans l'hypothèse d'une probabilité de sélection proportionnelle à la viabilité.

Si pour une génération donnée de n individus (à un temps t), on a m individus représentatifs d'un schéma H donné, on notera : $m = m(H,t)$. Le nombre d'individus représentatifs d'un schéma donné varie naturellement d'une génération à l'autre.

Dans les hypothèses que nous avons prises, un individu sera sélectionné et copié[9] pour former la génération suivante avec une probabilité proportionnelle à sa viabilité. La probabilité qu'un individu A_i soit sélectionné pour générer une copie de lui-même pour la génération suivante est donc le rapport entre sa viabilité et l'ensemble des viabilités des individus de la population, soit :

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

où f_i est la viabilité de l'individu A_i. Le nombre de représentants du schéma H que l'on s'attend en moyenne à retrouver à la génération t+1 est donc[10] :

$$m(H, t+1) = n \cdot \frac{\sum_{A_i \in H} f_i}{\sum_{j=1}^{n} f_j}$$

Pour exprimer l'évolution des représentants du schéma par rapport à la génération précédente, on peut écrire :

$$m(H, t+1) = \frac{m(H,t)}{m(H,t)} \cdot n \cdot \frac{\sum f_i}{\sum f_j}$$

ou :

$$m(H, t+1) = m(H,t) \cdot n \cdot \frac{\frac{\sum f_i}{m(H,t)}}{\sum f_j}$$

Si l'on note $f(H)$ la viabilité moyenne des représentants (c'est à dire $\frac{\sum f_i}{m(H,t)}$) du schéma H, on obtient

$$m(H, t+1) = m(H,t) \cdot n \cdot \frac{f(H)}{\sum f_j}$$

Enfin, en tenant compte du fait que la viabilité moyenne de la population est $\overline{f} = \frac{\sum f_j}{n}$, on peut écrire :

$$m(H, t+1) = m(H,t) \frac{f(H)}{\overline{f}} \tag{3.1}$$

Ce résultat montre que le nombre d'individus d'un schéma donné croît ou décroît d'une génération à l'autre d'une manière proportionnelle à leur viabilité moyenne. En d'autres termes, si la viabilité moyenne des représentants d'un

[9] Notons d'une manière générale que dans les algorithmes génétiques, chaque nouvelle génération est constituée *par remplacement*, ce qui signifie qu'un individu ne participe jamais à plusieurs générations de suite. Naturellement, dans le cas explicatif qui nous occupe, où les nouvelles générations sont créées par simple copie des individus sélectionnés, une ou plusieurs copies conformes d'un individu peuvent se trouver dans la génération suivante.

[10] Par facilité nous utilisons la notation ensembliste $A_i \in H$ pour exprimer que l'individu A_i est représentatif ou compatible avec le schéma H.

schéma donné est supérieure à la viabilité moyenne de l'ensemble de la population, leur nombre croîtra proportionnellement à la génération suivante.

On peut obtenir un résultat supplémentaire si l'on considère un schéma qui resterait, de manière constante, un facteur c au-dessus de la viabilité moyenne. Sa viabilité est donc toujours $(1+c)\overline{f}$. L'équation dégagée précédemment nous permet d'écrire :

$$m(H,t+1) = m(H,t)\frac{f(H)}{\overline{f}} = m(H,t)\frac{(1+c)\overline{f}}{\overline{f}} = (1+c)m(H,t)$$

Si l'on admet que c reste constant au cours du temps, en commençant au temps 0, on obtient l'équation

$$m(H,t) = m(H,0)(1+c)^t$$

Ce qui montre qu'un schéma de viabilité constamment supérieure à la moyenne voit sa population représentative croître d'une manière exponentielle au cours des générations.

3.3.5 Influence du crossover sur les schémas

Jusqu'ici nous avons évalué l'influence de la sélection sur la composition de la population. Cependant, la sélection seule ne permet pas d'améliorer la population, elle permet juste d'améliorer la viabilité moyenne en favorisant les individus de meilleure viabilité. Cependant, de meilleures solutions ne sont pas recherchées, et le meilleur individu créé aléatoirement dans la génération de base restera le meilleur individu au cours du temps.

Le rôle du crossover est de générer de nouveaux individus par recombinaison des anciens. Deux individus sont choisis au hasard dans la population sélectionnée, et un ou plusieurs points sur la chaîne de nucléotides sont choisis pour découper les chromosomes, après quoi les tronçons de chromosomes sont recombinés par échange d'un tronçon sur deux. L'exemple ci-dessous illustre un crossover à un point :

$$\begin{array}{c|c} 1111 & 1111 \\ \hline 0000 & 0000 \end{array} \quad \rightarrow \quad \begin{array}{c|c} 1111 & 0000 \\ \hline 0000 & 1111 \end{array}$$

Ci-dessous, un crossover à deux points :

$$\begin{array}{c|c|c} 01 & 010 & 10 \\ \hline 00 & 000 & 00 \end{array} \quad \rightarrow \quad \begin{array}{c|c|c} 01 & 000 & 10 \\ \hline 00 & 010 & 00 \end{array}$$

A l'instar de Goldberg, nous allons considérer uniquement le crossover à un point[11] dans notre démarche de formalisation de l'effet du crossover sur les schémas, et considérer les crossovers à points multiples comme des extensions des résultats que nous dégagerons. Considérons le chromosome suivant et deux schémas dont ce chromosome est représentatif :

		1	2	3	4	5	6	7
A	=	0	1	1	1	0	0	0
H_1	=	*	1	*	*	*	*	0
H_2	=	*	*	*	1	0	*	*

[11] Auquel nous nous référerons aussi sous le terme de "crossover simple".

En étudiant l'effet du crossover sur ces schémas, nous considérerons qu'en toute généralité l'individu avec lequel ce chromosome sera "croisé" est suffisamment différent pour ne pas préserver le schéma (c'est à dire qu'une partie de chromosome perdue lors du crossover ne sera pas remplacée par une partie identique, ce qui peut se produire, mais avec une probabilité faible).

Si l'on opère un crossover simple en position 3 (après le bit 3), on voit immédiatement que le schéma H_1 sera perdu (parce que le 1 et le 0 qui constituent les positions fixes du schéma se retrouveront séparés sur deux individus distincts), alors que le schéma H_2 sera préservé :

		1	2	3	4	5	6	7
A	=	0	1	1	1	0	0	0
H_1	=	*	1	*	*	*	*	0
H_2	=	*	*	*	1	0	*	*

Cet exemple permet de développer l'intuition que le schéma H_2 a une probabilité plus grande d'être préservé. En effet, une coupure en positions 2, 3, 4, 5 et 6 détruit le schéma H_1 (soit cinq chances sur six), alors que seule une coupure en 4 peut détruire le schéma H_2 (soit une chance sur six). Observons que H_1 a une longueur définissante de 5 et six sites de crossover possibles. On peut ainsi exprimer la probabilité de destruction d'un schéma en fonction de sa longueur définissante :

$$p_d = \frac{\delta(H_1)}{(l-1)}$$

Naturellement, la probabilité qu'un schéma soit préservé est inversement $p_s = 1 - p_d$ et donc

$$p_s = 1 - \frac{\delta(H_1)}{(l-1)}$$

Si l'on inclut également le fait qu'une probabilité existe que le schéma soit préservé par combinaison heureuse avec un autre chromosome, et si l'on considère que généralement le crossover est soumis à un tirage au sort selon une probabilité donnée p_c (de sorte que toutes les tentatives de crossover n'aboutissent pas à un crossover), on peut écrire la probabilité de survie d'un schéma :

$$p_s \geq 1 - p_c \cdot \frac{\delta(H)}{l-1} \tag{3.2}$$

Cette équation est identique à la précédente si le crossover est systématique, c'est à dire si sa probabilité p_c est de 1.0, et si l'on ignore les combinaisons heureuses préservant le schéma.

Si l'on combine ce résultat à l'équation 2.1. décrivant l'évolution du système sous l'effet unique de la reproduction, on obtient :

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\overline{f}} \left[1 - p_c \cdot \frac{\delta(H)}{l-1}\right] \tag{3.3}$$

Ceci donne donc la quantité minimale de représentants du schéma qui sont produits à la génération suivante sur base de la sélection et du crossover simple. On voit que l'évolution du nombre de représentants d'un schéma donné sous les conditions définies ici est dépendante d'un facteur composé

- du rapport entre la viabilité moyenne des représentants du schéma et la viabilité moyenne de la population d'une part,
- de la longueur définissante du schéma d'autre part.

3.3.6 Influence de la mutation sur les schémas

La mutation est le dernier des opérateurs élémentaires. L'effet de la mutation est une inversion aléatoire, dirigée par une probabilité donnée, d'un bit (nucléotide) donné. En d'autres termes, pour chaque chromosome de la population ayant subi la sélection et le crossover, on va effectuer un tirage au sort de probabilité faible sur chacun de ses bits constitutifs. Si le tirage est positif, le bit sélectionné est inversé. La probabilité doit donc être faible sans quoi la mutation deviendrait extrêmement disruptive et détruirait les effets combinatoires des autres opérateurs.

Notons p_m la probabilité associée à la mutation (comme p_c, cette probabilité est un des paramètres de la configuration de l'algorithme génétique et est fixée par l'utilisateur). La probabilité pour qu'un bit donné survive à la mutation est donc de $1 - p_m$. Pour que le schéma H survive, toutes les positions fixes du schéma doivent survivre. Si l'on note $O(H)$ le nombre de positions fixes du schéma, la probabilité de survie du schéma est exprimée en multipliant $1 - p_m$ par lui-même $O(H)$ fois, ce qui donne une probabilité de survie du schéma de :

$$(1 - p_m)^{O(H)} \tag{3.4}$$

Pour de petites valeurs de p_m (ce qui est cohérent avec l'hypothèse générale, puisque p_m doit rester petit pour éviter que la mutation ne détruise les effets de la sélection et du crossover), on peut exprimer cette probabilité de survie par :

$$1 - O(H) \cdot p_m \tag{3.5}$$

Dans l'équation 2.3, si l'on diminue l'effet du crossover de la probabilité de disruption du schéma par la mutation, on obtient l'équation finale exprimant les effets combinés de la sélection, du crossover et de la mutation :

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{\overline{f}} \left[1 - p_c \frac{\delta(H)}{l-1} - O(H) p_m\right] \tag{3.6}$$

Ceci change très peu la conclusion précédente, à savoir que les schémas courts et de viabilité au-dessus de la moyenne sont représentés en nombre croissant exponentiellement d'une génération à l'autre. On voit juste que c'est encore plus valable pour les schémas d'ordre faible. Cette conclusion est l'essence du *théorème des schémas*, ou *théorème fondamental des algorithmes génétiques.*

3.3.7 Approche qualitative

Le théorème des schémas permet de quantifier l'évolution statistique des schémas dans la population au cours de l'exécution de l'algorithme génétique. Elle ne donne cependant pas d'indication qualitative, en d'autres termes, est-ce vraiment intéressant de donner aux schémas de viabilité moyenne élevée un nombre de représentants croissant de manière exponentielle ?

Une bonne indication qualitative est obtenue en comparant le fonctionnement de l'algorithme génétique avec un problème classique du domaine de la théorie décisionnelle statistique, le problème du *bandit 2-manchot* ou *k-manchot* (*2-armed bandit* ou *k-armed bandit*).

Le *bandit manchot* est le nom trivialement donné aux machines à sous (*slot machine*) ayant traditionnellement un bras métallique sur lequel on tire pour

faire tourner des tambours ; si les tambours tombent sur une combinaison de valeurs donnée, le joueur reçoit un certain nombre de fois sa mise, voire l'ensemble du contenu de la cagnotte de la machine (*jackpot*).

Dans la théorie du bandit 2-manchot, la machine a deux parties distinctes, chacune avec son propre bras, sa propre probabilité de gain, et sa propre cagnotte. On sait d'avance que l'un des bras de la machine a une probabilité de gain plus importante – mais on ne sait pas lequel. L'idée est de maximiser le gain par un apprentissage par l'expérimentation. L'aspect intéressant de cette théorie est que l'apprentissage lui-même est une source de gain ou de pertes, et doit donc être optimisé pour garantir à la fois l'efficacité de l'apprentissage et un gain raisonnable (ou une limitation des pertes).

L'étude de ce problème peut être étendue à des situations plus complexes, où le nombre de dimensions du problème est supérieur à deux, on parlera donc alors de problème du bandit k-manchot, où k est la dimension du problème.

L'important pour traiter ce problème est de définir une bonne balance entre l'*exploration* (c'est à dire l'apprentissage) de l'*exploitation* (c'est à dire le ramassage des gains permis par l'apprentissage). La phase d'exploration permet de réunir un ensemble limité d'informations sur le problème, qui permet de prendre certaines décisions avec un minimum de connaissance de cause. Ces décisions vont donc conditionner la phase d'exploitation : si la phase d'exploration a été efficace (ou heureuse), la phase d'exploitation sera profitable. La phase d'exploitation, cependant, est encore l'occasion d'affiner les résultats obtenus précédemment.

Nous n'allons pas détailler plus loin les stratégies utilisées, ni développer la théorie statistique du bandit manchot, ce qui nous éloignerait trop de notre sujet, mais insistons sur l'analogie entre ce problème et le fonctionnement de l'algorithme génétique.

En effet, le fonctionnement de l'algorithme génétique est une manière d'apprendre en expérimentant. La création de la première génération est une première exploration. Elle permet de tirer des conclusions préliminaires sur la manière de résoudre le problème, et de ce fait, elle influence l'évolution future de la résolution du problème. Les conclusions préliminaires sont en fait tirées par la découverte de schémas favorables (ce qui est indiqué par la fonction de viabilité). Comme dans le cas du bandit k-manchot, ces conclusions préliminaires ne sont peut-être pas optimales, mais permettent de faire des choix limitant les dégâts.

L'algorithme va ensuite exploiter ces choix, en créant de nouvelles générations mettant en avant les choix effectués précédemment. Ces nouvelles générations apporteront de nouvelles informations permettant d'affiner les critères de choix, jusqu'à obtenir une solution idéale (ou presque).

3.4 Difficultés usuelles

3.4.1 Généralités

Les algorithmes génétiques sont des outils extrêmement versatiles et puissants pour régler des catégories de problèmes très variés, comme nous le verrons en plus grand détail dans les chapitres ultérieurs. Il est cependant utile de s'intéresser quelques instants aux limites et difficultés rencontrées par les al-

gorithmes génétiques, en particulier le problème extrêmement important de la représentation.

Ce problème est de deux ordres. D'abord, il n'est pas toujours trivial de trouver une manière de "plier" un problème à la représentation requise par les algorithmes génétiques. Ensuite, une représentation choisie peut biaiser l'algorithme génétique d'une manière sensible.

Nous évoquerons ensuite quels sont les problèmes typiquement difficiles à résoudre au moyen des algorithmes génétiques.

3.4.2 Adaptation des problèmes à la résolution par algorithme génétique

Prenons un exemple pour démontrer les problèmes que l'on peut rencontrer. Supposons que nous devions optimiser une fonction quelconque à n dimensions. Nous coderons les variables correspondant à chaque dimension comme autant de gènes sur un chromosome-individu. La viabilité sera calculée en appliquant la fonction sur les valeurs portées par les gènes (la manière dont la viabilité s'exprime dépend naturellement du problème à traiter).

Les gènes doivent être codés en fonction de l'espace de solutions que l'on veut investiguer, à savoir son étendue et la finesse de la résolution de l'espace que l'on veut considérer. D'une manière générale, on essaiera de diminuer le nombre de points de l'espace à traiter. Par exemple, si l'on veut déterminer la position d'un point sur un terrain (pour un problème quelconque dont la nature ne nous intéresse pas pour cet exemple), si ce terrain est un carré d'un kilomètre de coté, et si l'on veut une résolution de l'ordre du millimètre, on codera les deux gènes représentant les coordonnées x et y afin d'obtenir des valeurs entières entre 0 et 1.000.000, c'est à dire le nombre de millimètres que l'on trouve dans un kilomètre. Ceci peut être codé, en représentation binaire classique, sur 20 bits ($2^{20} = 1.048.576$).

Si l'on utilise une représentation standard en entier non signé, c'est à dire sur 32 bits, l'espace considéré par l'algorithme génétique est étendu à 4.294.967.296, soit un espace environ 4000 fois trop grand par dimension, soit 16.000.000 fois trop grand pour le problème à deux dimensions que nous considérons. Il est souhaitable dans ce cas de réduire la représentation au minimum requis afin d'améliorer le rendement de l'algorithme.

Même en négligeant ces contraintes, le rendement peut être très acceptable. Il est clair cependant qu'un problème complexe, par exemple à variables multiples, gagnera nettement en performance si l'on tient compte de ces contraintes.

En-dehors de la performance, un autre problème potentiellement gênant peut se poser. L'algorithme génétique est agnostique en ce qui concerne le type de problème qu'on lui soumet. Ce qui signifie que seul le codage des gènes détermine le domaine des variables traitées. Dans l'exemple général décrit ci-dessus, on pourrait imaginer que l'optimum global se trouve en-dehors du terrain, dans l'espace situé entre le domaine désiré des variables, soit [0,1.000.000] et le domaine réel du type utilisé, soit [0,1.048.576], ce qui représente un espace de 48.576 points par dimension. Selon les problèmes et leur codage, cette différence peut être encore plus notable. Il n'est probablement pas acceptable de considérer cette solution, qui amènerait dans un cas réel à placer la solution chez son voisin.

Deux solutions se présentent. La première consiste à redistribuer le domaine des variables sur le domaine réel du problème. C'est acceptable parce qu'on

met à l'échelle tous les paramètres du problème avec sa solution. Cette solution affinant en finale l'espace des solutions, elle force juste l'arrondi au point le plus proche de l'espace réel. Dans notre exemple, cette option est probablement praticable, mais certains problèmes peuvent ne pas se prêter à cette mise à l'échelle.

La seconde solution consiste à refuser purement et simplement les points se situant en-dehors du domaine réel. Nous l'avons dit, l'algorithme génétique est agnostique en ce qui concerne le domaine réel du problème, seul le domaine des gènes interprétés a du sens pour lui. Une solution est possible cependant, et elle consiste à gérer le domaine réel au niveau du calcul de la viabilité. La fonction de viabilité peut être prévue pour donner une viabilité extrêmement mauvaise à tout point se trouvant en-dehors de l'espace réel recherché. Ceci impose cependant une certaine prudence. En effet, si l'optimum se trouve à proximité de la limite, il peut devenir de ce fait plus difficile à atteindre, parce que certains chromosomes proches de la solution, et comportant donc des schémas utiles, peuvent être vus comme de mauvais candidats par la méthode de sélection. De sorte que certains "bons schémas" peuvent être perdus dans l'opération.

3.4.3 Problèmes liés à la représentation

Une fois la représentation fixée efficacement, d'autres problèmes plus subtils liés à celle-ci peuvent se produire. C'est par exemple le cas lorsqu'on utilise des gènes de type entier signé.

Dans cette représentation, les entiers positifs sont codés binairement de façon naturelle, le zéro étant une chaîne de bits tous à 0, le 1 une chaîne de bits tous à 0 à l'exception du bit de poids le plus faible, etc. Par contre, pour des raisons de traitement des relations entre nombres, les entiers négatifs sont codés en notation inversée, et donc -1 est représenté par une chaîne de bits tous à 1 (en ce inclus le bit de signe), -2 par une chaîne de bits tous à 1 sauf le bit le moins significatif, etc.

L'algorithme génétique travaille avec des schémas ou, pourrait-on dire, des *patterns*. En l'occurrence, on voit que la représentation des nombres entiers positifs proches du zéro est extrêmement éloignée en termes de schémas de la représentation des nombres négatifs proches. Et en effet, si un optimum pour une variable donnée se trouve à proximité du zéro (en variables entières), l'algorithme génétique va rencontrer là une barrière infranchissable.

Supposons pour fixer les idées que l'optimum se trouve sur le zéro. Si la population de départ contient en moyenne plus de bits à 0 que de bits à 1, l'algorithme a de bonnes chances de trouver l'optimum exact parce que la population se "gonflera" de schémas contenant des zéros, favorables à la convergence vers le zéro.

Si à l'inverse la population de départ a une majorité de bits à 1, la population a plus de chances de converger vers -1. En effet, pour un nombre dont le bit de signe est à 1 (indiquant qu'il s'agit d'un nombre négatif), plus il y a de 1 dans la représentation binaire, plus on sera proche de zéro[12]. Finalement, quand la solution du problème atteint -1, elle s'arrête de converger, parce que la population est dopée en bits à 1 et incapable par crossover (recombinaison de

[12] Le nombre composé de 0 binaires (excepté pour le bit de signe) est le plus petit nombre négatif, donc le plus éloigné de zéro ; au contraire, un nombre composé de 1 binaires représente -1, l'entier négatif le plus proche de zéro.

bits majoritairement à 1) ou mutation (qui ne modifiera pas la population de manière assez sensible) de converger vers un ensemble de bits à zéro.

C'est probablement souvent sans gravité, après tout, les algorithmes génétiques trouvent souvent dans le meilleur des cas une approximation satisfaisante de la solution, et rarement la solution elle-même, et donc une erreur d'une unité de résolution est généralement acceptable. Cependant, ce problème de représentation est une sorte de "barrière de potentiel" qui peut bloquer l'algorithme assez loin de l'optimum pour que ce soit problématique.

Des solutions peuvent être envisagées. Par exemple, on peut modifier l'interprétation de la valeur binaire de sorte que -1 soit représenté majoritairement par des zéros (en inversant tous les bits sauf le bit de signe). On peut aussi utiliser des nombres non-signés de même résolution et les reporter sur des valeurs signées lors du calcul de la viabilité. En fait, les solutions ne manquent pas, mais il faut avant tout être conscient du piège éventuellement créé par le codage binaire.

On peut également opter pour d'autres types, en particulier le type classique en virgule flottante (*float* ou *double*). Celle-ci pose également quelques problèmes assez subtils. Par exemple, l'interface entre la mantisse et l'exposant est un point de rupture pour les schémas, qui peut poser des problèmes d'efficacité à l'algorithme. Ensuite, à nombre de bits égal, la résolution d'un float est inférieure à celle d'un entier (du fait de l'espace occupé par l'exposant), bien que son domaine soit infiniment plus grand. En réalité, les nombres en virgule flottante sont des nombres à "résolution variable", ce qui n'est pas toujours acceptable. Enfin, le fait qu'un même schéma peut représenter des ordres de grandeur très éloignés en fonction du contenu de l'exposant doit inciter à une extrême prudence dans l'usage de ce genre de type.

La représentation des nombres pour les algorithmes génétiques implique une réflexion éduquée et parfois osée. Il peut parfois être utile de considérer des codages originaux des nombres, même s'ils ne sont pas supportées nativement par les machines ou les compilateurs, pour assurer un bon traitement des schémas.

3.4.4 Les problèmes difficiles pour les algorithmes génétiques

Les algorithmes génétiques sont des outils puissants mais cependant pas universels. Alors, quels types de problèmes sont-ils typiquement difficiles pour des algorithmes génétiques ?

Le problème de la représentation peut être une première difficulté pour un algorithme génétique, d'abord pour l'implémenteur (comment coder mon problème ?), ensuite pour l'algorithme génétique lui-même : une représentation mal choisie peut amener des solutions hors de l'espace recherché ou peut allonger indûment le temps d'exécution de l'algorithme, comme on l'a vu dans la section précédente.

Une fois le problème correctement implémenté, il peut quand même se révéler spécialement difficile à résoudre. La raison de cette difficulté est souvent une topologie particulière de l'espace des solutions. L'approche par schémas implique une certaine continuité topologique dans l'approche de l'optimum. Intuitivement, on peut se représenter que si l'optimum est isolé et entouré de solutions de faible qualité, il sera difficile à atteindre parce que les "mauvaises" solutions entourant l'optimum constitueront une sorte de "barrière de potentiel"

qui gènera l'approche de l'optimum. Le problème de la représentation de -1 en nombres entiers décrit plus haut en est un exemple simple.

Cette contrainte de continuité est cependant souvent trompeuse, parce que la continuité se marque au niveau des schémas, et non des valeurs interprétées des gènes. De fait, un espace des solutions relativement discontinu en valeurs mais cohérent au niveau des *patterns* de bits peut être un bon candidat pour un algorithme génétique. Prenons l'hypothèse d'un calcul de viabilité qui compte simplement le nombre de bits à 1 du chromosome (un exemple inspiré de [Mitchell 1996] et que nous utiliserons encore largement par la suite). L'espace des solutions est très discontinu si on le considère en valeurs entières. En effet, par exemple, la viabilité de la valeur 15 est 4 (la représentation binaire entière de 15 est 1111, soit quatre bits à 1), mais la viabilité de la valeur 16 est 1 (la représentation binaire entière de 16 est 10000, soit un bit à 1). L'espace des solutions de ce problème est rempli de discontinuités semblables.

L'algorithme génétique n'est cependant pas influencé par ces discontinuités apparentes, parce que dans ce cas la représentation de l'espace des solutions par une valeur numérique en abscisse n'est pas pertinente puisque que cette valeur n'apparaît pas dans le problème. En fait, une représentation plus correcte est celle d'un espace binaire à n dimensions, où n est la taille (en bits) du chromosome. Dans cette représentation, l'espace des solutions est continu : la formule de viabilité peut être exprimée par $f(x_i) = \sum_{i=1}^{n} x_i$ (comme x_i ne peut être égal qu'à 0 ou 1, ceci revient bien à compter le nombre de bits à 1), ce qui montre que la fonction de viabilité exprime un hyperplan à $n + 1$ dimensions.

Goldberg (in [Goldberg 1989]) étudie les problèmes ardus pour les algorithmes génétiques au moyen de ce qu'il appelle le "problème minimal trompeur" (*minimal deceptive problem*), c'est à dire le problème (de taille) minimum pouvant tromper un algorithme génétique et l'empêcher d'atteindre l'optimum. Le problème est défini artificiellement avec une barrière de potentiel de solutions à viabilité faible autour de l'optimum. Goldberg constate cependant que cet obstacle posé artificiellement pour le tromper n'est en pratique pas insurmontable pour un algorithme génétique.

Chapitre 4

Cuvier : un framework Java

4.1 Choix d'implémentation

4.1.1 Généralités

Le but final de ce travail est la mise au point d'un outil de démonstration clair, didactique, et aussi complet que possible. Ce but impose certaines contraintes mais laisse le champ très libre pour le choix des outils et les stratégies utilisés pour le développement.

En effet, en ce qui concerne par exemple la possibilité d'illustrer clairement le concept et les exemples, il y a actuellement pléthore de normes et outils permettant de réaliser une interface homme-machine très efficace. Notons à titre d'exemples HTML avec Javascript ou VBScript, Flash (de la société Macromedia), les interfaces graphiques natives (Windows, Gnome, KDE, etc), ou d'autres interfaces plus portables, comme QT, ou enfin des machines virtuelles portables proposant une interface graphique, comme Java/Swing.

En ce qui concerne la complétude, plusieurs stratégies sont possibles. On peut soit réaliser une implémentation *ad hoc* de différents problèmes et les présenter via l'interface choisie, ou utiliser des implémentations existantes de problèmes déterminés et les "enrober" d'une interface graphique, soit utiliser un moteur générique existant et le présenter d'une manière didactique, soit enfin réaliser un moteur qui réponde à nos contraintes.

Ce chapitre décrit les choix réalisés pour ce travail.

4.1.2 Stratégie générale

Utilité d'un framework

Il a paru intéressant, en dehors de l'aspect didactique du projet, de réaliser une base générale réutilisable et extensible.

Des ressources très nombreuses et diverses sont disponibles dans le domaine des algorithmes génétiques par le biais des institutions universitaires ou simplement sur Internet. Cependant, les recherches effectuées n'ont pas permis de localiser un candidat répondant complètement aux contraintes posées pour ce travail (la contrainte principale étant la démonstration d'un panel complet des

mécanismes de base)[1].

Quelques-uns des candidats étaient :

The GA Playground (arieldolan.com)Un excellent outil de démonstration qui vient avec un nombre impressionnant d'exemples, en particulier plusieurs implémentations du problème du voyageur de commerce, du problème de Steiner, de problèmes d'ordonnancement... C'est cependant une "boîte à outils" plutôt qu'un framework, et le contrôle que l'on a sur les classes de base est limité. De plus, la version actuelle n'implémente pas plusieurs mécanismes de sélection, ce qui est un point assez important dans un outil didactique. C'est cependant une des rares applications à vocation réellement didactique et de démonstration, et la seule tant soit peu généraliste.

Genocop (www.coe.uncc.edu)Développé par Zbigniew Michalewicz (www.coe.uncc.edu/~zbyszek/) de l'université de Caroline du Nord, est un moteur écrit en C ANSI portable (bien que principalement écrit pour les plates-formes Unix). Ce n'est pas un framework, et le moteur est principalement destiné à résoudre des problèmes d'optimisation à contraintes linéaires. Ce moteur était un temps une des références du domaine, mais les resources disparaissent d'Internet depuis qu'il est utilisé commercialement par la société NuTech (www.nutechsolutions.com). On peut trouver une applet Java utilisant ce moteur sur www2.ics.hawaii.edu.

Genitor Développé par le groupe de recherches sur les algorithmes génétiques de l'Université de l'Etat du Colorado (www.cs.colostate.edu/ genitor), Genitor est un ensemble de fonctionnalités développé en C. Celui-ci est cependant assez peu documenté, et fournit une variété de fonctionnalités assez limitée.

A défaut de disposer d'un outil présentant les qualités requises, il a donc fallu réaliser un nouveau moteur, en plus de l'interface. Restait à choisir l'outil de développement, et la philosophie d'implémentation du moteur.

4.1.3 Le choix du langage

Le langage Java (langage inventé par la société Sun Microsystems - voir [Java language specification 2000] pour la spécification du langage) a été choisi pour implémenter le moteur et l'application de démonstration. Ce choix n'est pas immédiat. En effet, beaucoup considèrent encore Java comme étant trop peu performant et robuste pour la réalisation de moteurs applicatifs. En ce qui concerne l'interface, ce choix n'est pas non plus immédiat, car comme indiqué plus haut, les alternatives ne manquent pas, dont certaines tout aussi gratuites que Java.

Avantages

Le choix de Java présente cependant plusieurs avantages. Le premier est la portabilité. Il n'est pratiquement plus à l'heure actuelle un système d'exploita-

[1] Ceci exclut naturellement les solutions commerciales qui sont généralement très spécialisées, très propriétaires et...très chères.

tion qui ne supporte pas Java, et sauf exception[2], le même code Java peut être exécuté sans recompilation sur toutes les plate-formes disposant d'une machine virtuelle.

Un autre avantage est le modèle orienté objet de Java, qui est extrêmement bien structuré et adapté à la réalisation de frameworks, comme il l'a prouvé amplement[3].

Un avantage non-négligeable est la disponibilité du langage comme une spécification publique, et les compilateurs comme des outils librement distribuables. Il existe également pas mal d'outils permettant de rédiger du code Java et gérer des projets Java dans différents modèles de distribution, dont certains *open source* ou *freeware*.

L'énorme avantage enfin est la disponibilité publique de bibliothèques professionnelles de fonctionnalités. Le kit de développement standard s'accompagne déjà d'une bibliothèque extensive, mais Sun Microsystems (propriétaire de Java) et des compagnies ou groupements tiers, en particulier la fondation software Apache (www.apache.org), proposent de nombreuses extensions gratuites très professionnelles, comme par exemple le projet Jakarta (jakarta.apache.org) et la bibliothèque Xerces de traitement de XML (xml.apache.org/xerces2-j). L'intérêt pour ce travail est la possibilité de réaliser facilement une interface graphique portable sur toute plate-forme graphique[4]. Un autre intérêt est la facilité d'implémenter du multithreading.

Désavantages

Un désavantage souvent cité pour Java n'est pas d'application ici, à savoir la lenteur relative de l'exécution. Il y a deux raisons principales à cela. La première est que le plus gros du travail d'exécution d'un algorithme génétique est peu influencé par la machine virtuelle. Ce genre de traitement est typiquement presque aussi rapide sur une machine virtuelle qu'avec du code natif. La seconde raison est due à l'optimisation des versions récentes de la machine virtuelle Java, qui est à présent extrêmement performante, en particulier en ce qui concerne la responsivité des interfaces graphiques.

Le choix de Java a cependant présenté certains désavantages, dont certains ne sont vraiment apparus qu'en cours de réalisation. Le premier est l'inexistence de types non-signés en Java. L'utilisation de ces types est un grand avantage pour certains algorithmes génétiques.

Le second désavantage majeur est le *garbage collector*, ce qui fut assez inattendu. Java ne permet pas à l'utilisateur de libérer la mémoire lui-même. Ce travail est pris en charge par la machine virtuelle d'une manière asynchrone dans un thread système dédié. En gros, Java détruit périodiquement les zones mémoires allouées mais non-référencées. Malheureusement, les algorithmes génétiques réclament idéalement beaucoup d'allocations/désallocations de mémoire, et très rapidement le garbage collector s'est retrouvé incapable de suivre, causant une dégradation très nette de la performance. Il a donc fallu

[2] Certaines conventions particulières à certaines plates-formes peuvent diminuer la portabilité, mais ces exceptions sont rares, et généralement des solutions existent pour éviter ces problèmes.

[3] Voir J2EE, Swing, XERCES, etc.

[4] L'interface de démonstration a été testée sur Windows et XWindows, ce qui couvre la grande majorité des interfaces graphiques dans le monde.

réaliser toute une série d'optimisations "non naturelles" du code pour que la performance ne soit pas dégradée par le garbage collector.

Enfin, le modèle orienté objet de Java ne connaît pas les notions C++ de *template* et *surcharge d'opérateurs* (voir[Stroustrup 1997] pour plus d'informations sur ces concepts), qui auraient eu pour effet de clarifier le framework.

Au final, une option intéressante aurait peut-être été de réaliser le framework en C++ portable (c'est à dire recompilable sur toute plate-forme disposant d'un compilateur C++) avec une interface utilisateur Java, liée au moteur au moyen de Corba, JNI ou même TCP/IP. La portabilité de C++ reste cependant limitée même si on fait l'effort de coder du C++ parfaitement ANSI/ISO, et pour ce projet le jeu n'en valait pas la chandelle.

Note

Le site de Sun Microsystems (java.sun.com) offre une source inépuisable d'informations utiles sur Java, le lecteur est invité à s'y reporter pour plus d'information sur les concepts discutés ici (en particulier java.sun.com/j2se/1.4.1/docs/api/index.html pour informations sur l'API et [Java language specification 2000] pour informations sur le langage, et java.sun.com/docs/books/tutorial/ pour un excellent tutoriel sur les fonctionnalités de l'API).

4.1.4 Choix particuliers

Philosophie orientée objet

Il m'a paru intéressant, dans le cadre d'une implémentation nouvelle d'un moteur d'algorithmes génétiques, d'en faire une implémentation résolument orientée objet, et d'utiliser pertinemment l'héritage, le polymorphisme, et les *patterns* typiques de cette approche programmative. La notion d'objet vient assez naturellement lorsqu'on travaille avec les concepts des algorithmes génétiques. Ceux-ci restent pourtant assez marqués par le code réalisé dans les années 70, en C ou en Pascal, et les implémentations modernes n'utilisent pas souvent pertinemment l'orienté objet.

Le choix des objets de base de l'implémentation est assez simple.

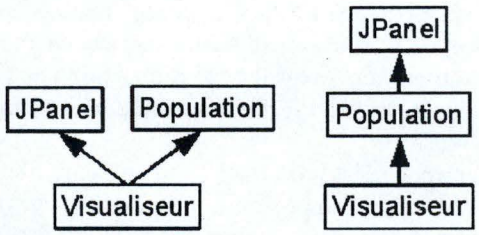
Un *chromosome* (strictement équivalent à un individu dans cette implémentation), est un objet qui s'impose naturellement. L'information portée par le chromosome peut se concevoir comme une chaîne de nucléotides (composants élémentaires mais non-signifiants de l'information génique) ou comme un stock de gènes (éléments signifiants élémentaires).

La représentation de la chaîne de nucléotides sera naturellement un champ de bits. On ne peut se passer de cette représentation parce que la plupart des opérateurs génétiques agissent directement sur les nucléotides. Par contre, les gènes peuvent être extraits programmativement de la chaîne de bits, ce qui permet d'éviter le dédoublement de l'information. Cependant, cette approche présente deux défauts : d'abord, elle représente un processus récurrent assez lourd, l'accès à l'information génique étant très fréquent ; ensuite, l'aspect pratique est fortement diminué, parce que cette méthode impose de développer un code d'extraction de l'information génique pour chaque implémentation de problème. Or, en séparant la représentation du gène de la chaîne de nucléotides,

la puissance de l'orienté objet permet de rendre le processus extrêmement simple, comme on le verra plus loin. Un objet *gène* a donc également été défini. Celui-ci est synchronisé avec la chaîne de nucléotides chaque fois que l'un ou l'autre est modifié.

Les algorithmes génétiques travaillent avec des *populations* d'individus. Il est assez naturel de considérer une population également comme un objet contenant un nombre donné d'individus (l'objet *chromosome*). La population contient également le code d'exécution de l'algorithme[5].

Enfin, une représentation graphique par défaut est proposée par une classe qui repose à la fois sur la classe population et la classe Java *JPanel*, lui permettant de s'intégrer dans n'importe quelle application graphique ou applet. Notons qu'en l'absence d'héritage multiple, ceci n'a pu être réalisé élégamment qu'en utilisant un "truc" : la classe population (non-visuelle) étend elle-même la classe JPanel, de sorte que la classe visuelle en hérite sans avoir à recourir à un double héritage (ce qui est impossible en Java).



Interdit en Java Solution choisie

La viabilité d'un individu est une donnée primordiale et fréquemment utilisée. Elle est fonction du contenu des gènes. Dans cette implémentation, la viabilité est une propriété du chromosome/individu et est mise à jour à chaque modification du chromosome (et non pas calculée à chaque accès).

Choix particuliers aux algorithmes génétiques

Le domaine des algorithmes génétiques est très actif et assez large ; de nouveaux domaines de recherche s'ouvrent régulièrement. Il fallait naturellement, pour ce travail, limiter l'étendue de l'implémentation pour obtenir un résultat praticable et opérationnel. Voici les présupposés qui sont d'application dans ce travail.

- Les individus d'une population sont tous de la même espèce (ils ont le même nombre de chromosomes, du même type). Ceci est en ligne avec la métaphore naturelle (des espèces distinctes ne s'apparient pas) et probablement avec la grande majorité des implémentations d'algorithmes génétiques, si pas toutes.
- Le mécanisme d'exécution utilise une seule population (de taille quelconque déterminée par l'utilisateur). Il n'est pas strictement impossible d'utiliser des stratégies à multiples populations avec échange d'individus sur base de ce framework mais il serait nécessaire de récrire la routine d'exécution.

[5] Ceci limite le framework aux stratégies utilisant une seule population, ce qui est en fait un choix limitatif. On aurait pu, en toute généralité, faire résider le code d'exécution dans une sous-classe de la classe implémentant la population. Ceci aurait cependant complexifié le framework pour atteindre un but dépassant les objectifs définis pour ce travail.

- Les opérateurs génétiques préservant le contenu des gènes mais agissant sur l'ordre de ceux-ci est un sous-groupe assez spécialisé des opérateurs génétiques, principalement utilisés pour résoudre le problème du voyageur de commerce et ses variantes. Ce sous-groupe est suffisamment important pour que ce framework l'implémente. Cependant, il est clair que les opérateurs agissant sur les nucléotides et ceux agissant sur les gènes sont mutuellement exclusifs, et donc le type de problème que l'on veut résoudre doit fixer le type d'opérateurs que l'on pourra utiliser.
- Mélanger la mutation et l'inversion dans un même problème n'ayant pas de sens (l'inversion a pour but de maintenir le contenu d'un gène alors que la mutation au contraire le détruit), la mutation et l'inversion ont été représentées comme deux valeurs distinctes de la même propriété de l'algorithme. Si l'algorithme est orienté "bit", la propriété *mutation/inversion* prendra la valeur "mutation", sinon, elle prendra la valeur "inversion".
- Bien qu'il soit généralement aisé de transposer un problème de minimisation en un problème de maximisation et vice-versa, il m'a semblé utile d'éviter cette peine à l'implémenteur de problèmes. Le framework permet de passer de l'un à l'autre sans programmation, et il est possible, quand c'est pertinent, de forcer le type d'un problème (c'est à dire empêcher l'utilisateur de chercher un minimum quand seul un maximum est pertinent, et vice-versa).
- Le framework n'est pas conçu pour implémenter un système d'intelligence artificielle, par exemple pour l'apprentissage des réseaux neuronaux ou pour créer un *classifier system* (voir [Goldberg 1989]). Le framework peut fournir les bases mais n'est pas conçu pour une implémentation immédiate de ce genre de système : il est prévu avant tout pour implémenter des problèmes de recherche d'optimum.

Choix relatifs à Java

La partie visuelle du framework et l'outil de démonstration sont basés sur les Java Foundation Classes (JFC/Swing). Celles-ci permettent d'obtenir un résultat bien plus professionnel que le classique AWT (Abstract Window Toolkit). Cependant, cela nécessite une machine virtuelle de version 1.2 ou supérieure. En particulier, tous les browsers d'Internet ne supportent pas d'office une version de la machine virtuelle qui soit compatible (il est cependant possible d'installer sur tous les browsers internet graphiques un *plug-in* compatible du *run-time engine*, très facilement et gratuitement).

Le lecteur ne disposant pas d'une machine virtuelle Java compatible pourra télécharger gratuitement le run-time engine (JRE) sur java.sun.com/j2se/1.4.1/download.html (dernière version à ce jour).

Il est à noter également que le développement a été axé sur un usage de l'outil de démonstration comme *application Java*, et non comme applet. Le portage est cependant trivial (mais certaines fonctionnalités ne sont pas disponibles, ou difficilement, sur une applet, comme par exemple la sauvegarde d'informations sur disque local). En cas d'implémentation du framework dans une applet, il est vivement conseillé d'utiliser la classe *javax.swing.JApplet* et non *java.awt.Applet*.

4.2 Présentation du framework

4.2.1 Généralités

Le but de ce chapitre n'est pas une présentation détaillée du framework (l'annexe A en présente une description complète), mais une introduction générale à son implémentation.

Ce chapitre est focalisé sur l'implémentation dans le langage Java, et les notions utilisées au cours de ce chapitre sont supposées connues, car il est tout à fait hors du propos de ce travail de donner un exposé sur Java. Le lecteur se reportera aux références données ci-dessus pour plus d'informations sur le langage et l'organisation de l'API[6].

Le framework est structuré en quelques packages :

cuvier est le package contenant la fonctionnalité purement dédiée aux algorithmes génétiques, c'est à dire les implémentations de chromosomes, gènes, population, ainsi que certaines implémentations spécialisées de ces fonctionnalités (en particulier le visualiseur par défaut de problèmes).

cuvier.util contient des classes utilitaires proposant des outils particuliers utilisés par le framework.

cuvier.problems est le package dédié aux problèmes implémentés par le framework. Son usage n'est pas obligatoire pour des implémentations tiers mais les problèmes implémentés pour l'outil de démonstration se trouvent dans ce package.

Le framework Cuvier et sa documentation sont disponibles sur Internet à l'adresse http ://www.caudron.info/Cuvier. L'application de démonstration et une copie électronique de ce rapport sont également disponibles à cette adresse.

4.2.2 Chaînes de nucléotides

La métaphore informatique pour le nucléotide est, nous l'avons vu, le bit. Les chaînes de nucléotides sont donc représentées par des champs de bits, implémentés sur base de la classe *java.util.BitSet* de l'API Java. Cette classe a été très légèrement étendue en une classe *cuvier.util.FixedBitSet* pour gérer la longueur du champ de bits. En effet, BitSet a une longueur dynamique, or il est nécessaire, pour la représentation de la chaîne de nucléotides, de fixer très précisément la longueur réelle du champ de bits, et d'éviter son extension implicite.

BitSet fournit la fonctionnalité de stockage et de manipulation de bits, optimisée de manière à permettre un stockage quasi optimal (avec perte de quelques octets) et une gestion dynamique évitant des allocations inutiles. Sa base de stockage est de 64 bits (8 octets).

Les bits individuels sont manipulés comme des booléens. C'est à la fois le standard usuel en Java, et la représentation idéale, que j'ai également choisie pour le framework. Ceci permet aux opérateurs binaires de fonctionner directement en évitant en grande partie la complication de l'arithmétique binaire.

[6] Il n'est cependant pas nécessaire d'être un spécialiste de Java pour pouvoir implémenter des problèmes au moyen de *cuvier*, comme on le verra plus loin.

4.2.3 Gènes

La classe *Gene* est une interface (une classe purement abstraite) fixant le contrat pour tous les types de gènes utilisés par les chromosomes. De cette manière les chromosomes peuvent contenir des gènes pouvant être résolus en différents types (entiers, réels en virgule flottante, champs de bits, booléens, etc.) mais pouvant être manipulés d'une manière standardisée.

Les points importants de ce contrat sont :

- Donner une représentation bit par bit de ce gène, c'est à dire retourner un bit donné à une position donnée (*getBitAt*), retourner le nombre de bits codant ce gène (*length*), et retourner le champ de bits dans une représentation différente (*getBitField*, *toString*).
- Modifier la valeur interne sur base d'un champ de bits (*setBits*), utilisé pour la synchronisation entre la chaîne de nucléotides et le gène.
- Appliquer la valeur interne sous forme d'un champ de bits, à un champ de bits donné (*applyGene*).
- Retourner une représentation numérique standard, si c'est pertinent (*byteValue*, *shortValue*, *intValue*, *longValue*, *floatValue*, *doubleValue*). Ces méthodes sont données pour fixer leur syntaxe, mais elles n'ont pas nécessairement de sens pour un type de gène donné. Si une de ces méthodes n'est pas pertinente dans un implémentation donnée, elle devra lancer une runtime exception (par convention interne au framework, cette exception devrait être de type *ClassCastException*, mais ce n'est pas une obligation).

Le framework fournit certains types de gènes basés sur des types standards (*ByteGene*, *IntGene*, *LongGene*), ou créés pour certains problèmes de démonstration (*PositiveLongGene*, *BitFieldGene*). Il est très facile d'ajouter des types de gènes supplémentaires et adaptés à des problèmes donnés.

4.2.4 Viabilité

La classe *Fitness* est une classe abstraite encapsulant la fonctionnalité nécessaire pour maintenir et calculer la viabilité. Une instance d'un descendant de cette classe doit être liée à une instance de chromosome (les constructeurs de la classe Chromosome doivent recevoir une instance d'un descendant de cette classe). De cette manière, la synchronisation entre le contenu du chromosome et la valeur de la viabilité peut être maintenue de manière optimale.

La classe de viabilité encapsule le type utilisé pour représenter la viabilité, de sorte qu'il est possible d'utiliser d'autres représentations que la virgule flottante pour la valeur de viabilité. Par exemple, utiliser une représentation de grands nombres en précision garantie (type *long* ou un autre type défini par l'utilisateur). Le choix d'encapsuler le type dans la classe de viabilité est assez naturel en général (c'est à dire en "philosophie" orienté objet) ; cependant, il manque à Java certains mécanismes existant dans d'autres langages, en particulier la surcharge d'opérateurs et l'héritage multiple, qui rendent cette opération assez peu élégante en Java. Bien qu'*a priori* il n'est pas certain que l'on puisse tirer avantage de la souplesse supplémentaire apportée par la possibilité de déterminer le type de la viabilité, j'ai préféré implémenter cette fonctionnalité malgré sa relative complexité pour ne pas fermer le champ des implémentations basées sur ce framework.

On sous-classera donc la classe abstraite de base, *Fitness*, afin de lui al-

louer un type. Le framework propose plusieurs viabilités "typées", notamment LongFitness et DoubleFitness. Il est à noter que ces sous-classes sont toujours des classes abstraites, la classe ne devenant concrète que quand la fonction de viabilité est implémentée, donc uniquement lorsqu'un problème est implémenté.

Fitness implémente l'interface de l'API Java *Comparable* de sorte que les viabilités peuvent être classées, si nécessaire, automatiquement par le framework de collections de Java. Cette fonctionnalité est également utilisée par Cuvier pour classer les chromosomes par ordre de viabilité.

Le contrat de *Fitness* comporte quatre grands points :

- le calcul de la viabilité, par la méthode *calculate(Chromosome)*, dans laquelle la valeur interne de la viabilité doit être déterminée au moyen de la méthode *setValue* ;
- des méthodes de valorisation, permettant de convertir la valeur interne en un type connu de Java. Cette conversion doit en principe être codée de manière signifiante pour le type *double* parce que certaines stratégies de sélection doivent utiliser ce type (uniquement le "sigma scaling" pour l'instant), et également parce que les statistiques de l'outil de visualisation sont en partie dépendantes de ce type. Ce n'est pas strictement une obligation mais il faut être conscient de quelques limitations si l'on choisit de ne pas le faire.
- des méthodes arithmétiques remplaçant les opérateurs équivalents qui ne peuvent pas être appliqués à une classe. Il s'agit de l'addition (*add*), la soustraction (*substract*) et de la division (*divide*). Ceux-ci doivent être implémentés de manière signifiante dans chaque descendant concret de *Fitness*.
- des méthodes de fixation de la valeur interne, permettant de la maximiser (lui donner la valeur maximale liée au type de donnée interne), la minimiser, la fixer au zéro de la représentation, et enfin de lui donner une valeur aléatoire. Ces méthodes *doivent* être implémentées de manière signifiante.

En-dehors de ce contrat, la classe propose plusieurs méthodes de convenance permettant par exemple d'"optimiser" ou "pessimiser" la valeur interne, c'est à dire la maximiser ou la minimiser en fonction du type de problème traité. Si le problème est une maximisation, optimiser la valeur la fixera à la plus haute valeur possible, et la pessimiser la fixera à la plus basse possible.

4.2.5 Chromosomes

La classe *Chromosome* contient la chaîne de nucléotides (bits) sous la forme d'un champ de bits (*FixedBitSet*).

Chromosome contient également les gènes qui le composent. Ceux-ci sont stockés dans un vecteur (*java.util.Vector*) afin de garantir leur position relative, et de permettre une ajoute ou un retrait dynamique de gènes le cas échéant.

Chaque implémentation devra composer son propre chromosome-type. Le constructeur permet de créer en une fois des chromosomes portant de 0 à cinq gènes. On conçoit aisément que cette limite de cinq gènes est arbitraire, mais il est impossible de fournir un constructeur pour tous les nombres de gènes possibles, et il faut donc établir une limite. Les fonctionnalités de la classe permettent d'ajouter encore des gènes supplémentaires si nécessaire.

Note Il n'est pas possible de passer au constructeur une liste de gènes sous

forme d'une chaîne ou d'un vecteur, parce que l'initialisation des objets de la classe courante s'effectue *après* l'appel du constructeur de l'ancêtre de la classe courante. Donc les gènes ne seraient pas instanciés au moment où ils seraient utilisés pour la première fois. Il existe des solutions, comme par exemple définir un modèle de gène sous la forme d'une variable statique, mais aucune de ces solutions n'est vraiment pratique et élégante.

Les gènes incluant la fonctionnalité nécessaire pour mettre à jour la chaîne de nucléotides, l'implémenteur ne manipulera jamais en direct la chaîne de bits/nucléotides. Il suffit donc, pour l'implémenteur, de construire un chromosome sur base des gènes qui le composent.

On l'a vu, la viabilité d'un chromosome est maintenue dans une instance d'un descendant concret de la classe *Fitness*. Les constructeurs de la classe *Chromosome* forcent l'inclusion d'un descendant du type *Fitness*. De cette manière, le chromosome peut automatiquement mettre à jour sa viabilité à chaque modification de son génotype.

Ceci implique cependant une certaine prudence. En effet, le constructeur du chromosome ne peut accepter que cinq gènes au maximum (on pourrait ajouter des constructeurs supplémentaires mais leur nombre resterait limité), et donc dans beaucoup de cas le chromosome peut être incomplet au moment où sa viabilité est calculée pour la première fois. La fonction de viabilité doit donc être protégée contre un chromosome incomplet. Par exemple, si le chromosome doit porter six gènes, le calcul de la viabilité peut retourner zéro ou un minimum si la fonction de viabilité détecte que le nombre de gènes est inférieur à six.

Chromosome est une classe concrète. Ses fonctionnalités principales incluent :

- la gestion du vecteur de gènes (ajout, retrait, et modification d'un gène par accès à sa référence) ;
- les opérations de bas niveau sur la chaîne de bits, pour la plupart cachées de l'utilisateur ;
- les opérateurs génétiques : crossovers (à un et deux points), mutation, crossovers géniques (PMX, OX, CX - détaillés au chapitre suivant), inversion, insertion, échange réciproque ;
- la réinitialisation du chromosome à un génotype aléatoire, soit par modification aléatoire des nucléotides (chaque bit est forcé à une valeur aléatoire), soit par échange aléatoire des gènes (les gènes sont échangés deux à deux, au hasard, un certain nombre de fois) ;
- des méthodes d'accès à la valeur de viabilité ;
- diverses fonctionnalités utiles, comme la copie conforme de chromosome (incluant la viabilité, avec et sans allocation[7]), la vérification de cohérence du chromosome, la comparaison de chromosomes pour déterminer s'ils sont de la même "espèce" (même nombre de gènes, de même type), etc.

4.2.6 Sélection

Les stratégies de sélection sont découplées du moteur d'exécution grâce à la classe *Selection*. Celle-ci permet d'implémenter des stratégies de sélection quelconques. Sa fonctionnalité principale est proposée par la méthode *run*, méthode

[7] Cette dernière fonctionalité est une optimisation permettant d'éviter les problèmes de *garbage collector*.

abstraite dans la classe de base devant être implémentée dans chaque nouvelle stratégie de sélection.

Cette méthode reçoit en paramètre une population (pour l'instant, considérons qu'il s'agit simplement d'un vecteur de chromosomes), une indication sur le type de problème (maximisation ou minimisation) qui influence généralement la stratégie de sélection, et un nombre d'individus à produire. La méthode retourne le nombre d'individus demandé dans une chaîne (*array*) de chromosomes.

Notons que le nombre de chromosomes à retourner par la stratégie de sélection n'est pas nécessairement identique au nombre d'individus composant la population. Ceci permettrait de mélanger plusieurs stratégies de sélection. Plus pragmatiquement, cette fonctionnalité permet d'appliquer de l'élitisme sur toutes les stratégies de sélection, c'est à dire présélectionner un nombre donné des meilleurs individus avant d'appliquer la stratégie de sélection choisie.

Le framework implémente d'ores et déjà plusieurs stratégies de sélection, que nous détaillerons dans le chapitre 5 décrivant l'outil de démonstration.

Par ailleurs, *Selection* propose quelques méthodes utilitaires permettant notamment de donner une description à la stratégie de sélection ou de sauvegarder la configuration de l'instance courante.

Enfin, cette classe étend la classe *javax.swing.JPanel* afin de permettre aisément la visualisation des propriétés d'une instance d'un descendant de cette classe.

4.2.7 Population

La classe *Population* remplit deux rôles :
- initialiser et gérer un ensemble d'individus (représentés par des chromosomes) ;
- exécuter l'algorithme génétique.

Constructeurs

Le constructeur de cette classe reçoit en principe un chromosome en paramètre (ainsi qu'une chaîne de caractères décrivant le problème que l'on cherche à résoudre). Ce chromosome servira de modèle à l'ensemble de la population. Les individus sont maintenus dans une chaîne, et sont créés automatiquement par copie du modèle (après quoi la copie est "brouillée" aléatoirement, comme expliqué dans la section sur la classe *Chromosome*, pour obtenir une population de départ totalement aléatoire).

Le problème créé est par défaut un problème de maximisation, est orienté bit (donc agit par recombinaison des nucléotides et non des gènes), et permet à l'utilisateur final de changer le type d'optimisation dynamiquement. Un paramètre supplémentaire des constructeurs permet de modifier ce comportement. Des variables statiques définies dans la classe permettent de forcer le type d'optimisation de départ du problème, ce sont *Population.MINIMIZATION* et *Population.MAXIMISATION* ; d'autres permettent de fixer le type de problème, orienté nucléotide ou orienté gène, ce sont *Population.BIT_ORIENTED* et *Population.GENE_ORIENTED*. Enfin, une variable permet d'empêcher l'utilisateur final de changer le type d'optimisation, il

s'agit de *Population.FIXEDTYPE*. Par exemple, passer au constructeur *Population.GENE_ORIENTED+Population.MINIMIZATION+Population.FIXEDTYPE* change complètement le comportement par défaut. Ce choix précis est par exemple adapté à l'implémentation du problème du voyageur de commerce.

Constructeur sans modèle

Il existe un constructeur qui ne reçoit aucun chromosome (seulement la chaîne de caractères de titre). Cela peut paraître aberrant, mais c'est en réalité indispensable. En effet, le chromosome, on l'a vu, doit contenir une liste de gènes et une instance de viabilité. Or, cette liste de gènes est limitée, dans la présente implémentation, à cinq unités[8]. Il est donc impossible de passer au constructeur de la population un chromosome modèle possédant une liste de six gènes ou plus.

De plus, pas mal de problèmes ne peuvent pas s'accommoder d'une définition de chromosome fixée une fois pour toutes par le constructeur. En effet, pour des problèmes pouvant recevoir des modèles différents comme le voyageur de commerce (liste de villes) ou le sac de contrebandier (liste d'items à passer en fraude), la liste de gènes peut changer à chaque fois que l'on charge de nouveaux paramètres de problème. Il est donc clair que la meilleure solution dans ce cas est de définir un constructeur ne recevant pas de modèle de chromosome, mais il est également clair que tant que la population n'est pas constituée de chromosomes valides, le problème ne peut pas être traité.

Moteur d'exécution

Le moteur d'exécution de l'algorithme est inclus dans cette classe. L'exécution s'effectue dans un *thread* séparé, ce qui permet notamment de garder une interface utilisateur responsive. Le thread est défini dans une classe interne appelée *Runner*.

L'exécution utilise un système de "drapeaux" (*flags*) afin de gérer les états d'exécution de manière asynchrone. Lorsqu'une condition est rencontrée, par exemple, si l'utilisateur désire arrêter l'exécution prématurément, ou une condition de succès de l'algorithme est rencontrée, un drapeau est levé qui stoppe d'une manière asynchrone le thread d'exécution et déclenche la procédure de finalisation.

Ce système de drapeaux est également utilisé afin de gérer finement l'exécution. Il est en effet possible de geler momentanément l'exécution (*pause*), d'exécuter pas à pas (avec arrêt à chaque génération), et de relancer à volonté l'exécution. Le diagramme d'états de la figure 4.1. montre les états et transitions possibles.

On démarre l'exécution en invoquant la méthode *run* ou la méthode *runStep*. Les deux méthodes sont équivalentes, si ce n'est que *runStep* lève le drapeau *stepping* qui génère une exécution pas à pas (comme le montre le diagramme d'états, l'état *stepping* est suivi de l'état *pausing*).

La méthode *run* vérifie la consistence des individus de la population (s'ils sont de même espèce et sont des individus "complets"), puis leur donne une

[8] Voir la section ci-dessus décrivant la classe *Chromosome* pour la justification de cette limite.

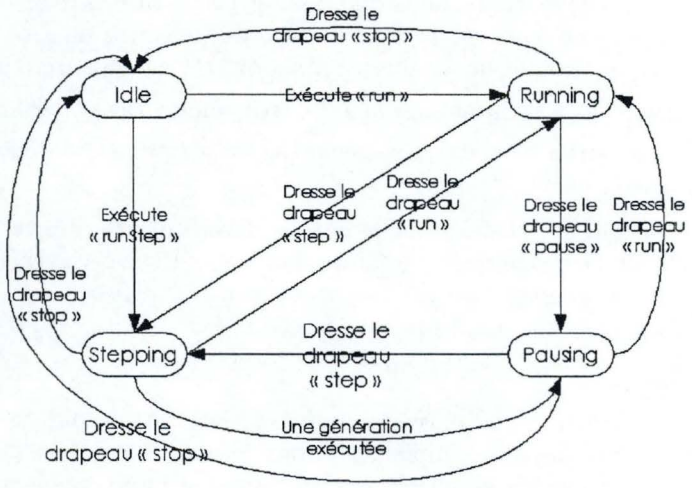


FIG. 4.1 – Etats d'exécution

valeur interne aléatoire de la manière expliquée dans la section sur les chromosomes. Ceci garantit que la population de départ est bien aléatoire, et que le problème ne sera pas biaisé par une valeur "survivante" d'une exécution précédente. Après avoir levé le drapeau d'exécution, la méthode active le thread d'exécution et se termine.

La méthode *run* du thread d'exécution (à ne pas confondre avec la méthode *run* de la classe Population) contient la boucle des générations. Les opérations sont cependant menées dans une méthode séparée afin de synchroniser les données (protéger la zone critique des données traitées contre un accès concurrent entre threads).

La boucle principale met à jour le compte de générations et vérifie si un critère d'arrêt a été configuré, et si celui-ci est atteint, auquel cas la boucle se termine. Si le drapeau *pausing* a été levé, la boucle *polle* (vérifie régulièrement) sa valeur, en dormant par intervalles de 100 millisecondes, et n'exécute naturellement plus l'algorithme.

Chaque étape d'exécution est menée par la méthode *act*. Cette méthode effectue les opérations suivantes :

1. Vérifie si une limite de viabilité existe et si celle-ci a été atteinte. Dans ce cas, la méthode retourne *false*, ce qui a pour effet de quitter la boucle de la méthode *run*.
2. Applique l'élitisme (si nécessaire) : commence à garnir la population de la génération suivante avec des copies des meilleurs individus de la génération courante.
3. Applique la sélection : termine le garnissage de la nouvelle population en appliquant la stratégie de sélection un nombre de fois suffisant.
4. Remplace le contenu de la population par la nouvelle génération.
5. Applique la technique de crossover choisie.
6. Applique la technique de mutation ou d'inversion choisie.

7. Vérifie le critère de stagnation, si nécessaire. Si c'est pertinent (si une quantité de stagnation a été indiquée), effectue un *kick*. Si les critères de stagnation et de nombre de *kicks* ont été atteints, retourne *false*.
8. Vérifie si le drapeau *stop* a été levé, auquel cas la méthode retourne *false*
9. Retourne *true*, ce qui a pour effet de réexécuter la boucle de générations principale.

Une variable d'état indique le type d'évènement qui a causé l'arrêt de l'algorithme, ce qui permet d'informer l'utilisateur sur cet évènement.

Toutes les opérations sont asynchrones du fait que la communication de l'état du drapeau d'exécution passe du thread principal au thread d'exécution. Il peut donc y avoir une latence limitée entre l'établissement d'un drapeau et l'effet escompté.

Population définit plusieurs *callbacks* invoqués durant l'exécution et pouvant être surchargés par les implémentations de cette classe. Ceci permet notamment de générer une visualisation de l'exécution en interceptant ses étapes les plus importantes : initialisation, générations, crossovers, mutations et inversions, fin d'exécution.

Divers

La classe propose également des méthodes permettant de gérer globalement les individus de la population. Notamment, *setPopulationCount* permet de modifier le nombre d'individus composant la population. Ceci agit sur le vecteur de chromosomes, soit en le tronquant, soit en l'étendant. Les nouveaux individus sont créés par copie aléatoire du modèle passé en paramètre du constructeur comme indiqué ci-dessus. Si aucun modèle n'a été passé en paramètre, le premier élément de la liste de chromosomes sert de modèle aux nouveaux individus.

D'autres méthodes permettent de modifier le génotype des individus d'une manière globale ou le type de leur viabilité. Des tests de cohérence sont également définis dans cette classe, assurant que tous les chromosomes sont de la même espèce avant d'accepter d'exécuter l'algorithme.

Enfin, la classe propose des méthodes permettant de configurer les paramètres de l'algorithme génétique, en particulier le type de crossover à appliquer, le type de mutation/inversion, le type de sélection, et le(s) critère(s) d'arrêt.

La classe est basée sur *javax.swing.JPanel* afin de permettre à ses descendants de proposer aisément une visualisation basée sur les JFC.

4.2.8 Visualisateur de population

Le framework propose une classe *PopulationViewer*, basée sur *Population*, qui fournit une visualisation de problème par défaut. Cette visualisation utilise un système d'onglets :

Settings permet de configurer l'exécution de l'algorithme, c'est à dire le type d'optimum recherché (si c'est autorisé pour ce problème), les critères d'arrêt, le type de crossover et de mutation/inversion, le type de sélection, l'élitisme, et la priorité du thread d'exécution par rapport au thread principal (ce qui permet de gérer finement le rapport entre la rapidité d'exécution et la responsivité de l'interface).

Statistics montre les statistiques en cours d'exécution, en particulier le nombre d'exécutions d'opérateurs, le nombre de générations, et des informations sur la viabilité moyenne et optimale de la population. Le même onglet propose une visualisation graphique de l'évolution de la viabilité de la population.

Visualization permet à l'implémenteur de problème de donner une représentation du problème traité en surchargeant la méthode *paintVisualization*. Par défaut l'onglet ne montre naturellement rien.

Trace donne une description textuelle de l'exécution de l'algorithme. Le niveau de détail de la description peut être fixé au moyen d'un menu de la barre d'outils. La "trace" peut être désactivée totalement pour améliorer le rendement de l'algorithme. Enfin, il est possible de sauvegarder ou d'exporter vers le presse-papiers le contenu du texte descriptif.

La visualisation possède également une barre d'outils permettant d'exécuter l'algorithme, le "geler", fixer le niveau de détail de la trace d'exécution, etc. Ceci sera décrit en détail dans le chapitre ultérieur consacré à l'outil de démonstration.

PopulationViewer est une implémentation-type de *Population* utilisant les *callbacks* invoqués à chaque événement lors de l'exécution, et les méthodes d'accès et de mutation des paramètres internes.

4.2.9 Implémentation d'un problème

Introduction

Voyons à présent comment implémenter un problème au moyen du framework. Considérons un problème trivial (inspiré de [Mitchell 1996]), qui nous permettra de nous focaliser sur l'implémentation en elle-même.

Considérons une chaîne de bits. Nous allons demander à l'algorithme génétique de trouver la chaîne de bits contenant le plus (ou le moins) de bits à "1". Il va de soi que la réponse à cette question est la chaîne de bits contenant uniquement des 1 (ou aucun si l'on minimise).

Comme l'algorithme génétique n'a pas de préjugé sur le problème qu'on lui soumet, ce problème est aussi bon qu'un autre (nous allons démontrer expérimentalement ce fait au chapitre 6). Par ailleurs, il faut considérer que l'espace des solutions n'est pas de petite taille : il est de 2^n unités, où n est la taille de la chaîne de bits. Pour 64 bits par exemple, il est de 18.446.744.073.709.551.616 unités - ce qui en fait un problème tout à fait intéressant à traiter. Fixons la taille du problème à 64 bits.

Représentation

Pour représenter le génotype des individus, nous allons utiliser un chromosome à gène unique, ce gène codant une chaîne de bits. La classe *BitFieldGene* du framework propose cette représentation et ses méthodes spécifiques.

Calcul de la viabilité

La viabilité étant un nombre entre 0 et 64, nous pouvons utiliser une représentation entière à un byte, soit le type natif *byte* en Java (dont le domaine

est $[-128, 127]$). Pour cela, nous pouvons définir un descendant de la classe *Fitness* implémentant un byte. Le framework dispose d'une classe *ByteFitness* à cet effet. Nous allons donc donner une implémentation concrète de cette classe abstraite en codant la méthode de calcul de la viabilité. Ce calcul est simple, il suffit de compter le nombre de bits à 1 du gène unique du chromosome.

```
class MyByteFitness extends ByteFitness
{
    protected synchronized boolean calculate(Chromosome cr)
    {
        byte value=0;
        for (int x=0; x<cr.getGene(0).getLength(); x++)
                if (cr.getGene(0).getBitAt(x)) value++;
        setValue(value);
        return true;
    }
}
```

Notons que *calculate* peut être exposée publiquement si l'on préfère, et la classe peut être finalisée et publique si on le désire. Comme l'exécution est multithread, il est préférable de synchroniser la méthode pour éviter d'accéder à des chromosomes dans des états instables (*in-doubt*).

Notons également que cette méthode fait preuve d'une entière confiance dans le fait que le chromosome a au moins un gène. Nous l'avons vu, ce n'est absolument pas garanti, et si le chromosome ne contient pas de gène cette méthode lancera une *ArrayIndexOutOfBoundsException* qui terminera l'application. Nous allons cependant faire en sorte que cette situation ne se produise pas, en fournissant le gène immédiatement au constructeur du chromosome.

Constitution de la population

Il n'est pas nécessaire de sous-classer la classe *Chromosome*. Nous allons simplement fournir à la population une instance de chromosome initialisée avec notre classe de viabilité, et le gène unique qui le compose. Cette opération doit être réalisée en une fois dans le constructeur de la population, sans quoi le modèle ne sera pas initialisé au moment où l'ancêtre du constructeur de la population sera invoqué[9], causant une exception.

Nous allons créer un descendant de *PopulationViewer* pour bénéficier de la visualisation par défaut. Ce n'est pas obligatoire, et l'on peut créer sa propre visualisation sur base de *Population*, mais c'est bien sûr plus complexe.

```
class TousAUn extends PopulationViewer
{
    public TousAUn()
    {
        super("Tous à un !",
            new Chromosome(new BitFieldGene(64), new MyByteFitness()));
    }
```

[9] Pour mémoire, l'appel à l'ancêtre du constructeur d'une classe doit être la toute première action se déroulant dans le constructeur d'une classe, et si ce n'est pas fait explicitement, un appel implicite a lieu, ce qui donne le même effet.

```
}
```

Le constructeur de la classe effectue une action unique : appeler un des constructeurs de l'ancêtre (*PopulationViewer*) en fixant le nom du problème ("Tous à un !") et en fournissant une instance de chromosome à utiliser comme modèle pour générer les individus de la population. Le chromosome, quant à lui, est instancié en utilisant le constructeur qui prend un seul gène (et une instance de viabilité) en paramètre.

Mise en place de l'application

Le problème est prêt à être résolu. Il nous reste juste à intégrer notre problème à une application Java pour pouvoir l'exécuter. Par exemple,

```
public class MonAG extends JFrame
{
    public MonAG()
    {
        PopulationViewer pop=new TousAUn();
        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(pop, Borderlayout.CENTER);
        setSize(500,500);
        setVisible(true);
    }

    public static void main(String args())
    {
        new MonAG();
    }
}
```

Ce code est suffisant pour exécuter l'algorithme génétique et solutionner ce problème.

Code complet de l'application

Voici le code complet de notre application :

```
import java.awt.*;      // Package de BorderLayout
import javax.swing.*;  // Package de JFrame
import cuvier.*;  // Package de PopulationViewer, ByteFitness et BitFieldGene.

public final class MonAG extends JFrame
{
    public MonAG()
    {
        PopulationViewer pop=new TousAUn();
        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(pop, BorderLayout.CENTER);
        setSize(500,500);
        setVisible(true);
```

```
    }

    public static void main(String[] args)
    {
        new MonAG();
    }
}

final class TousAUn extends PopulationViewer
{
    public TousAUn()
    {
        super("Tous à un !",
            new Chromosome(new BitFieldGene(64), new MyByteFitness()));
    }
}

final class MyByteFitness extends ByteFitness
{
    protected synchronized boolean calculate(Chromosome cr)
    {
        byte value=0;
        for (int x=0; x<cr.getGene(0).getLength(); x++)
                if (cr.getGene(0).getBitAt(x)) value++;
        setValue(value);
        return true;
    }
}
```

La figure 4.2. montre le résultat de ce code (onglet de statistiques).

En résumé

Voici en résumé les étapes à suivre afin d'implémenter un problème :

- Le cas échéant, créer une classe de gène spécialisée (si le problème réclame un gène d'un type pas encore présent dans le framework, ou si le comportement d'un gène existant doit être adapté).
- Créer une implémentation spécialisée de viabilité :
  - Si l'on veut définir une viabilité donnant une valeur d'un type particulier, sous-classer la classe abstraite *Fitness* pour lui donner un type.
  - Concrétiser un descendant de *Fitness* comme *LongFitness* ou *DoubleFitness*, ou encore une implémentation spécifique, en implémentant la méthode *calculate*[10].
  - La méthode *calculate* reçoit en paramètre une référence vers un chromosome, dont elle calcule la viabilité, qu'elle fixe dans l'instance de viabilité par la méthode *setValue*. La méthode retourne un booléen qui indique le succès ou l'échec du calcul de viabilité.

[10] On peut naturellement concrétiser en une fois *Fitness* en lui donnant un type et une méthode de calcul.

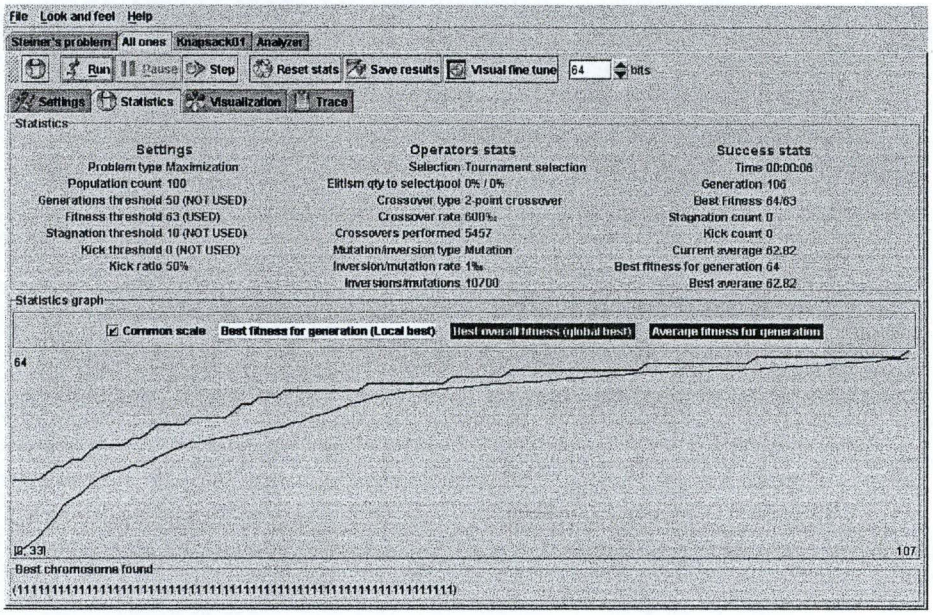


FIG. 4.2 – Exemple d'implémentation

- Sous-classer *PopulationViewer*, et invoquer l'ancêtre direct en lui donnant les informations nécessaires à son initialisation, en particulier un modèle de chromosome à utiliser pour générer la population.
- Intégrer le descendant de *PopulationViewer* dans une application Java.

Si on sous-classe *Population* plutôt que *PopulationViewer*, on doit fournir toute la visualisation nécessaire à la gestion du problème.

Visualisation du problème

Il est tout à fait possible, sur base de *PopulationViewer*, de fournir une représentation visuelle du problème que l'on traite. L'onglet *visualization* est prévu à cet effet. A titre d'exemple, la figure 4.3. montre une visualisation du problème de Steiner qui consiste à relier des batiments ou villes à une centrale électrique dont on doit déterminer l'emplacement idéal.

Il y a deux manières de fournir une visualisation spécifique d'un problème. La première consiste à récupérer une référence vers le panneau (*JPanel*) qui occupe l'onglet de visualisation, et de l'équiper de composants utiles à la visualisation, par exemple des labels donnant les résultats de l'exécution.

La seconde consiste à "peindre" directement sur le canevas du panneau, en utilisant les fonctionnalités de la classe Java *Graphics*. Celle-ci permet de faire du dessin sur base de primitives comme des lignes, rectangles, ovales, texte, en jouant avec les positions et les couleurs. Pour ce faire, il faut surcharger la méthode *paintVisualization* de la classe *PopulationViewer*. Celle-ci est invoquée chaque fois que le panneau doit être redessiné (ce qui est géré par la machine virtuelle). Cette méthode reçoit une référence vers le panneau (permettant notamment d'en déterminer les dimensions) et une référence vers le canevas de

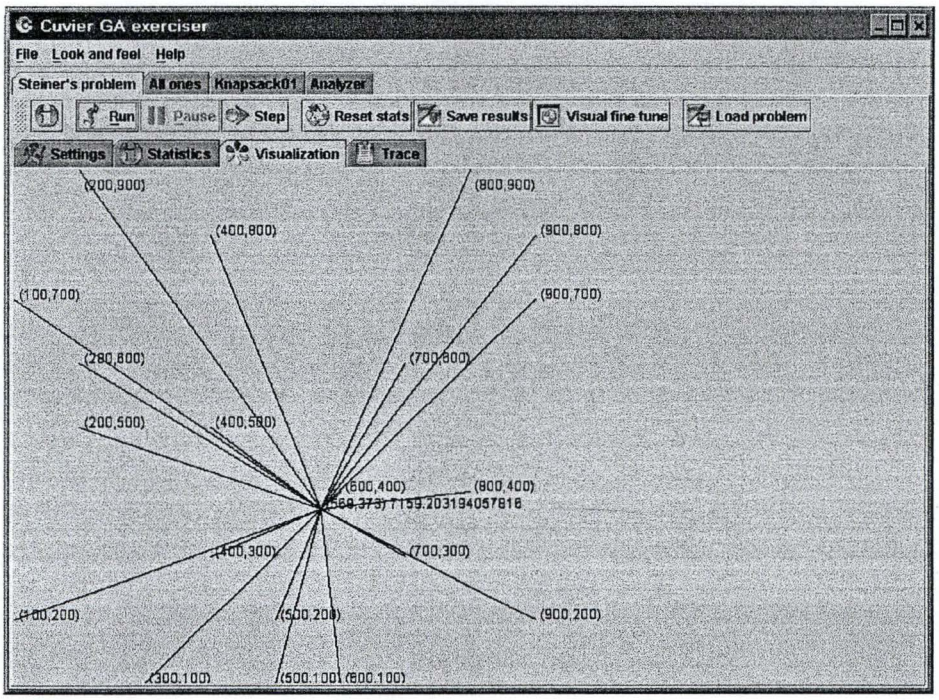


FIG. 4.3 – Exemple de visualisation

dessin du panneau (ce qui permet de dessiner directement en utilisant les primitives de la classe *Graphics*).

Toutes les fonctionnalités de cette classe sont disponibles quand on surcharge *paintVisualization*. Ces fonctionnalités sont décrites en détail dans la documentation de l'API Java (java.sun.com/j2se/1.4.1/docs/api/index.html) et dans le tutoriel des classes Java (java.sun.com/docs/books/tutorial/), et leur description sort totalement du cadre de ce travail.

Notons que la visualisation peut être assez complexe, si l'on utilise l'API Java2D (java.sun.com/products/java-media/2D/) ou même Java3D (java.sun.com/products/java-media/3D/). Elle peut proposer une visualisation interactive si nécessaire.

Configuration du problème

On peut permettre à l'utilisateur de configurer le problème traité en mettant à sa disposition des composants de configuration (comme des boutons, boîtes à cocher ou boîtes de texte). Ceci peut être réalisé de plusieurs manières différentes. La plus simple, si la configuration se compose de peu de composants de petite taille (par exemple un bouton et/ou un champ de texte), est de les ajouter à la barre d'outils de *PopulationViewer*. Il suffit d'invoquer la méthode *addToToolBar* de *PopulationViewer*.

Une autre option consiste à ajouter des composants de paramétrisation sur le panneau de visualisation. Cette option n'est cependant envisageable que si le problème n'a pas de visualisation ou une visualisation limitée à quelques

composants.

Pour les problèmes demandant une configuration complexe, la solution consiste à ajouter un onglet à la liste des onglets par défaut de *PopulationViewer*, en utilisant la méthode *addTab*. L'implémenteur a alors plein contrôle sur le contenu du nouvel onglet. Il est d'ailleurs possible d'en ajouter plusieurs si nécessaire.

Une option supplémentaire consiste à placer un bouton sur la barre d'outils qui fait apparaître une boîte de dialogue permettant de configurer le problème. C'est l'option idéale, par exemple, pour charger la configuration du problème depuis un fichier (par exemple, l'emplacement des villes pour le problème du voyageur de commerce).

Chapitre 5

Présentation de l'outil de démonstration

5.1 Introduction

Ce chapitre donne une description détaillée des fonctionnalités du framework, et en particulier des fonctionnalités génétiques qu'il supporte, au-travers de l'outil de démonstration.

Celui-ci est inclus dans le fichier d'archives java ("jar") qui contient les classes du framework. Pour démarrer l'outil de démonstration, il suffit de taper la commande *java -jar cuvier.jar* dans le répertoire où se trouve le fichier *cuvier.jar*, ou *java Cuvier* depuis un répertoire quelconque si *cuvier.jar* est déclaré dans la variable d'environnement *CLASSPATH*.

5.2 Généralités

La figure 5.1. ci-dessous montre la fenêtre générale de l'application, avec l'onglet de configuration de l'algorithme génétique d'un des problèmes sélectionnés. En-dehors de la fenêtre (conteneur) elle-même, dont l'aspect et les fonctionnalités peuvent différer d'un système d'exploitation à l'autre, le menu de l'application :

donne la possibilité de quitter l'application, de changer le *look and feel*[1] et d'afficher une fenêtre "à propos de...".

Sous le menu on trouve des onglets permettant de sélectionner un des problèmes de démonstration :

[1] Le framework *Swing* de Java permet d'adapter l'interface visuelle au système graphique du système d'exploitation, et d'utiliser si on le désire un système graphique différent de celui par défaut.

Chaque problème de démonstration est implémenté au moyen de la visualisation par défaut du framework Cuvier (la classe *PopulationViewer*). Cette visualisation est, comme nous l'avons vu, structurée également au moyen d'onglets. Nous allons décrire chacune des pages correspondant à l'un des onglets en détail.

5.3 Configuration de l'algorithme (onglet *Settings*)

5.3.1 Paramètres généraux

Voici une vue de la page de configuration d'un problème implémenté par *PopulationViewer* :

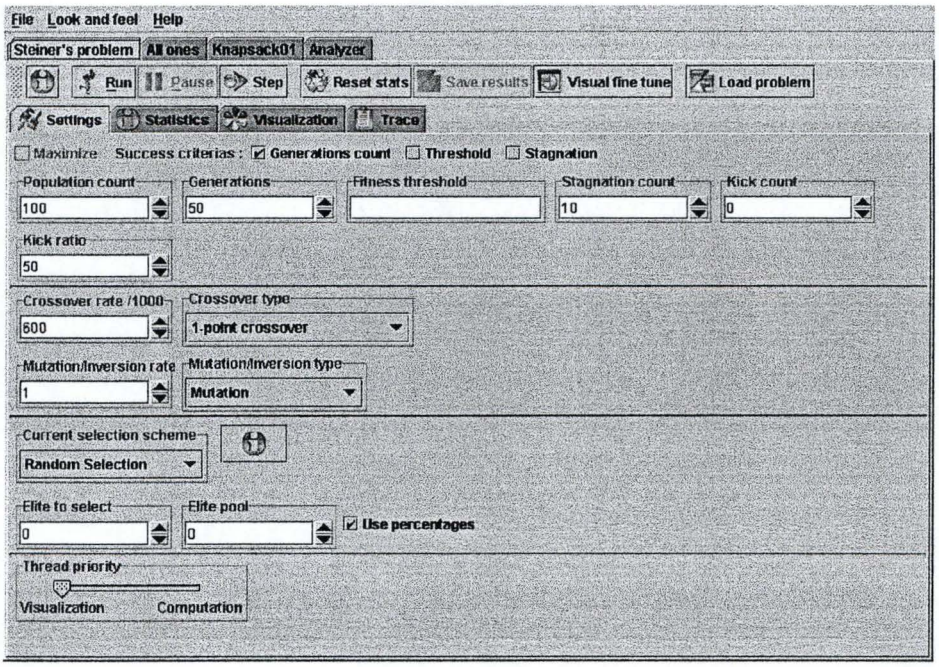
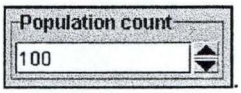


FIG. 5.1 – Onglet de configuration

La boîte à cocher ☑ Maximize permet de choisir si l'optimisation que l'on veut réaliser est une maximisation ou une minimisation. Pour que ce choix soit possible, il faut que l'implémenteur du problème ait décidé d'accepter les deux options, sans quoi cette boîte à cocher n'est pas modifiable : ☐ Maximize .

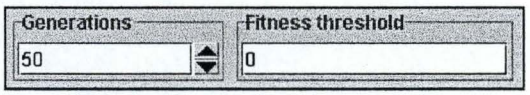
Une boîte de texte avec flèches de défilement (*spinner*) permet de fixer la taille de la population :



5.3.2 Critères d'arrêt

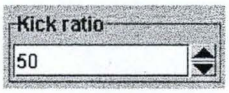
On peut définir à l'exécution plusieurs critères d'arrêt. Ceux-ci sont cumulatifs et sont contrôlés par les boîtes à cocher :

Le *nombre de générations maximal* et la *limite de viabilité* peuvent être fixés au moyen des boîtes de texte correspondants :

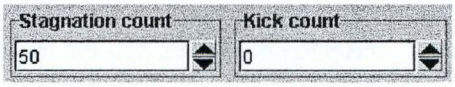


La mesure de *stagnation* indique le nombre de générations qui se sont écoulées sans amélioration de la *moyenne de viabilité* de la population. Une fois le seuil de stagnation atteint, un *kick* est effectué sur la population.

Le *kick* choisit totalement au hasard, sans préjuger de leur viabilité, un certain nombre d'individus de la population et les remplace par de nouveaux individus générés aléatoirement. Cette manoeuvre permet d'essayer de sortir le problème d'un optimum local où il serait coincé en rediversifiant la population. La proportion de la population à être remplacée de cette manière est configurable au moyen du champ de texte :



On peut programmer le critère de stagnation et le kick très finement au moyen des champs :

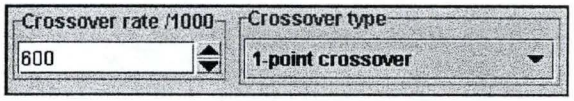


Les règles sont les suivantes :

- Si la stagnation n'est pas un critère d'arrêt (la boîte à cocher n'est pas sélectionnée) :
  - Si la limite de stagnation est à zéro, il n'y a jamais de kick.
  - Sinon, il y a kick à chaque fois que la limite de stagnation est atteinte.
- Si la limite de stagnation est un critère d'arrêt :
  - Si la limite de kick est à zéro, dès que la limite de stagnation est atteinte, l'algorithme se termine.
  - Sinon, un kick est produit chaque fois que la limite de stagnation est atteinte, à concurrence du nombre de kicks indiqués.

5.3.3 Configuration du crossover

Il est possible de déterminer la probabilité de crossover (exprimée *pro milla*). On choisira généralement une probabilité assez haute ; par défaut, elle est fixée à 0,6 (600/1000). On peut également déterminer le type de crossover, au moyen des composants :



On peut choisir de ne pas utiliser de crossover en sélectionnant *none* dans la boîte de sélection de type. Il y a deux types de crossovers, mutuellement exclusifs : les crossovers orientés nucléotides, et les crossovers orientés gènes. On ne peut sélectionner que les crossovers adaptés au problème que l'on traite[2].

Décrivons à présent les différents types de crossover proposés.

Le crossover à un point

Ce type de crossover a été essentiellement décrit dans le chapitre sur framework. Dans cette implémentation, deux individus de la population créée par la sélection sont choisis au hasard. D'abord, un tirage aléatoire détermine si le crossover doit avoir lieu ou pas en fonction de la probabilité donnée. Si le tirage est positif (par exemple, si le ratio de probabilité donné est de 600 pour mille, un tirage aléatoire x tel que $0 < x <= 600$ sera positif, et tout autre tirage négatif), le crossover est effectué. Un point de crossover est déterminé aléatoirement, et les individus échangent leurs gènes situés avant le point de crossover.

Le crossover à deux points

Ce crossover est identique au précédent si ce n'est que deux points de crossover sont choisis aléatoirement, et que l'échange de gènes se produit dans la portion située entre les deux points. Dans certains cas (en fonction du tirage aléatoire des points de crossover), le crossover à deux points agit comme un crossover à un point (si le point de crossover d'index faible est zéro, ou le point de crossover d'index fort est égal à la taille du chromosome).

Le framework permet de déterminer finement le comportement du crossover à deux points en déterminant la taille minimale et maximale de la portion de crossover. Cependant, cette fonctionnalité, bien qu'implémentée dans le framework, n'est pas encore présente dans la version actuelle de l'outil de démonstration.

Le *Partially Matched crossover (PMX)*

Goldberg introduit plusieurs types de *crossovers géniques* qui agissent par échange de gènes sans en modifier le contenu comme le ferait un crossover classique. Comme nous l'avons vu, il existe deux grands types d'opérateurs : ceux qui agissent sur les bits/nucléotides (crossover et mutation) et ceux qui agissent sur l'ordre des gènes. Originellement, le seul opérateur agissant sur l'ordre des gènes était l'inversion. Goldberg et d'autres, à partir de 1985, ont tenté de mettre au point un type de crossover préservant le contenu des gènes[3].

L'implémentation la plus classique et importante de ce type d'opérateurs est le problème du voyageur de commerce (*Traveling Salesman Problem* ou TSP),

[2] En principe, on ne devrait voir dans la liste que les crossovers adaptés au problème. Dans l'état actuel, on voit cependant tous les types, mais on ne peut sélectionner que ceux qui sont adaptés au problème. C'est un défaut mineur de l'interface.

[3] Goldberg classe l'inversion et ces crossovers "géniques" (ce terme n'est pas utilisé par Goldberg) sous l'appellation de *reordering operators*([Goldberg 1989], chapitre 5).

où un voyageur doit visiter un nombre donné de villes, en visitant chaque ville une et une seule fois, de manière à minimiser la longueur du trajet parcouru.

Ceci n'est pas aussi simple que le crossover par échange de nucléotides. En effet, considérons deux chromosomes constitués de 8 gènes contenant les nombres de 1 à 8 (chaque gène a une valeur différente). Dans cet exemple, nous ignorons la représentation des gènes et donc comment les gènes sont traduits en une chaîne de nucléotides. Considérons que ces individus sont constitués de la manière suivante :

1	2	3	4	5	6	7	8
2	1	8	7	6	4	5	3

Si l'on échange les gènes en position 3, on obtient

1	2	8	4	5	6	7	8
2	1	3	7	6	4	5	3

Cette situation n'est pas acceptable, puisque le premier chromosome a deux copies de la valeur 8 et le second deux copies de la valeur 3. En effet, ceci est équivalent à changer le contenu d'un des gènes, ce qui est justement ce que l'on veut éviter.

Il a donc fallu mettre au point des stratégies permettant de régler ce problème. Les trois crossovers géniques présentés ici utilisent des stratégies légèrement différentes, qui permettent des effets subtilement différents, que nous allons décrire. Ces trois opérateurs proviennent de [Goldberg 1989].

Note Les crossovers géniques demandent une certaine cohérence dans l'organisation des gènes. Pour le moins, chaque gène de chaque individu doit avoir un correspondant dans tous les individus de la population (c'est à dire un gène codé de manière identique et possédant la même valeur), mais naturellement pas nécessairement dans la même position.
Le framework vérifie uniquement que le nombre de gènes est identique et que le nombre de nucléotides est également identique. Ce n'est bien sûr pas une garantie absolue, mais de cette manière le framework est ouvert à des implémentations plus diverses.
La seule limitation est celle évoquée ci-dessus : si un individu porte un gène de type t et de valeur x, tous les chromosomes de la population doivent porter un gène de type t et de valeur x sans quoi l'implémentation refusera d'exécuter l'algorithme ou en arrêtera l'exécution lors de la première application d'un crossover génique (avec une erreur). Pour éviter des difficultés, on codera préférentiellement les individus avec des gènes tous de même type.

Note Les crossovers géniques sont également soumis à une certaine probabilité d'exécution, comme les autres crossovers.

Le *partially matched crossover* (crossover partiellement apparié), ou *PMX*, commence par déterminer un site de crossover en choisissant aléatoirement deux points de crossover :

1	2	3	4	5	6	7	8
2	1	8	7	6	4	5	3

Il procède ensuite à l'échange des gènes dans la portion de crossover, en utilisant la stratégie suivante pour garantir la consistance des chromosomes : pour chaque gène se trouvant dans la zone d'échange (de gauche à droite), il localise sur le

même chromosome le gène identique à son alter ego de l'autre chromosome, puis les échange. Il fait de même pour tous les gènes de la zone d'échange, puis effectue la même opération pour le second chromosome. A proprement parler, il n'y a pas d'échange de matériel génétique entre les chromosomes. Voici un exemple détaillé. Utilisons l'exemple ci-dessus :

0.	1	2	3	4	5	6	7	8
	2	1	8	7	6	4	5	3

Traitons d'abord le premier chromosome :

1.	1	2	3	4	5	6	7	8
			8	7	6			
2.	1	2	**8**	4	5	6	7	**3**
			8	7	6			
3.	1	2	**8**	**7**	5	6	**4**	**3**
			8	**7**	6			
4.	1	2	**8**	**7**	**6**	**5**	**4**	**3**
			8	**7**	**6**			

Puis le second :

5.			3	4	5			
	2	1	8	7	6	4	5	3
6.			**3**	4	5			
	2	1	**3**	7	6	4	5	**8**
7.			**3**	**4**	5			
	2	1	**3**	**4**	6	**7**	5	**8**
8.			**3**	**4**	**5**			
	2	1	**3**	**4**	**5**	**7**	**6**	**8**

Soit en finale :

1	2	**8**	**7**	**6**	**5**	**4**	**3**
2	1	**3**	**4**	**5**	**7**	**6**	**8**

Bien qu'il permette un échange de "schémas"[4] au niveau génique (combinaison de gènes dans un ordre donné) et permette donc de transposer la métaphore des schémas au niveau des gènes, le PMX impose une recombinaison des gènes situés en dehors de la zone d'échange qui reste assez aléatoire, et peut donc détruire des ordonnancements de gènes utiles. Les deux autres opérateurs proposent des stratégies permettant une meilleure sauvegarde de l'ordre des gènes.

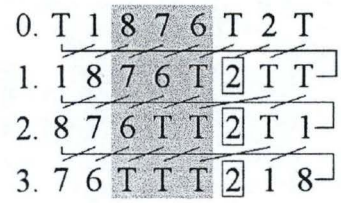
L'*Order crossover (OX)*

Le crossover "ordonné" propose une stratégie permettant de maintenir l'ordre relatif des gènes en-dehors du site de crossover. Considérons un nouvel exemple proche du précédent :

1	2	3	4	5	6	7	8
5	1	8	7	6	4	2	3

[4] Il s'agit naturellement ici d'un abus de langage.

Occupons-nous du second individu qui illustrera plus clairement notre propos. Remplaçons dans cet individu les gènes de valeur identique aux gènes occupant le site de crossover du premier individu par des "trous" indiqués par la lettre T : $T\ \ 1 \mid 8\ \ 7\ \ 6 \mid T\ \ 2\ \ T$. Nous allons déplacer ces trous vers le site de crossover par un glissement de tous les gènes à partir du trou qui suit directement la limite haute (ou de droite) du site de crossover, à l'exception des gènes se trouvant entre la limite haute et le trou déplacé. Par souci de clarté, nous allons illustrer ceci étape par étape :



Après quoi on remplace les trous par le contenu du site de crossover du partenaire de l'individu, ce qui donne : $7\ \ 6 \mid 3\ \ 4\ \ 5 \mid 2\ \ 1\ \ 8$.

On procède de la même manière pour le premier individu :

$$1\ \ 2 \mid 3\ \ 4\ \ 5 \mid T\ \ T\ \ T$$

Ceci donne facilement :

$$4\ \ 5 \mid T\ \ T\ \ T \mid 1\ \ 2\ \ 3$$

et donc, après échange :

$$4\ \ 5 \mid 8\ \ 7\ \ 6 \mid 1\ \ 2\ \ 3$$

Si l'on applique cette solution au problème du voyageur de commerce, l'ordre relatif de visite des villes est préservé par l'opérateur, mais la ville de départ change continuellement. Si l'implémentation n'impose pas la ville de départ, cet opérateur est tout-à-fait adapté à ce problème.

Le *Cycle crossover (CX)*

Le crossover cyclique tente au contraire de préserver la position de départ et les positions absolues d'une partie des gènes. Pour commencer, on sélectionne le premier gène de chaque individu :

1	2	3	4	5	6	7	8
5	1	8	7	6	4	2	3

Nous allons ensuite localiser, dans le premier chromosome, la position du gène identique à celui se trouvant en position 1 sur le second chromosome (soit la valeur 5 dans notre exemple), et créer ainsi une nouvelle paire :

1				2			
1	2	3	4	**5**	6	7	8
5	1	8	7	**6**	4	2	3

Et ainsi de suite :

		1				2	3	
1.	**1**	2	3	4	**5**	**6**	7	8
	5	1	8	7	**6**	**4**	2	3
		1			4	2	3	
2.	**1**	2	3	**4**	**5**	**6**	7	8
	5	1	8	**7**	**6**	**4**	2	3
		1			4	2	3	5
3.	**1**	2	3	**4**	**5**	**6**	**7**	8
	5	1	8	**7**	**6**	**4**	**2**	3
		1	6		4	2	3	5
4.	**1**	**2**	3	**4**	**5**	**6**	**7**	8
	5	**1**	8	**7**	**6**	**4**	**2**	3

Dans la dernière étape, on constate que 1 est un élément déjà traité (lors de la première sélection). On a donc bouclé un cycle de sélections. On constate à présent qu'en inversant les paires sélectionnées, les deux individus auront toujours un stock complet de gènes :

5	**1**	3	**7**	**6**	**4**	**2**	8
1	**2**	8	**4**	**5**	**6**	**7**	3

Selon la disposition des gènes, un nombre variable de gènes seront échangés (de 1 à l'ensemble des gènes - dans ces deux cas, les individus resteront inchangés).

Dans une implémentation du problème du voyageur de commerce, s'il y a une contrainte de départ d'une ville donnée, on pourra utiliser cet opérateur. Cette contrainte est cependant facilement découplable de l'opérateur, ce qui est préférable dans un souci de généralisation. Nous avons donc modifié le crossover cyclique en choisissant le premier gène à traiter au hasard sur la chaîne plutôt que de prendre systématiquement le premier. De cette manière, on pourra utiliser cet opérateur dans toutes les variantes du voyageur de commerce, et éventuellement sur d'autres problèmes.

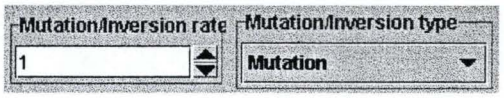
5.3.4 Configuration de la mutation ou de l'inversion

La mutation et l'inversion, bien qu'étant deux concepts très différents, ont été groupés sous un même item pour les raisons suivantes :

- Ils sont mutuellement exclusifs (un même problème ne devrait pas proposer à la fois la mutation et l'inversion).
- Ce sont tous deux des opérateurs qui agissent sur un individu unique (au contraire du crossover qui combine deux individus).

Sous cet item, la mutation est le seul opérateur agissant au niveau des bits/nucléotides. Les opérateurs agissant au niveau de l'ordre des gènes sont l'inversion, l'insertion et l'échange mutuel.

Le type d'opérateur et la probabilité d'exécution peuvent être configurés au moyen des champs suivants :



Pour rappel, on ne peut sélectionner que les opérateurs adaptés au type de problème traité (orienté nucléotide ou orienté gène), mais tous les opérateurs sont visibles dans la liste, ce qui est un défaut mineur de l'interface.

La mutation

Comme expliqué précédemment, la mutation inverse un bit donné selon une probabilité donnée. La probabilité s'applique à chaque bit individuellement, il est donc très important de garder une probabilité basse afin de ne pas détruire l'effet des autres opérateurs. Le sélecteur de probabilité permet cependant de donner une probabilité quelconque (au millième près).

L'inversion à un ou deux points

L'inversion est un opérateur opérant sur l'ordre des gènes. Son effet est d'inverser l'ordre des gènes d'une portion du chromosome. C'est à nouveau un opérateur adapté au problème du voyageur de commerce, et il a été principalement étudié dans ce but.

Par souci de complétude, le framework propose une inversion à un point, où la portion de chromosome inversée s'étend toujours d'un point donné à l'un des deux bouts du chromosome. Par exemple :

$$1\ 2\ 3\ 4\,|\,5\ 6\ 7\ 8 \rightarrow 4\ 3\ 2\ 1\,|\,5\ 6\ 7\ 8$$

ou :

$$1\ 2\ 3\ 4\,|\,5\ 6\ 7\ 8 \rightarrow 1\ 2\ 3\ 4\,|\,8\ 7\ 6\ 5$$

Le framework propose également une inversion à deux points (qui est en fait celle décrite par la littérature), où une portion située entre deux *loci* est inversée comme par exemple :

$$1\ 2\,|\,3\ 4\ 5\ 6\,|\,7\ 8 \rightarrow 1\ 2\,|\,6\ 5\ 4\ 3\,|\,7\ 8$$

Comme noté pour le crossover, dans certaines circonstances l'inversion a deux points a le même effet que l'inversion à un point (si l'une des bornes de l'inversion se situe sur une des limites du chromosome).

La probabilité d'inversion se situera dans le même ordre de grandeur que celle du crossover.

Le framework et l'application de démonstration permettent d'appliquer l'inversion et un crossover génique sur le même problème. Ce n'est cependant probablement pas une bonne idée, les effets cumulés des deux opérateurs pouvant en finale être trop disruptifs.

L'insertion et l'échange mutuel

Les crossovers géniques ont une certaine tendance à "plafonner" une fois atteint un certain seuil de viabilité moyenne, parfois assez éloigné de l'optimum. Il leur manque en effet l'équivalent de la mutation pour améliorer leur dynamisme. L'insertion et l'échange mutuel ont été introduits pour servir de contrepartie génique à la mutation. Ils seront donc utilisés en complément des crossovers géniques.

Ce sont des opérateurs très simples et très similaires. L'insertion consiste à sélectionner un gène, à l'extraire de sa position, et à l'insérer entre deux autres gènes, en déplaçant tous les gènes pour remplir le trou laissé par le gène déplacé. Par exemple :

$$\begin{array}{cccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \rightarrow & 1 & 2 & 3 & 5 & 6 & 4 & 7 & 8 \\ & & & \downarrow & & & & & & & & & \leftarrow & \leftarrow & \uparrow & & \end{array}$$

ou (autre exemple) :

$$\begin{array}{cccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \rightarrow & 1 & 2 & 7 & 3 & 4 & 5 & 6 & 8 \\ & & & & & & \downarrow & & & & & \uparrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \end{array}$$

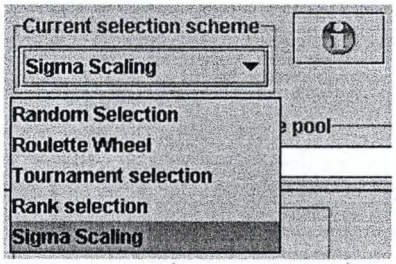
L'échange mutuel est pratiquement identique, sauf que dans ce cas deux gènes sont échangés (et donc deux et seulement deux gènes sont déplacés). Par exemple :

$$\begin{array}{cccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \rightarrow & 1 & 2 & 7 & 4 & 5 & 6 & 3 & 8 \\ & & \downarrow & & & & \downarrow & & & & & \uparrow & & & & \uparrow & \end{array}$$

La probabilité d'application de ces opérateurs est évaluée pour chaque gène individuellement. De ce fait, elle devrait être un peu plus élevée que celle de la mutation (puisqu'il y a typiquement plus de nucléotides que de gènes sur un chromosome), mais nettement inférieure à celle du crossover ou de l'inversion. Le défaut est de 0,01 (10 pour mille).

5.3.5 Configuration de la stratégie de sélection

On peut choisir la stratégie de sélection au moyen de la boîte de sélection :



Le bouton marqué du symbole permet d'obtenir un texte informatif sur la stratégie de sélection active (sous la forme d'une info-bulle en passant la souris sur le bouton ou sous la forme d'une boîte de dialogue en cliquant sur le bouton).

Certaines stratégies de sélection peuvent être configurées finement, comme nous allons le voir. Dans ce cas, les paramètres de configuration apparaissent à la droite du bouton d'information.

La sélection aléatoire

La sélection aléatoire est une stratégie de sélection un peu à part. En effet, les individus sont sélectionnés entièrement au hasard, et en conséquence la stratégie de sélection ne permet pas d'améliorer la population. Cette stratégie peut être utilisée comme base de référence pour une comparaison d'efficacité des stratégies

de sélection mais en fait son but est tout différent. Elle permet d'étudier l'effet de l'élitisme (expliqué dans la section suivante) en enlevant l'influence de la stratégie de sélection.

En fait, l'élitisme seul (c'est à dire couplé avec une sélection aléatoire) permet tout à fait de faire fonctionner l'algorithme génétique de manière très satisfaisante dans beaucoup de cas. En conséquence, il n'est pas inimaginable d'utiliser cette stratégie de sélection (couplée à l'élitisme) dans des situations réelles, et pas seulement des études de cas.

Cette stratégie de sélection n'a pas de paramètre de configuration.

La sélection proportionnelle à la viabilité (*roulette wheel*)

La sélection proportionnelle à la viabilité (*fitness-proportionate*, surnommée *roulette wheel* en référence au jeu de casino) est la stratégie de sélection la plus classique, décrite et généralement prise en référence par Goldberg et probablement la première à avoir été utilisée. Elle sert aussi de modèle au développement de base (ou restreint) du théorème des schémas.

Le principe en est simple : chaque individu reçoit une probabilité d'être sélectionné pour se reproduire directement proportionnelle à la mesure de sa viabilité.

Ce modèle classique et raisonnablement efficace souffre cependant de certains défauts assez contraignants :

- Il réclame des calculs assez lourds : totalisation de la viabilité de la population (avec le risque de dépassement de capacité en cas de représentation en type entier ou de perte de précision en cas de représentation en virgule flottante), totalisation des viabilités d'un nombre quelconque d'individus pour chaque sélection, mise à l'échelle éventuelle en cas de représentation de la viabilité par des grands nombres.
- Il pose problème pour le traitement de viabilités négatives. Goldberg semble considérer la positivité de la viabilité comme un prérequis, bien qu'il ne l'évoque pas spécifiquement, mais en toute généralité rien n'empêche de considérer des viabilités négatives. Dans ce cas, la sélection proportionnelle réclame une mise à l'échelle des valeurs afin que les valeurs négatives soient replacées dans le domaine du positif et les autres valeurs décalées vers le haut (avec une décision particulière à prendre pour la plus petite valeur, qui ne peut avoir une viabilité nulle). Cette mise à l'échelle alourdit à nouveau le traitement de cette stratégie de sélection.

L'implémentation de *Cuvier* n'accepte que des valeurs de viabilité positives pour cette stratégie de sélection (l'algorithme s'arrêtera s'il découvre un individu de viabilité négative), et il utilise en interne des valeurs en virgule flottante (codage par *double*), ce qui peut amener des pertes de résolution (en principe sans gravité dans la plus grande majorité des cas).

Cette stratégie de sélection n'a pas de paramètre de configuration.

La sélection par tournoi (*tournament*)

La stratégie proportionnelle à la viabilité n'est pas optimale parce qu'en début d'exécution, des différences importantes existent entre les individus en termes de viabilité. En conséquence, les individus les plus viables sont très rapidement sélectionnés, diminuant très (trop) rapidement la diversité de la po-

pulation, ce qui peut amener des problèmes de stagnation par "gavage" de la population en individus certes viables mais peut-être pas optimaux.

De même, une fois que la population s'approche de l'optimum, la variance de la viabilité de la population diminue du fait de la richesse de la population en individus performants. A ce moment, la sélection proportionnelle devient très faible puisque tous les individus ont une chance à peu près identique d'être sélectionnés, et donc la population ne s'améliore plus.

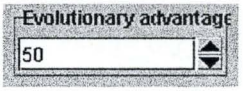
On l'a vu, idéalement, la pression de sélection ne doit pas être trop forte au début de l'exécution de l'algorithme, et devrait être plus forte en fin d'exécution. La sélection par tournoi propose une solution simple à ce problème.

Dans cette stratégie, deux individus sont choisis aléatoirement dans la population. Celui qui a la viabilité la plus élevée reçoit une probabilité de sélection plus élevée que l'autre. Un tirage au sort est alors effectué en fonction du rapport fixe entre les individus.

Ceci a pour effet d'aplanir la pression de sélection sur la durée d'exécution de l'algorithme. En effet, que les viabilités soient très différentes ou pas, le rapport de probabilités entre les deux individus est toujours le même (fixé par paramètre). La pression est donc moins forte en début d'algorithme, comparativement à la sélection proportionnelle, et plus forte en fin d'exécution (puisque le rapport de probabilités reste le même malgré le peu de différence entre les individus).

Cette stratégie a d'autres avantages. Elle ne nécessite pas du tout de calculs lourds (une simple comparaison entre les viabilités suffit), et accepte des valeurs de viabilité négatives sans difficulté. Elle est de plus tout à fait efficace.

L'avantage donné à l'individu de meilleure viabilité au cours des "tournois" peut être configuré au moyen de la boîte de texte :



La valeur, exprimée en pourcentage, indique un *avantage* : si la valeur est à 0, la probabilité de sélection pour chaque individu est de 0,5. Si la valeur est à 100, la probabilité de sélection pour l'individu le plus faible est de 0, et pour le plus fort de 1. Il est impossible de donner une probabilité plus faible à l'individu le plus fort. La valeur par défaut est de 50%, ce qui représente une probabilité de sélection de 0,75 pour l'individu le plus fort et 0,25 pour le plus faible.

La sélection proportionnelle au sigma (*sigma scaling*)

La stratégie proportionnelle au *sigma*, c'est à dire à l'écart-type (*standard deviation*), essaie de s'attacher encore plus précisément à la variation des écarts de viabilité dans la population en cours d'exécution de l'algorithme. Pour ce faire, il utilise l'écart-type, une mesure qui indique l'écart moyen d'un ensemble de valeurs à leur moyenne, par la formule :

$$V = \frac{\sum_{i=1}^{N}(x_i - M)}{N}$$

et

$$\sigma = \sqrt{V}$$

V est appelé la *variance* et σ, l'écart-type, est la racine carrée de la variance ; N est le nombre de valeurs présent dans l'ensemble, x_i est une valeur de l'ensemble, et M la moyenne des valeurs de l'ensemble.

Cette stratégie de sélection utilise la formule suivante :

$$val(i,t) = 1 + \frac{f(i) - \overline{f}(t)}{2\sigma(t)}$$

où $f(i)$ est la mesure de la viabilité d'un individu i, et $\overline{f}(t)$ est la moyenne des viabilités de la population au temps (à la génération) t. La probabilité de sélection d'un individu sera proportionnelle à la valeur retournée par cette formule. Cette valeur se situe autour de la valeur 1. Dans la plupart des cas, elle sera située entre 0 et 2.

Il est cependant possible, pour certaines populations très atypiques (par exemple, tous les individus ont une viabilité très faible sauf un, de viabilité très grande), d'obtenir un résultat inférieur à 0. Ceci pose un problème pour la sélection, qui réclame des valeurs toutes positives. La solution proposée par R. Tanese dans sa thèse de 1989 intitulée *Distributed Genetic Algorithms for Function Optimization*, citée dans [Mitchell 1996], est de ramener la valeur à $0,1$ pour donner à l'individu une chance de sélection. Ce framework implémente une solution légèrement différente, en calculant une valeur de correction basée sur le double de la valeur minimum calculée par la formule, ramené en positif. Si par exemple, cette valeur est de $-0,3$, on corrigera toutes les valeurs en leur ajoutant $0,6$. Ceci garantit à la fois la positivité de toutes les valeurs, et une correction des valeurs négatives proportionnelle, et donc plus réaliste.

Notons qu'il est extrêmement improbable que cette formule retourne des nombres négatifs très grands.

Si l'écart-type est de 0, la formule de calcul n'est pas valide. Cependant, ce cas spécial signifie que toutes les viabilités de tous les individus sont identiques. Dans ce cas, on fixe le résultat à 1 de sorte que naturellement tous les individus ont la même probabilité de sélection.

Enfin, les réserves exposées pour la stratégie proportionnelle à la viabilité sont d'application ici, à savoir que la viabilité de tous les individus doit être un nombre positif (sans quoi le calcul de la moyenne et de l'écart-type seront incorrects - le framework stoppe l'exécution dès qu'il rencontre des valeurs de viabilité négatives), et que l'exécution de cette stratégie de sélection est relativement lourde en termes de traitement.

Cette stratégie de sélection n'a pas de paramètre de configuration.

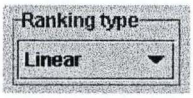
La sélection par classement (*rank*)

La sélection par classement tente, comme la sélection par tournoi, d'aplanir les différences entre viabilités en cours d'exécution de l'algorithme. Comme pour la sélection par tournoi, seul le classement des viabilités est important, et non pas leur valeur.

Dans ce modèle, les individus sont classés par ordre de viabilité. Après quoi la probabilité de sélection d'un individu est établie en fonction de son classement, selon un modèle donné. Cette implémentation propose deux modèles : un

modèle linéaire et un modèle exponentiel. Dans le modèle linéaire, la probabilité de sélection est directement proportionnelle à la position de l'individu dans le classement des viabilités. Dans le modèle exponentiel, la probabilité de sélection est proportionnelle à l'exponentielle du classement de l'individu, ce qui permet de donner une probabilité de sélection encore plus importante aux individus les plus viables.

On peut choisir le modèle de la sélection au moyen de la boîte de choix :



Le modèle exponentiel utilise la base 2, de manière à rester dans des limites de valeurs raisonnables tout en permettant une discrimination nette entre les individus (ce que n'aurait pas permis une base inférieure à 2). Il ne permet cependant pas un effectif de population supérieur à 62 individus, parce que cette sélection opère son tirage aléatoire dans le domaine du type entier long.

5.3.6 L'élitisme

On peut améliorer sensiblement le rendement des stratégies de sélection en leur adjoignant un système d'élitisme. Le principe de l'élitisme est de réserver aux plus viables des individus une garantie de sélection. On peut, par exemple, sélectionner un nombre donné de fois l'individu le plus performant avant de commencer la sélection en elle-même.

Dans cette implémentation, l'élitisme peut être appliqué au-dessus de toutes les stratégies de sélection, et est configuré au moyen des contrôles suivants :

L'élitisme sélectionne des individus parmi les plus viables un certain nombre de fois, indiqué par la première boîte de texte. Les individus sont sélectionnés dans un groupe des meilleurs individus de la population. Le nombre d'individus constituant ce groupe est indiqué par la seconde boîte de texte. Enfin, ces quantités peuvent être exprimées en nombre d'individus ou en pourcentage de la population, comme indiqué par la boîte à cocher.

Prenons un exemple simple. Si le nombre d'individus à sélectionner est de 10, et que le groupe d'individus est également de 10 (et supposons qu'il s'agit là de quantités et pas de pourcentages), la routine d'élitisme va localiser les 10 meilleurs individus de la population. Ensuite, elle va tirer au sort dix fois pour sélectionner un individu au hasard dans le groupe.

Si l'on veut privilégier uniquement le meilleur individu, par exemple, il suffit de donner une taille de groupe d'une unité, et d'indiquer le nombre de fois que cet individu devra être sélectionné.

Pour rappel, il est intéressant, pour évaluer les effets de l'élitisme, de le coupler à la stratégie de sélection aléatoire.

5.4 Statistiques

L'onglet de statistiques propose des informations sur l'exécution de l'algorithme, mises à jour dynamiquement en cours d'exécution. Il est divisé en deux parties, la partie supérieure est un "tableau de bord" donnant toutes sortes d'informations utiles, et la partie inférieure un graphique de visualisation de l'évolution de la viabilité de la population.

La figure 5.2 montre un exemple de contenu de l'onglet de statistiques (identique à la figure 4.2).

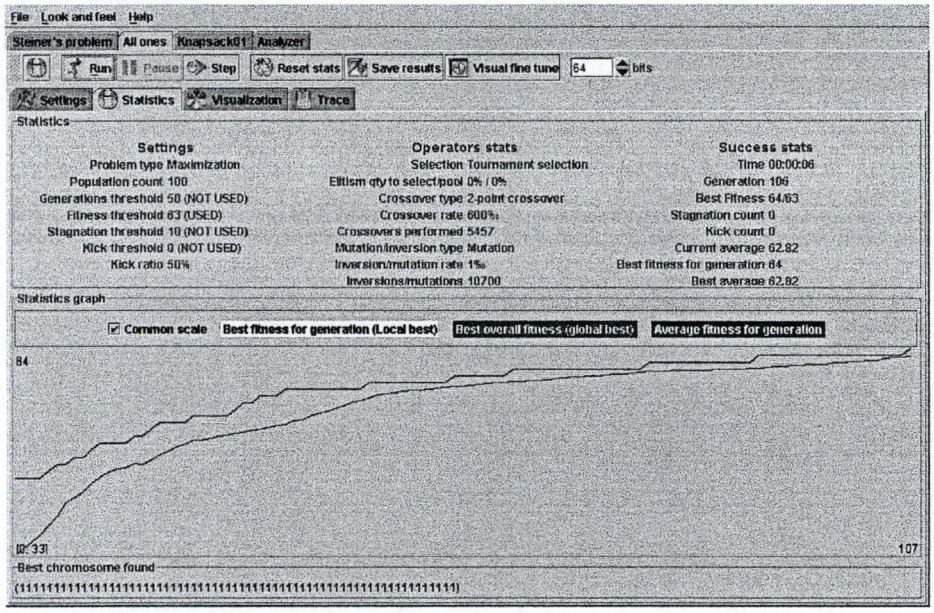
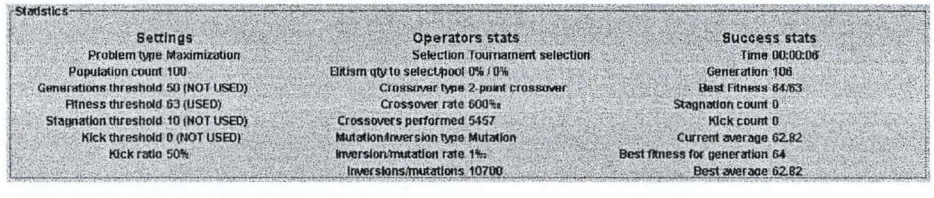


FIG. 5.2 – Onglet de statistiques

5.4.1 Tableau de bord

La partie "tableau de bord", dont voici une illustration, propose trois sections :



Paramètres (*settings*)

La section des paramètres est un rappel des paramètres généraux qui ont été fixés au moyen de l'onglet de configuration.

Problem type indique s'il s'agit d'une maximisation ou d'une minimisation.

Population count indique le nombre d'individus de la population.

Generations threshold indique le nombre de générations à partir duquel l'exécution de l'algorithme est considérée comme un succès, et si ce paramètre est pris en compte comme critère d'arrêt.

Fitness threshold indique le seuil de viabilité à partir duquel l'exécution est considérée comme un succès, et si ce paramètre est pris en compte.

Stagnation threshold indique le seuil de stagnation à partir duquel l'exécution est considérée comme un succès, et si ce paramètre est pris en compte.

Kick threshold indique après combien de *kicks* l'exécution est considérée comme un succès (dès que le seuil de stagnation est à nouveau atteint).

Kick ratio rappelle le pourcentage de la population réinitialisé par un *kick*.

Statistiques sur les opérateurs (*operators stats*)

La section de statistiques sur les opérateurs rappelle les paramètres qui ont été fixés en ce qui concerne les opérateurs génétiques, et donne des informations sur le nombre de fois que les opérateurs ont été appliqués.

Selection indique quelle stratégie de sélection est utilisée.

Elitism qty to select/pool indique le nombre d'individus que l'élitisme doit sélectionner, et la taille du groupe d'où ces individus sont choisis. S'il s'agit de pourcentages, le symbole % est indiqué.

Crossover type indique le type de crossover utilisé.

Crossover rate indique la probabilité d'application du crossover, *pro milla*.

Crossovers performed indique le nombre absolu de crossovers effectués au cours de l'exécution courante.

Mutation/inversion type indique le type d'opérateur unitaire utilisé (mutation, inversion, insertion, ou échange réciproque).

Inversion/mutation rate indique la probabilité d'application de l'opérateur unitaire sélectionné.

Inversions/Mutations indique le nombre absolu d'applications de l'opérateur unitaire sélectionné au cours de l'exécution courante.

Statistiques de performances (*success stats*)

Les statistiques de performances donnent des informations sur la qualité de l'algorithme et de la solution trouvée.

Time indique la durée d'exécution de l'algorithme.

Generation donne l'index de la génération courante. Si le nombre de générations est un critère d'arrêt, indique aussi le nombre maximum de générations.

Best fitness donne la meilleure viabilité trouvée jusqu'à présent.

Stagnation count indique le nombre de générations écoulées sans amélioration de la moyenne de la viabilité de la population.

Kick count indique le nombre de *kicks* effectués au cours de l'exécution courante.

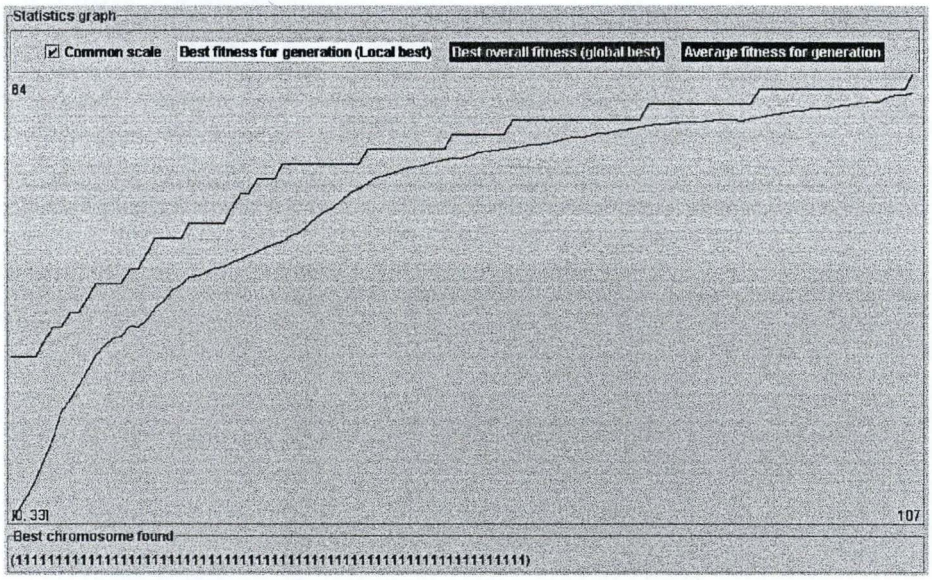
Current average donne la moyenne de viabilité de la population à la génération courante.

Best fitness for generation donne la meilleure mesure de viabilité de la génération courante.

Best average donne la meilleure moyenne de viabilité rencontrée.

5.4.2 Graphique

L'onglet de statistiques propose également une visualisation graphique des statistiques de performances :



L'axe des abscisses est l'axe du temps (c'est à dire des générations). Il est mis dynamiquement à l'échelle de sorte que l'ensemble des générations couvre toujours l'entièreté de la largeur du graphe (pour plus de lisibilité).

L'axe des ordonnées indique la mesure de viabilité.

Trois mesures sont montrées sur ce graphe : en jaune, la meilleure viabilité de la génération courante, en rouge, la meilleure viabilité trouvée (celle-ci recouvre souvent le graphe de meilleure viabilité locale), et enfin en bleu la viabilité moyenne de la population. Les trois mesures sont montrées sur une même échelle, mais elles peuvent être montrées dans leur propre échelle (de sorte que chacune des mesures occupe tout l'espace vertical). C'est parfois utile si les variations de meilleure viabilité sont assez faibles.

Note Un *kick*, à moins qu'il n'ait été programmé pour réinitialiser seulement une portion faible de la population, est toujours clairement identifiable sur le graphe car il se traduit par une diminution brusque et localisée de la viabilité moyenne.

En-dessous du graphe, on trouve encore une visualisation du meilleur individu trouvé. Selon le problème codé, cette visualisation peut ne pas être très lisible (d'où l'intérêt de l'onglet de visualisation) ou alors être trop longue pour être visualisée à l'écran. Dans ce cas, il est possible de faire défiler la vue du chromosome au moyen de la souris (par "glisser-déplacer").

5.5 Visualisation

L'onglet de visualisation dépend entièrement de l'implémenteur du problème, et est donc différent pour chaque implémentation. Quelques exemples seront montrés dans un chapitre ultérieur. Par défaut, la page est vide.

5.6 Rapport d'exécution

L'onglet de rapport d'exécution permet de visualiser un rapport détaillé de l'exécution de l'algorithme génétique. La figure 5.3 en montre un exemple.

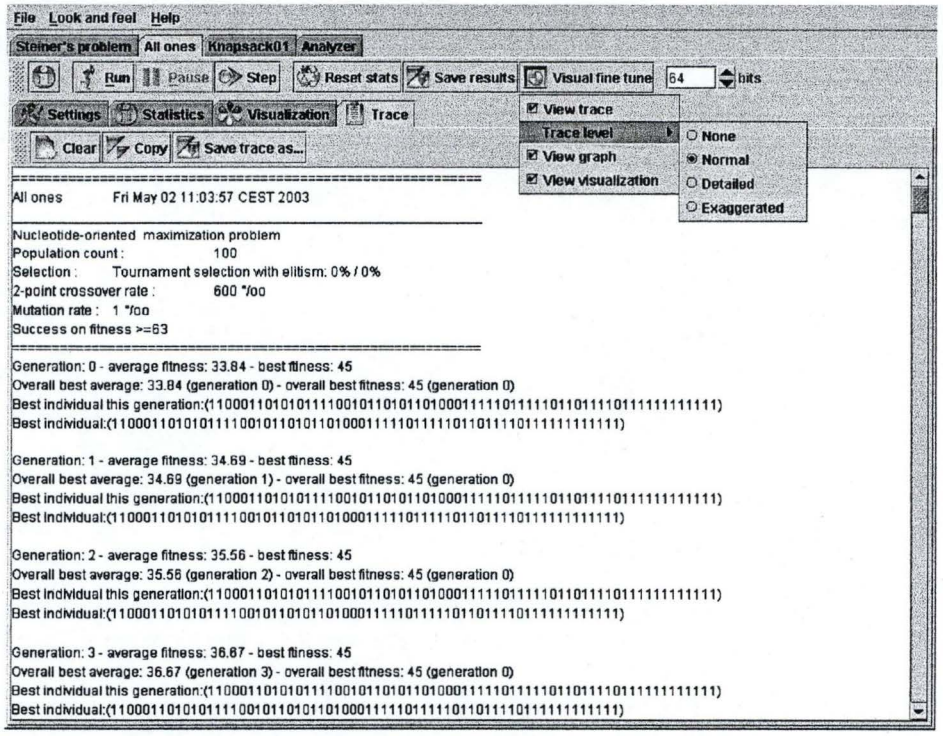


FIG. 5.3 – Rapport d'exécution

Le contenu du rapport est assez explicite et ne nécessite pas de commentaires. En voici un exemple (le contenu trop long ou trop répétitif a été supprimé pour une meilleure lisibilité) :

```
============================================================
All ones Fri May 02 11:03:57 CEST 2003
------------------------------------------------------------
Nucleotide-oriented  maximization problem
Population count : 100
Selection : Tournament selection with elitism: 0% / 0%
2-point crossover rate : 600 °/oo
Mutation rate : 1 °/oo
```

```
Success on fitness >=63
============================================================
Generation: 0 - average fitness: 33.84 - best fitness: 45
Overall best average: 33.84 (generation 0) -
                      overall best fitness: 45 (generation 0)
Best individual this generation:(...)
Best individual:(...)

...

Generation: 106 - average fitness: 62.82 - best fitness: 64
Overall best average: 62.82 (generation 106) -
                      overall best fitness: 64 (generation 106)
Best individual this generation:(...)
Best individual:(...)

============================================================
Success because fitness threshold reached !
Execution started at 11:03:57
Execution took 00:00:06
============================================================
```

Les boutons de la barre d'outils du rapport permettent respectivement d'effacer le rapport, de copier l'entièreté du rapport dans le presse-papiers du système, et de sauvegarder le contenu du rapport dans un fichier au choix de l'utilisateur.

Le menu "*visual fine tune*" de la barre d'outils permet de gérer finement le contenu du rapport. Les niveaux de détail sont les suivants :

None signifie que le rapport est vide. A utiliser si le rapport n'est pas utile et que l'on désire gagner un peu de performance à l'exécution.

Normal donne le rapport illustré ci-dessus. Il contient toutes les informations générales utiles et, pour chaque génération, des informations résumées.

Detailed donne les mêmes informations que le niveau normal mais ajoute une liste complète des chromosomes et de leur valeur de viabilité à chaque génération.

Exaggerated ajoute au niveau détaillé une indication de chaque application d'un opérateur génétique, en indiquant le ou les chromosome(s) concerné(s).

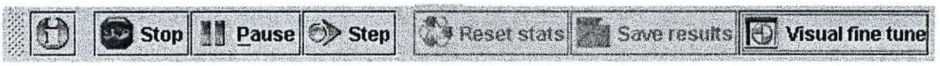
Il faut bien sûr rester prudent et raisonnable dans l'usage des deux derniers niveaux. Selon l'importance de la population et le nombre de générations, le rapport peut fortement ralentir l'exécution de l'algorithme, et à terme, il est tout-à-fait possible de planter l'application par une utilisation exagérée de mémoire.

5.7 La barre d'outils et divers

Chaque problème présenté dans l'outil de démonstration a sa propre barre d'outils, présentée sur l'onglet permettant de sélectionner le problème. Lorsque l'algorithme est inactif, la barre d'outils se présente sous cette forme :

Le bouton marqué du symbole permet d'obtenir une boîte de dialogue d'information sur le problème sélectionné.

Le bouton *run* permet de lancer l'exécution de l'algorithme. Le bouton *step* permet de lancer l'exécution de l'algorithme au pas à pas. Lorsque l'exécution est lancée, certains boutons sont modifiés et (dés)activés :



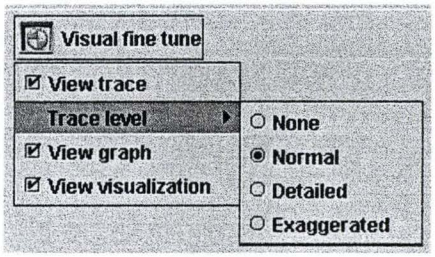
Le bouton *run* devient le bouton *stop* permettant d'arrêter manuellement l'exécution. Le bouton *pause* est activé, il permet de geler momentanément l'exécution de l'algorithme. Le bouton *step* est toujours disponible pour passer en mode pas à pas. Si l'on sélectionne le mode pas à pas, l'algorithme effectue une itération puis passe en mode "pause". En mode pause, la barre d'outils a cet aspect :

On a alors la possibilité de stopper l'exécution manuellement, la faire continuer (*unpause*) ou d'effectuer une itération en mode pas à pas.

Le bouton *reset stats* n'est disponible qu'en mode inactif. Par défaut, le résultat de la dernière exécution de l'algorithme reste affiché après la fin de l'exécution. Ce bouton permet de réinitialiser ces statistiques en remplaçant la population par une population aléatoire et en remettant les compteurs à zéro, et en effaçant le graphe de statistiques.

Le bouton *save results* n'est également disponible qu'en mode inactif, et permet de sauvegarder les résultats de la dernière exécution. Dans la version actuelle, ce bouton sauvegarde les trois séries illustrées sur le graphe de statistiques (meilleure viabilité de la génération, meilleure viabilité trouvée, et viabilité moyenne de la génération) sous forme de suites de valeurs en virgule flottante séparées par des tabulations, facilement importables dans un tableur.

Le bouton *visual fine tune* permet d'activer et de désactiver des fonctions d'affichage afin d'optimiser légèrement le traitement de l'algorithme en le délivrant des tâches annexes d'affichage.



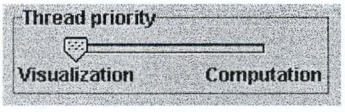
On peut ainsi désactiver l'affichage du graphe de statistiques et de la visualisation fournie par l'implémenteur du problème. On peut également désactiver le rapport d'exécution, ce qui en règle générale est assez intéressant en termes de performances (si l'on n'a pas besoin des détails fournis par le rapport).

L'activation et la désactivation des fonctions d'affichage peut s'effectuer à volonté pendant l'exécution de l'algorithme, ce qui fait que l'on peut tout désactiver pour améliorer le rendement et activer en cours de route l'affichage que l'on désire vérifier. Il faut cependant être prudent avec la désactivation du rapport d'exécution, parce que le rapport arrête de se compléter au moment où il est désactivé, et recommence au moment où il est réactivé, et il y a donc dans ce cas un trou dans le rapport.

Ce même menu propose également la configuration fine du niveau de rapport, que nous avons déjà décrite à la section précédente.

La barre d'outil propose généralement, à la droite des boutons que nous venons de décrire, des contrôles permettant de configurer le problème. Ces boutons sont différents pour chaque problème.

Enfin, une option de configuration se trouve au bas de l'onglet de configuration des paramètres de l'algorithme génétique :



Ce contrôle à glissière (*slider*) permet de gérer finement la priorité du thread d'exécution. S'il est positionné totalement à droite, l'interface sera très peu responsive parce que le thread d'exécution de l'algorithme recevra une très haute priorité. En règle générale il n'est pas nécessaire de modifier cette configuration mais pour lancer une exécution sans supervision jusqu'à ce qu'une condition de succès donnée soit atteinte, on peut donner plus de temps machine à l'exécution en utilisant ce paramètre.

Chapitre 6

Expérimentation

6.1 Introduction

Cette section d'expérimentation du framework et de l'outil de démonstration va nous permettre de réaliser une étude de l'influence des différents éléments de l'algorithme génétique (implémentation, effectif de la population, opérateurs, stratégies de sélection, etc.) sur son efficacité.

Par la même occasion, cette étude nous permettra de valider le framework lui-même, par l'observation de son efficacité sur divers problèmes, dans l'absolu et en comparaison d'autres implémentations.

Note Pour référence, toutes les expérimentations décrites dans ce chapitre ont été effectuées sur un ordinateur équipé d'un processeur AMD Duron 1,2 GHz et de 512 MO de mémoire vive, utilisant le système d'exploitation Microsoft Windows 2000.

6.2 Optimalisation d'une chaîne binaire

6.2.1 Description du problème

Ce problème trivial, déjà décrit à la section 4.2.9, *implémentation d'un problème*, et consistant à maximiser ou minimiser le nombre de bits à "1" dans une chaîne binaire, est un banc-test intéressant pour les algorithmes génétiques. L'espace des solutions, comme on l'a vu, est très grand (2^n où n est la taille de la chaîne binaire), et la solution à trouver est unique, donc la probabilité de tomber "par hasard" sur cette solution est de $1/2^n$. Cette probabilité diminue donc de manière exponentielle avec n.

Ce problème est intéressant parce que la solution en est connue, mais l'algorithme lui-même ne peut la deviner *a priori*. En effet, la population de base est générée aléatoirement avec une répartition équitable de 0 et de 1 (ce qui est garanti par la fonction de génération de nombres pseudo-aléatoire de la classe *java.util.Random*, initialisée sur l'horloge interne de la machine), et la population de départ a une probabilité minime de s'approcher de la solution.

L'expérimentation confirme d'ailleurs la répartition équitable des 1 et des 0 puisque la viabilité moyenne de la population à la première itération (la viabilité d'un individu dans ce problème est égale au nombre de 1 présents dans la chaîne

de bits) est toujours de 32 (64/2) ou une valeur proche, ce qui indique que la moitié environ des bits de la population est à 1.

Comme on pourrait penser que la solution cherchée influence la convergence de l'algorithme par sa structure particulière (ensemble de bits à 1), nous allons également confronter les mesures avec des contre-exemples nous permettant de valider le problème lui-même.

6.2.2 Influence de l'effectif de la population sur l'efficacité de l'algorithme

Nous allons mesurer l'influence de la quantité d'individus (l'effectif de la population) sur le rendement de l'algorithme. Le rendement sera mesuré sur le nombre de générations nécessaires à la découverte de l'optimum, et sur le temps nécessaire à cette découverte. En effet, l'effectif influence le temps nécessaire pour effectuer une itération de l'algorithme, et la durée d'exécution est dans ce cas une mesure au moins aussi utile que le nombre de générations.

Comme les algorithmes génétiques fonctionnent selon un schéma non-déterministe, chaque exécution à paramètres identiques est différente. On ne peut donc pas tirer de conclusions sur une exécution unique, qui peut avoir été particulièrement heureuse ou malheureuse. Afin de donner des mesures réalistes dans des conditions d'expérimentation praticables en termes de temps, nous allons nous baser sur des séries de dix expérimentations pour chaque cas de figure.

Afin de limiter les effets pervers sur cette mesure consacrée spécifiquement à l'effectif, nous allons utiliser les opérateurs les plus simples et les plus classiques, le crossover à un point, avec une probabilité d'application de $0,6$ (utilisée le plus souvent dans les exemples de Goldberg), et la mutation, avec une probabilité de $0,001$ (idem).

Pour la stratégie de sélection, nous utiliserons la *stratégie du tournoi*. Nous aurions pu choisir la *stratégie proportionnelle à la viabilité*, plus classique, mais celle-ci implique des procédures assez lourdes : totalisation des viabilités de la population, totalisation partielle à chaque sélection, etc. Bien qu'en moyenne ces procédures devraient représenter la même charge pour chaque expérimentation elles représentent un risque de fausser les mesures. La stratégie du tournoi, par contre, représente une charge linéaire (sélection de deux individus un nombre donné de fois, directement proportionnel à l'effectif de la population).

Les optimisations particulières sont désactivées (élitisme et kick), et on utilise une chaîne de 64 bits (équivalente au codage d'un entier de type long).

Les tables 6.1 et 6.2 détaillent les résultats des mesures effectuées, en termes de nombre de générations et de temps d'exécution.

M indique la moyenne des mesures pour un effectif donné, et σ est l'écart-type correspondant. Notons que le temps absolu d'exécution reste influencé par des facteurs extérieurs à l'algorithme : temps de latence du thread d'exécution, rafraîchissement de l'écran, mise à jour des statistiques. Comme le but ici n'est pas la vitesse absolue mais le résultat relatif des différentes expérimentations, ce n'est pas un problème.

La figure 6.1 donne une représentation graphique de ces mesures (la durée est exprimée en dixièmes de seconde pour une meilleure lisibilité).

La mesure de l'écart-type permet de voir que l'exécution est plus régulière, c'est à dire paraît plus déterministe, à mesure que la population grandit, parce

n	50	100	200	300	400	500	1000
1	195	98	84	71	65	48	54
2	311	140	86	67	69	63	44
3	205	119	86	79	48	62	48
4	193	165	84	70	58	59	48
5	223	166	75	76	61	67	53
6	193	126	105	68	58	67	51
7	284	143	82	64	72	42	44
8	128	192	72	64	67	56	43
9	167	107	65	89	62	51	51
10	233	130	94	71	60	58	50
M	213,2	138,6	83,3	71,9	62	57,3	48,6
σ	50,67	27,54	10,65	7,27	6,45	7,8	3,69

TAB. 6.1 – Mesure d'efficacité (nombre de générations) en fonction de l'effectif de la population (Crossover simple de $P_X = 0,6$ et mutation de $P_M = 0,001$)

n	50	100	200	300	400	500	1000
1	10	5	5	4	4	3	5
2	16	8	5	4	5	5	5
3	10	6	5	5	3	5	4
4	10	9	5	4	4	5	5
5	12	9	5	5	5	4	5
6	10	7	6	4	4	5	5
7	15	8	5	4	5	4	4
8	6	11	4	4	5	4	4
9	9	6	4	6	4	4	5
10	12	7	6	4	4	6	4
M	11	7,6	5	4,4	4,3	4,5	4,6
σ	2,76	1,69	0,63	0,66	0,64	0,81	0,49

TAB. 6.2 – Mesure d'efficacité (temps d'exécution) en fonction de l'effectif de la population (Crossover simple de $P_X = 0,6$ et mutation de $P_M = 0,001$)

que le nombre de générations nécessaire pour découvrir l'optimum est toujours proche de la même valeur dans ce cas[1]. A partir d'une population de 200 individus, l'écart-type sur la durée d'exécution ne varie cependant plus d'une manière notable.

De manière similaire, le nombre de générations nécessaire pour localiser l'optimum ne diminue de façon importante que tant que la population se compose de moins de 200 individus. A partir de ce point, la diminution reste constante, mais est nettement plus faible.

Cependant, une mesure utile est le temps absolu nécessaire pour atteindre ce point, puisque l'effectif de la population influence le temps nécessaire au traitement de chaque génération. On constate que le temps d'exécution absolu

[1] L'écart-type mesure l'écart moyen des individus à la moyenne de la population. Si cette valeur est faible, les individus sont tous, ou la plupart, proches de la moyenne.

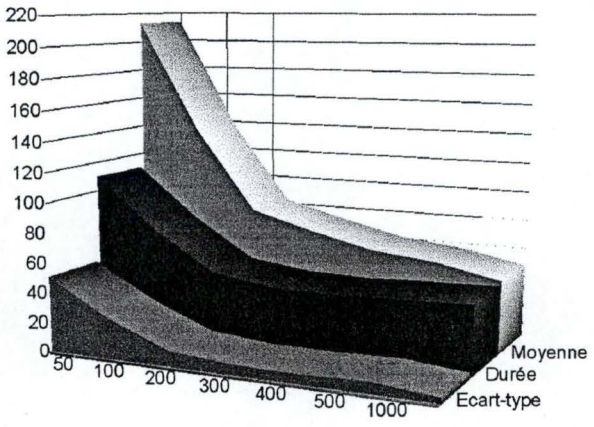


FIG. 6.1 – Moyenne et écart-type des expérimentations

connaît un minimum entre un effectif de 200 à 400 individus, puis se met à remonter sensiblement. En effet, un calcul rapide de rapport entre le temps absolu et le nombre de générations montre une évolution du temps d'exécution par génération de 60 millisecondes pour un effectif de 50 individus à 90 millisecondes pour 1000 individus. Le gain d'efficacité dû à l'augmentation de l'effectif de la population est compensé par l'augmentation du temps nécessaire pour traiter une génération.

Une population de 300 individus permettra donc, pour ce problème et pour cette taille de chaîne binaire, d'obtenir un résultat régulier (ce qu'indique un écart-type faible) pour un temps d'exécution et un nombre de générations optimal.

Ce résultat nous permet aussi de tirer des conclusions préliminaires sur l'efficacité de l'algorithme. En effet, si l'on considère une moyenne inférieure à 80 générations nécessaires à la localisation de l'optimum pour une population de 300 individus, on se rend compte qu'on a envisagé $80 * 300 = 24000 \approx 2 \cdot 10^4$ individus lors de l'exécution de l'algorithme[2], à comparer avec les $2^{64} \approx 2 \cdot 10^{19}$ combinaisons possibles, soit un rapport proche de 10^{15}, qui nous permet de constater que l'efficacité de l'algorithme est largement supérieure à ce qu'aurait obtenu une recherche exhaustive ou aléatoire.

6.2.3 Validation du problème

Afin de valider le fait que la trivialité de la solution n'influence pas la convergence de l'algorithme, nous allons reproduire l'un des tests précédents en modifiant le but recherché. Au lieu de rechercher une chaîne composée de 1, nous allons rechercher une chaîne composée alternativement de 1 et de 0.

[2] Et sans doute moins, parce que l'algorithme a probablement généré occasionnellement des copies conformes d'une génération à l'autre.

Nous allons prendre pour référence une configuration simple comme celle utilisée pour l'étude de l'effet de l'effectif sur le problème (crossover à un point en probabilité 0,6, mutation en probabilité 0,001, stratégie du tournoi, pas d'optimisations), en fixant l'effectif de la population sur la valeur optimale trouvée dans ce test, à savoir 300 individus.

La table 6.3 montre les résultats de l'expérimentation de cette recherche de chaîne "panachée" de 1 et de 0.

	g	t
1	58	4
2	76	5
3	75	5
4	80	6
5	87	6
6	68	5
7	78	5
8	88	6
9	75	5
10	75	5
M	76	5,2
σ	7,84	0,57

TAB. 6.3 – Résultats du problème de la chaîne binaire "panachée"

On constate que les résultats sont tout-à-fait cohérents avec ceux obtenus dans des conditions semblables par l'expérimentation de base (71,9 de moyenne contre 76 ici, et un écart-type de 7,27 pour 7,84 ici).

Nous allons faire une expérimentation supplémentaire pour achever la validation du problème. Au lieu de rechercher une valeur de structure donnée, nous allons demander au système de nous générer une chaîne aléatoire, que nous allons essayer de retrouver. Cette nouvelle expérimentation donne un résultat encore équivalent, avec une moyenne de 71,3 générations et un écart-type de 9,04, un peu plus élevé mais restant suffisamment proche.

Ces tests nous permettent de déduire que la convergence n'est pas influencée par la structure particulière du résultat final. Comme l'expérimentation a démontré également que la population de base était constituée de manière équitable de bits à 1 et à 0, on peut en effet considérer le problème comme n'étant pas "biaisé" et met donc à l'épreuve le moteur de résolution de manière indépendante.

6.2.4 Influence de la configuration du crossover

Nous allons à présent fixer l'effectif de la population à la valeur idéale trouvée à la section 6.2.2[3], soit 300 individus. Nous allons étudier les variations engendrées par une modification des paramètres des opérateurs génétiques.

[3]Cette valeur n'est peut-être pas idéale pour toutes les variantes que nous allons tester ; cependant, nous accepterons l'idée que l'approximation engendrée par ce choix reste dans des limites correctes. Nous ne pouvons nous permettre, dans cette étude limitée, d'étudier toutes les combinaisons de paramètres, d'autant que l'étude statistique demande un grand nombre d'expérimentations de chaque combinaison.

Pour ce test, nous nous limiterons à la stratégie du tournoi, sans optimisation spéciale, comme précédemment. Nous nous intéresserons uniquement au nombre de générations nécessaires pour trouver l'optimum. En effet, le nombre de crossovers ou de mutations effectués a un impact négligeable sur le temps d'exécution à population égale, et l'on peut considérer le temps d'exécution comme directement proportionnel au nombre de générations dans cette hypothèse.

Nous allons fixer de nouveau la probabilité de mutation à 0,001.

La table 6.4 détaille le résultat de l'expérimentation du crossover à un point. En titre est indiquée la probabilité de crossover utilisée pour l'expérimentation (P_X).

P_X	0	0,2	0,4	0,6	0,8	1
1	170	96	78	71	65	56
2	184	96	101	88	58	51
3	186	121	99	58	86	60
4	148	106	80	74	80	40
5	176	92	73	70	58	56
6	181	91	73	70	56	53
7	144	93	90	70	56	81
8	165	105	77	62	66	60
9	167	111	82	71	66	62
10	154	103	91	74	55	63
M	167,5	101,4	84,4	70,8	64,6	58,2
σ	14,13	9,16	9,72	7,48	10,13	9,92

TAB. 6.4 – Influence de la probabilité du crossover simple sur l'efficacité de l'algorithme ($n = 300$, mutation de $P_M = 0,001$)

On constate qu'un maximum de crossover améliore le résultat moyen, pour ce problème. Par contre, la variance est minimale pour une probabilité de crossover de 0,6.

Si le crossover est désactivé (sa probabilité est de zéro), l'algorithme converge toujours d'une manière satisfaisante, mais avec un rendement bien moins bon. La convergence s'effectue donc seulement par le biais de la sélection et de la mutation. Il ne faut cependant pas en tirer de conclusions sur l'utilité du crossover, car ce problème réagit peut-être particulièrement bien à la mutation seule du fait de sa typologie particulière.

Voyons si le crossover à deux points (à détermination de la section de crossover totalement aléatoire) réagit de manière comparable. La table 6.5 détaille une expérimentation identique à la précédente, mais utilisant le crossover à deux points (il n'y a pas de test à probabilité 0, les résultats étant naturellement identiques aux précédents, puisque dans ce cas le crossover n'est jamais appliqué).

On voit que les résultats sont sensiblement meilleurs, ce type de crossover est donc mieux adapté à ce problème.

6.2.5 Influence de la probabilité de mutation

La probabilité de mutation est un facteur un peu différent dans l'exécution de l'algorithme. En effet, sa présence est pratiquement indispensable. Pour ce

P_X	0,2	0,4	0,6	0,8	1
1	99	67	66	51	39
2	109	76	61	67	57
3	90	88	70	48	53
4	68	72	59	41	58
5	82	64	67	51	47
6	76	60	52	53	56
7	110	69	52	51	50
8	89	77	46	73	42
9	101	64	50	51	52
10	83	83	67	61	47
M	90,7	72	59	54,7	50,1
σ	13,24	8,51	8,06	9,03	6,04

TAB. 6.5 – Influence de la probabilité du crossover à deux points sur l'efficacité de l'algorithme ($n = 300$, mutation de $P_M = 0,001$)

problème, l'expérimentation montre que si l'on supprime la mutation, l'algorithme finit presque toujours par stagner, souvent très près de la solution. La raison de ce problème est le "gavage" de la population en individus (et donc en schémas) "presque" optimaux effectué par le crossover et la sélection. Sans le sang neuf apporté par la mutation, l'approche de l'optimum reste limitée (et on verra que le kick fournit un effet similaire d'une manière différente).

Comme l'expérimentation précédente a montré qu'une probabilité de crossover de 1 était favorable dans ce problème, de même que le crossover à deux points, nous allons utiliser cette configuration pour les tests sur la mutation, avec l'effectif de population déterminé précédemment (300) et la stratégie du tournoi.

P_M	0,001	0,003	0,005	0,007	0,01
1	39	55	57	70	98
2	54	51	52	59	72
3	62	55	52	59	96
4	53	45	55	58	97
5	58	48	54	65	92
6	55	55	53	66	86
7	47	55	49	65	90
8	45	38	43	58	74
9	48	48	53	58	95
10	45	44	43	64	70
M	50,6	49,4	51,1	62,2	87
σ	6,62	5,57	4,5	4,09	10,41

TAB. 6.6 – Influence de la probabilité de mutation sur l'efficacité de l'algorithme ($n = 300$, crossover à deux points de $P_X = 1$)

Le problème accepte des taux de mutation entre 0,001 et 0,005 environ. Comme indiqué ci-dessus, un taux de zéro est improductif. L'expérimentation

a également montré que les taux supérieurs à 0,01 sont également improductifs. L'évolution de la recherche de l'optimum montre que la moyenne ne s'améliore plus à partir d'un certain seuil, et si l'optimum est atteint, c'est purement par combinaison aléatoire (ce qui revient à une sorte de recherche aléatoire des environs de l'optimum). La figure 6.2 montre un exemple représentatif de l'exécution de la recherche pour un taux de mutation égal à 0,012. La courbe inférieure du graphe montre la moyenne, la courbe supérieure le meilleur résultat trouvé. On voit clairement que la moyenne stagne (aux environs de 56,9), et que donc seul une combinaison heureuse permet d'atteindre l'optimum.

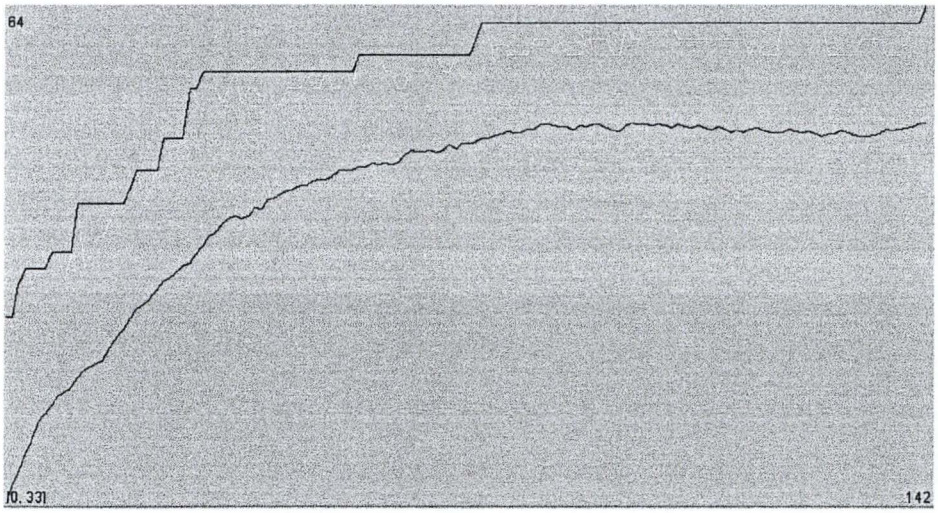


FIG. 6.2 – Effet d'un excès de mutation

A partir d'un taux situé aux environs de 0,015, la mutation gène tellement la convergence que l'optimum est rarement atteint.

6.2.6 Conclusions

Nous n'allons pas poursuivre l'expérimentation sur ce problème, parce que nous avons pu optimiser très finement les paramètres d'exécution, de sorte que les dernières expérimentations trouvaient l'optimum d'une manière presque linéaire, ce qui est totalement atypique de l'exécution d'un algorithme génétique. La raison de ce résultat est évidemment la simplicité du problème. Il serait un peu spécieux de "désoptimiser" le problème afin d'expérimenter d'autres fonctionnalités, nous allons donc nous pencher sur d'autres problèmes un peu plus ardus.

Jusqu'ici, l'expérimentation nous a permis de constater l'effet de l'augmentation de l'effectif de la population. Augmenter l'effectif de la population permet toujours, dans les limites de notre expérimentation, d'améliorer le rendement de l'algorithme en termes de nombre de générations. On note cependant qu'à partir d'un certain point, l'amélioration n'est plus très importante. Par contre, en termes de temps d'exécution, le problème connait un effectif de population optimal, parce que le gain en termes de nombre de générations est compensé par le temps d'exécution individuel de chaque génération.

Nous avons également étudié l'effet de la probabilité de crossover sur le rendement de l'algorithme. Le résultat est cependant atypique en ce sens que le meilleur rendement est obtenu avec une probabilité de crossover de 1. C'est probablement un effet de la simplicité du problème. Nous avons constaté que le crossover à deux points était plus efficace que le crossover simple. C'est un paramètre dépendant du problème traité, en ce sens que l'échange de matériel génétique situé au centre du chromosome est plus rentable que celui situé aux bords.

Enfin, nous avons constaté que la mutation est indispensable à ce problème, son absence amenant une stagnation quasi systématique et irréversible. Nous avons démontré expérimentalement un fait déjà mentionné, à savoir que la mutation doit s'effectuer à probabilité très faible.

6.3 Problème de Steiner

6.3.1 Description du problème

Le problème de Steiner propose de déterminer l'emplacement idéal d'une centrale électrique devant desservir un certain nombre de villes, de manière à minimiser la quantité de câblage électrique utilisée pour relier les villes à la centrale. Le problème suppose que toutes les villes sont reliées directement à la centrale, en d'autres termes, une même ligne ne peut pas desservir deux villes au moyen d'un relais.

Cette implémentation relie les villes à la centrale en ligne droite. Cependant, on peut affiner le problème en associant une pondération à une connection particulière. Cette pondération peut modéliser l'importance de la consommation électrique de la ville (dans ce cas, plus de câbles seront nécessaires, ou des câbles plus épais et plus chers), ou d'une manière simplifiée le surcoût causé par des accidents de terrain obligeant à des frais de contournement.

6.3.2 Notes sur l'implémentation

Ce problème est relativement élémentaire à implémenter. La solution recherchée est la coordonnée (x, y) de l'emplacement de la centrale, et la viabilité de la solution est simplement la somme des distances euclidiennes entre toutes les villes et la centrale, éventuellement corrigée de la pondération. On veut minimiser la viabilité.

Dans cette implémentation, nous utiliserons des coordonnées en nombres entiers. C'est une décision réaliste parce que si l'on considère un codage par entier long (64 bits signé), on peut représenter la surface de la Terre avec une précision de l'ordre du nanomètre (ce qui est clairement plus que nécessaire). Pour la facilité, nous utiliserons également des pondérations entières.

Par contre, le calcul de la somme des distances s'effectuera en virgule flottante. En effet, le calcul de la distance euclidienne fait intervenir une racine carrée dont le résultat, une fois cumulé sur toutes les villes, peut amener une erreur de troncation notable s'il est ramené à un nombre entier.

Nous avons considéré improductif de livrer à l'algorithme un espace de solutions équivalent au domaine d'un type de données particulier, alors que la solution se situe toujours (quelles que soient les pondérations et la topologie

des localisations des villes) entre les villes les plus extrêmes dans les deux dimensions de l'espace. Nous avons donc optimisé l'algorithme en donnant une représentation des gènes correspondant à l'étendue de l'espace dans chacune des dimensions, au bit près. Par exemple, si les villes les plus extrêmes sur la dimension x sont distantes de 300 unités (kilomètres, mètres, millimètres, etc. selon la représentation choisie par l'utilisateur), on codera cette dimension sur 9 bits, de sorte que le domaine sera limité à 512 unités (8 bits aurait été trop court, avec 256 unités, et 10 bits inutilement long). L'implémentation effectue cet ajustement automatiquement en fonction des coordonnées reçues.

6.3.3 Notes sur l'expérimentation

Nous avons utilisé pour cette expérimentation des paramètres de problème provenant de l'outil de démonstration *GAPlayground* disponible *via* Internet sur arieldolan.com. De cette manière, nous nous assurons que le choix des coordonnées des villes est neutre, et que nous avons une référence permettant de valider grossièrement les résultats de l'expérimentation.

Ce fichier de configuration contient un ensemble de 19 villes. Il était exprimé originellement en virgule flottante, nous l'avons modifié pour représenter des valeurs entières.

Notons que le problème de Steiner n'est pas influencé notablement par le nombre de villes qu'il contient : seule la taille de l'espace des solutions augmente la complexité de l'algorithme. L'influence du nombre de villes modifie uniquement le temps de calcul de la viabilité. Augmenter le nombre de villes ne ferait donc que biaiser les mesures de performance de l'algorithme génétique par un facteur qui lui est étranger, à savoir une somme de distances euclidiennes.

Avant de commencer l'expérimentation proprement dite, nous allons tenter de déterminer l'optimum par la force brute, en choisissant un effectif de population extrêmement élevé, et en laissant l'algorithme rechercher la solution pendant un grand nombre de générations. Nous allons également répéter cette expérience un nombre important de fois. Le détail de cette expérimentation préliminaire n'est pas important. L'optimum localisé de cette manière est situé en $(518, 424)$ et possède une viabilité (distances cumulées) de $7062,7974$ unités.

L'implémentation de *GAPlayground*, que nous utilisons à titre comparatif, trouve un optimum en $(500, 420)$ avec une viabilité de $7099,9997$[4]. Ce résultat, un peu moins bon, est cependant cohérent avec celui que nous avons obtenu.

6.3.4 Influence de la stratégie de sélection

Nous allons étudier comment les différentes stratégies de sélection se comportent vis-à-vis de ce problème. L'implémentation nous en propose cinq. La stratégie par choix aléatoire (*random selection*) n'est pertinente que pour étudier l'élitisme, nous ne l'évoquerons que dans la section suivante. Nous allons évoquer séparément les quatre autres.

Pour fixer les idées, nous utiliserons arbitrairement des paramètres de configuration classiques, à savoir, un crossover simple de probabilité de $0,6$, une probabilité de mutation de $0,001$, et une population de 300 individus.

[4]Cette implémentation est programmée pour s'arrêter à une approximation raisonnable de la solution, elle ne peut donc pas s'approcher aussi complètement de la solution optimale.

Nous allons limiter l'exécution à 50 générations et observer comment les stratégies de sélection s'approchent de l'optimum dans cette limite. Pour chaque sélection, nous effectuerons 5 mesures pour obtenir un résultat statistiquement correct. Les résultats détaillés des expérimentations se trouvent à l'annexe B.

Roulette Wheel Selection

Cette stratégie classique proportionnelle à la viabilité présente, comme nous l'avons vu, la limitation de n'accepter que des viabilités exprimées par des nombres positifs[5]. Le problème que nous traitons garantit ce point.

Par ailleurs, cette stratégie n'est pas optimale en termes de traitement parce qu'elle implique un parcours préliminaire de la population pour totaliser les valeurs de viabilité, et ensuite une totalisation partielle à chaque sélection. Comparativement à d'autres stratégies, le traitement est donc relativement lourd. De fait, l'exécution de 50 générations avec une population de 300 individus prend 9 à 10 secondes sur notre machine de référence.

La figure 6.3 montre l'évolution de la viabilité moyenne de la population pour chacun des cinq tests.

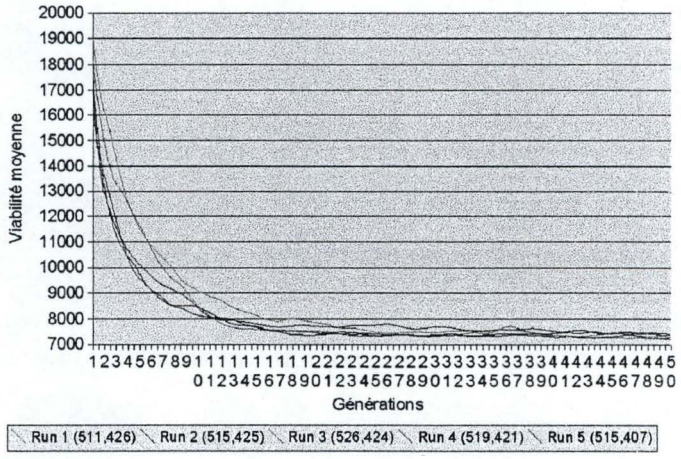


FIG. 6.3 – Viabilités moyennes sur 5 exécutions du problème de Steiner utilisant la stratégie proportionnelle. $P_X = 0,6$, $P_M = 0,001$, $n = 300$.

On constate une convergence rapide de la moyenne de la population. Celle-ci reste relativement éloignée de l'optimum : la moyenne des cinq tests donne une viabilité moyenne au terme des cinquante générations de 7337,95, à comparer avec l'optimum trouvé précédemment, à savoir 7062,7974. C'est assez naturel puisque la population doit conserver une certaine variété (en partie garantie par la mutation).

Le meilleur individu trouvé est assez proche de l'optimum, la meilleure viabilité trouvée étant, sur la moyenne des cinq tests, de 7064,53 (le meilleur des

[5] Naturellement, si la viabilité est exprimée par des nombres négatifs, il est trivial de transposer le problème pour que la contrainte de positivité soit respectée. Par contre, cette stratégie ne peut accepter un mélange de valeurs de viabilités positives et négatives.

cinq tests se place à 7062, 93), ce qui est excellent.

Tournament Selection

La stratégie du tournoi présente l'avantage d'être très légère d'un point de vue traitement informatique, puisqu'elle ne fait que sélectionner aléatoirement (sur base d'une probabilité donnée) un individu parmi deux choisis aléatoirement dans la population. Et en effet, on constate que l'exécution est nettement plus rapide que pour la stratégie proportionnelle, avec un temps d'exécution moyen de 3 secondes dans les paramètres du test (soit environ un tiers du temps moyen de la stratégie proportionnelle).

Cette stratégie peut être affinée au moyen d'un paramètre déterminant l'avantage évolutionnaire de l'individu le plus viable (lors d'un tournoi). Une étude comparative des valeurs de ce paramètre nous emmènerait trop loin, nous nous contenterons de le fixer à une valeur moyenne donnant de bons résultats, à savoir 50% d'avantage.

La figure 6.4 montre l'évolution de la viabilité moyenne de la population pour chacun des cinq tests.

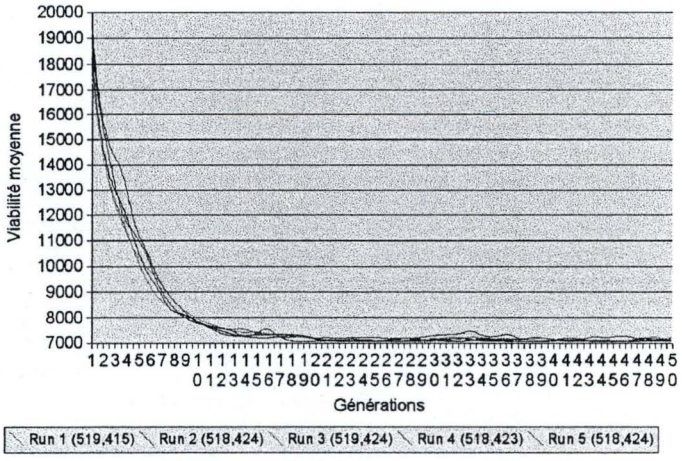


FIG. 6.4 – Viabilités moyennes sur 5 exécutions du problème de Steiner utilisant la stratégie du tournoi. $P_X = 0,6$, $P_M = 0,001$, $n = 300$.

La convergence est excellente, avec une valeur moyenne de la viabilité moyenne sur les 5 tests de 7149, 62. La moyenne de la meilleure viabilité se situe à 7063, 16, ce qui est très proche de l'optimum.

Rank Selection

La stratégie du classement présente le désavantage de forcer le classement des individus par ordre de viabilité, ce qui, dans tous les cas de figure, reste un traitement assez lourd[6]. Le reste du traitement est cependant moins exigeant

[6]Cette version de l'implémentation utilise un simple algorithme de type "tri à bulle" relativement peu performant. D'autres implémentations se sont révélées assez complexes à mettre

en ressources machines, l'individu sélectionné étant localisé par des méthodes mathématiques directes. De fait, à cause du tri, le temps d'exécution du protocole de test varie entre 13 et 16 secondes.

Cette stratégie propose deux variantes, un modèle linéaire et un modèle exponentiel. Nous n'utiliserons cependant pas ce dernier, parce que l'effectif de la population est limité à une soixantaine d'individus, ce qui n'est pas compatible avec notre protocole de tests.

La figure 6.5 montre l'évolution de la viabilité moyenne de la population pour chacun des cinq tests.

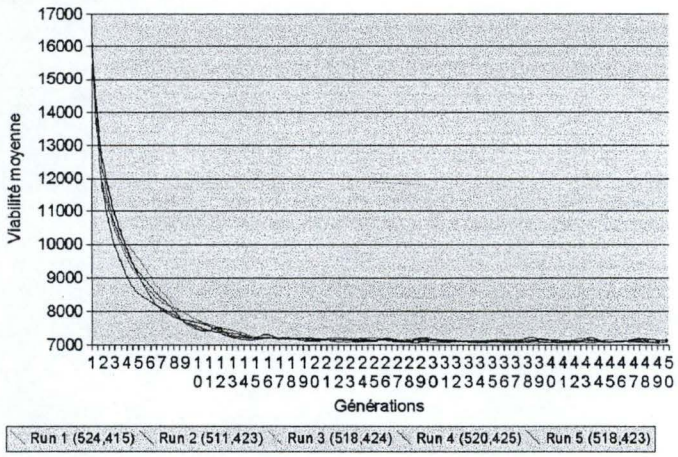


FIG. 6.5 – Viabilités moyennes sur 5 exécutions du problème de Steiner utilisant la stratégie du classement. $P_X = 0,6$, $P_M = 0,001$, $n = 300$.

Le résultat est de nouveau très bon, la moyenne de viabilité obtenue est de 7133,74. La meilleure viabilité est de 7063,41.

Sigma Scaling Selection

La stratégie proportionnelle à l'écart-type présente des limitations comparables à la stratégie proportionnelle, avec en plus, du fait de la complexité des calculs à effectuer, l'obligation de transposer la viabilité en un nombre à virgule flottante. Ceci ne pose pas de difficultés pour le problème de Steiner.

Pour des raisons probablement liées à l'implémentation, cette stratégie est plus performante que la stratégie proportionnelle, bien qu'elle nécessite un traitement similaire. En effet, le temps d'exécution a été de 3 secondes pour chacun des cinq tests.

La figure 6.6 montre l'évolution de la viabilité moyenne de la population pour chacun des cinq tests.

La courbe d'évolution de la moyenne de viabilité de la population est sensiblement plus linéaire que dans les tests précédents, ce qui est cohérent avec le modèle de cette stratégie de sélection. Probablement pour cette même raison,

en oeuvre dans ce contexte. Une prochaine version devrait remédier à cette difficulté, en implémentant un *quicksort* par exemple.

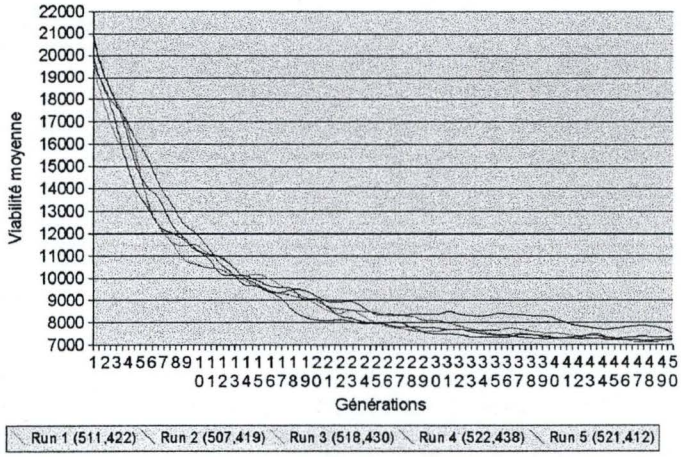


FIG. 6.6 – Viabilités moyennes sur 5 exécutions du problème de Steiner utilisant la stratégie du sigma. $P_X = 0,6$, $P_M = 0,001$, $n = 300$.

le résultat final est un peu moins bon : 7312,04 pour la moyenne, et 7065,16 pour la meilleure viabilité trouvée.

Comparaisons

Comparons d'abord les résultats absolus. La table 6.7 ci-dessous dresse un résumé des mesures globales.

	Roulette	Tournament	Rank	Sigma
Durée approx. (sec.)	9	3	14	3
Viabilité moyenne	7337,95	7149,62	7133,74	7312,04
Meilleure viabilité	7064,53	7063,16	7063,41	7065,16

TAB. 6.7 – Comparaison des résultats moyens des expérimentations de diverses stratégies de sélection sur le problème de Steiner.

On voit que la stratégie du tournoi se comporte particulièrement bien pour la résolution de ce problème. C'est un choix d'autant plus intéressant que la stratégie requiert très peu de ressources systèmes, et ne connaît pas de limitations en ce qui concerne le domaine des valeurs de la viabilité.

On peut encore tirer quelques conclusions d'une vue comparative de l'évolution de la viabilité moyenne pour chaque stratégie de sélection, que l'on peut voir sur la figure 6.7. Chaque courbe est la moyenne des cinq expérimentations effectuées pour chaque stratégie.

On constate que la stratégie proportionnelle à la viabilité (*roulette*) converge vite mais sa courbe de progression s'aplanit relativement loin de l'optimum. C'est tout à fait cohérent avec le reproche fait habituellement à cette stratégie, à savoir de ne pas gérer la pression de sélection de façon optimale.

La stratégie du classement (*rank*) converge très rapidement, puis a quelques

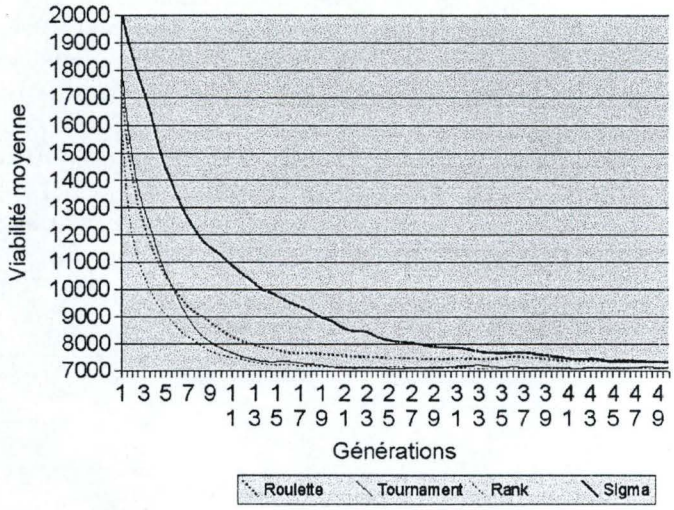


FIG. 6.7 – Comparaison de l'évolution des viabilités moyennes pour les différentes stratégies de sélection.

difficultés à s'approcher plus près de l'optimum. Ce résultat est un peu inattendu de la part de cette stratégie de sélection qui devrait montrer une courbe relativement plate, du fait que la sélection n'est pas proportionnelle à la viabilité mais au classement des individus. On peut supposer qu'une convergence assez linéaire mais trop rapide a gavé la population en individus presque optimaux qui ont rendu la convergence finale plus difficile.

La stratégie proportionnelle à l'écart-type (*sigma*) montre typiquement une courbe plus aplatie que les autres stratégies. C'est la seule stratégie qui n'a pas achevé sa convergence au bout des cinquante générations prévues, ce qui explique son résultat moins bon que celui de deux autres stratégies. L'approche plus linéaire de l'optimum peut cependant se révéler intéressante lorsque l'on traite des problèmes plus complexes.

Enfin, la stratégie du tournoi (*tournament*) propose un choix intéressant à plusieurs points de vue : contraintes minimales sur la représentation du problème, exigences de traitement minimales et excellentes performances en termes d'approche de l'optimum et de temps d'exécution. L'expérimentation semble montrer que son modèle de convergence est meilleur que celui de la stratégie proportionnelle (la convergence initiale est équivalente mais l'approche asymptotique est plus proche de l'optimum) et de la stratégie du classement (convergence moins rapide mais approche finale équivalente).

6.3.5 L'élitisme

Nous allons succinctement étudier l'élitisme en comparant les résultats d'un des tests ci-dessus (nous avons sélectionné arbitrairement le test de la stratégie du tournoi) avec le même test amélioré par une certaine quantité d'élitisme. Il serait intéressant en soi de faire une étude de différentes valeurs d'élitisme, mais malheureusement, ça nous amènerait trop loin. Nous allons

donc nous contenter de voir l'effet d'une valeur d'élitisme assez moyenne, à savoir que nous sélectionnerons une élite représentant 10% de la population, choisis aléatoirement parmi 10% de la population de départ (il peut donc y avoir, aléatoirement, plusieurs copies d'un même individu). Les 90% restants de la population sont complétés par la stratégie de sélection choisie.

Nous avons de nouveau effectué cinq expérimentations pour diminuer l'influence éventuelle d'un effet probabiliste. Le détail de chacune de ces expérimentations importe peu (les résultats bruts sont présentés à l'annexe B), comparons simplement la moyenne des cinq expérimentations avec la moyenne correspondante pour la stratégie du tournoi sans élitisme (figure 6.8).

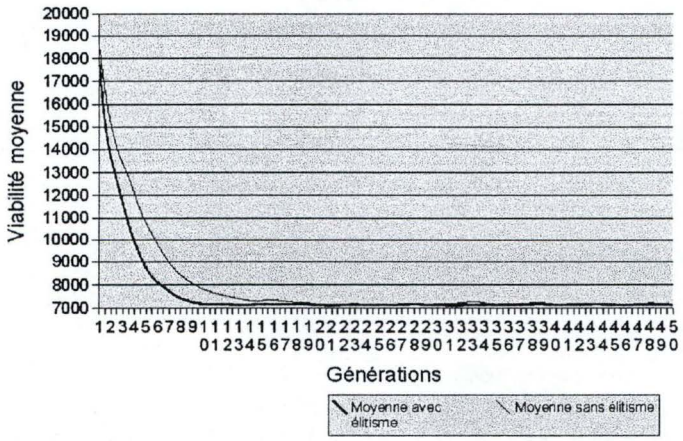


FIG. 6.8 – Comparaison de la moyenne des viabilités sur la stratégie du tournoi avec et sans élitisme. $P_X = 0,6$, $P_M = 0,001$, n=300.

On remarque que la convergence est plus rapide, mais que pour autant le résultat reste tout aussi proche de l'optimum. Le fait est que l'élitisme augmente la pression de sélection d'une manière constante au cours de l'exécution de l'algorithme.

L'élitisme comme stratégie de sélection

L'efficacité de l'élitisme peut être démontrée facilement en la testant indépendamment d'une autre stratégie de sélection, si on la couple à une sélection purement aléatoire. On vérifie facilement qu'une telle sélection purement aléatoire ne permettra pas au problème de converger, et qu'en conséquence les bons résultats du test "purement élitiste" sont attribuables uniquement à l'élitisme et non à un effet quelconque de la sélection aléatoire.

La figure 6.9 montre l'évolution de la moyenne de viabilité pour chacun des cinq tests.

On voit que dans ces conditions, l'exécution est un peu moins déterministe : les courbes montrent des différences sensiblement plus marquées que lors les tests précédents. Cependant, la convergence est toujours tout à fait visible. La viabilité moyenne finale est un peu moins bonne (7587, 2), mais le meilleur résultat

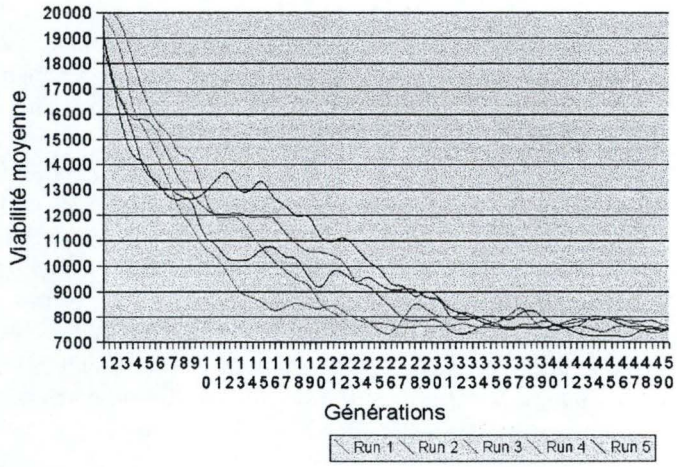


FIG. 6.9 – Comparaison de la moyenne des viabilités sur cinq tests de la stratégie aléatoire avec élitisme. $P_X = 0,6$, $P_M = 0,001$, n=300.

trouvé, du fait de la variance aléatoire des courbes, est plutôt bon (7063,71 en moyenne, avec trois tests ayant amené un meilleur résultat de 7062,8).

La figure 6.10 montre une comparaison du résultat moyen pour la stratégie du tournoi et la stratégie aléatoire utilisant de l'élitisme.

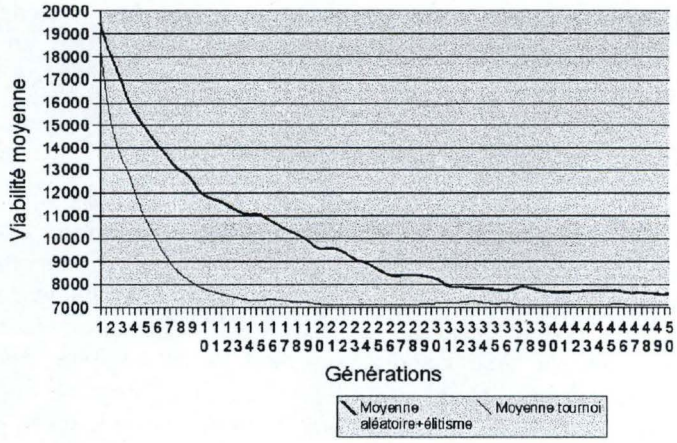


FIG. 6.10 – Comparaison de la moyenne des viabilités pour le test de la stratégie du tournoi et de la stratégie aléatoire avec élitisme. $P_X = 0,6$, $P_M = 0,001$, $n = 300$.

On voit que l'élitisme seul permet une convergence très correcte. Ce n'est pas le but poursuivi, mais on pourrait même envisager cette solution comme alternative aux autres stratégies de sélection dans des situations réelles.

6.3.6 Conclusions

On voit que les défauts de la stratégie proportionnelle à la viabilité ne se limitent pas aux contraintes d'implémentation, mais également à sa performance en termes de vitesse d'exécution et d'approche de l'optimum. Son utilité paraît donc contestable.

Ceci dit, les différences sont assez faibles sur un problème simple comme celui que nous avons traité. On peut cependant tirer certaines conclusions, même si un problème n'est pas l'autre et que ces conclusions devraient être revalidées à chaque implémentation. Le choix le plus sûr est probablement, d'après ces résultats, celui de la stratégie du tournoi, dont les contraintes sont pratiquement inexistantes tout en donnant un niveau de performance supérieur.

On peut enfin supposer, en extrapolant les tests effectués, que la stratégie proportionnelle à l'écart-type propose un modèle de convergence bien équilibré permettant une meilleure convergence à long terme.

6.4 Sac de contrebandier

6.4.1 Description du problème

Le problème du sac de contrebandier (*smuggler's knapsack*) est un problème d'ordonnancement classique. Un contrebandier dispose d'un sac à dos d'une capacité donnée (exprimée en poids maximum). Il dispose d'une série d'items qu'il peut passer en fraude. Chaque item a un poids et un rendement (prix de vente). Il faut maximiser le rendement des items pris dans le sac, en tenant compte de la capacité du sac, qui ne peut être dépassée.

Nous avons implémenté la variante simple de ce problème, où chaque item ne peut être dans le sac qu'une seule fois au maximum (donc zéro ou une fois). En toute généralité le contrebandier pourrait disposer d'un nombre donné ou d'un stock illimité de chaque objet. Ce n'est pas le cas ici.

6.4.2 Notes sur l'implémentation

Les items sont lus d'un fichier de configuration, chaque item est représenté par un poids et un rendement, exprimés par des nombres entiers positifs.

Comme chaque item peut être présent zéro ou une fois dans le sac, le problème est représenté par un gène unique, une chaîne de bits dont chaque bit représente un des items. Si le bit est à 1, l'item est dans le sac, sinon pas. La viabilité est donc la somme des rendements des items dont le bit est à 1.

La seule difficulté de ce problème est la limite de poids imposée. Un optimum proposant un poids total supérieur même d'une seule unité au poids maximum autorisé ne peut pas être pris en compte. Par contre, on ne peut pas refuser absolument ces solutions : d'abord, l'algorithme génétique n'a aucun moyen de les ignorer, ensuite, ils peuvent présenter des schémas utiles pour l'optimum qui se situe dans les limites de poids.

Pour régler cette difficulté, nous avons fait en sorte que la viabilité des solutions, dont le poids est supérieur à la limite, soit inférieure à celle de toutes les solutions se situant dans les limites de poids. Pour malgré tout pouvoir donner une chance aux schémas présents dans ces solutions, nous avons fait en sorte que plus le poids devient grand (au-dessus de la limite), plus la viabilité devient

faible. Cette stratégie proposera donc toujours une solution dans les limites de poids (puisque toutes les autres sont moins bonnes), sauf si c'est impossible, ce qui ne peut se produire que si tous les items ont un poids individuel supérieur à la capacité du sac.

6.4.3 Notes sur l'expérimentation

Nous avons à nouveau utilisé un fichier de paramètres de *GAPlayground*. Celui-ci propose 50 items de poids et rendements quelconques. Les poids et les rendements sont exprimés par des nombres entiers positifs, ce qui est cohérent avec notre implémentation. Le poids maximum autorisé est fixé à 625 unités.

Nous avons de nouveau effectué une expérimentation par la force brute, qui nous a délivré un rendement optimum de 966 unités pour un poids de 622 unités. *GAPlayground* a trouvé, après plusieurs expérimentations, un rendement optimum de 958 unités pour un poids de 619 unités, ce qui est un peu moins bon mais cohérent avec notre résultat.

6.4.4 Influence du *kick*

Nous allons utiliser ce problème sensiblement plus complexe pour étudier l'utilité du *kick*. Pour mémoire, la stratégie du *kick* consiste à surveiller la convergence et, si certaines conditions de stagnation sont réunies (si la moyenne de viabilité de la population ne s'est pas améliorée au bout d'un nombre donné de générations), re-diversifier la population en remplaçant une proportion donnée des individus par des individus nouvellement générés au hasard. Les individus à remplacer sont sélectionnés aléatoirement et indépendamment de leur valeur de viabilité. Ceci permet d'introduire de nouveaux schémas dans une population trop uniforme, typiquement en fin de traitement de l'algorithme.

Nous allons comparer les résultats d'une exécution de ce problème avec des paramètres classiques, à savoir la stratégie du tournoi, un crossover à deux points de probabilité 0,6, et une probabilité de mutation de 0,001. Nous fixerons l'effectif de la population à une valeur assez moyenne, soit 300 individus. Enfin, nous n'utiliserons pas d'élitisme pour ne pas influencer la résolution du problème par une optimisation particulière.

Nous allons limiter le nombre de générations à 500 pour avoir un point de référence où comparer les exécutions. En effet, laisser l'algorithme atteindre l'optimum exact (tout au moins celui déterminé par la force brute) n'est pas pertinent ici, parce que la complexité du problème fait que l'optimum est approché assez facilement mais difficilement atteint. En conséquence, le nombre de générations nécessaire pour atteindre l'optimum exact varie fortement d'une expérimentation à l'autre, et de manière tout à fait aléatoire, qui rend une mesure statistique peu réaliste.

La table 6.8 montre les résultats bruts de cette expérimentation.

Les résultats sont assez comparables, les expérimentations sans kick obtiennent un résultat légèrement inférieur aux expérimentations avec kick. Le point important est cependant que les résultats sans kick sont obtenus précocement (à la 110e génération en moyenne), ce qui montre qu'atteint un certain niveau proche de l'optimum, la convergence stagne de manière souvent irréversible. Par contre, le kick obtient son meilleur résultat dans les dernières

	Sans kick					Avec kick				
	R1	R2	R3	R4	R5	R1	R2	R3	R4	R5
Revenu	943	953	947	947	942	949	937	945	957	954
Génération	47	161	95	166	82	497	93	397	304	499
Kicks	N/A	N/A	N/A	N/A	N/A	4	0	3	2	4
Durée	34	34	34	33	33	29	32	34	30	21

TAB. 6.8 – Résultats de cinq expérimentations du problème du contrebandier avec et sans *kick*. $n = 300$, $P_X = 0,6$, $P_M = 0,001$.

générations (à la 358e en moyenne), parce que la re-diversification de la population par le kick permet d'utiliser des générations qui seraient sinon perdues dans la stagnation, à tenter la chance en explorant de nouvelles portions de l'espace des solutions.

Un choix raisonnable de paramètres de kick, en particulier la limite de stagnation, permet de s'assurer qu'il n'aura pas un effet disruptif, car dans la grande majorité des cas, un problème qui a stagné pendant une centaine de générations a relativement peu de chances d'améliorer encore sa solution, et le reste des itérations sera probablement improductif.

6.4.5 Conclusions

Le *kick* n'est sûrement pas la panacée, mais il permet de mettre à profit des itérations de l'algorithme qui, sinon, auraient été improductives. Son effet reste fortement aléatoire, mais il est important de noter qu'il n'influence pas négativement l'efficacité de l'algorithme (si on choisit une limite de stagnation raisonnable), ni d'ailleurs le temps d'exécution global, comme on le constate dans ce test. C'est donc une aide à envisager pour la résolution de problèmes ardus devant se rapprocher aussi près que possible de l'optimum.

6.5 Voyageur de commerce

6.5.1 Description du problème

Le problème classique du voyageur de commerce (*Traveling Salesman Problem* ou TSP) consiste, en toute généralité, à trouver un chemin ou un circuit dans un graphe pondéré en visitant chaque sommet une et une seule fois, tout en minimisant le poids total des arêtes traversées. En d'autres termes, il revient à rechercher un chemin ou un circuit hamiltonien minimum. Informellement, il s'agit de minimiser le coût du trajet d'un voyageur de commerce qui doit visiter un nombre donné de villes.

Ce problème connaît quelques variantes. Il peut être *symétrique* ou *asymétrique*, selon que le graphe est orienté ou pas. Il peut être *cyclique*, c'est à dire que le chemin doit aboutir au sommet de départ. Le point de départ peut être forcé ou pas.

La complexité du problème du voyageur de commerce est notoire, et c'est devenu un problème de référence qu'on a tenté de résoudre par des méthodes très diverses. C'est donc un banc-test intéressant pour notre framework, même

si nous n'avons pas l'ambition d'entrer en compétition avec les meilleures techniques de résolution de ce problème.

6.5.2 Notes sur l'implémentation

Cette implémentation considère uniquement des problèmes symétriques (graphes non orientés), et utilise un modèle simplifié qui ne permet pas de déterminer à volonté la pondération des arêtes du graphe, mais utilise les distances euclidiennes entre les sommets.

Un paramètre détermine si le chemin doit être cyclique ou pas. Ceci n'influence que le calcul de la viabilité, qui doit simplement ajouter une arête dans le calcul du poids total.

Un autre paramètre permet de forcer le point de départ. Celui-ci sera toujours le premier sommet dans la liste que l'algorithme reçoit. Le calcul de la viabilité est modifié pour extraire le premier sommet : celui-ci est pris comme point de départ d'office, et est ignoré lorsqu'il apparaît dans la liste des sommets.

La liste des sommets est en coordonnées exprimées en virgule flottante (de type *double*). La liste est chargée d'un fichier utilisant le format des fichiers de définition de la *TSPLib* (voir ci-dessous). Ceci implique que la liste de coordonnées soit préfixée de la mention *NODE_COORD_SECTION* sur une ligne séparée, que les coordonnées soient de la forme *I X Y* où I est l'index de la coordonnée, X la coordonnée x, Y la coordonnée Y. Les coordonnées en type entier sont transposées en valeurs de type *double*. La liste de coordonnées doit en principe être postfixée par *EOF* sur une ligne séparée.

Le fichier peut être lu directement même s'il est compressé par GZIP.

6.5.3 Notes sur l'expérimentation

Une source intéressante centralisant des problèmes divers est la *TSPLib* de l'université d'Heidelberg en Allemagne, que l'on peut trouver sur http ://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/index.html. Malheureusement, beaucoup de problèmes proposés utilisent une pondération d'arêtes, ou utilisent une fonction spéciale de calcul de distances, ce qui rend leurs résultats incomparables avec notre expérimentation. En fin de cette section, nous allons décrire les résultats de tests effectués sur certains de ces problèmes utilisant des distances euclidiennes, d'une manière cohérente avec notre implémentation.

L'un de ces problèmes est heureusement traité par *GAPlayground*, avec des résultats comparables aux nôtres. Il s'agit du problème assez classique des 48 capitales d'états américains "métropolitains"[7]. Nous utiliserons donc ce problème, qui en plus de permettre une validation de nos résultats par comparaison avec une autre application, a l'avantage d'être d'une taille conséquente mais pas exagérée.

Nous allons encore faire une estimation de l'optimum par la force brute. *GAPlayground* utilise une implémentation non cyclique, nous allons donc commencer par valider l'implémentation par une expérimentation non cyclique. Pour les tests ultérieurs, nous utiliserons une expérimentation cyclique, plus classique.

[7] Dans la *TSPLib*, ce problème utilise une fonction de calcul spéciale qui n'est pas compatible avec notre méthode de calcul, c'est pourquoi nous ne pourrons pas comparer les résultats de ce test précis avec le meilleur optimum connu.

Le test préliminaire non-cyclique donne un meilleur parcours de 31951 unités au bout de 2000 générations ($n = 2000$, stratégie du tournoi avec 10% d'élitisme, inversion à deux points à $P_I = 0,6$). *GAPlayground* trouve un meilleur parcours de 32280 unités après 10000 générations, ce qui est un peu moins bon mais cohérent avec notre résultat. La figure 6.11 montre une illustration du meilleur trajet trouvé.

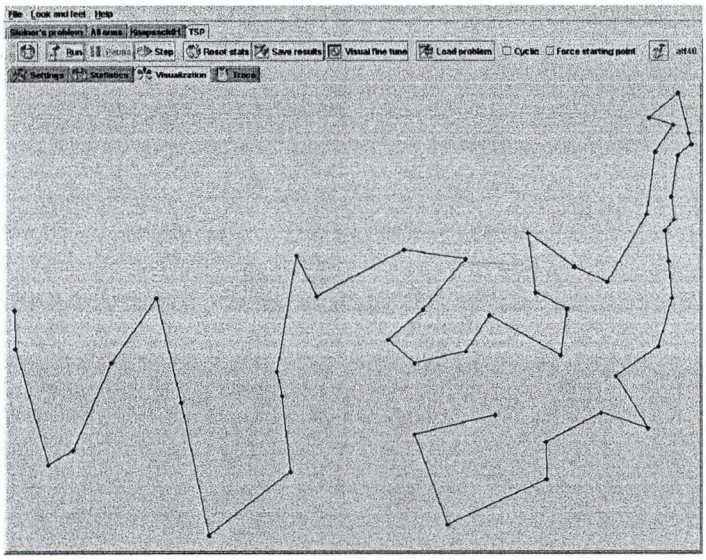


FIG. 6.11 – Meilleur trajet pour le problème du TSP non-cyclique des 48 capitales d'états américains.

Le test préliminaire cyclique donne un meilleur parcours de 33957 unités dans les mêmes conditions de test, nous l'utiliserons comme valeur de référence. La figure 6.12 illustre le meilleur trajet cyclique trouvé.

Dans les tests qui suivent, nous utiliserons une configuration plus légère que pour le test préliminaire, à savoir une population de 1000 individus. Nous nous baserons sur l'état de la convergence après 500 générations. Nous garderons la stratégie du tournoi avec 10% d'élitisme pour tous les tests. Nous n'utiliserons pas de *kick* pour ne pas fausser les résultats. Il est à noter que pour le test de force brute qui tente d'approcher le plus près possible de l'optimum, le *kick* s'est révélé très utile pour gagner quelques unités en fin de parcours. Ce n'est cependant pas pertinent pour les tests qui suivent.

6.5.4 Résultats des tests comparatifs

Nous avons testé les différents opérateurs génétiques adaptés au problème du voyageur de commerce, à savoir l'inversion, le crossover partiellement apparié, le crossover ordonné, et le crossover cyclique.

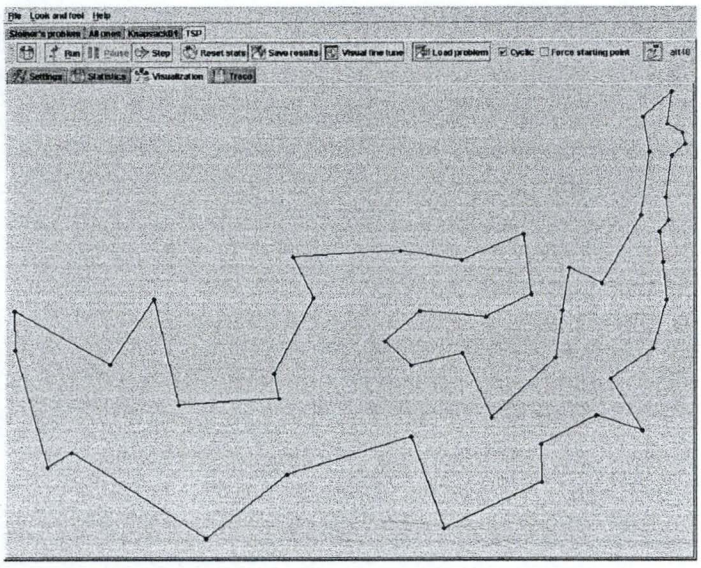


FIG. 6.12 – Meilleur trajet pour le problème du TSP cyclique des 48 capitales d'états américains.

Limitations des crossovers géniques

Les crossovers géniques ou recombinants (*reordering*) posent un problème déjà rencontré pour les autres crossovers lorsqu'ils sont utilisés sans mutation : ils finissent par composer la population de copies d'un même individu (quelle que soit la stratégie de sélection) et donc stagner d'une manière irréversible. Goldberg ne mentionne pas ce problème dans son ouvrage de référence. Cependant, si l'on observe les figures 5.17 et 5.18 de [Goldberg 1989], on voit clairement que les problèmes traités par le crossover partiellement apparié amènent une stagnation irréversible. Goldberg estime que l'approche de l'optimum est satisfaisante, et, probablement pour cette raison, ignore la sévère stagnation observée.

Nous nous sommes cependant livrés à une série de tests comparatifs sur le problème des 48 capitales. Dans la configuration la plus favorable que nous ayons identifiée (stratégie du tournoi sans élitisme, $n = 1000$, $P_X = 600$), la stagnation irréversible se produit, sur la moyenne des tests, à une valeur de viabilité de 51749 et après 277 générations. Cela reste très éloigné du meilleur résultat obtenu (au moyen de l'inversion), à savoir 33957. On peut supposer que la taille limitée des problèmes traités par Goldberg (10 et 33 sommets) ont permis une meilleure approche de l'optimum, qui n'aurait peut-être pas été possible pour des problèmes plus grands.

Il faut également noter que ces crossovers et en particulier la limite de stagnation irréversible sont extrêmement sensibles à la pression de sélection. Si elle est trop forte, la stagnation peut se produire très précocement. Dans le cas contraire, l'algorithme ne convergera pas.

Pour éviter ce problème de stagnation sévère, nous avons ajouté un équivalent génique à la mutation, à savoir l'insertion[8], à une probabilité faible

[8] L'échange réciproque donne des résultats similaires, nous ne l'avons pas inclus dans ces

(1%). Ceci permet, comme on va le voir, de passer la limite de stagnation sévère.

Résultats des tests

La table 6.9 montre le résultat des tests comparatifs.

	Inv. 1pt $P_I = 0,6$	Inv. 2pts $P_I = 0,6$	PMX $P_X = 0,6$	OX $P_X = 0,6$	CX $P_X = 0,6$
1	44064	36330	46115	59703	50587
2	40874	36348	46032	54071	45995
3	40484	35555	41850	51878	43965
4	41056	35260	44489	51950	45241
5	46971	34497	45468	55281	45536
M	42689,8	35598	44790,8	54576,6	46264,8
σ	2491,95	696,65	1580,63	2871	2263,74
$\overline{T}$	2 : 24	2 : 26,2	2 : 40,4	3 : 17,2	2 : 30

TAB. 6.9 – Résultats des tests comparatifs des opérateurs recombinants sur le problème TSP des 48 capitales américaines. $n = 1000$.

On constate que l'inversion à un point donne des résultats nettement moins bons que l'inversion à deux points, et de fait, les possibilités recombinatoires de l'inversion à un point sont plus limitées. En conséquence, cet opérateur est mal adapté au problème du voyageur de commerce.

Les meilleurs résultats sont obtenus avec l'inversion à deux points, qui s'approche, dans la moyenne des tests, à 4,8% de l'optimum dans le délai de 500 générations alloué, avec l'écart-type le plus faible du test (696,65). Ce résultat n'est pas cohérent avec [Goldberg 1989], qui constate une stagnation assez éloignée de l'optimum quand on utilise l'inversion pour un problème à 33 sommets. Ce résultat est peut-être partiellement imputable à la stratégie proportionnelle utilisée.

Les crossovers recombinatoires affichent des résultats décevants. Parmi eux, le crossover partiellement apparié montre les meilleurs résultats, mais reste à 31,9% de l'optimum. On constatera cependant que grâce à l'apport de l'insertion, on a pu systématiquement (dans le cas du PMX) dépasser la limite de stagnation sévère (51749 unités). Aucun des opérateurs n'a d'ailleurs causé de stagnation, mais la plupart ont simplement montré une efficacité relativement faible.

6.5.5 Problèmes de tailles diverses

Nous allons à présent tester l'application sur des problèmes de tailles diverses. Cette expérimentation a une moindre valeur statistique parce qu'il n'a pas été possible d'effectuer plusieurs expérimentations de chaque problème, chaque expérimentation étant assez longue. Ceci nous permettra cependant de donner une estimation de la performance que l'on peut attendre de l'application quand elle est confrontée à des problèmes plus complexes.

---

tests.

Le protocole de test utilise la stratégie du tournoi avec 10% d'élitisme (sur un pool de 10% de la population), l'inversion à deux points avec une probabilité de $P_I = 0,6$ et une population de 2000 individus. L'application est configurée pour s'arrêter après une stagnation de 250 générations, et n'effectue pas de *kick* pour ne pas fausser les mesures.

La table 6.10 montre le résultat des expérimentations. Le nombre de sommets est indiqué dans le nom du problème. Le nom des problèmes indique encore à qui ils sont attribués, selon la clé suivante : eil=Christofides et Eilon, pr=Padberg et Rinaldi, lin=Lin et Kernighan. Des captures d'écrans des meilleurs trajets obtenus sont illustrées en annexe C.

	G	T	f	$\overline{f}$	TSPLib	δ (%)	δ/ns
Eil51	1385	00 :13 :35	437	518,8	426	2,5172	4,9356
Eil76	1900	00 :27 :10	576	658,48	538	6,5972	8,6806
Eil101	1531	00 :27 :23	717	812,25	629	12,2734	12,1518
Pr76	1125	00 :15 :30	113246	133534,62	108159	4,492	5,9105
Pr107	1237	00 :22 :37	47808	62054,74	44303	7,3314	6,8518
Pr124	2658	00 :55 :13	63816	79360,3	59030	7,4997	6,0481
Pr152	2259	00 :56 :57	82310	102872,2	73682	10,4823	6,8963
Pr226	3806	02 :24 :31	94209	116972,6	80369	14,6907	6,5003
Lin105	2321	00 :44 :35	15632	18628,01	14379	8,0156	7,6339
Lin318	4043	03 :29 :01	62503	67869,52	42029	32,7568	10,3009

TAB. 6.10 – Résultats d'expérimentations de problèmes du voyageur de commerce de tailles diverses.

Les colonnes de la table 6.10 ont la signification suivante :

G Nombre de générations au moment de l'arrêt de l'algorithme.

T Durée d'exécution de l'algorithme (cette mesure est purement indicative).

f Meilleure mesure de viabilité trouvée.

$\overline{f}$ Meilleure moyenne de viabilité atteinte en cours d'exécution.

TSPLib Meilleure valeur de viabilité possible, ainsi que rapportée par la bibliothèque *TSPLib* dont ces problèmes sont issus.

δ **(%)** Ecart, exprimé en pourcentage, entre la meilleure viabilité trouvée et celle rapportée par la *TSPLib*.

δ/ns Rapport entre l'écart à la meilleure valeur et le nombre de sommets du problème.

La figure 6.13 montre le rapport entre le nombre de générations nécessaire pour obtenir le résultat et la taille du problème en nombre de sommets. On voit que, les variations statistiques en moins, le nombre de générations nécessaires pour atteindre le meilleur résultat reste grossièrement proportionnelle à la taille du problème.

La figure 6.14 montre l'écart entre le meilleur résultat obtenu et le meilleur résultat théorique comme rapporté par la *TSPLib*. L'évolution de l'écart semble également propotionnelle à la taille du problème, tandis que le rapport entre la précision du résultat de l'expérimentation et la taille du problème (nombre de sommets) paraît relativement constant.

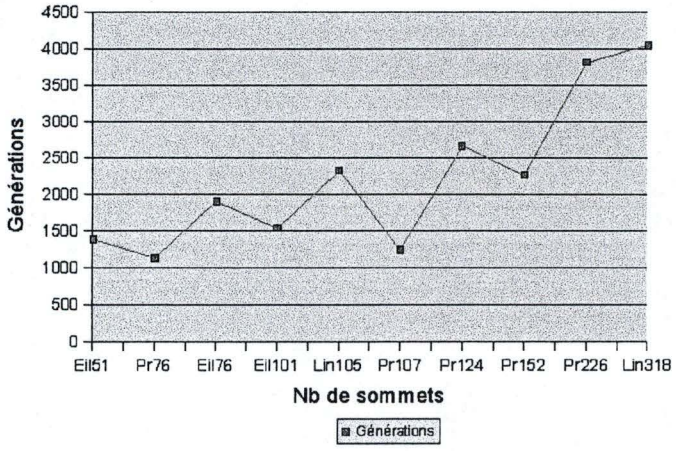


FIG. 6.13 – Rapport entre le nombre de générations et la taille du problème pour diverses expérimentations du TSP.

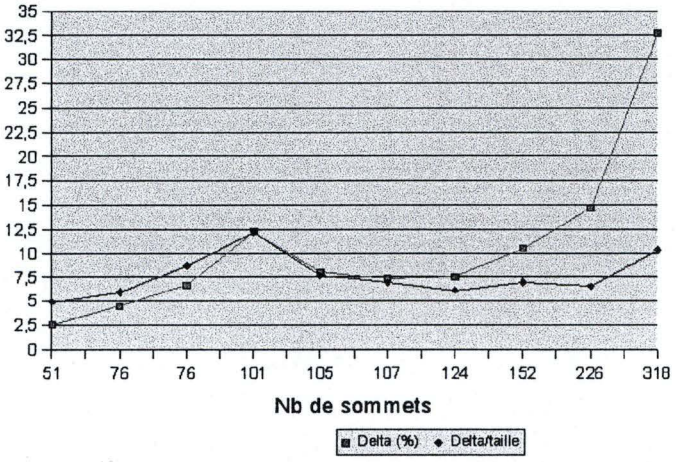


FIG. 6.14 – Ecart au meilleur optimum connu de diverses expérimentations du TSP, et rapport au nombre de générations de l'expérimentation.

6.5.6 Conclusions

Malgré des résultats relativement décevants des crossovers recombinants, le moteur d'exécution se comporte bien, même sur des problèmes de taille conséquente. En effet, même si l'on considère que l'expérimentation ayant traité le problème le plus complexe (lin318) a utilisé 4043 générations pour arriver à son but, et a donc généré environ huit millions d'individus (l'effectif de la population étant de 2000 unités)[9], ce chiffre est à comparer avec le nombre de

[9]Il est probable que certains de ces individus étaient identiques.

variantes possibles (en d'autres termes, la taille de l'espace des solutions), qui est de 317! unités[10], soit environ $6 \cdot 10^{656}$ unités.

[10] Pour un problème non cyclique, il est de $n!$, où n est le nombre de sommets. Pour un problème cyclique, il est de $n!/n$ unités parce que l'on a un même cycle si tous les sommets sont visités dans le même ordre, quel que soit le sommet de départ. $n!/n = (n-1)!$.

Chapitre 7

Conclusion

Un exercice comme la création d'un framework est toujours un exercice frustrant parce qu'il y a toujours quelque chose d'important qui manque au moment où l'on doit le clôturer. Il en est de même en ce qui concerne un rapport comme celui-ci. Mais il est nécessaire de se fixer des limites bien définies, sans quoi, pour l'un et pour l'autre de ces exercices, on n'en finit jamais.

Le but principal de ce mémoire était de proposer une implémentation de référence d'un panel raisonnablement complet des mécanismes de base des algorithmes génétiques, structurée d'une manière claire et facilement manipulable, et d'un outil de démonstration validant cette implémentation tout en montrant de manière visuelle et didactique quelques exemples de la puissance des algorithmes génétiques.

Aucun outil existant (à notre connaissance) ne remplit ces conditions. Les applications existantes proposent des fonctionnalités très spécialisées ou limitées, et/ou ne proposent aucune possibilité d'illustration. Dans tous les cas, les produits sont pauvrement documentés (et parfois pas du tout), souvent mal structurés ou d'une manière obsolète, et occasionnellement manquent de suivi.

Or un outil clair permettant d'expérimenter les possibilités des algorithmes génétiques n'est pas une fantaisie. L'expérience des algorithmes génétiques est souvent frustrante quand on doit la pratiquer sur des outils peu clairs, mal documentés ou complexes, parce qu'il est souvent très difficile d'obtenir des résultats lors des premiers essais ; or c'est d'autant plus difficile si on travaille avec un outil qui ne donne pas d'informations sur les erreurs que l'on a faites. Reste l'option de tout programmer de zéro : option longue et tout aussi frustrante...

Les buts fixés, permettant de pallier à ce manque, nous paraissent entièrement atteints.

- La structure est obtenue au moyen du langage Java, qui a permis d'implémenter un framework clair encourageant la réutilisation et l'extension au moyen des techniques orientées objet.
- La transparence est obtenue par une documentation complète, composée du détail des classes du framework entièrement commenté (dont une copie se trouve en annexe A), d'un tutoriel permettant une mise en route rapide (section 4.2.9 de ce rapport), et d'implémentations de référence donnant plusieurs exemples très différents.
- Le framework a été validé par des tests variés, dont une partie est détaillée au chapitre 6 de ce rapport. La neutralité des tests est garantie par l'uti-

lisation de sources de paramètres extérieures à l'application (*TSPLib*, par exemple), et les résultats des tests ont pu être comparés avec une application indépendante (*GAPlayground*).

Le lecteur jugera, nous l'espérons, que ces buts ont été atteints dans le respect des contraintes de professionnalisme et d'efficacité que nous nous sommes fixés.

7.1 Perspectives

La version du framework au moment de la clôture de ce rapport est étiquetée 0.9*b* (c'est à dire bêta). Ce qui signifie que quelques points de détail peuvent encore être retravaillés, mais la fonctionnalité requise est implémentée et ne présente pas de dysfonction apparente après une série de tests assez complète. Il n'est pas douteux qu'il puisse être utilisé en confiance dans l'état. Il va de soi que c'est un travail que j'espère continuer à peaufiner.

En-dehors du polissage de l'application existante, plusieurs fonctionnalités pourraient être ajoutées à terme pour un produit plus complet et efficace :

- Découpler le moteur d'exécution de la classe de population pour permettre une plus grande flexibilité dans les variantes d'exécution. Pour l'instant il n'est cependant pas prouvé que ce soit possible en gardant une cohérence dans le framework.
- Ajouter la fonctionnalité permettant d'utiliser des populations multiples. C'est un projet en soi, parce que les conséquences sur le framework ne sont pas triviales.
- Découpler l'implémentation des opérateurs génétiques des chromosomes (comme c'est déjà le cas pour la stratégie de sélection). Encore une fois, il faut pouvoir le faire en garantissant la cohérence du framework, ce qui n'est pas trivial.
- Ajouter des optimisations supplémentaires configurables (éviter la compétition entre jumeaux par exemple) et peut-être étendre le framework à des représentations plus pointues (diploïdie par exemple).
- Ajouter des fonctionnalités simplifiant l'implémentation d'une illustration graphique d'un problème implémenté. Le système actuel reposant sur les fonctionnalités de Java est aussi souple que possible mais demande une bonne compétence en Java, il serait intéressant de proposer des fonctionnalités simplifiées pour les personnes ne connaissant pas Java.
- Enfin, il serait plus qu'intéressant d'implémenter un langage simplifié permettant de coder un problème sans passer par Java. C'est aussi un projet en soi, car il faut définir les limites de ce langage, sa syntaxe la plus expressive, en implémenter la grammaire et les fonctionnalités.

Ces extensions ne pouvaient raisonnablement pas être inclues dans la fonctionnalité de base, sans quoi il aurait été impossible de terminer ce travail dans des délais acceptables. Tout du moins la base implémentée est-elle suffisamment bien structurée et suffisamment robuste pour pouvoir être étendue proprement et efficacement. Je ne peux que conclure en espérant qu'elle le sera.

Glossaire

Abstract Window Toolkit (AWT) Un framework de l'API Java permettant de créer des interfaces graphiques simples et portables au moyen des primitives et fonctionnalités les plus courantes. Ce framework est à présent dépassé par *Swing*.

ADN ou Acide Désoxyribonucléique Nom donné aux variantes des acides nucléiques formant la base de l'hérédité, localisées dans les noyaux des cellules, et composées d'une double hélice de bases de purine et de pyrimidine maintenues par des liens hydrogènes, et portant des liens alternés de désoxyribose et de phosphate sur leur partie externe.

Allèle L'une des deux formes d'un gène donné chez les organismes diploïdes. En toute généralité, l'ensemble des formes que peut prendre un gène donné.

API Application Progam Interface. Un ensemble de routines, de protocoles, et d'outils permettant de construire des applications logicielles. Un ensemble de blocs de construction permettant à un programmeur de construire une application. Ce terme est particulièrement usité dans le domaine orienté objet, mais n'est pas limité à ce domaine (par exemple, API Win32).

Applet Java Un programme écrit en Java dans le but d'être exécuté par la machine virtuelle sous le contrôle d'un browser Internet et intégré (en principe) dans une page Web, par opposition à une application Java. Une applet voit ses fonctionnalités restreintes par rapport à celles d'une application du fait des contraintes de sécurité imposées par l'environnement Internet.

Application Java Un programme écrit en Java dans le but d'être exécuté par la machine virtuelle sous le contrôle direct du système d'exploitation, par opposition à une applet Java. Les programmes Java ont pratiquement les mêmes droits sur le système d'exploitation hôte que tout programme natif.

ARN ou Acide Ribonucléique Nom donné aux variantes des acides nucléiques contenant de la ribose et de l'uracile comme éléments structurels, associés au contrôle de certaines activités chimiques cellulaires.

Chromosome Chaîne continue de nucléotides porteuse de l'information génétique.

Classifier System Voir Système de classification.

Convergence prématurée Se dit de la situation où un algorithme génétique s'est rapidement focalisé sur une solution sans avoir eu le temps d'explorer l'espace des solutions d'une manière satisfaisante. L'entièreté de la population étant occupée par cette solution, l'algorithme n'évolue plus, mais la solution peut ne pas être optimale.

Crossover Un échange de gènes entre chromosomes homologues. Dans la nature, l'échange d'une ou plusieurs portions d'un chromosome. Simulé dans les algorithmes génétiques par l'échange d'une ou plusieurs parties de la chaîne de bits codant un chromosome. Les *partially matched*, *cycle* et *order* crossovers sont des extrapolations du crossover (n'existant pas dans la nature) manipulant les gènes sans détruire l'ordre des nucléotides qui les composent.

CX Voir Cycle Crossover.

Cycle Crossover Un type de crossover recombinant ou génique, permettant de croiser les gènes de deux individus tout en garantissant l'intégrité du patrimoine génétique de chacun des individus (voir 5.3.3).

Diploïde (organisme) Se dit d'un organisme possédant un stock dédoublé de chromosomes, c'est à dire deux versions de chaque gène, portées par deux chromosomes distincts.

Fitness (function) Voir Viabilité.

Framework Terme spécifiquement orienté objet désignant un ensemble de classes permettant de créer des applications complexes par réutilisation, extension et combinaison de ces classes. Ce terme est proche du terme *API*.

G.A. Abréviation de *Genetic Algorithm(s)*, algorithme génétique.

Gène L'unité fonctionnelle d'héritage contrôlant la transmission et l'expression d'une ou plusieurs caractéristiques d'un individu en spécifiant la structure d'un polypeptide particulier (spécialement une protéine) ou en contrôlant le fonctionnement d'autres matériaux génétiques. Les gènes sont portés par les chromosomes et l'information qu'ils portent est codée au moyen de nucléotides.

Genetic programming Voir programmation génétique.

Génotype L'ensemble des gènes d'un individu (qu'ils s'expriment en une caractéristique de l'individu formé ou pas).

Haploïde (organisme) Se dit d'un organisme portant un stock simple de chromosomes, par opposition aux organismes diploïdes.

Java Fundation Classes (JFC) L'ensemble des classes de l'API Java fournissant les fonctionnalités permettant de créer des interfaces utilisateurs professionnelles et portables. Les JFC incluent AWT, Swing, Java2D (une API permettant de créer des applications graphiques complexes), Accessibility (une API implémentant des technologies assistives pour les personnes moins valides) et Internationalization (une API permettant d'adapter l'interface à un environnement international).

Locus Position, emplacement. Ce terme désigne l'emplacement d'un gène sur un chromosome. Par extension, désigne également l'emplacement d'un nucléotide sur une chaîne polynucléotidique (comme l'ADN et l'ARN). Pour les AG, l'emplacement d'un bit ou d'un gène sur un chromosome.

Neural Networks Voir Réseaux Neuronaux.

Nucléotide Constituant de base des acides nucléiques comme l'ARN et l'ADN, et porteurs de l'information génétique. Dans la nature, un nucléotide est constitué d'un sucre (ribose ou désoxyribose) joint à une base purique ou pyrimidique et à un groupe phosphate.

Order Crossover Un type de crossover recombinant ou génique, permettant de croiser les gènes de deux individus tout en garantissant l'intégrité du patrimoine génétique de chacun des individus (voir 5.3.3).

OX Voir Order Crossover.

Partially Matched Crossover Un type de crossover recombinant ou génique, permettant de croiser les gènes de deux individus tout en garantissant l'intégrité du patrimoine génétique de chacun des individus (voir 5.3.3).

Phénotype Les propriétés visibles d'un organisme produites par l'intéraction de son génotype avec l'environnement.

PMX Voir Partially Matched Crossover.

Programmation Génétique Technique permettant de générer des programmes informatiques sur base d'une série de fonctions données. Les fonctions sont combinées aléatoirement au départ pour former un programme, la validité du programme est évaluée, puis on fait évoluer les combinaisons, sur base de cette mesure de validité, au moyen de techniques similaires à celles utilisées dans les algorithmes génétiques. Cette technique est également utilisée pour la création et l'optimisation de circuits électroniques.

Recuit Simulé Technique de recherche d'optimum s'inspirant du principe de résilience observé dans la nature (dont un exemple est la trempe en métallurgie). On fait évoluer une solution aléatoirement tout en observant la "température" du système. Plus celle-ci est basse, plus on approche de la solution. L'idée est d'augmenter artificiellement l'entropie du système (donc sa chaleur) lorsqu'on atteint un minimum de température, afin d'explorer de nouveaux minima éventuels.

Réseaux Neuronaux Technique d'intelligence artificielle simulant la manière de fonctionner des neurones du cerveau. Les réseaux de neurones relient des éléments de processus au moyen de connections pondérées. L'état de sortie du réseau de neurones dépend de l'état d'entrée, de l'organisation des connections, et des pondérations associées aux connections. La détermination de la topologie des connexions et de leur pondérations est essentielle pour les réseaux neuronaux. Cet apprentissage peut être notamment effectué sous le contrôle d'un algorithme génétique.

Sélection (stratégie de) Dans les algorithmes génétiques, la technique utilisée pour simuler la sélection naturelle déterminant les individus amenés à se reproduire. C'est généralement une technique probabiliste basée exclusivement sur la mesure de viabilité des individus.

Simulated Annealing Voir Recuit Simulé.

Stagnation Terme utilisé pour désigner un arrêt de la convergence durant l'exécution d'un algorithme génétique. Si la stagnation est précoce, elle est généralement le signe d'une convergence prématurée. Sinon, elle peut être considérée comme un signe d'approche de l'optimum.

Swing Le nom donné à la partie des Java Fundation Classes qui fournit les fonctionnalités permettant de créer des interfaces utilisateurs complexes et modernes, mais cependant portables. Swing étend et remplace AWT.

Système de classification Une technique d'intelligence artificielle, basée sur un ensemble de règles simples que l'on fait évoluer au moyen d'un al-

gorithme génétique. La validité des règles (équivalente à la viabilité) est évaluée au moyen d'un système de récompense/punition.

Viabilité (fonction de) Mesure simulant dans les algorithmes génétiques la capacité des individus à prévaloir dans la lutte à la reproduction. Cette mesure est généralement exprimée par une fonction dépendante du type de problème et basée sur le contenu des gènes de l'individu.

Table des figures

Liste des tableaux

Bibliographie

Algorithmes génétiques

[Goldberg 1989] Goldberg D., *Genetic algorithms in search, optimization and machine learning*, Addison-Wesley, United States of America, 1989.

[Mitchell 1996] Mitchell M., *An introduction to genetic algorithms*, The MIT Press, Cambridge, United States of America, 1996.

[Bauer 1994] Bauer, R., *Genetic algorithms and investment strategies*, John Wiley and Sons, United States of America, 1994.

[Koza 2003] Koza J., Keane M. et Streeter M., "L'informatique évolutionniste", *Pour la science*, 62 – 67, Mai 2003.

Informatique

[Stroustrup 1997] Stroustrup B., *The C++ programming language (third edition)*, Addison-Wesley, United States of America, 1997.

[Java language specification 2000] Gosling J. et al., *The Java^{TM} Language Specification, Second Edition*, ftp.javasoft.com/docs/specs/langspec-2.0.pdf (avril 2000).

Génétique et généralités

[Rostand et Tétry 1972] Rostand J. et Tétry A., *L'homme - Initiation à la biologie*, Paris, France, 1972.

[Merriam-Webster] Merriam-Webster on-line Dictionary, http ://www.m-w.com/.

[Encyclopedia Universalis, Génétique, 1979] L'Héritier P., "Génétique", in *Encyclopedia Universalis*, Encyclopedia Universalis France, Paris, volume 9, pages 548-559, 1979.

[Encyclopedia Universalis, Cuvier, 1979] Piveteau J., "Cuvier (Georges)", in *Encyclopedia Universalis*, Encyclopedia Universalis France, Paris, volume 5, pages 252-254, 1979.

Annexe A

Description de classes du framework

Contents

1 Cuvier Hierarchical Index

1.1 Cuvier Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

2 Cuvier Compound Index

2.1 Cuvier Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

3 Cuvier Page Index

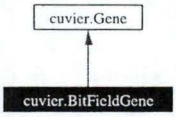
3.1 Cuvier Related Pages

Here is a list of all related documentation pages:

4 Cuvier Class Documentation

4.1 cuvier.BitFieldGene Class Reference

Inheritance diagram for cuvier.BitFieldGene:



4.1.1 Detailed Description

BitFieldGene is an implementation of Gene that holds an internal value that is a plain bit field and hence can be interpreted in any way. The numeric accessors are all enabled and will interpret the bit field.

Author:
Olivier Caudron

Date:
2002-2003

Todo
implement internal representation as a java.util.BitSet for optimal memory usage

Public Methods

- BitFieldGene ()
- BitFieldGene (int size)
- BitFieldGene (boolean[ ] bits)
- byte byteValue ()
- short shortValue ()
- int intValue ()
- long longValue ()
- long signedLongValue ()
- float floatValue ()
- double doubleValue ()
- void setValue (boolean[ ] newVal)
- boolean setValue (String newVal)
- void setBits (java.util.BitSet bits)
- int setBits (java.util.BitSet bits, int pos)
- boolean exchange (Gene g)
- Gene copy ()
- boolean copyInto (Gene g)
- int getLength ()
- boolean[ ] getBitField ()
- int applyGene (java.util.BitSet bits, int pos)
- boolean getBitAt (int pos)
- int compareTo (Object o)
- String toString ()

4.1.2 Constructor & Destructor Documentation

4.1.2.1 cuvier.BitFieldGene.BitFieldGene () [inline]

No-parameter constructor; internal value set to 8 booleans all set to false (default for boolean).

4.1.2.2 cuvier.BitFieldGene.BitFieldGene (int *size*) [inline]

Constructor that only sets the size of the internal representation. All nucleotides are set to false (default for boolean).

4.1.2.3 cuvier.BitFieldGene.BitFieldGene (booleanbits[ ]) [inline]

Constructor that initializes the bit field based on a given bit field.

Note:
The instance copies the provided bit field, and does not use the reference directly (that would be dangerous).

4.1.3 Member Function Documentation

4.1.3.1 int cuvier.BitFieldGene.applyGene (java.util.BitSet *bits*, int *pos*) [inline]

Apply Gene on a bit field at the given offset

Returns:
offset that follows the Gene in the chromosome Note if the bit list offset has not enough bits for Gene, the remaining bits are not applied

Implements cuvier.Gene.

4.1.3.2 byte cuvier.BitFieldGene.byteValue () [inline]

Returns a byte representation of a gene.

Note:
Conversion is not accepted if value is longer than 8 bits to avoid a loss of precision that might cause a meaningless return value.

Exceptions:
ClassCastException in case of bit field size issue.

Implements cuvier.Gene.

4.1.3.3 int cuvier.BitFieldGene.compareTo (Object *o*) [inline]

Comparison method (inherited from interface Comparable) Note : the important point is to report 0 if all bits have the same values. However Comparable imposes that the comparison be coherent, so we say that the gene with the highest-order bit set to 1 is greater than the other. Bit order is counted left-to right (that is, lower indexes in the internal array are lowest-order bits)

Implements cuvier.Gene.

4.1.3.4 Gene cuvier.BitFieldGene.copy () [inline]

Creates a copy of this Gene.

Returns:
a newly-allocated Gene.

Implements cuvier.Gene.

4.1.3.5 boolean cuvier.BitFieldGene.copyInto (Gene *g*) [inline]

Copies this Gene into another without allocation. The target Gene must be pre-allocated.

Returns:
false if target is not pre-allocated or target is not a BitFieldGene.

Implements cuvier.Gene.

4.1.3.6 double cuvier.BitFieldGene.doubleValue () [inline]

Returns a double representation of a gene

Note:
Conversion is not accepted if value is longer than 64 bits to avoid a loss of precision that might cause a meaningless return value.

Exceptions:
ClassCastException in case of bit field size issue.

Implements cuvier.Gene.

4.1.3.7 boolean cuvier.BitFieldGene.exchange (Gene *g*) [inline]

Switch the contents of the genes with no new allocation

Returns:
true if succeeded.

Implements cuvier.Gene.

4.1.3.8 float cuvier.BitFieldGene.floatValue () [inline]

Returns a float representation of a gene

Note:
Conversion is not accepted if value is longer than 32 bits to avoid a loss of precision that might cause a meaningless return value.

Exceptions:
ClassCastException in case of bit field size issue.

Implements cuvier.Gene.

4.1.3.9 boolean cuvier.BitFieldGene.getBitAt (int *pos*) [inline]

Returns a bit at a given position as a boolean; 0 if pos is out of range !!!

Implements cuvier.Gene.

4.1.3.10 boolean [ ] cuvier.BitFieldGene.getBitField () [inline]

Returns an array of booleans equivalent to the bit values

Implements cuvier.Gene.

4.1.3.11 int cuvier.BitFieldGene.getLength () [inline]

Returns the length of the Gene, that is, the size of the representation of the gene

Implements cuvier.Gene.

4.1.3.12 int cuvier.BitFieldGene.intValue () [inline]

Returns an int representation of a gene

Note:

Conversion is not accepted if value is longer than 32 bits to avoid a loss of precision that might cause a meaningless return value.

Exceptions:

ClassCastException in case of bit field size issue.

Implements cuvier.Gene.

4.1.3.13 long cuvier.BitFieldGene.longValue () [inline]

Returns a long representation of a gene

Note:

Conversion is not accepted if value is longer than 64 bits to avoid a loss of precision that might cause a meaningless return value.

Exceptions:

ClassCastException in case of bit field size issue.

Implements cuvier.Gene.

4.1.3.14 int cuvier.BitFieldGene.setBits (java.util.BitSet *bits*, int *pos*) [inline]

Apply the provided array of boolean to the gene's internal value bitwise, starting at a given position in the provided array.

Returns:
Position in the bit field just after the Gene

Note:
The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implements cuvier.Gene.

4.1.3.15 void cuvier.BitFieldGene.setBits (java.util.BitSet *bits*) [inline]

Apply the provided array of boolean to the gene's internal value bitwise.

Note:
The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implements cuvier.Gene.

4.1.3.16 boolean cuvier.BitFieldGene.setValue (String *newVal*) [inline]

Tries to interpret the value passed as a string, by assigning "false" to zeroes and "true" otherwise.

Warning:
if there is a size mismatch between the internal bit field and the string, the internal bit field will be reallocated with the new values !

Implements cuvier.Gene.

4.1.3.17 void cuvier.BitFieldGene.setValue (boolean *newVal*[ ]) [inline]

Sets the internal value according to the provided bit field.

Note:
this may change the size of the bit field (longer or shorter).

4.1.3.18 short cuvier.BitFieldGene.shortValue () [inline]

Returns a short representation of a gene

Note:
Conversion is not accepted if value is longer than 16 bits to avoid a loss of precision that might cause a meaningless return value.

Exceptions:
ClassCastException in case of bit field size issue.

Implements cuvier.Gene.

4.1.3.19 long cuvier.BitFieldGene.signedLongValue () [inline]

Returns a long representation of a gene. If the internal value is shorter than 64 bits, the sign bit is expanded, that is, all bits between the end of the internal value and bit 64 are copies of the sign bit. That way, for instance, 11111110 on a 8-bit representation will be interpreted as -2L: bit 8 is considered as a sign bit and is expanded. Otherwise the value will be 254L. The choice is left to the user which version is suitable for him/her.

Note:
Conversion is not accepted if value is longer than 64 bits to avoid a loss of precision that might cause a meaningless return value.

Exceptions:
ClassCastException in case of bit field size issue.

Implements cuvier.Gene.

4.1.3.20 String cuvier.BitFieldGene.toString () [inline]

Returns a String consisting of an array of bits (0/1) of the length of the Gene Note High-level bits appear to the left as it should

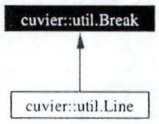
Implements cuvier.Gene.

The documentation for this class was generated from the following file:

- BitFieldGene.java

4.2 cuvier::util.Break Class Reference

Inheritance diagram for cuvier::util.Break:



4.2.1 Detailed Description

The class break is used in the ControlledFlowLayout to mark a line break. Add a reference to an instance of this class in the list of components of ControlledFlowLayout to force a line break.

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

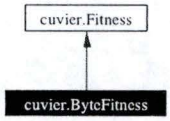
- Break ()
  No-argument constructor, keeps the default alignment properties.
- Break (int newHAlign, int newVAlign)
  Constructor that changes the alignment.
- int getVAlign ()
  Retrieves the vertical alignment property of this object.
- int getHAlign ()
  Retrieves the horizontal alignment property of this object.
- Dimension getPreferredSize ()
  Returns a zero preferred size because the component is non-visual.
- Dimension getMinimumSize ()
  Returns a zero minimum size because the component is non-visual.

The documentation for this class was generated from the following file:

- ControlledFlowLayout.java

4.3 cuvier.ByteFitness Class Reference

Inheritance diagram for cuvier.ByteFitness:



4.3.1 Detailed Description

This is an implementation of the abstract class Fitness with an internal value of type byte. Still needs the calculate method to become a concrete class !

Author:
Olivier Caudron

Date:
25-avr.-03 12:05:03

Public Methods

- ByteFitness ()
- ByteFitness (boolean mini)
- ByteFitness (byte newVal)
- long longValue ()
- double doubleValue ()
- void setValue (String newVal)
- void setValue (long newVal)
- void setValue (double newVal)
- void setValue (Fitness newVal)
- void minimize ()
- void maximize ()
- void zero ()
- void randomize (Fitness f)
- int compare (Fitness f)
- void add (Fitness f)
- void divide (long i)
- String toString ()

4.3.2 Constructor & Destructor Documentation

4.3.2.1 cuvier.ByteFitness.ByteFitness () [inline]

Default constructor. The internal value keeps its default.

4.3.2.2 cuvier.ByteFitness.ByteFitness (boolean *mini*) [inline]

Convenience constructor that forces minimization or maximization of the internal value.

Parameters:
mini boolean true means minimize, false maximize.

4.3.2.3 cuvier.ByteFitness.ByteFitness (byte *newVal*) [inline]

Constructor that sets the internal value.

4.3.3 Member Function Documentation

4.3.3.1 void cuvier.ByteFitness.add (Fitness *f*) [inline, virtual]

See also:
cuvier.Fitness#add(Fitness)

Implements cuvier.Fitness.

4.3.3.2 int cuvier.ByteFitness.compare (Fitness *f*) [inline, virtual]

See also:
cuvier.Fitness#compare(Fitness)

Implements cuvier.Fitness.

4.3.3.3 void cuvier.ByteFitness.divide (long *i*) [inline, virtual]

See also:
cuvier.Fitness#divide(long)

Implements cuvier.Fitness.

4.3.3.4 double cuvier.ByteFitness.doubleValue () [inline, virtual]

See also:
cuvier.Fitness#doubleValue()

Implements cuvier.Fitness.

4.3.3.5 long cuvier.ByteFitness.longValue () [inline, virtual]

See also:
cuvier.Fitness#longValue()

Implements cuvier.Fitness.

4.3.3.6 void cuvier.ByteFitness.maximize () [inline, virtual]

See also:
cuvier.Fitness#maximize()

Implements cuvier.Fitness.

4.3.3.7 void cuvier.ByteFitness.minimize () [inline, virtual]

See also:
cuvier.Fitness#minimize()

Implements cuvier.Fitness.

4.3.3.8 void cuvier.ByteFitness.randomize (Fitness *f*) [inline, virtual]

See also:
cuvier.Fitness#randomize(Fitness)

Implements cuvier.Fitness.

4.3.3.9 void cuvier.ByteFitness.setValue (Fitness *newVal*) [inline, virtual]

See also:
cuvier.Fitness#setValue(Fitness)

Implements cuvier.Fitness.

4.3.3.10 void cuvier.ByteFitness.setValue (double *newVal*) [inline, virtual]

See also:
cuvier.Fitness#setValue(double)

Implements cuvier.Fitness.

4.3.3.11 void cuvier.ByteFitness.setValue (long *newVal*) [inline, virtual]

See also:
cuvier.Fitness#setValue(long)

Implements cuvier.Fitness.

4.3.3.12 void cuvier.ByteFitness.setValue (String *newVal*) [inline, virtual]

See also:
cuvier.Fitness#setValue(String)

Implements cuvier.Fitness.

4.3.3.13 String cuvier.ByteFitness.toString () [inline]

Returns the internal value as a String.

4.3.3.14 void cuvier.ByteFitness.zero () [inline, virtual]

See also:
cuvier.Fitness#zero()

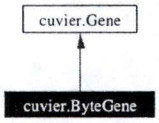
Implements cuvier.Fitness.

The documentation for this class was generated from the following file:

- ByteFitness.java

4.4 cuvier.ByteGene Class Reference

Inheritance diagram for cuvier.ByteGene:



4.4.1 Detailed Description

ByteGene is an implementation of Gene that holds an internal value of type byte (8-bit signed).

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- ByteGene ()
- ByteGene (byte newVal)
- ByteGene (boolean[ ] bits)
- byte byteValue ()
- short shortValue ()
- int intValue ()

- long longValue ()
- long signedLongValue ()
- float floatValue ()
- double doubleValue ()
- void setValue (byte newVal)
- boolean setValue (String newVal)
- void setBits (java.util.BitSet bits)
- int setBits (java.util.BitSet bits, int pos)
- boolean exchange (Gene g)
- Gene copy ()
- boolean copyInto (Gene g)
- int getLength ()
- boolean[ ] getBitField ()
- int applyGene (java.util.BitSet bits, int pos)
- boolean getBitAt (int pos)
- int compareTo (Object o)
- String toString ()

4.4.2 Constructor & Destructor Documentation

4.4.2.1 cuvier.ByteGene.ByteGene () [inline]

No-parameter constructor; internal value set to 0.

4.4.2.2 cuvier.ByteGene.ByteGene (byte *newVal*) [inline]

Constructor that sets the internal value.

4.4.2.3 cuvier.ByteGene.ByteGene (booleanbits[ ]) [inline]

Constructor that sets the internal value using a bit field.

Note:
If the bit field is too long, extra bits are ignored.

4.4.3 Member Function Documentation

4.4.3.1 int cuvier.ByteGene.applyGene (java.util.BitSet *bits*, int *pos*) [inline]

Apply Gene on a bit field at the given offset

Returns:
offset that follows the Gene in the chromosome

Note:
if the bit list offset has not enough bits for Gene, the remaining bits are not applied

Implements cuvier.Gene.

4.4.3.2 byte cuvier.ByteGene.byteValue () [inline]

Returns a byte representation of a gene

Implements cuvier.Gene.

4.4.3.3 int cuvier.ByteGene.compareTo (Object *o*) [inline]

Comparison method (inherited from interface Comparable)

Returns:
positive int if this>o, 0 if this==o, negative int otherwise.

Implements cuvier.Gene.

4.4.3.4 Gene cuvier.ByteGene.copy () [inline]

Creates a copy of this Gene.

Returns:
a newly-allocated Gene.

Implements cuvier.Gene.

4.4.3.5 boolean cuvier.ByteGene.copyInto (Gene *g*) [inline]

Copies this Gene into another without allocation. The target Gene must be pre-allocated.

Returns:
false if target is not pre-allocated or target is not a ByteGene.

Implements cuvier.Gene.

4.4.3.6 double cuvier.ByteGene.doubleValue () [inline]

Returns a double representation of a gene using a simple cast.

Implements cuvier.Gene.

4.4.3.7 boolean cuvier.ByteGene.exchange (Gene *g*) [inline]

Switch the contents of the genes with no new allocation

Returns:
true if succeeded

Implements cuvier.Gene.

4.4.3.8 float cuvier.ByteGene.floatValue () [inline]

Returns a float representation of a gene using a simple cast.

Implements cuvier.Gene.

4.4.3.9 boolean cuvier.ByteGene.getBitAt (int *pos*) [inline]

Returns a bit at a given position as a boolean.

Returns:
bit value (true or false).

Exceptions:
IndexOutOfBoundsException if pos is less than 0 or greater than 7.

Implements cuvier.Gene.

4.4.3.10 boolean [ ] cuvier.ByteGene.getBitField () [inline]

Returns an array of booleans equivalent to the bit values

Implements cuvier.Gene.

4.4.3.11 int cuvier.ByteGene.getLength () [inline]

Returns the length of the Gene, that is, the size of the representation of the gene

Implements cuvier.Gene.

4.4.3.12 int cuvier.ByteGene.intValue () [inline]

Returns an int representation of a gene

Implements cuvier.Gene.

4.4.3.13 long cuvier.ByteGene.longValue () [inline]

Returns a long representation of a gene

Implements cuvier.Gene.

4.4.3.14 int cuvier.ByteGene.setBits (java.util.BitSet *bits*, int *pos*) [inline]

Apply the provided array of boolean to the gene's internal value bitwise, starting at a given position in the provided array.

Returns:
Position in the bit field just after the Gene

Note:
The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implements cuvier.Gene.

4.4.3.15 void cuvier.ByteGene.setBits (java.util.BitSet *bits*) [inline]

Apply the provided array of boolean to the gene's internal value bitwise.

Note:
The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implements cuvier.Gene.

4.4.3.16 boolean cuvier.ByteGene.setValue (String *newVal*) [inline]

Tries to interpret the value passed as a string using the parsing method of Byte.

Exceptions:
IllegalArgumentException in case of error.

Implements cuvier.Gene.

4.4.3.17 void cuvier.ByteGene.setValue (byte *newVal*) [inline]

Mutator that uses the right internal type

4.4.3.18 short cuvier.ByteGene.shortValue () [inline]

Returns a short representation of a gene

Implements cuvier.Gene.

4.4.3.19 long cuvier.ByteGene.signedLongValue () [inline]

Returns a long representation of a gene.

Note:
Identical to longValue() in this implementation.

Implements cuvier.Gene.

4.4.3.20 String cuvier.ByteGene.toString () [inline]

Returns a String consisting of an array of bits (0/1) of the length of the Gene

Note:
High-level bits appear to the left as they should

Implements cuvier.Gene.

The documentation for this class was generated from the following file:

- ByteGene.java

4.5 cuvier.Chromosome Class Reference

4.5.1 Detailed Description

A Chromosome contains the genetic material of an individual in the form of an array of nucleotides (a BitField) and a vector of Gene. It provides the implementation of the genetic operators.

A chromosome is a bit string (internally represented as an array of boolean) that maps to a structure of Genes (represented as a Vector of Genes). It holds a fitness value that is held in an instance of the Fitness class. The instance of the Fitness class also provides the fitness calculation method.

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- Chromosome (Fitness f)
- Chromosome (Gene gene, Fitness f)
- Chromosome (Gene gene1, Gene gene2, Fitness f)
- Chromosome (Gene gene1, Gene gene2, Gene gene3, Fitness f)
- Chromosome (Gene gene1, Gene gene2, Gene gene3, Gene gene4, Fitness f)
- Chromosome (Gene gene1, Gene gene2, Gene gene3, Gene gene4, Gene gene5, Fitness f)
- int getLength ()
- boolean getBit (int pos)
- void addGene (Gene gene)
- void setGene (int pos, Gene g)
- void removeGene (int pos)
- void removeAllGenes ()

- int getGenesQty ()
- int numGenes ()
- Gene getGene (int pos)
- byte byteGetGene (int pos)
- short shortGetGene (int pos)
- int intGetGene (int pos)
- long longGetGene (int pos)
- float floatGetGene (int pos)
- double doubleGetGene (int pos)
- Fitness getFitness ()
- void setFitness (Fitness f)
- int compareTo (Object o)
- synchronized boolean randomize (boolean shuffle)
- synchronized boolean randomize ()
- synchronized boolean shuffle ()
- Chromosome copy ()
- Chromosome randomCopy (boolean shuffle)
- synchronized boolean copyInto (Chromosome c)
- void crossOver (Chromosome c, int start, int end)
- TraceX crossOver1 (Chromosome c, int rate)
- TraceX crossOver2 (Chromosome c, int rate)
- TraceX crossOver2 (Chromosome c, int rate, int minSize, int maxSize)
- TraceX pmx (Chromosome c, int start, int end)
- TraceX pmx (Chromosome c, int rate)
- TraceX pmx (Chromosome c)
- TraceX ox (Chromosome c, int start, int end)
- TraceX ox (Chromosome c, int rate)
- TraceX ox (Chromosome c)
- TraceX cyclex (Chromosome c, int locus)
- TraceX cx (Chromosome c, int rate)
- TraceX cx (Chromosome c)
- boolean mutate (int pos)
- TraceI mutation (int rate)
- boolean invert (int start, int end)
- TraceI inversion2 (int rate)
- TraceI inversion1 (int rate)
- boolean exchange (int pos1, int pos2)
- TraceI exchange (int rate)
- boolean insert (int orig, int dest)
- TraceI insert (int rate)
- String toString ()
- boolean sameSpecies (Chromosome c)
- boolean isValid ()
- String toGenesTuple ()

Protected Methods

- void setFitnessValue (String newVal)
- void setFitnessValue (long newVal)
- void setFitnessValue (double newVal)
- void setFitnessValue (Fitness newVal)

4.5.2 Constructor & Destructor Documentation

4.5.2.1 cuvier.Chromosome.Chromosome (Fitness *f*) [inline]

Default constructor. Sets the fitness type.

4.5.2.2 cuvier.Chromosome.Chromosome (Gene *gene*, Fitness *f*) [inline]

Convenience constructor that takes one predefined gene and the fitness type.

4.5.2.3 cuvier.Chromosome.Chromosome (Gene *gene1*, Gene *gene2*, Fitness *f*) [inline]

Convenience constructor that takes 2 predefined genes and the fitness type.

4.5.2.4 cuvier.Chromosome.Chromosome (Gene *gene1*, Gene *gene2*, Gene *gene3*, Fitness *f*) [inline]

Convenience constructor that takes 3 predefined genes and the fitness type.

4.5.2.5 cuvier.Chromosome.Chromosome (Gene *gene1*, Gene *gene2*, Gene *gene3*, Gene *gene4*, Fitness *f*) [inline]

Convenience constructor that takes 4 predefined genes and the fitness type.

4.5.2.6 cuvier.Chromosome.Chromosome (Gene *gene1*, Gene *gene2*, Gene *gene3*, Gene *gene4*, Gene *gene5*, Fitness *f*) [inline]

Convenience constructor that takes 5 predefined genes and the fitness type.

4.5.3 Member Function Documentation

4.5.3.1 void cuvier.Chromosome.addGene (Gene *gene*) [inline]

Add a Gene to the chromosome. Automatically reconstructs the underlying bit field and updates the fitness.

4.5.3.2 byte cuvier.Chromosome.byteGetGene (int *pos*) [inline]

Positionally get a Gene as a byte.

Warning:
May throw a runtime exception if the operation is not relevant to the type of Gene !

4.5.3.3 int cuvier.Chromosome.compareTo (Object *o*) [inline]

Implementation of Comparable's compareTo method, based on the fitness.

Note:
the chromosome types don't have to match (but naturally they must be both Chromosomes), but the fitness types must or a class cast exception will be thrown (depending on the implementation of Fitness).

4.5.3.4 Chromosome cuvier.Chromosome.copy () [inline]

Clone this chromosome.

Note:
Method cannot be named "clone", this is already a method of Object
This is an exact copy contents included
Makes a copy of the Fitness object. Actually uses a clone() (see Fitness)

4.5.3.5 synchronized boolean cuvier.Chromosome.copyInto (Chromosome *c*) [inline]

Clone this chromosome into another already allocated Chromosome. Used to avoid useless memory allocations because the gc is slow to follow.

Note:
Both chromosomes (source and target) must have the same number of genes and the same fitness class or the copy will fail. If genes at a given locus differ, the target gene at that locus will receive a newly-allocated copy of the source gene. In this case the nucleotide count for the chromosome is updated to match the (possibly new) length of the replacement gene.

4.5.3.6 void cuvier.Chromosome.crossOver (Chromosome *c*, int *start*, int *end*) [inline]

Crossover this chromosome with another - crosses a positional subset of nucleotides

Note:
if positional start is <0 it is forced to 0, or if end is > length, it is forced to length-1.
no check is performed whether the Genes match but length must be identical

4.5.3.7 TraceX cuvier.Chromosome.crossOver1 (Chromosome *c*, int *rate*) [inline]

Random one-point crossover based on the given probability (rate)

Returns:
the point of crossover or length of left portion for statistics

4.5.3.8 TraceX cuvier.Chromosome.crossOver2 (Chromosome *c*, int *rate*, int *minSize*, int *maxSize*) [inline]

Random two-point crossover based on the given probability (rate) The length of the crossover portion is always between minSize and maxSize.

Returns:
the length of the crossover portion for statistics

4.5.3.9 TraceX cuvier.Chromosome.crossOver2 (Chromosome *c*, int *rate*) [inline]

Random two-point crossover based on the given probability (rate)

Returns:
the length of the crossover portion for statistics

4.5.3.10 TraceX cuvier.Chromosome.cx (Chromosome *c*) [inline]

Performs cycle crossover at a random locus

4.5.3.11 TraceX cuvier.Chromosome.cx (Chromosome *c*, int *rate*) [inline]

Performs cycle crossover at a given rate at a random locus

4.5.3.12 TraceX cuvier.Chromosome.cyclex (Chromosome *c*, int *locus*) [inline]

Cycle crossover (cx) : exchanges genes at a given locus then cascades inversions to ensure consistency of chromosomes in terms of genes values.

Returns:
number of exchanges, -1 in case of error.

4.5.3.13 double cuvier.Chromosome.doubleGetGene (int *pos*) [inline]

Positionally get a Gene as a double.

Warning:
May throw a runtime exception if the operation is not relevant to the type of Gene !

4.5.3.14 TraceI cuvier.Chromosome.exchange (int *rate*) [inline]

Exchange genes at random loci based on the given rate.

Returns:
TraceI.

4.5.3.15 boolean cuvier.Chromosome.exchange (int *pos1*, int *pos2*) [inline]

Exchange the genes at the given loci. Position 1 is not required to be < position 2.

Returns:
boolean true if all goes well, false in case of error (loci out of bounds or identical)

4.5.3.16 float cuvier.Chromosome.floatGetGene (int *pos*) [inline]

Positionally get a Gene as a float.

Warning:
May throw a runtime exception if the operation is not relevant to the type of Gene !

4.5.3.17 boolean cuvier.Chromosome.getBit (int *pos*) [inline]

Get an individual bit (nucleotide)

4.5.3.18 Fitness cuvier.Chromosome.getFitness () [inline]

Returns a reference to this class' Fitness instance.

4.5.3.19 Gene cuvier.Chromosome.getGene (int *pos*) [inline]

Get a Gene at a given locus

4.5.3.20 int cuvier.Chromosome.getGenesQty () [inline]

Returns the number of genes in chromosome (synonym to numGenes)

4.5.3.21 int cuvier.Chromosome.getLength () [inline]

Number of bits in chromosome

4.5.3.22 Tracel cuvier.Chromosome.insert (int *rate*) [inline]

Insert a random-chosen gene at a random-chosen locus based on the given rate.

Returns:
int 1 if operation succeeded, 0 otherwise (to be coherent with stats).

4.5.3.23 boolean cuvier.Chromosome.insert (int *orig*, int *dest*) [inline]

Moves the gene at the given locus to some new locus and shifts all genes in between.
For instance : 12345678, we insert gene 4 between 6 and 7 and it gives : 12356478.

Returns:
boolean true if all goes well, false if loci out of bounds or identical.

4.5.3.24 int cuvier.Chromosome.intGetGene (int *pos*) [inline]

Positionally get a Gene as an int.

Warning:
May throw a runtime exception if the operation is not relevant to the type of Gene !

4.5.3.25 TraceI cuvier.Chromosome.inversion1 (int *rate*) [inline]

Conditionally invert genes in the chromosome based on the given rate. The inversion site is randomly determined based on one locus and an indication of start or end to invert

Returns:

The length of the inversion site in number of genes or 0 if inversion did not happen. Also returns 0 in case the inversion failed due to an error (needed by statistics).

4.5.3.26 TraceI cuvier.Chromosome.inversion2 (int *rate*) [inline]

Conditionally invert genes in the chromosome based on the given rate. The inversion site is randomly determined based on 2 loci

Returns:

The length of the inversion site in number of genes, or 0 if inversion did not happen. Also returns 0 in case the inversion failed due to an error (needed by statistics).

4.5.3.27 boolean cuvier.Chromosome.invert (int *start*, int *end*) [inline]

Invert the order of genes in the given locus interval Gene loci are inclusive.

Returns:

false if loci are out of bounds or identical. true otherwise.

4.5.3.28 boolean cuvier.Chromosome.isValid () [inline]

A valid chromosome has non-null fitness and has at least one gene.

Returns:

boolean True if the chromosome is valid.

4.5.3.29 long cuvier.Chromosome.longGetGene (int *pos*) [inline]

Positionally get a Gene as a long.

Warning:

May throw a runtime exception if the operation is not relevant to the type of Gene !

4.5.3.30 boolean cuvier.Chromosome.mutate (int *pos*) [inline]

Mutate a given nucleotide (bit)

4.5.3.31 TraceI cuvier.Chromosome.mutation (int *rate*) [inline]

Conditionally mutate random nucleotides based on the provided rate

Returns:
The number of mutated bits

4.5.3.32 int cuvier.Chromosome.numGenes () [inline]

Returns the number of genes in chromosome (synonym to getGenesQty)

4.5.3.33 TraceX cuvier.Chromosome.ox (Chromosome *c*) [inline]

Performs order crossover unconditionally on a random locus extent

4.5.3.34 TraceX cuvier.Chromosome.ox (Chromosome *c*, int *rate*) [inline]

Performs order crossover at a given rate on a random locus extent

4.5.3.35 TraceX cuvier.Chromosome.ox (Chromosome *c*, int *start*, int *end*) [inline]

Order crossover (ox) : exchanges genes at a given locus while keeping the relative order of the genes at the other loci.

Returns:
the size of the crossover site, -1 in case of error.

4.5.3.36 TraceX cuvier.Chromosome.pmx (Chromosome *c*) [inline]

Performs pmx unconditionally on a random locus extent

4.5.3.37 TraceX cuvier.Chromosome.pmx (Chromosome *c*, int *rate*) [inline]

Performs pmx at a given rate on a random locus extent

4.5.3.38 TraceX cuvier.Chromosome.pmx (Chromosome *c*, int *start*, int *end*) [inline]

Partially Matched Crossover (pmx) : performs crossover of Genes in the given locus extent and ensures integrity by switching Genes outside of the given extent with values in the given extent. Exchanges at least 1 pair (if alleles are identical) and at most 2 pairs (if alleles are different) for each locus exchange. Example : *x***y* at locus 1 give *y****x*. *y*x*** *x*y***

Returns:
length of crossover site, -1 in case of incoherence in the Genes values or types.

4.5.3.39 Chromosome cuvier.Chromosome.randomCopy (boolean *shuffle*) [inline]

Make a copy of this chromosome and randomizes its contents

4.5.3.40 synchronized boolean cuvier.Chromosome.randomize () [inline]

Randomizes the chromosome, that is, randomizes the nucleotides (bits)

4.5.3.41 synchronized boolean cuvier.Chromosome.randomize (boolean *shuffle*) [inline]

Randomizes the chromosomes. If the parameter is false, randomizes the nucleotides (bits), else shuffles the loci of the genes whilst keeping the genes values.

4.5.3.42 void cuvier.Chromosome.removeAllGenes () [inline]

Removes all genes from this chromosome.

Warning:
Use this with caution !!!

4.5.3.43 void cuvier.Chromosome.removeGene (int *pos*) [inline]

Removes a gene at a given locus

Exceptions:
ArrayIndexOutOfBoundsException if pos is not in range.

4.5.3.44 boolean cuvier.Chromosome.sameSpecies (Chromosome *c*) [inline]

Determines whether the Chromosome passed as parameter is of the same species as this chromosome, that is, it is of the same class, with the same fitness class, and an identical genes collection.

Note:

That does not mean both chromosomes hold the same value!
If the chromosomes have no gene or the fitness is null, the method will return true.
It only checks if the chromosomes are of the same species, not if they are internally consistent.

Parameters:

c Chromosome to compare with this

Returns:

boolean True if chromosomes are of the same species

4.5.3.45 void cuvier.Chromosome.setFitness (Fitness *f*) [inline]

Changes this class' Fitness instance and recalculates the fitness. End-users should not use this most of the time.

4.5.3.46 void cuvier.Chromosome.setFitnessValue (Fitness *newVal*) [inline, protected]

Convenience method equivalent to getFitness().setValue(Fitness)

4.5.3.47 void cuvier.Chromosome.setFitnessValue (double *newVal*) [inline, protected]

Convenience method equivalent to getFitness().setValue(double)

4.5.3.48 void cuvier.Chromosome.setFitnessValue (long *newVal*) [inline, protected]

Convenience method equivalent to getFitness().setValue(long)

4.5.3.49 void cuvier.Chromosome.setFitnessValue (String *newVal*) [inline, protected]

Convenience method equivalent to getFitness().setValue(String)

4.5.3.50 void cuvier.Chromosome.setGene (int *pos*, Gene *g*) [inline]

Changes a gene at a given locus. Automatically reconstructs the underlying bit field and updates the fitness.

Exceptions:
ArrayIndexOutOfBoundsException if pos is not in range.

4.5.3.51 short cuvier.Chromosome.shortGetGene (int *pos*) [inline]

Positionally get a Gene as a short.

Warning:
May throw a runtime exception if the operation is not relevant to the type of Gene !

4.5.3.52 synchronized boolean cuvier.Chromosome.shuffle () [inline]

Shuffles the genes at random whilst preserving the genes values

4.5.3.53 String cuvier.Chromosome.toGenesTuple () [inline]

Return a tuple of Gene values. Best conversion is performed for all genes

4.5.3.54 String cuvier.Chromosome.toString () [inline]

toString invokes toString on every Gene. Every Gene is bound in square brackets.

The documentation for this class was generated from the following file:

- Chromosome.java

4.6 cuvier::util.ColumnedLister Class Reference

4.6.1 Detailed Description

ColumnedLister is a custom component that lists information in columns.

The information is not editable by the user but can be modified dynamically. There are two types of information: labeled values and titles.

Author:
Olivier Caudron

Date:
17-mai-03 17:05:02

Note:
The functionality of this class is a still a bit barebone and should be extended.

Public Methods

- boolean setLabeledValue (String label, String value)
- boolean setAllLabeledValues (String[ ] values)
- boolean setVisible (String label, boolean visible)
- void setTitleFontName (String name)
- void setTitleStyle (int style)
- void setTitleSize (int size)
- void setTitleColor (Color col)
- void setTitleFont (Font font)
- void setLabelFontName (String name)
- void setLabelStyle (int style)
- void setLabelSize (int size)
- void setLabelColor (Color col)
- void setLabelFont (Font font)
- void setValueFontName (String name)
- void setValueStyle (int style)
- void setValueSize (int size)
- void setValueColor (Color col)
- void setValueFont (Font font)
- void setGutter (int gutter)
- void setInterval (int interval)
- String getValue (String label)
- LabeledValue getLabeledValue (String label)
- LabeledValue getLabeledValue (int index)
- int getNumberOfLabeledValues ()
- boolean isVisible (String label)
- boolean isVisible (int index)
- int getGutter ()
- int getInterval ()
- void addTitle (String title)
- void addLabeledValue (String label, String value)
- void addLabeledValue (String label, int value)
- void addLabeledValue (String label, long value)
- void addLabeledValue (String label, double value)
- void addColumnBreak ()
- Dimension getPreferredSize ()
- void updateUI ()

Protected Methods

- void paintComponent (Graphics g)

4.6.2 Member Function Documentation

4.6.2.1 void cuvier.util.ColumnedLister.addColumnBreak () [inline]

Adds a column break to the end of the list of items.

4.6.2.2 void cuvier.util.ColumnedLister.addLabeledValue (String *label*, double *value*) [inline]

Adds a labeled value to the list of items. The value is a double.

Parameters:

label String the name of the label

value double the value of the label.

4.6.2.3 void cuvier.util.ColumnedLister.addLabeledValue (String *label*, long *value*) [inline]

Adds a labeled value to the list of items. The value is a long.

Parameters:

label String the name of the label

value long the value of the label.

4.6.2.4 void cuvier.util.ColumnedLister.addLabeledValue (String *label*, int *value*) [inline]

Adds a labeled value to the list of items. The value is an int.

Parameters:

label String the name of the label

value int the value of the label.

4.6.2.5 void cuvier.util.ColumnedLister.addLabeledValue (String *label*, String *value*) [inline]

Adds a labeled value to the list of items. The value is a String.

Parameters:

label String the name of the label

value String the value of the label.

4.6.2.6 void cuvier.util.ColumnedLister.addTitle (String *title*) [inline]

Adds a title at the end of the list of items.

4.6.2.7 int cuvier.util.ColumnedLister.getGutter () [inline]

Returns the size of the gutter between a label and its value, in pixels.

4.6.2.8 int cuvier.util.ColumnedLister.getInterval () [inline]

Returns the size of the interval between columns, in pixels.

4.6.2.9 LabeledValue cuvier.util.ColumnedLister.getLabeledValue (int *index*) [inline]

Get a reference to a LabeledValue (a pair label/value) identified by its index.

4.6.2.10 LabeledValue cuvier.util.ColumnedLister.getLabeledValue (String *label*) [inline]

Get a reference to a LabeledValue (a pair label/value) identified by its label.

4.6.2.11 int cuvier.util.ColumnedLister.getNumberOfLabeledValues () [inline]

Returns the count of labeled values.

4.6.2.12 Dimension cuvier.util.ColumnedLister.getPreferredSize () [inline]

See also:
java.awt.Component#getPreferredSize()

4.6.2.13 String cuvier.util.ColumnedLister.getValue (String *label*) [inline]

Get the value of a given label.

4.6.2.14 boolean cuvier.util.ColumnedLister.isVisible (int *index*) [inline]

Returns the visibility of a label identified by its index.

4.6.2.15 boolean cuvier.util.ColumnedLister.isVisible (String *label*) [inline]

Returns the visibility of a label identified by its name.

4.6.2.16 void cuvier.util.ColumnedLister.paintComponent (Graphics *g*) [inline, protected]

See also:
javax.swing.JComponent#paintComponent(Graphics)

4.6.2.17 boolean cuvier.util.ColumnedLister.setAllLabeledValues (String *values*[ ]) [inline]

Sets the values of all the labels in the order they appear in the list.

Parameters:
values String[] An array of values.

Returns:
boolean False if the number of values was different than the number of labels.

4.6.2.18 void cuvier.util.ColumnedLister.setGutter (int *gutter*) [inline]

Sets the gutter between the label and its value to a given pixel count.

Parameters:
gutter int The gutter to set

4.6.2.19 void cuvier.util.ColumnedLister.setInterval (int *interval*) [inline]

Sets the interval between columns to a given pixel count.

Parameters:
interval int The interval to set.

4.6.2.20 void cuvier.util.ColumnedLister.setLabelColor (Color *col*) [inline]

Sets the font color for labels

4.6.2.21 boolean cuvier.util.ColumnedLister.setLabeledValue (String *label*, String *value*) [inline]

Sets a labeled value identified by its label.

Parameters:

label String The label of the labeled value to change.

value String The new value of this label.

Returns:

boolean Was the label found, and could the value be changed?

4.6.2.22 void cuvier.util.ColumnedLister.setLabelFont (Font *font*) [inline]

Sets the font for labels based on a Font object

4.6.2.23 void cuvier.util.ColumnedLister.setLabelFontName (String *name*) [inline]

Uses a given font (identified by its name) for labels

4.6.2.24 void cuvier.util.ColumnedLister.setLabelSize (int *size*) [inline]

Sets the font size for labels

4.6.2.25 void cuvier.util.ColumnedLister.setLabelStyle (int *style*) [inline]

Sets the font style for labels

4.6.2.26 void cuvier.util.ColumnedLister.setTitleColor (Color *col*) [inline]

Sets the font color for titles

4.6.2.27 void cuvier.util.ColumnedLister.setTitleFont (Font *font*) [inline]

Sets the font for titles based on a Font object

4.6.2.28 void cuvier.util.ColumnedLister.setTitleFontName (String *name*) [inline]

Uses a given font (identified by its name) for titles

4.6.2.29 void cuvier.util.ColumnedLister.setTitleSize (int *size*) [inline]

Sets the font size for titles

4.6.2.30 void cuvier.util.ColumnedLister.setTitleStyle (int *style*) [inline]

Sets the font style for titles

4.6.2.31 void cuvier.util.ColumnedLister.setValueColor (Color *col*) [inline]

Sets the font color for values

4.6.2.32 void cuvier.util.ColumnedLister.setValueFont (Font *font*) [inline]

Sets the font for values based on a Font object

4.6.2.33 void cuvier.util.ColumnedLister.setValueFontName (String *name*) [inline]

Uses a given font (identified by its name) for values

4.6.2.34 void cuvier.util.ColumnedLister.setValueSize (int *size*) [inline]

Sets the font size for values

4.6.2.35 void cuvier.util.ColumnedLister.setValueStyle (int *style*) [inline]

Sets the font style for values

4.6.2.36 boolean cuvier.util.ColumnedLister.setVisible (String *label*, boolean *visible*) [inline]

Turns on or off the visibility of a label identified by its name.

Parameters:

label String The name of the label.

visible boolean The visibility of the label

Returns:

boolean false if the label was not found.

4.6.2.37 void cuvier.util.ColumnedLister.updateUI () [inline]

See also:
javax.swing.JComponent#updateUI()

Warning:
NOT IMPLEMENTED AT THE MOMENT!

Todo
Implement.

The documentation for this class was generated from the following file:

- ColumnedLister.java

4.7 cuvier::util.ControlledFlowLayout Class Reference

4.7.1 Detailed Description

ControlledFlowLayout is a custom layout manager. It allows listing components in a line and manages line breaks, either implicitely when the container is too narrow or explicitely by interpreting a Break component as a forced line break. Breaks can also generate horizontal lines.

Author:
Olivier Caudron

Date:
17-mai-03 17:08:06

Todo
This layout manager only has the basic functionality required and could be enhanced and extended with more eye candy.

Public Methods

- ControlledFlowLayout ()
- ControlledFlowLayout (int newHGap, int newVGap)
- ControlledFlowLayout (int orientation)
- ControlledFlowLayout (int newVGap, int newHGap, int orientation)
- ControlledFlowLayout (int newHGap, int newVGap, int newHAlign, int newVAlign)
- void addLayoutComponent (String name, Component comp)
- void removeLayoutComponent (Component comp)
- Dimension preferredLayoutSize (Container parent)
- Dimension minimumLayoutSize (Container parent)
- void layoutContainer (Container parent)

Static Public Methods

- Break createCR ()
- Break createCR (int hAlign, int vAlign)
- Line createHR ()
- Line createHR (int hAlign, int vAlign)

4.7.2 Constructor & Destructor Documentation

4.7.2.1 cuvier.util.ControlledFlowLayout.ControlledFlowLayout() [inline]

No-parameter constructor. Gaps are defaulted to 4 pixels, orientation is NORTH_WEST.

4.7.2.2 cuvier.util.ControlledFlowLayout.ControlledFlowLayout (int *newHGap*, int *newVGap*) [inline]

Constructor that sets the horizontal and vertical gap between components, in pixels. Orientation is the default, that is, NORTH_WEST.

4.7.2.3 cuvier.util.ControlledFlowLayout.ControlledFlowLayout (int *orientation*) [inline]

Constructor that sets the orientation of the component.

Parameters:

orientation int the orientation requested. Use SwingConstants.NORTH_EAST, NORTH, NORTH_WEST, etc.

4.7.2.4 cuvier.util.ControlledFlowLayout.ControlledFlowLayout (int *newVGap*, int *newHGap*, int *orientation*) [inline]

Constructor that sets the gaps and the orientation.

4.7.2.5 cuvier.util.ControlledFlowLayout.ControlledFlowLayout (int *newHGap*, int *newVGap*, int *newHAlign*, int *newVAlign*) [inline]

Constructor that sets the gaps and the orientation in 2 separate entries, one for horizontal alignment (use SwingConstants.LEFT, CENTER or RIGHT), one for vertical alignment (use SwingConstants.TOP, CENTER or BOTTOM).

4.7.3 Member Function Documentation

4.7.3.1 void cuvier.util.ControlledFlowLayout.addLayoutComponent (String *name*, Component *comp*) [inline]

See also:
java.awt.LayoutManager#addLayoutComponent(String, Component)

4.7.3.2 Break cuvier.util.ControlledFlowLayout.createCR (int *hAlign*, int *vAlign*) [inline, static]

Creates a break object with new alignment properties. Convenience method.

4.7.3.3 Break cuvier.util.ControlledFlowLayout.createCR () [inline, static]

Creates a break object that maintains the alignment properties. Convenience method.

4.7.3.4 Line cuvier.util.ControlledFlowLayout.createHR (int *hAlign*, int *vAlign*) [inline, static]

Creates a lined break object with new alignment properties. Convenience method.

4.7.3.5 Line cuvier.util.ControlledFlowLayout.createHR () [inline, static]

Creates a lined break object that maintains the alignment properties. Convenience method.

4.7.3.6 void cuvier.util.ControlledFlowLayout.layoutContainer (Container *parent*) [inline]

See also:
java.awt.LayoutManager#layoutContainer(Container)

4.7.3.7 Dimension cuvier.util.ControlledFlowLayout.minimumLayoutSize (Container *parent*) [inline]

See also:
java.awt.LayoutManager#minimumLayoutSize(Container)

4.7.3.8 Dimension cuvier.util.ControlledFlowLayout.preferredLayoutSize (Container *parent*) [inline]

See also:
java.awt.LayoutManager#preferredLayoutSize(Container)

4.7.3.9 void cuvier.util.ControlledFlowLayout.removeLayoutComponent (Component *comp*) [inline]

See also:
java.awt.LayoutManager#removeLayoutComponent(Component)

The documentation for this class was generated from the following file:

- ControlledFlowLayout.java

4.8 Cuvier Class Reference

4.8.1 Detailed Description

Cuvier is the entry point for the sample demontration tool based on the Cuvier framework.

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- Cuvier ()

Static Public Methods

- void main (String args[])

4.8.2 Constructor & Destructor Documentation

4.8.2.1 Cuvier.Cuvier () [inline]

Constructor - creates the general visual layout of the application.

4.8.3 Member Function Documentation

4.8.3.1 void Cuvier.main (String *args*[ ]) [inline, static]

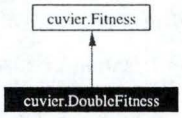
Entry point for demonstration tool of Cuvier.

The documentation for this class was generated from the following file:

- Cuvier.java

4.9 cuvier.DoubleFitness Class Reference

Inheritance diagram for cuvier.DoubleFitness:



4.9.1 Detailed Description

This is an implementation of the abstract class Fitness with an internal value of type double. Still needs the calculate method to become a concrete class !

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- DoubleFitness ()
- DoubleFitness (boolean mini)
- DoubleFitness (double newVal)
- int compare (Fitness f)
- int compare (double d)

- long longValue ()
- double doubleValue ()
- void setValue (String newVal)
- void setValue (long newVal)
- void setValue (double newVal)
- void setValue (Fitness newVal)
- void minimize ()
- void zero ()
- void maximize ()
- void randomize (Fitness f)
- void add (Fitness f)
- void divide (long i)
- String toString ()

4.9.2 Constructor & Destructor Documentation

4.9.2.1 cuvier.DoubleFitness.DoubleFitness () [inline]

Default constructor. The internal value keeps its default.

4.9.2.2 cuvier.DoubleFitness.DoubleFitness (boolean *mini*) [inline]

Convenience constructor that forces minimization or maximization of the internal value.

Parameters:

mini boolean true means minimize, false maximize.

4.9.2.3 cuvier.DoubleFitness.DoubleFitness (double *newVal*) [inline]

Constructor that sets the internal value.

4.9.3 Member Function Documentation

4.9.3.1 void cuvier.DoubleFitness.add (Fitness *f*) [inline, virtual]

Add the internal value of f to the internal value of this.

Note:

Does nothing if the parameter is not a DoubleFitness !

Implements cuvier.Fitness.

4.9.3.2 int cuvier.DoubleFitness.compare (double *d*) [inline]

Compare is equivalent to Comparable's compareTo method.

Returns:
int <0 if this.value<f.value, 0 if this.value==f.value, >0 otherwise.

4.9.3.3 int cuvier.DoubleFitness.compare (Fitness *f*) [inline, virtual]

Compare is equivalent to Comparable's compareTo method.

Returns:
int <0 if this<f, 0 if this==f, >0 otherwise.

Exceptions:
ClassCastException if f is not a DoubleFitness.

Implements cuvier.Fitness.

4.9.3.4 void cuvier.DoubleFitness.divide (long *i*) [inline, virtual]

Divide the internal value by the parameter cast to a double.

Implements cuvier.Fitness.

4.9.3.5 double cuvier.DoubleFitness.doubleValue () [inline, virtual]

Returns the internal value.

Implements cuvier.Fitness.

4.9.3.6 long cuvier.DoubleFitness.longValue () [inline, virtual]

Not relevant in this implementation.

Exceptions:
ClassCastException Always.

Implements cuvier.Fitness.

4.9.3.7 void cuvier.DoubleFitness.maximize () [inline, virtual]

Set internal value to the maximum for datatype double

Implements cuvier.Fitness.

4.9.3.8 void cuvier.DoubleFitness.minimize () [inline, virtual]

Set internal value to the minimum for datatype double

Implements cuvier.Fitness.

4.9.3.9 void cuvier.DoubleFitness.randomize (Fitness *f*) [inline, virtual]

Sets the internal value to a random long between 0 and f.value.

Exceptions:
IllegalArgumentException if f is not a DoubleFitness.

Implements cuvier.Fitness.

4.9.3.10 void cuvier.DoubleFitness.setValue (Fitness *newVal*) [inline, virtual]

Sets the internal value to that of the fitness passed as parameter, if it is a DoubleFitness.

Exceptions:
IllegalArgumentException if the argument is not a DoubleFitness.

Implements cuvier.Fitness.

4.9.3.11 void cuvier.DoubleFitness.setValue (double *newVal*) [inline, virtual]

Sets the internal value.

Implements cuvier.Fitness.

4.9.3.12 void cuvier.DoubleFitness.setValue (long *newVal*) [inline, virtual]

Not relevant to this implementation.

Exceptions:
IllegalArgumentException always.

Implements cuvier.Fitness.

4.9.3.13 void cuvier.DoubleFitness.setValue (String *newVal*) [inline, virtual]

Translates the string into a double using parseDouble and sets the internal value with the result.

Exceptions:
IllegalArgumentException if the string does not represent a valid double.

Implements cuvier.Fitness.

4.9.3.14 String cuvier.DoubleFitness.toString () [inline]

Implementation of toString returns the internal value as a string.

4.9.3.15 void cuvier.DoubleFitness.zero () [inline, virtual]

Set internal value to O.0d

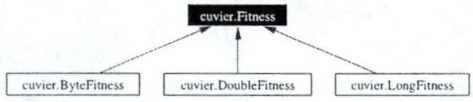
Implements cuvier.Fitness.

The documentation for this class was generated from the following file:

- DoubleFitness.java

4.10 cuvier.Fitness Class Reference

Inheritance diagram for cuvier.Fitness:



4.10.1 Detailed Description

Fitness defines a common representation for the fitnesses calculated by the Genetic Algorithm.

The result of the fitness function of a Genetic Algorithm may be of different data types (integral and floating-point, and possibly others). Fitness proposes a common contract that provides all functionalities required by the Population class to use the fitness returned by the fitness function. It might be useful, for instance, if the fitness calculation in a given problem returns integral numbers that are larger than the largest supported datatype in Java. In that case you may define a Fitness class that will use your specific datatype. Any representation is acceptable (one might even imagine a fitness that is a character string), as long as it provides the base functionality required by this interface, that is, the capability of delivering a minimum, a maximum and a zero, and of comparing 2 values.

Note:

The bounds properties and comparisons must be implemented coherently for the algorithm to work correctly. On top of that, the add and divide methods should be implemented to allow accurate statistics in PopulationViewer (not needed if you provide your own visualization module).

Warning:
Population relies heavily on copying chromosomes accurately. In some rare cases, specific data in your implementation of fitness may not be cloned correctly if you don't provide cloning code for them. Be specially cautious on this issue because it may cause strange misbehaviors.

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- abstract long longValue ()
- abstract double doubleValue ()
- abstract void setValue (String newVal)
- abstract void setValue (long newVal)
- abstract void setValue (double newVal)
- abstract void setValue (Fitness newVal)
- abstract void minimize ()
- abstract void maximize ()
- abstract void zero ()
- abstract void randomize (Fitness f)
- abstract int compare (Fitness f)
- int compareTo (Object o)
- final void optimize (boolean maxProb)
- final void pessimize (boolean maxProb)
- final boolean better (Fitness f, boolean maxProb)
- final boolean worse (Fitness f, boolean maxProb)
- final boolean better (Fitness f)
- final boolean worse (Fitness f)
- Fitness copy ()
- abstract void add (Fitness f)
- abstract void divide (long i)

Protected Methods

- abstract boolean calculate (Chromosome cr)

4.10.2 Member Function Documentation

4.10.2.1 abstract void cuvier.Fitness.add (Fitness *f*) [pure virtual]

Method used for statistics. Should provide a way to accumulate values in this fitness.

Note:

Must not be implemented if not using PopulationViewer. However, if using PopulationViewer, this method shoud not throw a runtime exception.

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.2 final boolean cuvier.Fitness.better (Fitness *f*) [inline]

Same as better(f,true) (maximization problem)

4.10.2.3 final boolean cuvier.Fitness.better (Fitness *f*, boolean *maxProb*) [inline]

Compare this with f based on the type of problem.

Parameters:

f Fitness fitness to compare to.

maxProb boolean true means maximization problem, false minimization problem.

Returns:

true if it's a maximization problem and this>f, or it's a minimization problem and this<f; false otherwise.

4.10.2.4 abstract boolean cuvier.Fitness.calculate (Chromosome *cr*) [protected, pure virtual]

To be implemented in final concrete class. Must set the internal value based on the chromosome passed as parameter.

4.10.2.5 abstract int cuvier.Fitness.compare (Fitness *f*) [pure virtual]

Compare is equivalent to Comparable's compareTo method.

Returns:

int <0 if this<f, 0 if this==f, >0 otherwise.

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.6 int cuvier.Fitness.compareTo (Object *o*) [inline]

Implementation of Comparable's compareTo method. Depends on abstract method compare.

4.10.2.7 Fitness cuvier.Fitness.copy () [inline]

Returns a newly-allocated copy of this fitness.

Exceptions:

NullPointerException if the new instantiation of the class failed (if an InstantiationException or an IllegalAccessException was thrown). This avoids to force exception handling each time copy is invoked.

4.10.2.8 abstract void cuvier.Fitness.divide (long *i*) [pure virtual]

Method used for statistics (averages). Should provide a way to divide the internal value by a long.

Note:

Must not be implemented if not using PopulationViewer. However, if using PopulationViewer, this method shoud not throw a runtime exception.

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.9 abstract double cuvier.Fitness.doubleValue () [pure virtual]

Returns the internal value as a double. This is used by statistics in the default visualization. Certain selection schemes may rely on this value (if the selection uses operators that are not provided by the contract of Fitness), so you should make this method return a meaningful value. If you don't, the default visualization will miss some statistics, and some selection schemes may not work as expected.

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.10 abstract long cuvier.Fitness.longValue () [pure virtual]

Returns the internal value as a long.

Note:

It may not be relevant, and in such case should return 0 (for coherence with doubleValue).

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.11 abstract void cuvier.Fitness.maximize () [pure virtual]

Set internal value to the maximum relevant to the internal type

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.12 abstract void cuvier.Fitness.minimize () [pure virtual]

Set internal value to the minimum relevant to the internal type

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.13 final void cuvier.Fitness.optimize (boolean *maxProb*) [inline]

Maximize if it's a maximization problem, minimize otherwise

Parameters:

maxProb boolean true means maximization problem, false minimization problem.

4.10.2.14 final void cuvier.Fitness.pessimize (boolean *maxProb*) [inline]

Minimize if it's a maximization problem, maximize otherwise

Parameters:

maxProb boolean true means maximization problem, false minimization problem.

4.10.2.15 abstract void cuvier.Fitness.randomize (Fitness *f*) [pure virtual]

Set the value of this to a random value between the zero of this implementation and f. Required by some selection schemes (roulette).

Exceptions:

IllegalArgumentException if the type of f is different to this.

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.16 abstract void cuvier.Fitness.setValue (Fitness *newVal*) [pure virtual]

Sets the internal value to that of the Fitness passed as parameter.

Note:

Should only accept Fitnesses of the same type !

Exceptions:

IllegalArgumentException if the type of Fitness is different.

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.17 abstract void cuvier.Fitness.setValue (double *newVal*) [pure virtual]

Sets the internal value by translating the double passed as parameter, if relevant.

Exceptions:

IllegalArgumentException if double not supported.

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.18 abstract void cuvier.Fitness.setValue (long *newVal*) [pure virtual]

Sets the internal value by translating the long passed as parameter, if relevant.

Exceptions:

IllegalArgumentException if long not supported.

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.19 abstract void cuvier.Fitness.setValue (String *newVal*) [pure virtual]

Sets the internal value by interpreting the string passed as parameter.

Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

4.10.2.20 final boolean cuvier.Fitness.worse (Fitness *f*) [inline]

Same as worse(f,true) (maximization problem)

4.10.2.21 final boolean cuvier.Fitness.worse (Fitness *f*, boolean *maxProb*) [inline]

Compare this with f based on the type of problem.

Parameters:

f Fitness Fitness to compare to.

maxProb boolean true means maximization problem, false minimization problem.

Returns:

true if it's a minimization problem and this>f, or it's a maximization problem and this<f; false otherwise.

4.10.2.22 abstract void cuvier.Fitness.zero () [pure virtual]

Set internal value to the zero relevant to the internal type

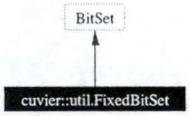
Implemented in cuvier.ByteFitness, cuvier.DoubleFitness, and cuvier.LongFitness.

The documentation for this class was generated from the following file:

- Fitness.java

4.11 cuvier::util.FixedBitSet Class Reference

Inheritance diagram for cuvier::util.FixedBitSet:



4.11.1 Detailed Description

FixedBitSet is a limited extension of the BitSet Java API class. It fixes the length of the bit set.

Author:
Olivier Caudron

Date:
06-avr.-03 12:22:41

Public Methods

- FixedBitSet (int nbits)
- void clear (int bitIndex)
- boolean get (int bitIndex)
- int length ()
- void set (int bitIndex)
- void set (int bitIndex, boolean value)
- void invert (int bitIndex)
- void exchange (FixedBitSet bs, int bitIndex)

4.11.2 Constructor & Destructor Documentation

4.11.2.1 cuvier.util.FixedBitSet.FixedBitSet (int *nbits*) [inline]

This constructor forces setting the length of the BitSet (stored internally). It prevents using the other constructor of BitSet.

Parameters:
nbits The number of bits we initially need.

4.11.3 Member Function Documentation

4.11.3.1 void cuvier.util.FixedBitSet.clear (int *bitIndex*) [inline]

See also:
java.util.BitSet#clear(int) Modifies the ancestor's behavior by throwing an IndexOutOfBoundsException when trying to access a bit above the bit field length (and not only when the index is negative)

Parameters:
bitIndex int The index of the bit we want to clear (set to false).

Exceptions:
IndexOutOfBoundsException If bitIndex>length

4.11.3.2 void cuvier.util.FixedBitSet.exchange (FixedBitSet *bs*, int *bitIndex*) [inline]

Exchanges a bit at a given position with the given FixedBitSet

Exceptions:
IndexOutOfBoundsException if bitIndex is outside the bounds of either this or bs.

Parameters:
bs A bitSet to exchange a bit with

bitIndex Index of the bit to exchange

4.11.3.3 boolean cuvier.util.FixedBitSet.get (int *bitIndex*) [inline]

See also:
java.util.BitSet#get(int) Modifies the ancestor's behavior by throwing an IndexOutOfBoundsException when trying to access a bit above the bit field length (and not only when the index is negative)

Parameters:
bitIndex int The index of the bit we want to retrieve the value.

Returns:
boolean The value of the bit at the given index.

Exceptions:
IndexOutOfBoundsException If bitIndex>length

4.11.3.4 void cuvier.util.FixedBitSet.invert (int *bitIndex*) [inline]

Inverts the bit at the given position

Parameters:
bitIndex The position of the bit to invert

Exceptions:
IndexOutOfBoundsException if bitIndex is less than 0 or greater than length

4.11.3.5 int cuvier.util.FixedBitSet.length () [inline]

See also:
java.util.BitSet#length() Modifies the ancestor's behavior by returning a fixed length instead of one that could be implicitely extended.

4.11.3.6 void cuvier.util.FixedBitSet.set (int *bitIndex*, boolean *value*) [inline]

Convenience method to set the bit at a given position to either true or false.

Parameters:
bitIndex Bit to set
value Setting for the bit

Exceptions:
IndexOutOfBoundsException if bitIndex is less than 0 or greater than length

4.11.3.7 void cuvier.util.FixedBitSet.set (int *bitIndex*) [inline]

See also:
java.util.BitSet#set(int) Modifies the ancestor's behavior by throwing an IndexOutOfBoundsException when trying to access a bit above the bit field length (and not only when the index is negative)

Parameters:
bitIndex int The index of the bit we want to set (set to true).

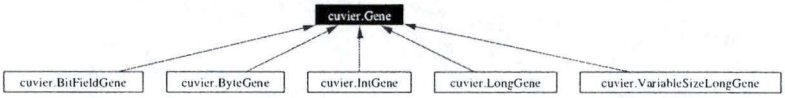
Exceptions:
IndexOutOfBoundsException If bitIndex>length

The documentation for this class was generated from the following file:

- FixedBitSet.java

4.12 cuvier.Gene Interface Reference

Inheritance diagram for cuvier.Gene:



4.12.1 Detailed Description

Gene is an interface that defines the contract for all actual gene types. A Gene is a meaningful component of a chromosome. It must map to nucleotides (bits) that can be applied to the chromosome it is a part of.

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- byte byteValue ()
- short shortValue ()
- int intValue ()
- long longValue ()
- long signedLongValue ()
- float floatValue ()
- double doubleValue ()
- void setBits (java.util.BitSet newBits)
- int setBits (java.util.BitSet newBits, int pos)
- boolean setValue (String newVal)
- boolean exchange (Gene g)
- Gene copy ()
- boolean copyInto (Gene g)
- int getLength ()
- boolean[ ] getBitField ()
- int applyGene (java.util.BitSet bitField, int pos)
- boolean getBitAt (int pos)
- int compareTo (Object o)
- String toString ()

4.12.2 Member Function Documentation

4.12.2.1 int cuvier.Gene.applyGene (java.util.BitSet *bitField*, int *pos*)

Apply Gene on a bit field at the given offset

Returns:
offset that follows the Gene in the chromosome

Warning:
if the bit list offset has not enough bits for Gene, the remaining bits are not applied

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.2 byte cuvier.Gene.byteValue ()

Returns a byte representation of a gene

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.3 int cuvier.Gene.compareTo (Object *o*)

Comparison method (inherited from interface Comparable)

Warning:
MUST BE IMPLEMENTED IN A COHERENT FASHION ! Please refer to the contract of Comparable for more information.

Returns:
positive int if this>o, 0 if this==o, negative int otherwise.

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.4 Gene cuvier.Gene.copy ()

Creates a copy of this Gene.

Returns:
a newly-allocated Gene.

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.5 boolean cuvier.Gene.copyInto (Gene *g*)

Copies this Gene into another without allocation.

Returns:
true if succeeded.

Note:
The target Gene must be pre-allocated. Implementation should check pre-allocation and return false if not pre-allocated. May also throw a null pointer exception.

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.6 double cuvier.Gene.doubleValue ()

Returns a double representation of a gene

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.7 boolean cuvier.Gene.exchange (Gene *g*)

Switch the contents of the genes with no new allocation

Returns:
true if succeeded

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.8 float cuvier.Gene.floatValue ()

Returns a float representation of a gene

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.9 boolean cuvier.Gene.getBitAt (int *pos*)

Returns a bit at a given position as a boolean;

Returns:
bit value.

Warning:
Implementation must throw an IndexOutOfBoundsException if index out of range

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.10 boolean [ ] cuvier.Gene.getBitField ()

Returns an array of booleans equivalent to the bit values

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.11 int cuvier.Gene.getLength ()

Returns the length of the Gene, that is, the size of the representation of the gene

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.12 int cuvier.Gene.intValue ()

Returns an int representation of a gene

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.13 long cuvier.Gene.longValue ()

Returns a long representation of a gene

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.14 int cuvier.Gene.setBits (java.util.BitSet *newBits*, int *pos*)

Apply the provided array of boolean to the gene's internal value bitwise, starting at a given position in the provided array.

Returns:
Position in the bit field just after the Gene

Note:
The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.15 void cuvier.Gene.setBits (java.util.BitSet *newBits*)

Apply the provided array of boolean to the gene's internal value bitwise.

Note:
The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.16 boolean cuvier.Gene.setValue (String *newVal*)

Tries to interpret the value passed as a string to the correct internal type of the implementation of this interface.

Returns:
true if succeeded.

Note:
If it fails, or if this operation is meaningless or not supported, may throw an IllegalArgumentException.

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.17 short cuvier.Gene.shortValue ()

Returns a short representation of a gene

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.18 long cuvier.Gene.signedLongValue ()

Returns a long representation of a gene. If the internal value is shorter than 64 bits, the sign bit is expanded, that is, all bits between the end of the internal value and bit 64 are copies of the sign bit. That way, for instance, 11111110 on a 8-bit representation will be interpreted as -2L: bit 8 is considered as a sign bit and is expanded. Otherwise the value will be 254L. The choice is left to the user which version is suitable for him/her.

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

4.12.2.19 String cuvier.Gene.toString ()

Returns a String consisting of an array of bits (0/1) of the length of the Gene

Note:
High-level bits appear to the left as they should

Implemented in cuvier.BitFieldGene, cuvier.ByteGene, cuvier.IntGene, cuvier.LongGene, and cuvier.VariableSizeLongGene.

The documentation for this interface was generated from the following file:

- Gene.java

4.13 cuvier::util.Graph Class Reference

4.13.1 Detailed Description

Class meant to visualize in a plain graph a series of points. Uses the class Serie defined in the package.

Todo

This class does not use the Java2D framework. It should be later extended to the richer functionality of this framework.
Is prepared for the zooming capability but doesn't have it yet - will be easier to implement using Java2D.

Public Methods

- Graph ()
- synchronized void addSerie (String name, Color col)
- synchronized void setColor (int serie, Color col)
- synchronized void setName (int serie, String name)
- synchronized void addPoint (int serie, double posy)
- synchronized void setCommonScale (boolean set)
- synchronized void setEnabled (boolean set)
- Color getColorOfSerie (int serie)
- Serie getSerie (int serie)
- String serieToString (int serie)
- String getNameOfSerie (int serie)
- int getSeriesCount ()
- boolean isCommonScale ()
- boolean isEnabled ()
- void clearAllSeries ()
- void save ()
- void paintComponent (Graphics g)

4.13.2 Constructor & Destructor Documentation

4.13.2.1 cuvier.util.Graph.Graph () [inline]

The constructor prepares the listeners needed by the zooming facility (not yet fully implemented)

4.13.3 Member Function Documentation

4.13.3.1 synchronized void cuvier.util.Graph.addPoint (int *serie*, double *posy*) [inline]

Adds a point to a serie identified by its index.

Parameters:

serie int the serie's index

posy double the point to add (y coordinate)

4.13.3.2 synchronized void cuvier.util.Graph.addSerie (String *name*, Color *col*) [inline]

To add a serie to the graph.

Parameters:

name String a name for the serie.

col Color a color for the serie.

4.13.3.3 void cuvier.util.Graph.clearAllSeries () [inline]

Removes the contents of all series, while keeping the series themselves.

4.13.3.4 Color cuvier.util.Graph.getColorOfSerie (int *serie*) [inline]

Retrieves the color of a serie identified by its index.

4.13.3.5 String cuvier.util.Graph.getNameOfSerie (int *serie*) [inline]

Get the name of a serie identified by its index.

4.13.3.6 Serie cuvier.util.Graph.getSerie (int *serie*) [inline]

Retrieves a reference to a serie identified by its index.

4.13.3.7 int cuvier.util.Graph.getSeriesCount () [inline]

Gets how many series are defined on this graph.

4.13.3.8 boolean cuvier.util.Graph.isCommonScale () [inline]

Indicates whether series show on a common scale.

4.13.3.9 boolean cuvier.util.Graph.isEnabled () [inline]

indicates whether the graph is enabled (painting).

4.13.3.10 void cuvier.util.Graph.paintComponent (Graphics *g*) [inline]

See also:

javax.swing.JComponent#paintComponent(Graphics)

4.13.3.11 void cuvier.util.Graph.save () [inline]

Saves the contents of the graph in a file selected using a JFileChooser.

Todo
NOT COMPLETE - DO NOT USE!!!

4.13.3.12 String cuvier.util.Graph.serieToString (int *serie*) [inline]

Converts a given serie, identified by its index, to a String. The format of the string is defined by the contract of Serie.toString().

4.13.3.13 synchronized void cuvier.util.Graph.setColor (int *serie*, Color *col*) [inline]

Changes the color of a serie indentified by its index.

4.13.3.14 synchronized void cuvier.util.Graph.setCommonScale (boolean *set*) [inline]

Sets the visualization on a common scale (some series may appear as lines if their scale is less than others)

4.13.3.15 synchronized void cuvier.util.Graph.setEnabled (boolean *set*) [inline]

Sets the graph as enabled, that is, it is painting. Otherwise the paintComponent method is short-circuited.

4.13.3.16 synchronized void cuvier.util.Graph.setName (int *serie*, String *name*) [inline]

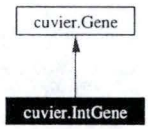
Changes the name of a serie identified by its index.

The documentation for this class was generated from the following file:

- Graph.java

4.14 cuvier.IntGene Class Reference

Inheritance diagram for cuvier.IntGene:



4.14.1 Detailed Description

IntGene is an implementation of Gene that holds an internal value of type int (32-bit signed).

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- IntGene ()
- IntGene (int newVal)
- IntGene (boolean[ ] bits)
- byte byteValue ()
- short shortValue ()
- int intValue ()
- long longValue ()
- long signedLongValue ()
- float floatValue ()
- double doubleValue ()
- void setValue (int newVal)
- boolean setValue (String newVal)
- void setBits (java.util.BitSet bits)
- int setBits (java.util.BitSet bits, int pos)
- boolean exchange (Gene g)
- Gene copy ()
- boolean copyInto (Gene g)
- int getLength ()
- boolean[ ] getBitField ()
- int applyGene (java.util.BitSet bits, int pos)
- boolean getBitAt (int pos)
- int compareTo (Object o)
- String toString ()

4.14.2 Constructor & Destructor Documentation

4.14.2.1 cuvier.IntGene.IntGene () [inline]

No-parameter constructor; internal value set to 0.

4.14.2.2 cuvier.IntGene.IntGene (int *newVal*) [inline]

Constructor that sets the internal value.

4.14.2.3 cuvier.IntGene.IntGene (booleanbits[ ]) [inline]

Constructor that sets the internal value using a bit field.

Note:
If the bit field is too long, extra bits are ignored.

4.14.3 Member Function Documentation

4.14.3.1 int cuvier.IntGene.applyGene (java.util.BitSet *bits*, int *pos*) [inline]

Apply Gene on a bit field at the given offset

Returns:
offset that follows the Gene in the chromosome

Note:
if the bit list offset has not enough bits for Gene, the remaining bits are not applied

Implements cuvier.Gene.

4.14.3.2 byte cuvier.IntGene.byteValue () [inline]

This implementation throws an unconditional ClassCastException

Exceptions:
ClassCastException always. This return value is illegal.

Implements cuvier.Gene.

4.14.3.3 int cuvier.IntGene.compareTo (Object *o*) [inline]

Comparison method (inherited from interface Comparable)

Returns:
positive int if this>o, 0 if this==o, negative int otherwise.

Implements cuvier.Gene.

4.14.3.4 Gene cuvier.IntGene.copy () [inline]

Creates a copy of this Gene.

Returns:
a newly-allocated Gene.

Implements cuvier.Gene.

4.14.3.5 boolean cuvier.IntGene.copyInto (Gene *g*) [inline]

Copies this Gene into another without allocation. The target Gene must be pre-allocated.

Returns:
false if target is not pre-allocated or target is not a IntGene.

Implements cuvier.Gene.

4.14.3.6 double cuvier.IntGene.doubleValue () [inline]

Returns a double representation of a gene using a simple cast.

Implements cuvier.Gene.

4.14.3.7 boolean cuvier.IntGene.exchange (Gene *g*) [inline]

Switch the contents of the genes with no new allocation

Returns:
true if succeeded

Implements cuvier.Gene.

4.14.3.8 float cuvier.IntGene.floatValue () [inline]

Returns a float representation of a gene using a simple cast.

Implements cuvier.Gene.

4.14.3.9 boolean cuvier.IntGene.getBitAt (int *pos*) [inline]

Returns a bit at a given position as a boolean

Returns:
bit value (true or false).

Exceptions:
IndexOutOfBoundsException if pos is less than 0 or greater than 31.

Implements cuvier.Gene.

4.14.3.10 boolean [ ] cuvier.IntGene.getBitField () [inline]

Returns an array of booleans equivalent to the bit values

Implements cuvier.Gene.

4.14.3.11 int cuvier.IntGene.getLength () [inline]

Returns the length of the Gene, that is, the size of the representation of the gene

Implements cuvier.Gene.

4.14.3.12 int cuvier.IntGene.intValue () [inline]

Returns an int representation of a gene

Implements cuvier.Gene.

4.14.3.13 long cuvier.IntGene.longValue () [inline]

Returns a long representation of a gene

Implements cuvier.Gene.

4.14.3.14 int cuvier.IntGene.setBits (java.util.BitSet *bits*, int *pos*) [inline]

Apply the provided array of boolean to the gene's internal value bitwise, starting at a given position in the provided array.

Returns:
Position in the bit field just after the Gene

Note:
The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implements cuvier.Gene.

4.14.3.15 void cuvier.IntGene.setBits (java.util.BitSet *bits*) [inline]

Apply the provided array of boolean to the gene's internal value bitwise.

Note:
The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implements cuvier.Gene.

4.14.3.16 boolean cuvier.IntGene.setValue (String *newVal*) [inline]

Tries to interpret the value passed as a string using the parsing method of Integer.

Exceptions:
IllegalArgumentException in case of error.

Implements cuvier.Gene.

4.14.3.17 void cuvier.IntGene.setValue (int *newVal*) [inline]

Mutator that uses the right internal type

4.14.3.18 short cuvier.IntGene.shortValue () [inline]

This implementation throws an unconditional ClassCastException

Exceptions:
ClassCastException always. This return value is illegal.

Implements cuvier.Gene.

4.14.3.19 long cuvier.IntGene.signedLongValue () [inline]

Returns a long representation of a gene. Identical to longValue in this implementation.

Implements cuvier.Gene.

4.14.3.20 String cuvier.IntGene.toString () [inline]

Returns a String consisting of an array of bits (0/1) of the length of the Gene

Note:
High-level bits appear to the left as they should

Implements cuvier.Gene.

The documentation for this class was generated from the following file:

- IntGene.java

4.15 cuvier::util.JLongField Class Reference

4.15.1 Detailed Description

Text field specialized to edit long values in a given range. Used by JSpinEdit.

Author:
Olivier Caudron

Date:
06-mars-03 15:37:41

Note:

When the edited value is out of range, the field appears in a different color. If the component loses the focus when displaying an out of range value, the original value is restored.

Todo

short-circuit text-setting functionalities of JTextField

Public Methods

- JLongField ()
- JLongField (int columns)
- JLongField (long newMin, long newMax, long newVal)
- JLongField (long newMin, long newMax, long newVal, int columns)
- void addChangeListener (ChangeListener cl)
- void removeChangeListener (ChangeListener cl)
- void setMinimum (long newMin)
- void setMaximum (long newMax)
- void setLimits (long newMin, long newMax)
- boolean setValue (long newVal)
- void setLargeIncrement (long step)
- boolean increment (long step)
- boolean increment ()
- boolean decrement (long step)
- boolean decrement ()
- void setBackground (Color bg)
- void setValidColor (Color bg)
- void setInvalidColor (Color bg)
- Color getValidColor ()
- Color getInvalidColor ()
- long getValue ()
- long getMinimum ()
- long getMaximum ()
- long getLargeIncrement ()

Protected Methods

- void fireStateChanged ()
- javax.swing.text.Document createDefaultModel ()

4.15.2 Constructor & Destructor Documentation

4.15.2.1 cuvier.util.JLongField.JLongField () [inline]

This constructor takes a default number of 10 columns as a field width, a default minimum value of 0L, a default maximum value of 100L, and a default value of 0L.

4.15.2.2 cuvier.util.JLongField.JLongField (int *columns*) [inline]

This constructor sets the number of columns in the field width and uses the default minimum value of 0L, the default maximum value of 100L, and the default value of 0L.

4.15.2.3 cuvier.util.JLongField.JLongField (long *newMin*, long *newMax*, long *newVal*) [inline]

This constructor uses the default number of 10 columns as a field width, and provides values for the minimum, maximum and displayed value.

4.15.2.4 cuvier.util.JLongField.JLongField (long *newMin*, long *newMax*, long *newVal*, int *columns*) [inline]

This constructor provides values for number of columns, minimum, maximum, and displayed values.

4.15.3 Member Function Documentation

4.15.3.1 void cuvier.util.JLongField.addChangeListener (ChangeListener *cl*) [inline]

Allows adding a ChangeListener to the component.

4.15.3.2 javax.swing.text.Document cuvier.util.JLongField.createDefaultModel () [inline, protected]

Modifies the document model behavior to check whether the edit value is valid.

4.15.3.3 boolean cuvier.util.JLongField.decrement () [inline]

Decrements the edit value of one unit.

Returns:

boolean false if the minimum was underflowed.

4.15.3.4 boolean cuvier.util.JLongField.decrement (long *step*) [inline]

Decrement the internal value by the given increment. Stops exactly at minimum.

Parameters:

step long the increment.

Returns:

boolean false if adding the increment would underflow the minimum. In this case the edit value is set to the minimum (NOT the previous value).

4.15.3.5 void cuvier.util.JLongField.fireStateChanged () [inline, protected]

Allows firing a state change event

4.15.3.6 Color cuvier.util.JLongField.getInvalidColor () [inline]

gets the current background color of a field with an invalid value.

4.15.3.7 long cuvier.util.JLongField.getLargeIncrement () [inline]

Gets the value of the large increment

4.15.3.8 long cuvier.util.JLongField.getMaximum () [inline]

Gets the current maximum

4.15.3.9 long cuvier.util.JLongField.getMinimum () [inline]

Gets the current minimum

4.15.3.10 Color cuvier.util.JLongField.getValidColor () [inline]

Gets the current background color of a field with a valid value.

4.15.3.11 long cuvier.util.JLongField.getValue () [inline]

Gets the internal value

4.15.3.12 boolean cuvier.util.JLongField.increment () [inline]

Increments the edit value of one unit.

Returns:

boolean false if the maximum was overflowed.

4.15.3.13 boolean cuvier.util.JLongField.increment (long *step*) [inline]

Increment the internal value by the given increment. Stops exactly at maximum.

Parameters:

step long the increment.

Returns:

boolean false if adding the increment would overflow the maximum. In this case the edit value is set to the maximum (NOT the previous value).

4.15.3.14 void cuvier.util.JLongField.removeChangeListener (ChangeListener *cl*) [inline]

Allows removing a ChangeListener from the list of ChangeListeners of the component.

4.15.3.15 void cuvier.util.JLongField.setBackground (Color *bg*) [inline]

Overrides JComponent's method

4.15.3.16 void cuvier.util.JLongField.setInvalidColor (Color *bg*) [inline]

Sets the background color of a field with an invalid value/

4.15.3.17 void cuvier.util.JLongField.setLargeIncrement (long *step*) [inline]

Changes the large increment step.

Parameters:

step long the new increment.

4.15.3.18 void cuvier.util.JLongField.setLimits (long *newMin*, long *newMax*) [inline]

Change dynamically the minimum and maximum values at once.

Parameters:

newMin long the new minimum

newMax long the new maximum

4.15.3.19 void cuvier.util.JLongField.setMaximum (long *newMax*) [inline]

Change dynamically the maximum value of this component.

Parameters:
newMax long the new maximum.

4.15.3.20 void cuvier.util.JLongField.setMinimum (long *newMin*) [inline]

Change dynamically the minimum value of this component.

Parameters:
newMin long the new minimum.

4.15.3.21 void cuvier.util.JLongField.setValidColor (Color *bg*) [inline]

Sets the background color of a field with a valid value.

4.15.3.22 boolean cuvier.util.JLongField.setValue (long *newVal*) [inline]

Sets the edit value and ensures it is coherent with the minimum and maximum.

Parameters:
newVal long the new value.

Returns:
boolean true if the value is in range.

The documentation for this class was generated from the following file:

- JLongField.java

4.16 cuvier::util.JSpinEdit Class Reference

4.16.1 Detailed Description

A spin edit is missing in the Java APIs before 1.4. This is a simple implementation of one. It uses long values and the JLongField custom component.

Author:
Olivier Caudron

Date:
06-mars-03 15:35:17

Public Methods

- JSpinEdit ()
- JSpinEdit (int columns)
- JSpinEdit (long newMin, long newMax, long newVal)
- JSpinEdit (long newMin, long newMax, long newVal, int columns)
- void addChangeListener (ChangeListener cl)
- void removeChangeListener (ChangeListener cl)
- void setMinimum (long newMin)
- void setMaximum (long newMax)
- void setLimits (long newMin, long newMax)
- boolean setValue (long newVal)
- void setLargeIncrement (long step)
- boolean increment (long step)
- boolean increment ()
- boolean decrement (long step)
- boolean decrement ()
- void setValidColor (Color bg)
- void setInvalidColor (Color bg)
- Color getValidColor ()
- Color getInvalidColor ()
- long getValue ()
- long getMinimum ()
- long getMaximum ()
- long getLargeIncrement ()

4.16.2 Constructor & Destructor Documentation

4.16.2.1 cuvier.util.JSpinEdit.JSpinEdit () [inline]

Constructor using defaults

See also:
JLongField

4.16.2.2 cuvier.util.JSpinEdit.JSpinEdit (int *columns*) [inline]

Constructor setting the number of columns of text field.

See also:
JLongField.

4.16.2.3 cuvier.util.JSpinEdit.JSpinEdit (long *newMin*, long *newMax*, long *newVal*) [inline]

Constructor setting the bounds and the internal value.

See also:
JLongField.

4.16.2.4 cuvier.util.JSpinEdit.JSpinEdit (long *newMin*, long *newMax*, long *newVal*, int *columns*) [inline]

Constructor setting the number of columns, the minimum, maximum and internal values.

See also:
JLongField.

4.16.3 Member Function Documentation

4.16.3.1 void cuvier.util.JSpinEdit.addChangeListener (ChangeListener *cl*) [inline]

Method allowing to add a ChangeListener to this component.

See also:
JLongField.

4.16.3.2 boolean cuvier.util.JSpinEdit.decrement () [inline]

Decrement the internal value.

See also:
JLongField.

4.16.3.3 boolean cuvier.util.JSpinEdit.decrement (long *step*) [inline]

Decrement the internal value.

See also:
JLongField.

4.16.3.4 Color cuvier.util.JSpinEdit.getInvalidColor () [inline]

Gets the color for invalid values.

See also:
JLongField.

4.16.3.5 long cuvier.util.JSpinEdit.getLargeIncrement () [inline]

Gets the large increment value.

4.16.3.6 long cuvier.util.JSpinEdit.getMaximum () [inline]

Gets the maximum value.

4.16.3.7 long cuvier.util.JSpinEdit.getMinimum () [inline]

Gets the minimum value.

4.16.3.8 Color cuvier.util.JSpinEdit.getValidColor () [inline]

Gets the color for valid values.

See also:
JLongField.

4.16.3.9 long cuvier.util.JSpinEdit.getValue () [inline]

Gets the internal value.

4.16.3.10 boolean cuvier.util.JSpinEdit.increment () [inline]

Increment the internal value.

See also:
JLongField.

4.16.3.11 boolean cuvier.util.JSpinEdit.increment (long *step*) [inline]

Increment the internal value.

See also:
JLongField

4.16.3.12 void cuvier.util.JSpinEdit.removeChangeListener (ChangeListener *cl*) [inline]

Method allowing to remove a ChangeListener from the list.

See also:
JLongField.

4.16.3.13 void cuvier.util.JSpinEdit.setInvalidColor (Color *bg*) [inline]

Sets the color for invalid values.

See also:
JLongField.

4.16.3.14 void cuvier.util.JSpinEdit.setLargeIncrement (long *step*) [inline]

Sets the large increment.

4.16.3.15 void cuvier.util.JSpinEdit.setLimits (long *newMin*, long *newMax*) [inline]

Sets the bounds in one go.

4.16.3.16 void cuvier.util.JSpinEdit.setMaximum (long *newMax*) [inline]

Sets the maximum value.

4.16.3.17 void cuvier.util.JSpinEdit.setMinimum (long *newMin*) [inline]

Sets the minimum value.

4.16.3.18 void cuvier.util.JSpinEdit.setValidColor (Color *bg*) [inline]

Sets the color for valid values.

See also:
JLongField.

4.16.3.19 boolean cuvier.util.JSpinEdit.setValue (long *newVal*) [inline]

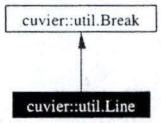
Sets the internal value, if the new value is within bounds.

The documentation for this class was generated from the following file:

- JSpinEdit.java

4.17 cuvier::util.Line Class Reference

Inheritance diagram for cuvier::util.Line:



4.17.1 Detailed Description

The class Line is used in the ControlledFlowLayout to mark a line break with an horizontal line. Add a reference to an instance of this class in the list of components of ControlledFlowLayout to force a line break with an horizontal line.

See also:
Break.

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- Line ()
  No-argument constructor, keeps the default alignment properties.
- Line (int newHAlign, int newVAlign)
  Constructor that changes the alignment.
- void paintComponent (Graphics g)

4.17.2 Member Function Documentation

4.17.2.1 void cuvier.util.Line.paintComponent (Graphics *g*) [inline]

See also:
javax.swing.JComponent#paintComponent(Graphics) Paints the line.

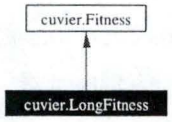
Todo
Modify to draw a line that is coherent with the PLAF.

The documentation for this class was generated from the following file:

- ControlledFlowLayout.java

4.18 cuvier.LongFitness Class Reference

Inheritance diagram for cuvier.LongFitness:



4.18.1 Detailed Description

This is an implementation of the abstract class Fitness with an internal value of type long. Still needs the calculate method to become a concrete class !

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- LongFitness ()
- LongFitness (boolean mini)
- LongFitness (long newVal)
- int compare (Fitness f)
- int compare (long l)
- long longValue ()
- double doubleValue ()
- void setValue (String newVal)

- void setValue (long newVal)
- void setValue (double newVal)
- void setValue (Fitness newVal)
- void minimize ()
- void zero ()
- void maximize ()
- void randomize (Fitness f)
- void add (Fitness f)
- void divide (long i)
- String toString ()

4.18.2 Constructor & Destructor Documentation

4.18.2.1 cuvier.LongFitness.LongFitness () [inline]

Default constructor. The internal value keeps its default.

4.18.2.2 cuvier.LongFitness.LongFitness (boolean *mini*) [inline]

Convenience constructor that forces minimization or maximization of the internal value.

Parameters:

mini boolean true means minimize, false maximize.

4.18.2.3 cuvier.LongFitness.LongFitness (long *newVal*) [inline]

Constructor that sets the internal value.

4.18.3 Member Function Documentation

4.18.3.1 void cuvier.LongFitness.add (Fitness *f*) [inline, virtual]

Add the internal value of f to the internal value of this.

Note:

Does nothing if the parameter is not a LongFitness !

Implements cuvier.Fitness.

4.18.3.2 int cuvier.LongFitness.compare (long *l*) [inline]

Compare is equivalent to Comparable's compareTo method.

Returns:
int <0 if this.value<f.value, 0 if this.value==f.value, >0 otherwise.

4.18.3.3 int cuvier.LongFitness.compare (Fitness *f*) [inline, virtual]

Compare is equivalent to Comparable's compareTo method.

Returns:
int <0 if this<f, 0 if this==f, >0 otherwise.

Exceptions:
ClassCastException if f is not a LongFitness.

Implements cuvier.Fitness.

4.18.3.4 void cuvier.LongFitness.divide (long *i*) [inline, virtual]

Divide the internal value by the parameter.

Implements cuvier.Fitness.

4.18.3.5 double cuvier.LongFitness.doubleValue () [inline, virtual]

Returns the internal value cast to a double.

Implements cuvier.Fitness.

4.18.3.6 long cuvier.LongFitness.longValue () [inline, virtual]

Returns the internal long value.

Implements cuvier.Fitness.

4.18.3.7 void cuvier.LongFitness.maximize () [inline, virtual]

Set internal value to the maximum for datatype long

Implements cuvier.Fitness.

4.18.3.8 void cuvier.LongFitness.minimize () [inline, virtual]

Set internal value to the minimum for datatype long

Implements cuvier.Fitness.

4.18.3.9 void cuvier.LongFitness.randomize (Fitness *f*) [inline, virtual]

Sets the internal value to a random long between 0 and f.value.

Exceptions:
IllegalArgumentException if f is not a LongFitness.

Implements cuvier.Fitness.

4.18.3.10 void cuvier.LongFitness.setValue (Fitness *newVal*) [inline, virtual]

Sets the internal value to that of the fitness passed as parameter, if it is a LongFitness.

Exceptions:
IllegalArgumentException if the argument is not a LongFitness.

Implements cuvier.Fitness.

4.18.3.11 void cuvier.LongFitness.setValue (double *newVal*) [inline, virtual]

Not relevant to this implementation.

Exceptions:
IllegalArgumentException always.

Implements cuvier.Fitness.

4.18.3.12 void cuvier.LongFitness.setValue (long *newVal*) [inline, virtual]

Sets the internal value.

Implements cuvier.Fitness.

4.18.3.13 void cuvier.LongFitness.setValue (String *newVal*) [inline, virtual]

Translates the string into a long using parseLong and sets the internal value with the result.

Exceptions:
IllegalArgumentException if the string does not represent a valid long.

Implements cuvier.Fitness.

4.18.3.14 String cuvier.LongFitness.toString () [inline]

Implementation of toString returns the internal value as a string.

4.18.3.15 void cuvier.LongFitness.zero () [inline, virtual]

Set internal value to OL

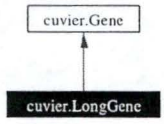
Implements cuvier.Fitness.

The documentation for this class was generated from the following file:

- LongFitness.java

4.19 cuvier.LongGene Class Reference

Inheritance diagram for cuvier.LongGene:



4.19.1 Detailed Description

LongGene is an implementation of Gene that holds an internal value of type long (64-bit signed).

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- LongGene ()
- LongGene (long newVal)
- LongGene (boolean[ ] bits)
- byte byteValue ()
- short shortValue ()
- int intValue ()
- long longValue ()
- long signedLongValue ()
- float floatValue ()
- double doubleValue ()
- void setValue (long newVal)
- boolean setValue (String newVal)

- void setBits (java.util.BitSet bits)
- int setBits (java.util.BitSet bits, int pos)
- boolean exchange (Gene g)
- Gene copy ()
- boolean copyInto (Gene g)
- int getLength ()
- boolean[ ] getBitField ()
- int applyGene (java.util.BitSet bits, int pos)
- boolean getBitAt (int pos)
- int compareTo (Object o)
- String toString ()

4.19.2 Constructor & Destructor Documentation

4.19.2.1 cuvier.LongGene.LongGene () [inline]

No-parameter constructor; internal value set to 0.

4.19.2.2 cuvier.LongGene.LongGene (long *newVal*) [inline]

Constructor that sets the internal value.

4.19.2.3 cuvier.LongGene.LongGene (booleanbits[ ]) [inline]

Constructor that sets the internal value using a bit field.

Note:
If the bit field is too long, extra bits are ignored.

4.19.3 Member Function Documentation

4.19.3.1 int cuvier.LongGene.applyGene (java.util.BitSet *bits*, int *pos*) [inline]

Apply Gene on a bit field at the given offset

Returns:
offset that follows the Gene in the chromosome

Note:
if the bit list offset has not enough bits for Gene, the remaining bits are not applied

Implements cuvier.Gene.

4.19.3.2 byte cuvier.LongGene.byteValue () [inline]

This implementation throws an unconditional ClassCastException

Exceptions:
ClassCastException always. This return value is illegal.

Implements cuvier.Gene.

4.19.3.3 int cuvier.LongGene.compareTo (Object *o*) [inline]

Comparison method (inherited from interface Comparable)

Returns:
positive int if this>o, 0 if this==o, negative int otherwise.

Implements cuvier.Gene.

4.19.3.4 Gene cuvier.LongGene.copy () [inline]

Creates a copy of this Gene.

Returns:
a newly-allocated Gene.

Implements cuvier.Gene.

4.19.3.5 boolean cuvier.LongGene.copyInto (Gene *g*) [inline]

Copies this Gene into another without allocation. The target Gene must be pre-allocated.

Returns:
false if target is not pre-allocated or target is not a ByteGene.

Implements cuvier.Gene.

4.19.3.6 double cuvier.LongGene.doubleValue () [inline]

Returns a double representation of a gene

Implements cuvier.Gene.

4.19.3.7 boolean cuvier.LongGene.exchange (Gene *g*) [inline]

Switch the contents of the genes with no new allocation

Returns:
true if succeeded

Implements cuvier.Gene.

4.19.3.8 float cuvier.LongGene.floatValue () [inline]

This implementation throws an unconditional ClassCastException

Exceptions:
ClassCastException always. This return value is illegal.

Implements cuvier.Gene.

4.19.3.9 boolean cuvier.LongGene.getBitAt (int *pos*) [inline]

Returns a bit at a given position as a boolean

Returns:
bit value (true or false).

Exceptions:
IndexOutOfBoundsException if pos is less than 0 or greater than 64.

Implements cuvier.Gene.

4.19.3.10 boolean [ ] cuvier.LongGene.getBitField () [inline]

Returns an array of booleans equivalent to the bit values

Implements cuvier.Gene.

4.19.3.11 int cuvier.LongGene.getLength () [inline]

Returns the length of the Gene, that is, the size of the representation of the gene

Implements cuvier.Gene.

4.19.3.12 int cuvier.LongGene.intValue () [inline]

This implementation throws an unconditional ClassCastException

Exceptions:
ClassCastException always. This return value is illegal.

Implements cuvier.Gene.

4.19.3.13 long cuvier.LongGene.longValue () [inline]

Returns a long representation of a gene

Implements cuvier.Gene.

4.19.3.14 int cuvier.LongGene.setBits (java.util.BitSet *bits*, int *pos*) [inline]

Apply the provided array of boolean to the gene's internal value bitwise, starting at a given position in the provided array.

Returns:
Position in the bit field just after the Gene

Note:
The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implements cuvier.Gene.

4.19.3.15 void cuvier.LongGene.setBits (java.util.BitSet *bits*) [inline]

Apply the provided array of boolean to the gene's internal value bitwise.

Note:
The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implements cuvier.Gene.

4.19.3.16 boolean cuvier.LongGene.setValue (String *newVal*) [inline]

Tries to interpret the value passed as a string using the parsing method of Long.

Exceptions:
IllegalArgumentException in case of error.

Implements cuvier.Gene.

4.19.3.17 void cuvier.LongGene.setValue (long *newVal*) [inline]

Mutator that uses the right internal type

4.19.3.18 short cuvier.LongGene.shortValue () [inline]

This implementation throws an unconditional ClassCastException

Exceptions:
ClassCastException always. This return value is illegal.

Implements cuvier.Gene.

4.19.3.19 long cuvier.LongGene.signedLongValue () [inline]

Returns a long representation of a gene. Identical to longValue() in this implementation.

Implements cuvier.Gene.

4.19.3.20 String cuvier.LongGene.toString () [inline]

Returns a String consisting of an array of bits (0/1) of the length of the Gene

Note:
High-level bits appear to the left as they should

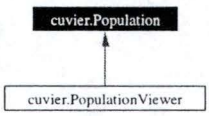
Implements cuvier.Gene.

The documentation for this class was generated from the following file:

- LongGene.java

4.20 cuvier.Population Class Reference

Inheritance diagram for cuvier.Population:



4.20.1 Detailed Description

A population composed of individuals and defining a problem.

A population is a given number of individuals represented by single chromosomes (this framework does not implement diploidy and its consequent properties like dominance, nor does it distinguish genotype from phenotype beyond fitness calculation). It also represents a simple GA problem fully, because the Chromosome class contains its fitness calculation (and value), and this class provides the GA execution code.

Note:
Population extends JPanel to allow its descendants (and especially PopulationViewer) to provide a visualization of the GA problem (required because multiple inheritance does not exist in Java...)

Author:
Olivier Caudron

Date:
06-mars-03 10:23:18

Public Methods

- Population (String name, Chromosome template)
- Population (String name, int settings, Chromosome template)
- Population (String name, int settings)
- Population (String name)
- String getName ()
- String getDescription ()
- Selection getSelection ()
- synchronized boolean isMaximizeProblem ()
- synchronized boolean isTypeFixed ()
- synchronized int getPopulationCount ()
- synchronized int getPopulationSize ()
- synchronized boolean isUsingThreshold ()
- synchronized boolean isUsingStagnation ()
- synchronized boolean isUsingGenerations ()
- synchronized Fitness getFitnessThreshold ()
- synchronized int getGenerationsThreshold ()
- synchronized int getCurrentGeneration ()
- synchronized int getStagnationCount ()
- synchronized int getStagnationThreshold ()
- synchronized int getKickCount ()
- synchronized int getKickCountThreshold ()
- synchronized int getKickRatio ()
- synchronized boolean isGeneOriented ()
- synchronized Chromosome getIndividual (int which)
- synchronized Fitness getFitness (int which)
- synchronized Chromosome getBestIndividual ()
- synchronized java.util.Collection getIndividuals ()
- synchronized int getElitismQtyToSelect ()
- synchronized int getElitismPoolSize ()
- synchronized boolean isElitismUsingRatios ()
- synchronized int getCrossOverStyle ()
- synchronized int getCrossOverRate ()
- synchronized int getMinimumCrossOverSize ()
- synchronized int getMaximumCrossOverSize ()
- synchronized boolean isUsingCrossOverBounds ()
- synchronized int getInversionStyle ()
- synchronized int getInversionRate ()
- synchronized int getMinimumInversionSize ()
- synchronized int getMaximumInversionSize ()
- synchronized boolean isUsingInversionBounds ()
- synchronized Fitness getBestFitness ()
- synchronized Fitness getAllBestFitness ()
- synchronized int getBestFitnessGeneration ()
- synchronized double getAverageFitness ()
- synchronized double getBestAverageFitness ()

- synchronized int getBestAverageGeneration ()
- synchronized Chromosome getOverallBestIndividual ()
- synchronized void setDescription (String desc)
- synchronized boolean setSelection (Selection sel)
- synchronized boolean setMaximize (boolean maxi)
- synchronized void setUseThreshold (boolean set)
- synchronized void setUseStagnation (boolean set)
- synchronized void setUseGenerationCount (boolean set)
- synchronized void setStagnationThreshold (int value)
- synchronized void setKickCountThreshold (int value)
- synchronized void setKickRatio (int value)
- synchronized boolean setIndividual (int which, Chromosome newChrome)
- synchronized boolean setPopulationCount (int count)
- synchronized boolean setElitismQtyToSelect (int number)
- synchronized boolean setElitismPoolSize (int number)
- synchronized void setElitismUseRatios (boolean set)
- synchronized void setCrossOverStyle (int style)
- synchronized boolean setCrossOverRate (int newRate)
- synchronized void setUseCrossOverBounds (boolean set)
- synchronized boolean setMinimumCrossOverSize (int size)
- synchronized boolean setMaximumCrossOverSize (int size)
- synchronized void setInversionStyle (int style)
- synchronized boolean setInversionRate (int newRate)
- synchronized void setUseInversionBounds (boolean set)
- synchronized boolean setMinimumInversionSize (int size)
- synchronized boolean setMaximumInversionSize (int size)
- synchronized boolean setGenerationsThreshold (int iter)
- synchronized boolean setFitnessThreshold (Fitness newThreshold)
- synchronized boolean setFitnessThreshold (String newThreshold)
- synchronized boolean isConsistent ()
- synchronized boolean addGene (Gene newGene)
- synchronized boolean removeGene (int pos)
- synchronized boolean changeGene (int pos, Gene newGene)
- synchronized boolean setFitness (Fitness f)
- synchronized boolean setPopulation (Chromosome template)
- synchronized boolean randomize ()
- synchronized boolean isRunning ()
- synchronized boolean isPaused ()
- synchronized boolean isStopped ()
- synchronized int getState ()
- synchronized void setThreadPriority (int newPriority)
- synchronized int getThreadPriority ()
- synchronized byte getCause ()
- synchronized void kick (int percentage)
- void run ()
- void runStep ()

- void resetStats ()
- synchronized boolean makePool (int size)
- synchronized Chromosome randomGetFromPool ()
- synchronized Chromosome[ ] survive ()
- void sort ()
- void setProperties (java.util.Properties props)
- void getProperties (java.util.Properties props)
- boolean storeSettings (String fname)
- boolean loadSettings (String fname)

Public Attributes

- final int STATE_IDLE = 0
- final int STATE_RUNNING = 1
- final int STATE_PAUSED = 2
- final int STATE_STEPPING = 3

Static Public Attributes

- final int BIT_ORIENTED = 0

  Constant to set the problem as bit (nucleotide)-oriented.

- final int GENE_ORIENTED = 1

  Constant to set the problem as gene-oriented.

- final int MAXIMIZATION = 0

  Constant to set the problem as a maximization problem.

- final int MINIMIZATION = 2

  Constant to set the problem as a minimization problem.

- final int FIXEDTYPE = 4

  Constant to prevent changing the problem type set in the constructor.

- final int CAUSE_ERROR = -1

  Constant that indicates the last run was stopped by an error.

- final int CAUSE_USERSTOP = 0

  Constant that indicates the last run was stopped by the user.

- final int CAUSE_GENERATIONS = 1

  Constant that indicates the last run was stopped by reaching a generations threshold.

- final int CAUSE_FITNESS = 2

  Constant that indicates the last run was stopped by reaching a fitness threshold.

- final int CAUSE_STAGNATION = 3

  Constant that indicates the last run was stopped by reaching a stagnation threshold.

- final int XSTYLE_NONE = 0

  Constant used to set the crossover style to no crossover.

- final int XSTYLE_1POINT = 1

  Constant used to set the crossover style to 1-point crossover.

- final int XSTYLE_2POINT = 2

  Constant used to set the crossover style to 2-point crossover.

- final int XSTYLE_PMX = 3

  Constant used to set the crossover style to Partially Matched Crossover.

- final int XSTYLE_OX = 4

  Constant used to set the crossover style to Order Crossover.

- final int XSTYLE_CX = 5

  Constant used to set the crossover style to Cycle Crossover.

- final int INVSTYLE_NONE = 0

  Constant used to set the mutation/inversion style to no mutation/inversion.

- final int INVSTYLE_MUTATION = 1

  Constant used to set the mutation/inversion style to mutation.

- final int INVSTYLE_1POINT = 2

  Constant used to set the mutation/inversion style to 1-point inversion.

- final int INVSTYLE_2POINT = 3

  Constant used to set the mutation/inversion style to 2-point inversion.

- final int INVSTYLE_RECIPROCALEXCHANGE = 4

  Constant used to set the mutation/inversion style to reciprocal exchange.

- final int INVSTYLE_INSERTION = 5

  Constant used to set the mutation/inversion style to insertion.

- final String xStyleNames [ ]
- final String invStyleNames [ ]

Protected Methods

- synchronized void setRunning ()
- synchronized void setPaused ()
- synchronized void switchPaused ()
- synchronized boolean isStepping ()
- synchronized void setStepping ()
- synchronized void setStopped ()
- synchronized void setState (int newState)
- synchronized boolean traceInit (Stats stats)
- synchronized boolean traceIteration (Stats stats)
- void traceCrossover (int style, Chromosome c1, Chromosome c2, int pos1, int pos2)
- void traceInversion (int style, Chromosome chromosome, int len)
- synchronized void traceResult ()
- synchronized void debug (String trace)

Protected Attributes

- Chromosome[ ] chromes = new Chromosome[100]

  Array of individuals.

- Chromosome[ ] cr2 = new Chromosome[100]

  Temporary holder of individuals for new generation.

- Selection select = new RandomSelection()

  Current Selection strategy holder.

4.20.2 Constructor & Destructor Documentation

4.20.2.1 cuvier.Population.Population (String *name*, Chromosome *template*) [inline]

The minimum constructor for Population requires a Population name and a template chromosome that will be copied to create the population. The default problem is bit-oriented and maximizing.

4.20.2.2 cuvier.Population.Population (String *name*, int *settings*, Chromosome *template*) [inline]

The full constructor requires a problem name, general settings (bit or gene-oriented problem, maximization or minimization problem), and a chromosome template that will be copied to create the population.

4.20.2.3 cuvier.Population.Population (String *name*, int *settings*) [inline]

No-template constructor. After calling this constructor the population is not fully-wrought, it needs to be added valid chromosomes to be runnable. This constructor is required by most implementations because it is impossible to parameterize properly a problem if the chromosomes (hence the fitness) are created in the constructor, because the instance variables of the problem are not accessible from here.

4.20.2.4 cuvier.Population.Population (String *name*) [inline]

Minimal constructor that takes no template and has default settings. Valid chromosomes must be added before the problem can be run.

4.20.3 Member Function Documentation

4.20.3.1 synchronized boolean cuvier.Population.addGene (Gene *newGene*) [inline]

Add a gene to all individuals in the population.

Parameters:
newGene

Returns:
boolean false if problem is running or population is inconsistent, true otherwise.

4.20.3.2 synchronized boolean cuvier.Population.changeGene (int *pos*, Gene *newGene*) [inline]

Changes a gene at a given locus in all the chromosomes in the population.

Parameters:
pos
newGene

Returns:
boolean false if problem is running or population is inconsistent, true otherwise.

4.20.3.3 synchronized void cuvier.Population.debug (String *trace*) [inline, protected]

Method used under exceptional circumstances.

Reimplemented in cuvier.PopulationViewer.

4.20.3.4 synchronized Fitness cuvier.Population.getAllBestFitness () [inline]

Returns a reference to the current copy of the overall best fitness, located in the statistics object. It is not a reference to the fitness object of the best chromosome.

4.20.3.5 synchronized double cuvier.Population.getAverageFitness () [inline]

Returns the average fitness of the current generation as a double.

Warning:
this relies on the Fitness object yielding a correct double value.

4.20.3.6 synchronized double cuvier.Population.getBestAverageFitness () [inline]

Returns the best average so far.

Warning:
this relies on the Fitness object yielding a correct double value.

4.20.3.7 synchronized int cuvier.Population.getBestAverageGeneration () [inline]

Returns the index of the generation where the best average so far was found.

4.20.3.8 synchronized Fitness cuvier.Population.getBestFitness () [inline]

Returns a reference to the current copy of the best fitness for the current generation, located in the statistics object. It is not a reference to the fitness object of the best chromosome.

4.20.3.9 synchronized int cuvier.Population.getBestFitnessGeneration () [inline]

Returns index of the generation where the best fitness so far was found.

4.20.3.10 synchronized Chromosome cuvier.Population.getBestIndividual () [inline]

Locates and returns a reference to the individual with the best fitness in the population. If there are several individuals with the same fitness, the first one is returned.

Returns:

Chromosome a reference to the best-performing individual in the population.

4.20.3.11 synchronized byte cuvier.Population.getCause () [inline]

Retrieves the value of the cause property that gives an indication of how the last run was stopped. Check static variables for possible causes.

4.20.3.12 synchronized int cuvier.Population.getCrossOverRate () [inline]

Returns the current crossover rate (or probability). It is expressed *pro milla* .

4.20.3.13 synchronized int cuvier.Population.getCrossOverStyle () [inline]

Returns the current set crossover style (see the crossover style constants).

4.20.3.14 synchronized int cuvier.Population.getCurrentGeneration () [inline]

Returns the current generation index (useful during a run).

4.20.3.15 String cuvier.Population.getDescription () [inline]

Returns the description of the population (that is, of the problem).

4.20.3.16 synchronized int cuvier.Population.getElitismPoolSize () [inline]

Returns the size of the pool of fittest individuals the elitism functionality randomly chooses individuals from. Whether this figure is a quantity or ratio is determined by isElitismUsingRatios().

4.20.3.17 synchronized int cuvier.Population.getElitismQtyToSelect () [inline]

Returns the number or percentage of individuals the elitism functionality will preselect. Whether this figure is a quantity or ratio is determined by isElitismUsingRatios().

4.20.3.18 synchronized Fitness cuvier.Population.getFitness (int *which*) [inline]

Returns the fitness of an individual identified by its index. If the individual is null, returns null.

Parameters:
which the index of the individual

Returns:
Fitness a reference to the fitness object of the individual.

4.20.3.19 synchronized Fitness cuvier.Population.getFitnessThreshold () [inline]

Returns the fitness at which the run is considered a success. Only taken into account if isUsingThreshold() is true.

4.20.3.20 synchronized int cuvier.Population.getGenerationsThreshold () [inline]

Returns the number of generations at which the run is considered a success. Only taken into account if isUsingGenerations() is true.

4.20.3.21 synchronized Chromosome cuvier.Population.getIndividual (int *which*) [inline]

Returns an individual in the population at the given index.

Parameters:
which index of the individual

Returns:
Chromosome a reference to the individual at the given index

4.20.3.22 synchronized java.util.Collection cuvier.Population.getIndividuals () [inline]

Returns the population as a Collection.

Returns:
Collection a HashSet of references to the individuals of the population.

4.20.3.23 synchronized int cuvier.Population.getInversionRate () [inline]

Returns the current mutation/inversion rate or probability. It is expressed *pro milla* .

Note:
For mutation, the probability is expressed at each bit, not each chromosome!

4.20.3.24 synchronized int cuvier.Population.getInversionStyle () [inline]

Returns the current mutation/inversion style (see the inversion style constants)

4.20.3.25 synchronized int cuvier.Population.getKickCount () [inline]

Returns the current kick count.

4.20.3.26 synchronized int cuvier.Population.getKickCountThreshold () [inline]

Returns the number of kicks at which the run is considered a success after the stagnation threshold is reached again.

4.20.3.27 synchronized int cuvier.Population.getKickRatio () [inline]

Returns the kick ratio, that is, the percentage of the population that is randomized when a kick is performed.

4.20.3.28 synchronized int cuvier.Population.getMaximumCrossOverSize () [inline]

Returns the maximum crossover size.

4.20.3.29 synchronized int cuvier.Population.getMaximumInversionSize () [inline]

Returns the maximum inversion size.

4.20.3.30 synchronized int cuvier.Population.getMinimumCrossOverSize () [inline]

Returns the minimum crossover size.

4.20.3.31 synchronized int cuvier.Population.getMinimumInversionSize () [inline]

Returns the minimum inversion size.

4.20.3.32 String cuvier.Population.getName () [inline]

Returns the name of the population (that is, of the problem) as defined in the contructor.

4.20.3.33 synchronized Chromosome cuvier.Population.getOverallBestIndividual () [inline]

Returns a reference to the copy of the overall best individual found in the statistics object. Does not reference the best individual itself!

4.20.3.34 synchronized int cuvier.Population.getPopulationCount () [inline]

Returns the current number of individuals. Synonym of getPopulationSize.

4.20.3.35 synchronized int cuvier.Population.getPopulationSize () [inline]

Returns the current number of individuals. Synonym of getPopulationCount.

4.20.3.36 void cuvier.Population.getProperties (java.util.Properties *props*) [inline]

Sets the properties of the population based on the properties passed as parameter in a Properties object.

4.20.3.37 Selection cuvier.Population.getSelection () [inline]

Returns a reference to the current selection scheme.

4.20.3.38 synchronized int cuvier.Population.getStagnationCount () [inline]

Returns the current stagnation count, that is, the number of generations that did not improve the **average** fitness.

4.20.3.39 synchronized int cuvier.Population.getStagnationThreshold () [inline]

Returns the stagnation count at which the run is considered a success. Only taken into account if isUsingStagnation() is true.

Note:

The kick count is also taken into account. When the stagnation threshold is reached, a kick is performed and the stagnation count reset as long as the kick threshold is not reached.

4.20.3.40 synchronized int cuvier.Population.getState () [inline]

Gets the internal state

4.20.3.41 synchronized int cuvier.Population.getThreadPriority () [inline]

Gets the value of the threadPriority property used to set the priority of the running thread

4.20.3.42 synchronized boolean cuvier.Population.isConsistent () [inline]

Checks whether the population is consistent, that is, the individuals are non-null and have fully-wrought and all compatible chromosomes.

Note:
this method is quite heavy (several nested loops) and should not be overused.

Returns:
boolean true if the population is consistent.

4.20.3.43 synchronized boolean cuvier.Population.isElitismUsingRatios () [inline]

Indicates whether the elitism figures are ratios (percentages) or quantities.

4.20.3.44 synchronized boolean cuvier.Population.isGeneOriented () [inline]

Indicates whether the problem is gene or nucleotide-oriented. This is set in the settings of the constructor and cannot be changed (this is an essential property of a problem)

4.20.3.45 synchronized boolean cuvier.Population.isMaximizeProblem () [inline]

Indicates whether this problem is currently set to maximize or minimize.

4.20.3.46 synchronized boolean cuvier.Population.isPaused () [inline]

Checks if the internal status is "paused"

4.20.3.47 synchronized boolean cuvier.Population.isRunning () [inline]

Checks if the internal status is "running"

4.20.3.48 synchronized boolean cuvier.Population.isStepping () [inline, protected]

Checks if the internal status is "stepping"

4.20.3.49 synchronized boolean cuvier.Population.isStopped () [inline]

Checks if the internal status is "idle"

4.20.3.50 synchronized boolean cuvier.Population.isTypeFixed () [inline]

Indicates whether changing the problem type is prevented as set in the constructor.

4.20.3.51 synchronized boolean cuvier.Population.isUsingCrossOverBounds () [inline]

Indicates whether crossover is using the minimum and maximum sizes (2-point only)

4.20.3.52 synchronized boolean cuvier.Population.isUsingGenerations () [inline]

Indicates whether generations count is a success criterium.

4.20.3.53 synchronized boolean cuvier.Population.isUsingInversionBounds () [inline]

Indicates whether inversion is using minimum and maximum sizes (2-point inversion only)

4.20.3.54 synchronized boolean cuvier.Population.isUsingStagnation () [inline]

Indicates whether stagnation (including possible kicks) is a success criterium.

4.20.3.55 synchronized boolean cuvier.Population.isUsingThreshold () [inline]

Indicates whether the fitness threshold is a success criterium.

4.20.3.56 synchronized void cuvier.Population.kick (int *percentage*) [inline]

A kick randomizes a given percentage of the population, randomly chosen (can be good performers as well as bad performers)

4.20.3.57 boolean cuvier.Population.loadSettings (String *fname*) [inline]

Loads the settings of the population from a file structured in the usual way for a Properties object serialization.

Returns:

boolean If the load was successful.

Note:

If the load is unsuccessful under rare circumstances the set of properties loaded may be incomplete.

Reimplemented in cuvier.PopulationViewer.

4.20.3.58 synchronized boolean cuvier.Population.makePool (int *size*) [inline]

Create pool of best individuals. Used by elitism (replaces total sort)

4.20.3.59 synchronized Chromosome cuvier.Population.randomGetFromPool () [inline]

Randomly choose an individual from pool

4.20.3.60 synchronized boolean cuvier.Population.randomize () [inline]

Convenience method that randomizes the whole population. Takes the type of problem (gene or bit oriented) into account.

Returns:
boolean false if problem is running or population is inconsistent, true otherwise.

4.20.3.61 synchronized boolean cuvier.Population.removeGene (int *pos*) [inline]

Remove a gene at a given position from all the individuals in the population.

Parameters:
pos

Returns:
boolean false if problem is running or population is inconsistent, true otherwise.

4.20.3.62 void cuvier.Population.resetStats () [inline]

Reinitializes all statistics in one go.

4.20.3.63 void cuvier.Population.run () [inline]

Entry point for executing the genetic algorithm. Checks and sets states and then spawns the execution thread.

4.20.3.64 void cuvier.Population.runStep () [inline]

Entry point for executing the genetic algorithm. Checks and sets states and then spawns the execution thread. Used for a step-by-step run: unlike run() sets the running status to STATE_STEPPING and not STATE_RUNNING.

4.20.3.65 synchronized boolean cuvier.Population.setCrossOverRate (int *newRate*) [inline]

Sets the crossover rate (or probability). It is expressed *pro milla* .

Parameters:
newRate The crossover rate (or probability).

Returns:
boolean false if the problem is running or the rate is incorrect.

Reimplemented in cuvier.PopulationViewer.

4.20.3.66 synchronized void cuvier.Population.setCrossOverStyle (int *style*) [inline]

Sets the style of the crossover. This method is valid for both gene-oriented and nucleotide-oriented problems but only accepts the relevant styles for a given problem type.

Parameters:
style the crossover style as defined in the constants.

Reimplemented in cuvier.PopulationViewer.

4.20.3.67 synchronized void cuvier.Population.setDescription (String *desc*) [inline]

Creates a description for this population (or problem).

Parameters:
desc String

4.20.3.68 synchronized boolean cuvier.Population.setElitismPoolSize (int *number*) [inline]

Sets the size of the pool of fittest individuals the elitism feature chooses individuals from. Whether the number is a quantity or a percentage is defined by isElitismUsingRatios() and set by setElitismUseRatios().

Parameters:
number the ratio or number of the fittest individuals.

Returns:
boolean false if number is out of bounds (less than zero, greater than 100 in case of ratios or greater than the population count in case of quantity).

Reimplemented in cuvier.PopulationViewer.

4.20.3.69 synchronized boolean cuvier.Population.setElitismQtyToSelect (int *number*) [inline]

Sets the number or ratio of individuals pre-selected by the elitism feature. Whether the number is a quantity or percentage is defined by isElitismUsingRatios() and set by setElitismUseRatios().

Parameters:

number the ratio or quantity of pre-selected individuals.

Returns:

boolean false if number is out of bounds (less than zero, greater than 100 in case of ratios or greater than the population count in case of quantity).

Reimplemented in cuvier.PopulationViewer.

4.20.3.70 synchronized void cuvier.Population.setElitismUseRatios (boolean *set*) [inline]

Defines the elitism figures as being ratios or quantities. When changing this setting, a ratio is converted to a quantity of individuals by applying the percentage, and vice-versa.

Parameters:

set true means use ratios, false means use quantities.

Reimplemented in cuvier.PopulationViewer.

4.20.3.71 synchronized boolean cuvier.Population.setFitness (Fitness *f*) [inline]

Changes the fitness class of all chromosomes.

Parameters:

f

Returns:

boolean

4.20.3.72 synchronized boolean cuvier.Population.setFitnessThreshold (String *newThreshold*) [inline]

Sets the fitness threshold to a given fitness expressed as a String.

Reimplemented in cuvier.PopulationViewer.

4.20.3.73 synchronized boolean cuvier.Population.setFitnessThreshold (Fitness *newThreshold*) [inline]

Sets the fitness threshold to a given fitness.

Reimplemented in cuvier.PopulationViewer.

4.20.3.74 synchronized boolean cuvier.Population.setGenerationsThreshold (int *iter*) [inline]

Sets the number of generations threshold.

Reimplemented in cuvier.PopulationViewer.

4.20.3.75 synchronized boolean cuvier.Population.setIndividual (int *which*, Chromosome *newChrome*) [inline]

Changes an individual at a given index.

Parameters:

which Index of the individual to change.

newChrome Reference to the new chromosome to use in that location.

Returns:

boolean false if problem is running.

4.20.3.76 synchronized boolean cuvier.Population.setInversionRate (int *newRate*) [inline]

Sets the inversion/mutation rate (or probability). It is expressed *pro milla* .

Note:

For mutation, the probability is checked for every bit (not every individual).

Parameters:

newRate The inversion rate (or probability).

Returns:

boolean false if the problem is running or the rate is incorrect.

Reimplemented in cuvier.PopulationViewer.

4.20.3.77 synchronized void cuvier.Population.setInversionStyle (int *style*) [inline]

Sets the style of the mutation/inversion. This method is valid for both gene-oriented and nucleotide-oriented problems but only accepts the relevant styles for a given problem type.

Parameters:
style the inversion style as defined in the constants.

Reimplemented in cuvier.PopulationViewer.

4.20.3.78 synchronized void cuvier.Population.setKickCountThreshold (int *value*) [inline]

Sets the kick count threshold. If zero or negative, will be ignored. If stagnation is a success criterium, when both the stagnation and kick thresholds are reached, the run is considered a success.

Reimplemented in cuvier.PopulationViewer.

4.20.3.79 synchronized void cuvier.Population.setKickRatio (int *value*) [inline]

Sets the kick ratio, that is, the percentage of th population that will be randomized when a kick occurs.

Reimplemented in cuvier.PopulationViewer.

4.20.3.80 synchronized boolean cuvier.Population.setMaximize (boolean *maxi*) [inline]

Sets the problem to maximize or minimize.

Parameters:
maxi boolean true means maximize, false minimize.

Returns:
boolean false if the problem is currently running or if the problem type cannot be changed (as defined in the settings of the constructor). True otherwise.

Reimplemented in cuvier.PopulationViewer.

4.20.3.81 synchronized boolean cuvier.Population.setMaximumCrossOverSize (int *size*) [inline]

Sets the maximum crossover size.

Returns:
boolean false if the given size is incorrect.

4.20.3.82 synchronized boolean cuvier.Population.setMaximumInversionSize (int *size*) [inline]

Sets the maximum size of the inversion site.

4.20.3.83 synchronized boolean cuvier.Population.setMinimumCrossOverSize (int *size*) [inline]

Sets the minimum crossover size.

Returns:
boolean false if the given size is incorrect.

4.20.3.84 synchronized boolean cuvier.Population.setMinimumInversionSize (int *size*) [inline]

Sets the minimum size of the inversion site.

4.20.3.85 synchronized void cuvier.Population.setPaused () [inline, protected]

Sets the internal status to "paused"

Note:
It is protected, and not private, to allow descendants to override this method to perform some actions (typically on the visualization). Proceed with caution anyway.

Reimplemented in cuvier.PopulationViewer.

4.20.3.86 synchronized boolean cuvier.Population.setPopulation (Chromosome *template*) [inline]

Sets all chromosomes in the population to randomized copies of the given template.

Returns:
boolean false if the template is null or the problem is running, true otherwise.

4.20.3.87 synchronized boolean cuvier.Population.setPopulationCount (int *count*) [inline]

Changes the populationcount. Produces randomized copies of the individual at index 0 (a random individual).

Parameters:
count new population count.

Returns:
boolean false if problem is running or if the count is less than 1.

Reimplemented in cuvier.PopulationViewer.

4.20.3.88 void cuvier.Population.setProperties (java.util.Properties *props*) [inline]

Adds the properties of the population to the Properties object passed as parameter.

4.20.3.89 synchronized void cuvier.Population.setRunning () [inline, protected]

Sets the internal status to "running"

Note:

It is protected, and not private, to allow descendants to override this method to perform some actions (typically on the visualization). Proceed with caution anyway.

Reimplemented in cuvier.PopulationViewer.

4.20.3.90 synchronized boolean cuvier.Population.setSelection (Selection *sel*) [inline]

Changes the current selection scheme by refernecing another selection instance.

Parameters:

sel Selection a reference to the new selection.

Returns:

boolean false if the problem is currently running (cannot change the selection scheme while the problem is running). true otherwise.

Reimplemented in cuvier.PopulationViewer.

4.20.3.91 synchronized void cuvier.Population.setStagnationThreshold (int *value*) [inline]

Sets the stagnation count threshold. If zero or negative, will be ignored. When the count is reached, a kick will be performed and the count reset. If stagnation is a success criterium, when both the stagnation and kick thresholds are reached, the run is considered a success.

Reimplemented in cuvier.PopulationViewer.

4.20.3.92 synchronized void cuvier.Population.setState (int *newState*) [inline, protected]

Sets the internal status. Check static variables for possible values.

Reimplemented in cuvier.PopulationViewer.

4.20.3.93 synchronized void cuvier.Population.setStepping () [inline, protected]

Sets the internal status to "stepping"

Reimplemented in cuvier.PopulationViewer.

4.20.3.94 synchronized void cuvier.Population.setStopped () [inline, protected]

Sets the internal status to "idle"

Reimplemented in cuvier.PopulationViewer.

4.20.3.95 synchronized void cuvier.Population.setThreadPriority (int *newPriority*) [inline]

Sets the value of the threadPriority property used to set the priority of the running thread

4.20.3.96 synchronized void cuvier.Population.setUseCrossOverBounds (boolean *set*) [inline]

Sets the problem to use the minimum and maximum crossover sizes.

4.20.3.97 synchronized void cuvier.Population.setUseGenerationCount (boolean *set*) [inline]

Sets the problem to consider the generations count threshold as a success criterium.

Note:

Nothing prevents this setting from being changed during a run, but this is as yet untested.

Reimplemented in cuvier.PopulationViewer.

4.20.3.98 synchronized void cuvier.Population.setUseInversionBounds (boolean *set*) [inline]

Sets the use of the Inversion bounds on or off.

4.20.3.99 synchronized void cuvier.Population.setUseStagnation (boolean *set*) [inline]

Sets the problem to consider the stagnation threshold as a success criterium (taking into acocunt the possible kick count).

Note:

Nothing prevents this setting from being changed during a run, but this is as yet untested.

Reimplemented in cuvier.PopulationViewer.

4.20.3.100 synchronized void cuvier.Population.setUseThreshold (boolean *set*) [inline]

Sets the problem to consider the fitness threshold as a success criterium.

Note:
Nothing prevents this setting from being changed during a run, but this is as yet untested.

Reimplemented in cuvier.PopulationViewer.

4.20.3.101 void cuvier.Population.sort () [inline]

Sort individuals in the chromes array according to their fitness (from worst to best) (Required by some selection schemes)

Note:
Uses bubble sort, may be optimized someday...

4.20.3.102 boolean cuvier.Population.storeSettings (String *fname*) [inline]

Stores the properties of the population in a property file based on the usual serialization of a java.util.Properties object.

Returns:
boolean If the properties have been succesfully stored.

4.20.3.103 synchronized Chromosome [] cuvier.Population.survive () [inline]

Select a given number of the best individuals.

4.20.3.104 synchronized void cuvier.Population.switchPaused () [inline, protected]

Switches between the "paused" and the "running" states

Reimplemented in cuvier.PopulationViewer.

4.20.3.105 void cuvier.Population.traceCrossover (int *style*, Chromosome *c1*, Chromosome *c2*, int *pos1*, int *pos2*) [inline, protected]

Method traceCrossover. Public callback for crossover type events.

Parameters:
style int XSTYLE_1POINT, XSTYLE_2POINT, XSTYLE_CX, XSTYLE_OX, XSTYLE_PMX. XSTYLE_NONE in case of error.

c1 Chromosome

c2 Chromosome

pos1 int Lower bound of the crossover site. Depends whether the crossover is bit- or gene-oriented.

pos2 int Upper bound of the crossover site.

Note:
This callback is not called if no crossover style is selected.

Reimplemented in cuvier.PopulationViewer.

4.20.3.106 synchronized boolean cuvier.Population.traceInit (Stats *stats*) [inline, protected]

Method traceInit. Convenience callback invoked for all intents and purposes before execution begins.

Reimplemented in cuvier.PopulationViewer.

4.20.3.107 void cuvier.Population.traceInversion (int *style*, Chromosome *chromosome*, int *len*) [inline, protected]

Method traceInversion. Public callback for the mutation/inversion type events.

Parameters:
style INVSTYLE_MUTATION, INVSTYLE_1POINT, INVSTYLE_2POINT, INVSTYLE_INSERTION, INVSTYLE_RECIPROCALEXCHANGE. INVSTYLE_NONE in case of error.

chromosome Chromosome

len int number of mutations or length of inversion site.

Note:
This callback is not invoked if no mutation/inversion style is selected.

Reimplemented in cuvier.PopulationViewer.

4.20.3.108 synchronized boolean cuvier.Population.traceIteration (Stats *stats*) [inline, protected]

Method traceIteration. Callback invoked at each generation.

Parameters:
stats Stats statistics for iteration.

Reimplemented in cuvier.PopulationViewer.

4.20.3.109 synchronized void cuvier.Population.traceResult () [inline, protected]

Method traceResult. Callback for one of the stop causes : error (-1), user(0), generations (1), threshold(2), stagnation(3). The cause is retrieved from the instance variable "cause".

Reimplemented in cuvier.PopulationViewer.

4.20.4 Member Data Documentation

4.20.4.1 final int cuvier.Population.BIT_ORIENTED = 0 [static]

Constant to set the problem as bit (nucleotide)-oriented.

General setting constants

4.20.4.2 final int cuvier.Population.FIXEDTYPE = 4 [static]

Constant to prevent changing the problem type set in the constructor.

General setting constants

4.20.4.3 final int cuvier.Population.GENE_ORIENTED = 1 [static]

Constant to set the problem as gene-oriented.

General setting constants

4.20.4.4 final int cuvier.Population.INVSTYLE_1POINT = 2 [static]

Constant used to set the mutation/inversion style to 1-point inversion.

Inversion styles constants (includes mutation)

4.20.4.5 final int cuvier.Population.INVSTYLE_2POINT = 3 [static]

Constant used to set the mutation/inversion style to 2-point inversion.

Inversion styles constants (includes mutation)

4.20.4.6 final int cuvier.Population.INVSTYLE_INSERTION = 5 [static]

Constant used to set the mutation/inversion style to insertion.

Inversion styles constants (includes mutation)

4.20.4.7 final int cuvier.Population.INVSTYLE_MUTATION = 1 [static]

Constant used to set the mutation/inversion style to mutation.

Inversion styles constants (includes mutation)

4.20.4.8 final int cuvier.Population.INVSTYLE_NONE = 0 [static]

Constant used to set the mutation/inversion style to no mutation/inversion.

Inversion styles constants (includes mutation)

4.20.4.9 final int cuvier.Population.INVSTYLE_RECIPROCALEXCHANGE = 4 [static]

Constant used to set the mutation/inversion style to reciprocal exchange.

Inversion styles constants (includes mutation)

4.20.4.10 final String cuvier.Population.invStyleNames[] [static]

Initial value:

```
{"None","Mutation","1-point inversion",
       "2-point inversion","Reciprocal exchange","Insertion"}
```

Inversion styles names

4.20.4.11 final int cuvier.Population.MAXIMIZATION = 0 [static]

Constant to set the problem as a maximization problem.

General setting constants

4.20.4.12 final int cuvier.Population.MINIMIZATION = 2 [static]

Constant to set the problem as a minimization problem.

General setting constants

4.20.4.13 final int cuvier.Population.STATE_IDLE = 0

Internal running states constants

4.20.4.14 final int cuvier.Population.STATE_PAUSED = 2

Internal running states constants

4.20.4.15 final int cuvier.Population.STATE_RUNNING = 1

Internal running states constants

4.20.4.16 final int cuvier.Population.STATE_STEPPING = 3

Internal running states constants

4.20.4.17 final int cuvier.Population.XSTYLE_1POINT = 1 [static]

Constant used to set the crossover style to 1-point crossover.

Crossover styles constants

4.20.4.18 final int cuvier.Population.XSTYLE_2POINT = 2 [static]

Constant used to set the crossover style to 2-point crossover.

Crossover styles constants

4.20.4.19 final int cuvier.Population.XSTYLE_CX = 5 [static]

Constant used to set the crossover style to Cycle Crossover.

Crossover styles constants

4.20.4.20 final int cuvier.Population.XSTYLE_NONE = 0 [static]

Constant used to set the crossover style to no crossover.

Crossover styles constants

4.20.4.21 final int cuvier.Population.XSTYLE_OX = 4 [static]

Constant used to set the crossover style to Order Crossover.

Crossover styles constants

4.20.4.22 final int cuvier.Population.XSTYLE_PMX = 3 [static]

Constant used to set the crossover style to Partially Matched Crossover.

Crossover styles constants

4.20.4.23 final String cuvier.Population.xStyleNames[] [static]

Initial value:

```
{"None","1-point crossover","2-point crossover",
        "Partially matched crossover","Order crossover","Cycle crossover"}
```

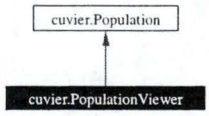
Crossover styles names

The documentation for this class was generated from the following file:

- Population.java

4.21 cuvier.PopulationViewer Class Reference

Inheritance diagram for cuvier.PopulationViewer:



4.21.1 Detailed Description

This class provides standard visualization for the Population engine.

Author:
Olivier Caudron

Date:
05-mars-03 17:38:14

Public Methods

- PopulationViewer (String name, Chromosome template)
- PopulationViewer (String name, int settings)
- PopulationViewer (String name)
- PopulationViewer (String name, int settings, Chromosome template)
- JPanel getVisualizationPanel ()
- void setVisualizationPanel (JPanel newPanel)
- boolean setSelection (Selection sel)
- boolean setMaximize (boolean maxi)
- void setUseThreshold (boolean set)
- void setUseStagnation (boolean set)
- void setUseGenerationCount (boolean set)
- void setStagnationThreshold (int value)
- void setKickCountThreshold (int value)
- void setKickRatio (int value)
- boolean setPopulationCount (int count)
- boolean setElitismQtyToSelect (int number)
- boolean setElitismPoolSize (int number)
- void setElitismUseRatios (boolean set)
- void setCrossOverStyle (int style)
- boolean setCrossOverRate (int newRate)
- void setInversionStyle (int style)
- boolean setInversionRate (int newRate)
- boolean setGenerationsThreshold (int iter)
- boolean setFitnessThreshold (Fitness newThreshold)

- boolean setFitnessThreshold (String newThreshold)
- void addToToolBar (JComponent c)
- void addSeparatorToToolBar ()
- synchronized void setTraceActive (boolean active)
- boolean isTraceActive ()
- synchronized void setTraceLevel (int level)
- int getTraceLevel ()
- void refreshVisualization ()
- void resetAll ()
- boolean loadSettings (String fname)

Public Attributes

- final int TRACE_NONE = 0

  Trace level: no trace.

- final int TRACE_NORMAL = 1

  Trace level: normal (summary information for every generation).

- final int TRACE_DETAILED = 2

  Trace level: detailed (normal + all individuals are listed at each generation).

- final int TRACE_EXAGGERATED = 3

  Trace level: exxagerated (detailed + every operator generates a trace).

Protected Methods

- synchronized void setRunning ()
- synchronized void setStepping ()
- synchronized void setPaused ()
- synchronized void switchPaused ()
- synchronized void setStopped ()
- synchronized void setState (int newState)
- synchronized void debug (String trace)
- void paintVisualization (JPanel pan, Graphics g)
- boolean traceInit (Stats stats)
- synchronized boolean traceIteration (Stats stats)
- void traceCrossover (int style, Chromosome c1, Chromosome c2, int pos1, int pos2)
- void traceInversion (int style, Chromosome chromosome, int len)
- void traceResult ()

4.21.2 Constructor & Destructor Documentation

4.21.2.1 cuvier.PopulationViewer.PopulationViewer (String *name*, Chromosome *template*) [inline]

See also:
cuvier.Population#Population(String, Chromosome)

4.21.2.2 cuvier.PopulationViewer.PopulationViewer (String *name*, int *settings*) [inline]

See also:
cuvier.Population#Population(String, int)

4.21.2.3 cuvier.PopulationViewer.PopulationViewer (String *name*) [inline]

See also:
cuvier.Population#Population(String)

4.21.2.4 cuvier.PopulationViewer.PopulationViewer (String *name*, int *settings*, Chromosome *template*) [inline]

See also:
cuvier.Population#Population(String, int, Chromosome)

4.21.3 Member Function Documentation

4.21.3.1 void cuvier.PopulationViewer.addSeparatorToToolBar () [inline]

Adds a separator to the toolbar.

4.21.3.2 void cuvier.PopulationViewer.addToToolBar (JComponent *c*) [inline]

Method addToToolBar.

Parameters:
c component to add to the toolbar.

4.21.3.3 synchronized void cuvier.PopulationViewer.debug (String *trace*) [inline, protected]

See also:
cuvier.Population#debug(String)

Reimplemented from cuvier.Population.

4.21.3.4 int cuvier.PopulationViewer.getTraceLevel () [inline]

Gets the level of tracing (

See also:
TRACE_NONE,
TRACE_NORMAL,
TRACE_DETAILED,
TRACE_EXAGGERATED)

4.21.3.5 JPanel cuvier.PopulationViewer.getVisualizationPanel () [inline]

Returns a reference to the visualization panel to make it available to the descendants.

Returns:
JPanel

4.21.3.6 boolean cuvier.PopulationViewer.isTraceActive () [inline]

Checks if the trace is active.

4.21.3.7 boolean cuvier.PopulationViewer.loadSettings (String *fname*) [inline]

See also:
cuvier.Population#loadSettings(String)

Reimplemented from cuvier.Population.

4.21.3.8 void cuvier.PopulationViewer.paintVisualization (JPanel *pan*, Graphics *g*) [inline, protected]

Callback invoked by paintComponent of pVisual, used to provide visualization. It is naturally meant to be implemented in descendants (although not mandatory)

4.21.3.9 void cuvier.PopulationViewer.refreshVisualization () [inline]

Allows client application to force repainting of visualization panel.

4.21.3.10 void cuvier.PopulationViewer.resetAll () [inline]

Resets all visual statistics, then invokes traceInit.

4.21.3.11 boolean cuvier.PopulationViewer.setCrossOverRate (int *newRate*) [inline]

See also:

cuvier.Population#setCrossOverRate(int) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.12 void cuvier.PopulationViewer.setCrossOverStyle (int *style*) [inline]

See also:

cuvier.Population#setCrossOverStyle(int) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.13 boolean cuvier.PopulationViewer.setElitismPoolSize (int *number*) [inline]

See also:

cuvier.Population#setElitismPool(int) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.14 boolean cuvier.PopulationViewer.setElitismQtyToSelect (int *number*) [inline]

See also:

cuvier.Population#setElitismSurvivors(int) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.15 void cuvier.PopulationViewer.setElitismUseRatios (boolean *set*) [inline]

See also:

cuvier.Population#setElitismUsePercent(boolean) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.16 boolean cuvier.PopulationViewer.setFitnessThreshold (String *newThreshold*) [inline]

See also:

cuvier.Population#setFitnessThreshold(String) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.17 boolean cuvier.PopulationViewer.setFitnessThreshold (Fitness *newThreshold*) [inline]

See also:

cuvier.Population#setFitnessThreshold(Fitness) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.18 boolean cuvier.PopulationViewer.setGenerationsThreshold (int *iter*) [inline]

See also:

cuvier.Population#setIterations(int) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.19 boolean cuvier.PopulationViewer.setInversionRate (int *newRate*) [inline]

See also:

cuvier.Population#setInversionRate(int) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.20 void cuvier.PopulationViewer.setInversionStyle (int *style*) [inline]

See also:
cuvier.Population#setInversionStyle(int) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.21 void cuvier.PopulationViewer.setKickCountThreshold (int *value*) [inline]

See also:
cuvier.Population#setKickRepeat(int) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.22 void cuvier.PopulationViewer.setKickRatio (int *value*) [inline]

See also:
cuvier.Population#setKickRatio(int) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.23 boolean cuvier.PopulationViewer.setMaximize (boolean *maxi*) [inline]

See also:
cuvier.Population#setMaximize(boolean) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.24 synchronized void cuvier.PopulationViewer.setPaused () [inline, protected]

See also:
cuvier.Population#setPaused() This implementation calls ancestor and takes care of visual feedback

Reimplemented from cuvier.Population.

4.21.3.25 boolean cuvier.PopulationViewer.setPopulationCount (int *count*) [inline]

See also:
cuvier.Population#setPopulationCount(int) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.26 synchronized void cuvier.PopulationViewer.setRunning () [inline, protected]

See also:
cuvier.Population#setRunning() This implementation calls ancestor and takes care of visual feedback

Reimplemented from cuvier.Population.

4.21.3.27 boolean cuvier.PopulationViewer.setSelection (Selection *sel*) [inline]

See also:
cuvier.Population#setSelection(Selection) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.28 void cuvier.PopulationViewer.setStagnationThreshold (int *value*) [inline]

See also:
cuvier.Population#setStagnationThreshold(int) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.29 synchronized void cuvier.PopulationViewer.setState (int *newState*) [inline, protected]

See also:
cuvier.Population#setState(int)

Reimplemented from cuvier.Population.

4.21.3.30 synchronized void cuvier.PopulationViewer.setStepping () [inline, protected]

See also:

cuvier.Population#setStepping() This implementation calls ancestor and takes care of visual feedback

Reimplemented from cuvier.Population.

4.21.3.31 synchronized void cuvier.PopulationViewer.setStopped () [inline, protected]

See also:

cuvier.Population#setStopped() This implementation calls ancestor and takes care of visual feedback

Reimplemented from cuvier.Population.

4.21.3.32 synchronized void cuvier.PopulationViewer.setTraceActive (boolean *active*) [inline]

Activates or deactivates the trace.

4.21.3.33 synchronized void cuvier.PopulationViewer.setTraceLevel (int *level*) [inline]

Sets the level of tracing (

See also:

TRACE_NONE,
TRACE_NORMAL,
TRACE_DETAILED,
TRACE_EXAGGERATED)

4.21.3.34 void cuvier.PopulationViewer.setUseGenerationCount (boolean *set*) [inline]

See also:

cuvier.Population#setUseGenerationCount(boolean) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.35 void cuvier.PopulationViewer.setUseStagnation (boolean *set*) [inline]

See also:

cuvier.Population#setUseStagnation(boolean) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.36 void cuvier.PopulationViewer.setUseThreshold (boolean *set*) [inline]

See also:

cuvier.Population#setUseThreshold(boolean) Calls ancestor and updates statistics panel.

Reimplemented from cuvier.Population.

4.21.3.37 void cuvier.PopulationViewer.setVisualizationPanel (JPanel *newPanel*) [inline]

Replaces the visualization panel by providing a reference to another.

4.21.3.38 synchronized void cuvier.PopulationViewer.switchPaused () [inline, protected]

See also:

cuvier.Population#switchPaused() This implementation calls ancestor and takes care of visual feedback

Reimplemented from cuvier.Population.

4.21.3.39 void cuvier.PopulationViewer.traceCrossover (int *style*, Chromosome *c1*, Chromosome *c2*, int *pos1*, int *pos2*) [inline, protected]

See also:

cuvier.Population#traceCrossover(int, Chromosome, Chromosome, int, int)

Reimplemented from cuvier.Population.

4.21.3.40 boolean cuvier.PopulationViewer.traceInit (Stats *stats*) [inline, protected]

Method traceInit. Convenience callback invoked for all intents and purposes before execution begins.

Reimplemented from cuvier.Population.

4.21.3.41 void cuvier.PopulationViewer.traceInversion (int *style*, Chromosome *chromosome*, int *len*) [inline, protected]

See also:
cuvier.Population#traceInversion(int, Chromosome, int)

Reimplemented from cuvier.Population.

4.21.3.42 synchronized boolean cuvier.PopulationViewer.traceIteration (Stats *stats*) [inline, protected]

Method traceIteration. Callback invoked at each generation.

Parameters:
stats Stats statistics for iteration.

Reimplemented from cuvier.Population.

4.21.3.43 void cuvier.PopulationViewer.traceResult () [inline, protected]

See also:
cuvier.Population#traceResult()

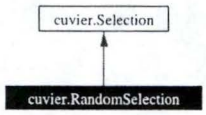
Reimplemented from cuvier.Population.

The documentation for this class was generated from the following file:

- PopulationViewer.java

4.22 cuvier.RandomSelection Class Reference

Inheritance diagram for cuvier.RandomSelection:



4.22.1 Detailed Description

Selection strategy that selects individuals randomly. To use exclusively with elitism!

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- boolean init (Population popul, boolean maximize)
- Chromosome[ ] run (int number, Population popul, boolean maximize)
- String getName ()
- String getDescription ()
- void deserializeSettings (String set)
- String serializeSettings ()

4.22.2 Member Function Documentation

4.22.2.1 void cuvier.RandomSelection.deserializeSettings (String *set*) [inline, virtual]

See also:
cuvier.Selection#deserializeSettings(String)

Implements cuvier.Selection.

4.22.2.2 String cuvier.RandomSelection.getDescription () [inline, virtual]

See also:
cuvier.Selection#getDescription()

Implements cuvier.Selection.

4.22.2.3 String cuvier.RandomSelection.getName () [inline, virtual]

See also:
java.awt.Component#getName()

Implements cuvier.Selection.

4.22.2.4 boolean cuvier.RandomSelection.init (Population *popul*, boolean *maximize*) [inline, virtual]

Init can be implemented in concrete classes to perform Selection-specific initialization.

Implements cuvier.Selection.

4.22.2.5 Chromosome [ ] cuvier.RandomSelection.run (int *number*, Population *popul*, boolean *maximize*) [inline, virtual]

Run is the method that performs the selection.

Returns:
Chromosome[]. If length<>number or if returns null, execution stops.

Implements cuvier.Selection.

4.22.2.6 String cuvier.RandomSelection.serializeSettings () [inline, virtual]

See also:
cuvier.Selection#serializeSettings()

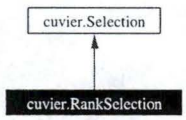
Implements cuvier.Selection.

The documentation for this class was generated from the following file:

- RandomSelection.java

4.23 cuvier.RankSelection Class Reference

Inheritance diagram for cuvier.RankSelection:



4.23.1 Detailed Description

A selection strategy that selects individuals with a probability proportional to their rank in the population (based on fitness) instead of the fitness itself.

The rank selection has the individuals sorted by fitness. The probability of an individual being selected is proportional to its rank, not its fitness. This tends to level irregularities

in the fitness distribution in the population, especially at the end of the processing when approaching the optimum, where all indiviuals have fitnesses that differ little. It can improve the algorithm by both slowing down convergence and preventing stagnation.

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- RankSelection ()
- String getName ()
- String getDescription ()
- synchronized boolean init (Population popul, boolean maximize)
- Chromosome[ ] runLinear (int number, Population popul, boolean maximize)
- Chromosome[ ] runExponential (int number, Population popul, boolean maximize)
- synchronized Chromosome[ ] run (int number, Population popul, boolean maximize)
- void deserializeSettings (String set)
- String serializeSettings ()

4.23.2 Constructor & Destructor Documentation

4.23.2.1 cuvier.RankSelection.RankSelection () [inline]

Constructor. Calls ancestor and prepares visualization of property.

4.23.3 Member Function Documentation

4.23.3.1 void cuvier.RankSelection.deserializeSettings (String *set*) [inline, virtual]

See also:
cuvier.Selection#deserializeSettings(String)

Implements cuvier.Selection.

4.23.3.2 String cuvier.RankSelection.getDescription () [inline, virtual]

See also:
cuvier.Selection#getDescription()

Implements cuvier.Selection.

4.23.3.3 String cuvier.RankSelection.getName () [inline, virtual]

See also:
java.awt.Component#getName()

Implements cuvier.Selection.

4.23.3.4 synchronized boolean cuvier.RankSelection.init (Population *popul*, boolean *maximize*) [inline, virtual]

See also:
cuvier.Selection#init(Population, boolean)

Implements cuvier.Selection.

4.23.3.5 synchronized Chromosome [] cuvier.RankSelection.run (int *number*, Population *popul*, boolean *maximize*) [inline, virtual]

Overrides Selection.run(...). Based on the settings, invokes runLinear() or runExponential().

Implements cuvier.Selection.

4.23.3.6 Chromosome [] cuvier.RankSelection.runExponential (int *number*, Population *popul*, boolean *maximize*) [inline]

method to execute the selection based on an exponential model.

4.23.3.7 Chromosome [] cuvier.RankSelection.runLinear (int *number*, Population *popul*, boolean *maximize*) [inline]

Method to execute the selection based on a linear model.

Todo
The "a" setting is useless.

4.23.3.8 String cuvier.RankSelection.serializeSettings () [inline, virtual]

See also:
cuvier.Selection#serializeSettings()

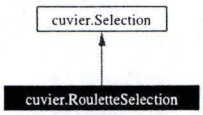
Implements cuvier.Selection.

The documentation for this class was generated from the following file:

- RankSelection.java

4.24 cuvier.RouletteSelection Class Reference

Inheritance diagram for cuvier.RouletteSelection:



4.24.1 Detailed Description

Roulette wheel selection or fitness-proportionate selection is the most classical selection scheme described by David Goldberg. It gives individuals a selection probability that is proportional to their fitness.

Public Methods

- RouletteSelection ()
- boolean init (Population popul, boolean maximize)
- Chromosome[ ] run (int number, Population popul, boolean maximize)
- String getName ()
- String getDescription ()
- void deserializeSettings (String set)
- String serializeSettings ()

4.24.2 Constructor & Destructor Documentation

4.24.2.1 cuvier.RouletteSelection.RouletteSelection () [inline]

As there are no specific settings in this selection strategy, the constructor only invokes the ancestor's.

4.24.3 Member Function Documentation

4.24.3.1 void cuvier.RouletteSelection.deserializeSettings (String *set*) [inline, virtual]

See also:
cuvier.Selection#deserializeSettings(String)

Implements cuvier.Selection.

4.24.3.2 String cuvier.RouletteSelection.getDescription () [inline, virtual]

See also:
cuvier.Selection#getDescription()

Implements cuvier.Selection.

4.24.3.3 String cuvier.RouletteSelection.getName () [inline, virtual]

See also:
java.awt.Component#getName()

Implements cuvier.Selection.

4.24.3.4 boolean cuvier.RouletteSelection.init (Population *popul*, boolean *maximize*) [inline, virtual]

Init is not needed by this implementation.

Implements cuvier.Selection.

4.24.3.5 Chromosome [] cuvier.RouletteSelection.run (int *number*, Population *popul*, boolean *maximize*) [inline, virtual]

Note:
This implementation requires summing the fitnesses. It may throw an overflow if the fitnesses are very large compared to the maximum value of the fitness type.

Warning:

Other selection schemes may accept mixed (positive and negative) values (for instance rank, tournament), this one only accepts positive values! A check is performed on fitness values, if one is negative, this method returns null.

Returns:

Chromosome[]. If length<>number or if returns null, execution stops.

Implements cuvier.Selection.

4.24.3.6 String cuvier.RouletteSelection.serializeSettings () [inline, virtual]

See also:

cuvier.Selection#serializeSettings()

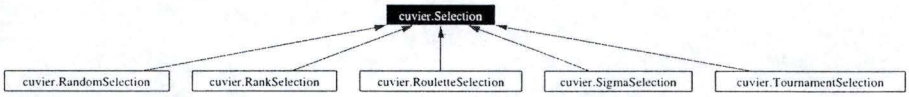
Implements cuvier.Selection.

The documentation for this class was generated from the following file:

- RouletteSelection.java

4.25 cuvier.Selection Class Reference

Inheritance diagram for cuvier.Selection:



4.25.1 Detailed Description

Base class for all selection strategy implementing classes.

Public Methods

- Selection ()
- abstract boolean init (Population popul, boolean maximize)
- abstract Chromosome[ ] run (int number, Population popul, boolean maximize)
- abstract String getName ()
- abstract String getDescription ()
- String getProperty (String propName)

- String toString ()
- void setProperty (String propName, String propValue)
- void setProperty (String propName, int propValue)
- void setProperty (String propName, boolean propValue)
- abstract String serializeSettings ()
- abstract void deserializeSettings (String set)

Protected Attributes

- java.util.Properties props = new java.util.Properties()

  Holder of properties of the selection, for convenience.

4.25.2 Constructor & Destructor Documentation

4.25.2.1 cuvier.Selection.Selection () [inline]

The base constructor of Selection prepares the JPanel to receive specialized setting components, and also adds the information button.

Warning:

It must be invoked by the descendant's constructor!!! (otherwise the layout will not be correct, and the informative button will not be present).

4.25.3 Member Function Documentation

4.25.3.1 abstract void cuvier.Selection.deserializeSettings (String *set*) [pure virtual]

This method is used to deserialize settings previously serialized using serializeSettings and apply them to the current instance.

Parameters:

set a serialized version of the settings that this method must

Implemented in cuvier.RandomSelection, cuvier.RankSelection, cuvier.RouletteSelection, cuvier.SigmaSelection, and cuvier.TournamentSelection.

4.25.3.2 abstract String cuvier.Selection.getDescription () [pure virtual]

This method is used to give an access to a description of the strategy as set by the concrete descendants of this class.

Returns:
String The description of the strategy.

Implemented in cuvier.RandomSelection, cuvier.RankSelection, cuvier.RouletteSelection, cuvier.SigmaSelection, and cuvier.TournamentSelection.

4.25.3.3 abstract String cuvier.Selection.getName () [pure virtual]

This method is used to give an access to the name of the strategy as set by the concrete descendants of this class.

Returns:
String The name of the strategy.

Implemented in cuvier.RandomSelection, cuvier.RankSelection, cuvier.RouletteSelection, cuvier.SigmaSelection, and cuvier.TournamentSelection.

4.25.3.4 String cuvier.Selection.getProperty (String *propName*) [inline]

Allows access to a property of the selection strategy using its name (as a String).

Parameters:
propName String The name of the property

Returns:
String The value of the property.

Note:
This may not be implemented fully in descendants.

4.25.3.5 abstract boolean cuvier.Selection.init (Population *popul*, boolean *maximize*) [pure virtual]

Init can be implemented in concrete classes to perform Selection-specific initialization.

Implemented in cuvier.RandomSelection, cuvier.RankSelection, cuvier.RouletteSelection, cuvier.SigmaSelection, and cuvier.TournamentSelection.

4.25.3.6 abstract Chromosome [ ] cuvier.Selection.run (int *number*, Population *popul*, boolean *maximize*) [pure virtual]

Run is the method that performs the selection. Its contract is to provide "number" Chromosomes from Population "popul" in an array of references (returned by the method). There can be duplicate references in the array, the next generation will be created using copies of the Chromosomes. This base implementation choses surviving Chromosomes randomly and is meant solely to use with elitism.

Warning:
Descendants of this class must NOT call this ancestor !!!

Returns:
Chromosome[]. If length<>number or if returns null, execution stops.

Implemented in cuvier.RandomSelection, cuvier.RankSelection, cuvier.RouletteSelection, cuvier.SigmaSelection, and cuvier.TournamentSelection.

4.25.3.7 abstract String cuvier.Selection.serializeSettings () [pure virtual]

This method is used to serialize the settings of the selection in order to store them.

Returns:
String contains a serialized version of the settings of this selection. The way the settings are coded are left to the selection programmer, and must only be decodable using deserializeSettings.

Note:
May return an empty string or null.

Implemented in cuvier.RandomSelection, cuvier.RankSelection, cuvier.RouletteSelection, cuvier.SigmaSelection, and cuvier.TournamentSelection.

4.25.3.8 void cuvier.Selection.setProperty (String *propName*, boolean *propValue*) [inline]

Sets one of the properties of the Selection, given as a boolean, in a Property object.

Parameters:
propName String Name of the property
propValue int Value of the property

4.25.3.9 void cuvier.Selection.setProperty (String *propName*, int *propValue*) [inline]

Sets one of the properties of the Selection, given as an int, in a Property object.

Parameters:
propName String Name of the property
propValue int Value of the property

4.25.3.10 void cuvier.Selection.setProperty (String *propName*, String *propValue*) [inline]

Sets one of the properties of the Selection, given as a String, in a Property object.

Parameters:

propName String Name of the property

propValue String Value of the property

4.25.3.11 String cuvier.Selection.toString () [inline]

toString is overriden from Object to invoke getName()

The documentation for this class was generated from the following file:

- Selection.java

4.26 cuvier::util.Serie Class Reference

4.26.1 Detailed Description

Class used to hold series in a Graph. Used internally by Graph, but may be used as a reference when handling directly the contents of a serie.

Public Methods

- Serie (String name, Color col)
- void setColor (Color col)
- void setName (String name)
- boolean add (double l)
- Color getColor ()
- String getName ()
- Iterator iterator ()
- int size ()
- double get (int index)
- double getMin ()
- double getMax ()
- void clear ()
- String toString ()

4.26.2 Constructor & Destructor Documentation

4.26.2.1 cuvier.util.Serie.Serie (String *name*, Color *col*) [inline]

Constructor: creates an instance with a name and a color.

4.26.3 Member Function Documentation

4.26.3.1 boolean cuvier.util.Serie.add (double *l*) [inline]

Adds a point to this serie.

4.26.3.2 void cuvier.util.Serie.clear () [inline]

Removes all entries in this Serie and sets the maximum and minimum to their respective INFINITies.

4.26.3.3 double cuvier.util.Serie.get (int *index*) [inline]

Get a point at a given index in this Serie.

4.26.3.4 Color cuvier.util.Serie.getColor () [inline]

Returns the color of this Serie.

4.26.3.5 double cuvier.util.Serie.getMax () [inline]

Retrieves the maximum point in this Serie.

4.26.3.6 double cuvier.util.Serie.getMin () [inline]

Retrieves the minimum point in this Serie.

4.26.3.7 String cuvier.util.Serie.getName () [inline]

Returns the name of this Serie.

4.26.3.8 Iterator cuvier.util.Serie.iterator () [inline]

Returns an iterator on the items in this Serie.

Note:
The iterator returns objects of instance Double.

4.26.3.9 void cuvier.util.Serie.setColor (Color *col*) [inline]

Changes the color of this Serie.

4.26.3.10 void cuvier.util.Serie.setName (String *name*) [inline]

Changes the name of this Serie.

4.26.3.11 int cuvier.util.Serie.size () [inline]

Returns the number of points in this Serie.

4.26.3.12 String cuvier.util.Serie.toString () [inline]

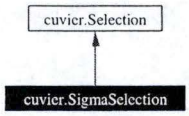
Returns a LF-separated list of values

The documentation for this class was generated from the following file:

- Graph.java

4.27 cuvier.SigmaSelection Class Reference

Inheritance diagram for cuvier.SigmaSelection:



4.27.1 Detailed Description

Sigma selection is a selection strategy that gives a probability of selection to individuals that is propotional to their fitness but also corrected according to the standard deviation of the fitnesses of the population, to beter balance the pression of selection during the execution of the algorithm.

Warning:

This selection strategy requires a fitness type that correctly maps to a double!
This selection strategy requires positive fitnesses!

Public Methods

- SigmaSelection ()
- boolean init (Population popul, boolean maximize)
- Chromosome[ ] run (int number, Population popul, boolean maximize)
- String getName ()
- String getDescription ()
- void deserializeSettings (String set)
- String serializeSettings ()

4.27.2 Constructor & Destructor Documentation

4.27.2.1 cuvier.SigmaSelection.SigmaSelection () [inline]

Constructor for SigmaSelection.

4.27.3 Member Function Documentation

4.27.3.1 void cuvier.SigmaSelection.deserializeSettings (String *set*) [inline, virtual]

See also:
cuvier.Selection#deserializeSettings(String)

Implements cuvier.Selection.

4.27.3.2 String cuvier.SigmaSelection.getDescription () [inline, virtual]

See also:
cuvier.Selection#getDescription()

Implements cuvier.Selection.

4.27.3.3 String cuvier.SigmaSelection.getName () [inline, virtual]

See also:
java.awt.Component#getName()

Implements cuvier.Selection.

4.27.3.4 boolean cuvier.SigmaSelection.init (Population *popul*, boolean *maximize*) [inline, virtual]

See also:
cuvier.Selection#init(Population, boolean)

Implements cuvier.Selection.

4.27.3.5 Chromosome [] cuvier.SigmaSelection.run (int *number*, Population *popul*, boolean *maximize*) [inline, virtual]

See also:
cuvier.Selection#run(int, Population, boolean)

Implements cuvier.Selection.

4.27.3.6 String cuvier.SigmaSelection.serializeSettings () [inline, virtual]

See also:
cuvier.Selection#serializeSettings()

Implements cuvier.Selection.

The documentation for this class was generated from the following file:

- SigmaSelection.java

4.28 cuvier.Stats Class Reference

4.28.1 Detailed Description

Stats provides a central repository for statistics used by the Cuvier framework. It has mainly an internal use.

Public Methods

- void initialize (Fitness fitnessTemplate, boolean maximize)
- void initialize (boolean maximize)
- void setStatsForIteration (Chromosome[ ] chromes, int iteration)
- Fitness getBestFitnessForIteration ()
- Chromosome getBestIndividual ()
- int getBestFitnessIteration ()
- double getAverageFitnessForIteration ()
- double getBestOverallAverage ()
- int getBestAverageIteration ()
- Fitness getBestOverallFitness ()
- Chromosome getBestOverallIndividual ()
- Fitness getStagnationFitness ()
- void resetStagnation ()
- int getStagnationCount ()

4.28.2 Member Function Documentation

4.28.2.1 double cuvier.Stats.getAverageFitnessForIteration () [inline]

Returns the averageFitnessForIteration.

Returns:
double

4.28.2.2 int cuvier.Stats.getBestAverageIteration () [inline]

Returns the bestAverageIteration.

Returns:
int

4.28.2.3 Fitness cuvier.Stats.getBestFitnessForIteration () [inline]

Returns the bestFitnessForIteration.

Returns:
Fitness

4.28.2.4 int cuvier.Stats.getBestFitnessIteration () [inline]

Returns the bestFitnessIteration.

Returns:
int

4.28.2.5 Chromosome cuvier.Stats.getBestIndividual () [inline]

Returns the bestIndividual.

Returns:
Chromosome

4.28.2.6 double cuvier.Stats.getBestOverallAverage () [inline]

Returns the bestOverallAverage.

Returns:
double

4.28.2.7 Fitness cuvier.Stats.getBestOverallFitness () [inline]

Returns the bestOverallFitness.

Returns:
Fitness

4.28.2.8 Chromosome cuvier.Stats.getBestOverallIndividual () [inline]

Returns the bestOverallIndividual.

Returns:
Chromosome

4.28.2.9 int cuvier.Stats.getStagnationCount () [inline]

Returns the stagnationCount.

Returns:
int

4.28.2.10 Fitness cuvier.Stats.getStagnationFitness () [inline]

Returns the stagnationFitness.

Returns:
Fitness

4.28.2.11 void cuvier.Stats.initialize (boolean *maximize*) [inline]

Method initialize.

Parameters:
maximize

4.28.2.12 void cuvier.Stats.initialize (Fitness *fitnessTemplate*, boolean *maximize*) [inline]

Method initialize.

Parameters:

fitnessTemplate

maximize

4.28.2.13 void cuvier.Stats.resetStagnation () [inline]

Method resetStagnationFitness.

4.28.2.14 void cuvier.Stats.setStatsForIteration (Chromosome *chromes*[ ], int *iteration*) [inline]

Method setStatsForIteration.

Parameters:

chromes

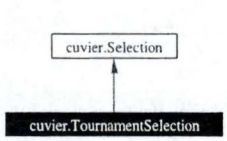
iteration

The documentation for this class was generated from the following file:

- Stats.java

4.29 cuvier.TournamentSelection Class Reference

Inheritance diagram for cuvier.TournamentSelection:



4.29.1 Detailed Description

Tournament selection scheme generates a new population based on **tournaments** between 2 individuals randomly chosen in the original population. The fittest individual in the tournament gets more chance to become a parent of the next generation. The advantage given to the fittest individual is a fixed ratio.

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- TournamentSelection ()
- TournamentSelection (int newAdvantage)
- void setAdvantage (int newAdvantage)
- String getName ()
- String getDescription ()
- int getAdvantage ()
- synchronized boolean init (Population popul, boolean maximize)
- synchronized Chromosome[ ] run (int number, Population popul, boolean maximize)
- void deserializeSettings (String set)
- String serializeSettings ()

4.29.2 Constructor & Destructor Documentation

4.29.2.1 cuvier.TournamentSelection.TournamentSelection () [inline]

This constructor calls the ancestor and prepares the visualization of the settings. The evolutionary advantage is kept at the default (50%)

4.29.2.2 cuvier.TournamentSelection.TournamentSelection (int *newAdvantage*) [inline]

This constructor calls the ancestor and prepares the visualization of the settings.

Parameters:
newAdvantage int Sets the evolutionary advantage.

4.29.3 Member Function Documentation

4.29.3.1 void cuvier.TournamentSelection.deserializeSettings (String *set*) [inline, virtual]

See also:
cuvier.Selection#deserializeSettings(String)

Implements cuvier.Selection.

4.29.3.2 int cuvier.TournamentSelection.getAdvantage () [inline]

Returns:
int The current value of the evolutionary advantage ratio.

4.29.3.3 String cuvier.TournamentSelection.getDescription () [inline, virtual]

See also:
cuvier.Selection#getDescription()

Implements cuvier.Selection.

4.29.3.4 String cuvier.TournamentSelection.getName () [inline, virtual]

See also:
java.awt.Component#getName()

Implements cuvier.Selection.

4.29.3.5 synchronized boolean cuvier.TournamentSelection.init (Population *popul*, boolean *maximize*) [inline, virtual]

See also:
cuvier.Selection#init(Population, boolean)

Implements cuvier.Selection.

4.29.3.6 synchronized Chromosome [] cuvier.TournamentSelection.run (int *number*, Population *popul*, boolean *maximize*) [inline, virtual]

See also:
cuvier.Selection#run(int, Population, boolean)

Implements cuvier.Selection.

4.29.3.7 String cuvier.TournamentSelection.serializeSettings () [inline, virtual]

See also:
cuvier.Selection#serializeSettings()

Implements cuvier.Selection.

4.29.3.8 void cuvier.TournamentSelection.setAdvantage (int *newAdvantage*) [inline]

Sets the evolutionary advantage ratio.

Parameters:
newAdvantage int Sets the evolutionary advantage.

The documentation for this class was generated from the following file:

- TournamentSelection.java

4.30 cuvier.TraceI Class Reference

4.30.1 Detailed Description

Convenience class used to store trace information on the mutations and inversions.

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- void set (int style, Chromosome c, int len)

Public Attributes

- Chromosome chromosome
  Holds a reference to the chromosome that has been inverted/mutated.
- int style = Population.INVSTYLE_NONE
  Style of inversion/mutation traced.
- int length = -1
  Length of inversion or number of mutations.

4.30.2 Member Function Documentation

4.30.2.1 void cuvier.TraceI.set (int *style*, Chromosome *c*, int *len*) [inline]

Sets the internal values of this trace object.

The documentation for this class was generated from the following file:

- Population.java

4.31 cuvier.TraceX Class Reference

4.31.1 Detailed Description

Convenience class used to store trace information on the crossovers.

Author:
Olivier Caudron

Date:
2002-2003

Public Methods

- void set (int style, Chromosome c1, Chromosome c2, int p1, int p2)

Public Attributes

- Chromosome chromosome1
  Holds a reference to the 1st chromosome in a crossover.
- Chromosome chromosome2
  Holds a reference to the 2nd chromosome in a crossover.
- int style = Population.XSTYLE_NONE
  Style of the crossover traced.
- int pos1 = -1
  First position of crossover.
- int pos2 = -1
  Second position of crossover.

4.31.2 Member Function Documentation

4.31.2.1 void cuvier.TraceX.set (int *style*, Chromosome *c1*, Chromosome *c2*, int *p1*, int *p2*) [inline]

Sets the internal values of this trace object.

The documentation for this class was generated from the following file:

- Population.java

4.32 cuvier::util.Utils Class Reference

4.32.1 Detailed Description

This static class delivers some application-wide utilities (tracing and random numbers).

Static Public Methods

- void setTraceLevel (int level)
- int getTraceLevel ()
- void traceNormal (String s)
- void norm (String s)
- void traceVerbose (String s)
- void verb (String s)
- final long randLong (long max)
- final ImageIcon loadSystemIcon (String name)

Static Public Attributes

- final int TRACE_QUIET = 0
- final int TRACE_NORMAL = 1
- final int TRACE_VERBOSE = 2
- int traceLevel = TRACE_NORMAL
- final java.util.Random rand = new java.util.Random()

4.32.2 Member Function Documentation

4.32.2.1 int cuvier.util.Utils.getTraceLevel () [inline, static]

Get the current trace level

4.32.2.2 final ImageIcon cuvier.util.Utils.loadSystemIcon (String *name*) [inline, static]

Allows loading an icon whether the application resides in a directory or a jar file. Tries first to load from the file system, then from the resources of the application (assuming it is in a jar file). This method assumes that the system icons are located in icons/ (a highest-level directory of the jar file, equivalent to /icons/, or a subdirectory of the current directory, equivalent to ./icons/)

Parameters:

name String Name of the icon to load without path.

Returns:

ImageIcon The created icon resource, null in case of error.

4.32.2.3 void cuvier.util.Utils.norm (String *s*) [inline, static]

Convenience shortcut for traceNormal

4.32.2.4 final long cuvier.util.Utils.randLong (long *max*) [inline, static]

Convenience method that returns a random long that is bonded. This method is not available from java.util.Random.

4.32.2.5 void cuvier.util.Utils.setTraceLevel (int *level*) [inline, static]

Modify the trace level

4.32.2.6 void cuvier.util.Utils.traceNormal (String *s*) [inline, static]

Display trace on console if the level is normal or above

4.32.2.7 void cuvier.util.Utils.traceVerbose (String *s*) [inline, static]

Display trace on console if the level is verbose

4.32.2.8 void cuvier.util.Utils.verb (String *s*) [inline, static]

Convenience shortcut for traceVerbose

4.32.3 Member Data Documentation

4.32.3.1 final java.util.Random cuvier.util.Utils.rand = new java.util.Random() [static]

Application-wide randomizer. Picking up randoms at this unique source guarantees there won't be funny patterns in the random values.

4.32.3.2 final int cuvier.util.Utils.TRACE_NORMAL = 1 [static]

Constants used to set the trace level

4.32.3.3 final int cuvier.util.Utils.TRACE_QUIET = 0 [static]

Constants used to set the trace level

4.32.3.4 final int cuvier.util.Utils.TRACE_VERBOSE = 2 [static]

Constants used to set the trace level

4.32.3.5 int cuvier.util.Utils.traceLevel = TRACE_NORMAL [static]

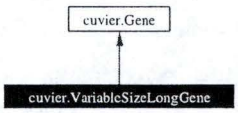
Trace level for application

The documentation for this class was generated from the following file:

- Utils.java

4.33 cuvier.VariableSizeLongGene Class Reference

Inheritance diagram for cuvier.VariableSizeLongGene:



4.33.1 Detailed Description

Implements a Gene with a decimal value coded on a given number of bits, that may vary.

Author:
Olivier Caudron

Date:
06-mars-03 18:25:00

Public Methods

- VariableSizeLongGene ()
- VariableSizeLongGene (int size)
- VariableSizeLongGene (boolean[ ] bits)
- byte byteValue ()
- short shortValue ()
- int intValue ()
- long longValue ()
- long signedLongValue ()
- float floatValue ()
- double doubleValue ()
- void setValue (boolean[ ] newVal)
- boolean setValue (String newVal)
- void setBits (java.util.BitSet bits)
- int setBits (java.util.BitSet bits, int pos)
- boolean exchange (Gene g)
- Gene copy ()
- boolean copyInto (Gene g)
- int getLength ()
- boolean[ ] getBitField ()
- int applyGene (java.util.BitSet bits, int pos)
- boolean getBitAt (int pos)
- int compareTo (Object o)
- String toString ()

4.33.2 Constructor & Destructor Documentation

4.33.2.1 cuvier.VariableSizeLongGene.VariableSizeLongGene () [inline]

No-parameter constructor; internal value set to 8 booleans all set to false (default for boolean).

4.33.2.2 cuvier.VariableSizeLongGene.VariableSizeLongGene (int *size*) [inline]

Constructor that only sets the size of the internal representation. All nucleotides are set to false (default for boolean).

4.33.2.3 cuvier.VariableSizeLongGene.VariableSizeLongGene (booleanbits[ ]) [inline]

Constructor that initializes the bit field based on a given bit field.

Note:
The instance copies the provided bit field, and does not use the reference directly (that would be dangerous).

4.33.3 Member Function Documentation

4.33.3.1 int cuvier.VariableSizeLongGene.applyGene (java.util.BitSet *bits*, int *pos*) [inline]

Apply Gene on a bit field at the given offset

Returns:
offset that follows the Gene in the chromosome Note if the bit list offset has not enough bits for Gene, the remaining bits are not applied

Implements cuvier.Gene.

4.33.3.2 byte cuvier.VariableSizeLongGene.byteValue () [inline]

Returns a byte representation of a gene. Not relevant to this implementation.

Exceptions:
ClassCastException always in this implementation.

Implements cuvier.Gene.

4.33.3.3 int cuvier.VariableSizeLongGene.compareTo (Object *o*) [inline]

Comparison method (inherited from interface Comparable) Note : the important point is to report 0 if all bits have the same values. However Comparable imposes that the comparison be coherent, so we say that the gene with the highest-order bit set to 1 is greater than the other. Bit order is counted left-to right (that is, lower indexes in the internal array are lowest-order bits)

Implements cuvier.Gene.

4.33.3.4 Gene cuvier.VariableSizeLongGene.copy () [inline]

Creates a copy of this Gene.

Returns:
a newly-allocated Gene.

Implements cuvier.Gene.

4.33.3.5 boolean cuvier.VariableSizeLongGene.copyInto (Gene *g*) [inline]

Copies this Gene into another without allocation. The target Gene must be pre-allocated.

Returns:
false if target is not pre-allocated or target is not a BitFieldGene.

Implements cuvier.Gene.

4.33.3.6 double cuvier.VariableSizeLongGene.doubleValue () [inline]

Returns a double representation of a gene. Not relevant to this implementation.

Exceptions:
ClassCastException always in this implementation.

Implements cuvier.Gene.

4.33.3.7 boolean cuvier.VariableSizeLongGene.exchange (Gene *g*) [inline]

Switch the contents of the genes with no new allocation

Returns:
true if succeeded.

Implements cuvier.Gene.

4.33.3.8 float cuvier.VariableSizeLongGene.floatValue () [inline]

Returns a float representation of a gene. Not relevant to this implementation.

Exceptions:
ClassCastException always in this implementation.

Implements cuvier.Gene.

4.33.3.9 boolean cuvier.VariableSizeLongGene.getBitAt (int *pos*) [inline]

Returns a bit at a given position as a boolean; 0 if pos is out of range !!!

Implements cuvier.Gene.

4.33.3.10 boolean [] cuvier.VariableSizeLongGene.getBitField () [inline]

Returns an array of booleans equivalent to the bit values

Implements cuvier.Gene.

4.33.3.11 int cuvier.VariableSizeLongGene.getLength () [inline]

Returns the length of the Gene, that is, the size of the representation of the gene

Implements cuvier.Gene.

4.33.3.12 int cuvier.VariableSizeLongGene.intValue () [inline]

Returns an int representation of a gene. Not relevant to this implementation.

Exceptions:

ClassCastException always in this implementation.

Implements cuvier.Gene.

4.33.3.13 long cuvier.VariableSizeLongGene.longValue () [inline]

Returns a long representation of a gene

Note:

This implementation is always signed, hence it only invokes signedLongValue().

Exceptions:

ClassCastException in case of bit field size issue.

Implements cuvier.Gene.

4.33.3.14 int cuvier.VariableSizeLongGene.setBits (java.util.BitSet *bits*, int *pos*) [inline]

Apply the provided array of boolean to the gene's internal value bitwise, starting at a given position in the provided array.

Returns:

Position in the bit field just after the Gene

Note:

The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implements cuvier.Gene.

4.33.3.15 void cuvier.VariableSizeLongGene.setBits (java.util.BitSet *bits*) [inline]

Apply the provided array of boolean to the gene's internal value bitwise.

Note:

The length must be coherent : if it is too long, values beyond the array size will be ignored, and if too short, only the first values will be updated

Implements cuvier.Gene.

4.33.3.16 boolean cuvier.VariableSizeLongGene.setValue (String *newVal*) [inline]

Tries to interpret the value passed as a string, by assigning "false" to zeroes and "true" otherwise.

Warning:
if there is a size mismatch between the internal bit field and the string, the internal bit field will be reallocated with the new values !

Implements cuvier.Gene.

4.33.3.17 void cuvier.VariableSizeLongGene.setValue (boolean *newVal*[]) [inline]

Sets the internal value according to the provided bit field.

Note:
this may change the size of the bit field (longer or shorter).

4.33.3.18 short cuvier.VariableSizeLongGene.shortValue () [inline]

Returns a short representation of a gene. Not relevant to this implementation.

Exceptions:
ClassCastException always in this implementation.

Implements cuvier.Gene.

4.33.3.19 long cuvier.VariableSizeLongGene.signedLongValue () [inline]

Returns a long representation of a gene. If the internal value is shorter than 64 bits, the sign bit is expanded, that is, all bits between the end of the internal value and bit 64 are copies of the sign bit. That way, for instance, 11111110 on a 8-bit representation will be interpreted as -2L: bit 8 is considered as a sign bit and is expanded. Otherwise the value will be 254L. The choice is left to the user which version is suitable for him/her.

Note:
Conversion is not accepted if value is longer than 64 bits to avoid a loss of precision that might cause a meaningless return value.

Exceptions:
ClassCastException in case of bit field size issue.

Implements cuvier.Gene.

4.33.3.20 String cuvier.VariableSizeLongGene.toString () [inline]

Returns a numeric value of this gene.

Implements cuvier.Gene.

The documentation for this class was generated from the following file:

- VariableSizeLongGene.java

5 Cuvier Page Documentation

5.1 Todo List

Class cuvier.BitFieldGene implement internal representation as a java.util.BitSet for optimal memory usage

Member cuvier::util::ColumnedLister::updateUI () Implement.

Class cuvier::util.ControlledFlowLayout This layout manager only has the basic functionality required and could be enhanced and extended with more eye candy.

Class cuvier::util.Graph This class does not use the Java2D framework. It should be later extended to the richer functionality of this framework.

Is prepared for the zooming capability but doesn't have it yet - will be easier to implement using Java2D.

Member cuvier::util::Graph::save () NOT COMPLETE - DO NOT USE!!!

Class cuvier::util.JLongField short-circuit text-setting functionalities of JTextField

Member cuvier::util::Line::paintComponent (Graphics g) Modify to draw a line that is coherent with the PLAF.

Member cuvier::RankSelection::runLinear (int number, Population popul, boolean maximize) The "a" setting is useless.

Index

Annexe B

Données d'expérimentation du problème de Steiner

Viabilités moyennes Roulette Wheel

Gén.	Run 1 (511,426)	Run 2 (515,425)	Run 3 (526,424)	Run 4 (519,421)	Run 5 (515,407)	Moy.
1	19050,34	17547,87	18137,46	17221,14	16180,67	17627,5
2	16933,67	14374,82	16119,36	13602,69	14367,63	15079,63
3	15220,54	12645,1	13584,36	12073,92	11767,53	13058,29
4	13113,73	10850,17	12921,7	10941,4	10641,55	11693,71
5	11799,92	9706,37	11962,11	10283,29	10103,33	10771
6	11040,31	9287,36	11087,24	9823,67	9203,99	10088,51
7	10507,19	8778,86	10145,06	9361,21	8941,56	9546,78
8	10069,14	8494,9	9561,82	9163,86	8483,09	9154,56
9	9386,35	8399,45	9276,34	8925,19	8512,66	8900
10	9168,2	8139,43	8587,55	8564,33	8503,21	8592,55
11	8879,5	8078,71	8194,11	8074,36	8304,13	8306,16
12	8768,2	7937,78	7844,36	8045,1	8019,06	8122,9
13	8458,78	7959,49	7665,32	7801,49	8015,83	7980,18
14	8295,16	7883,71	7596,18	7727,98	7796,87	7859,98
15	8146,05	7815,11	7595,61	7733,71	7665,58	7791,21
16	7960,69	7721,92	7536,51	7566,71	7551,97	7667,56
17	7865,27	7657,4	7526,03	7488,85	7560,46	7619,6
18	8039,85	7720,66	7532,6	7375,73	7514,59	7636,69
19	7917,31	7763,5	7484,53	7367,57	7537,02	7613,99
20	7854,12	7705,67	7523,94	7361,2	7464,25	7581,83
21	7788,63	7695,11	7491,73	7398,15	7414,86	7557,7
22	7622,54	7657,31	7403,81	7427,53	7492,86	7520,81
23	7657,74	7750,03	7343	7383,07	7464,95	7519,76
24	7565,99	7757,82	7323,51	7406,85	7433,49	7497,53
25	7468,93	7747,22	7345,14	7323,59	7415,75	7460,13
26	7437,24	7815,5	7384,7	7350,01	7370,02	7471,5
27	7448,71	7706,92	7395,69	7407,25	7394,34	7470,58
28	7479,94	7597,71	7356,52	7361,1	7371,65	7433,39
29	7481,08	7629,42	7341,66	7302,04	7353,7	7421,58
30	7443,39	7696,6	7330,74	7319,36	7379,03	7433,82
31	7530,05	7676,62	7356,57	7433,93	7363,1	7472,05
32	7514,61	7577,56	7380,51	7374,84	7371,75	7443,85
33	7547,07	7527,87	7484,55	7332,99	7300,91	7438,68
34	7527,66	7500,06	7404,94	7377,03	7338,67	7429,67
35	7579,83	7576,07	7526,19	7397,93	7317,38	7479,48
36	7555,04	7564,21	7762,29	7423,79	7376,8	7536,43
37	7513,6	7588,58	7575,04	7379,22	7404,02	7492,09
38	7460,08	7631,48	7444,19	7356,38	7401,89	7458,8
39	7407,79	7570,63	7379,52	7338,44	7410,7	7421,42
40	7349,11	7499,37	7336,79	7260,89	7431,71	7375,57
41	7332,46	7457,43	7346,05	7334,46	7513,79	7396,84
42	7321,97	7425,79	7290,74	7259,87	7575,5	7374,77
43	7344,98	7516,53	7244	7254,84	7469,17	7365,9
44	7403,15	7422,91	7276,99	7291,32	7403,41	7359,56
45	7424,75	7481,26	7267,07	7314,19	7428,9	7383,23
46	7397,29	7491,69	7244,13	7422,02	7480,02	7407,03
47	7338,27	7449,42	7270,48	7294,16	7383,97	7347,26
48	7322,46	7458,99	7226,8	7266,02	7417,22	7338,3
49	7353,72	7435,52	7231,23	7234,8	7328,79	7316,81
50	7387,51	7390,47	7192,1	7306,37	7413,3	7337,95

Meilleures viabilités Roulette Wheel

Gén.	Run 1 (511,426)	Run 2 (515,425)	Run 3 (526,424)	Run 4 (519,421)	Run 5 (515,407)	Moy.
1	7134,01	7102,23	7077,24	7099,13	7087,34	7099,99
2	7134,01	7074,17	7077,24	7079,28	7087,34	7090,41
3	7134,01	7074,17	7077,24	7079,28	7087,34	7090,41
4	7134,01	7074,17	7066,32	7079,28	7087,34	7088,22
5	7134,01	7072,39	7066,32	7071,73	7087,34	7086,36
6	7093,48	7072,39	7066,32	7070,19	7087,34	7077,94
7	7063,6	7072,39	7066,32	7070,19	7087,34	7071,97
8	7063,6	7072,39	7066,32	7070,19	7087,34	7071,97
9	7063,6	7072,39	7066,32	7070,19	7082,13	7070,93
10	7063,6	7072,39	7064,98	7070,19	7072,87	7068,81
11	7063,6	7072,39	7064,98	7066,07	7072,87	7067,98
12	7063,6	7072,39	7064,98	7066,07	7072,87	7067,98
13	7063,6	7062,93	7064,98	7066,07	7072,87	7066,09
14	7063,6	7062,93	7064,98	7066,07	7072,28	7065,97
15	7063,6	7062,93	7064,98	7066,07	7072,28	7065,97
16	7063,6	7062,93	7063,95	7064,91	7071,16	7065,31
17	7063,6	7062,93	7063,95	7064,91	7070,09	7065,1
18	7063,6	7062,93	7063,95	7064,91	7070,09	7065,1
19	7063,6	7062,93	7063,95	7064,71	7070,09	7065,06
20	7063,6	7062,93	7063,95	7064,71	7070,09	7065,06
21	7063,6	7062,93	7063,95	7064,64	7070,09	7065,04
22	7063,6	7062,93	7063,95	7064,64	7070,09	7065,04
23	7063,6	7062,93	7063,95	7064,64	7070,09	7065,04
24	7063,6	7062,93	7063,95	7064,64	7070,09	7065,04
25	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
26	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
27	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
28	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
29	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
30	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
31	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
32	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
33	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
34	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
35	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
36	7063,6	7062,93	7063,95	7063,11	7070,09	7064,74
37	7063,54	7062,93	7063,95	7063,11	7069,37	7064,58
38	7063,54	7062,93	7063,95	7063,11	7069,37	7064,58
39	7063,54	7062,93	7063,95	7063,02	7069,37	7064,56
40	7063,54	7062,93	7063,95	7063,02	7069,37	7064,56
41	7063,54	7062,93	7063,95	7063,02	7069,37	7064,56
42	7063,54	7062,93	7063,95	7063,02	7069,37	7064,56
43	7063,54	7062,93	7063,95	7063,02	7069,37	7064,56
44	7063,54	7062,93	7063,95	7063,02	7069,37	7064,56
45	7063,54	7062,93	7063,95	7063,02	7069,37	7064,56
46	7063,54	7062,93	7063,95	7063,02	7069,37	7064,56
47	7063,54	7062,93	7063,95	7063,02	7069,37	7064,56
48	7063,54	7062,93	7063,77	7063,02	7069,37	7064,53
49	7063,54	7062,93	7063,77	7063,02	7069,37	7064,53
50	7063,54	7062,93	7063,77	7063,02	7069,37	7064,53

Viabilités moyennes Tournament

Gén.	Run 1 (519,415)	Run 2 (518,424)	Run 3 (519,424)	Run 4 (518,423)	Run 5 (518,424)	Moy.
1	17319,93	19257,16	17526,73	18059,84	18603,38	18153,41
2	15259,76	16544,84	15645,6	15630,76	16451,36	15906,46
3	12855,24	14546,54	13014,66	13290,94	14047,26	13550,93
4	11661,39	13828,5	11853,99	12450,06	11895,88	12337,96
5	10545,73	11559,76	10515,3	10963	11293,05	10975,37
6	9947,3	10607,83	9433,48	9760,12	10493,4	10048,42
7	9458,05	9469,91	8749,11	9125,86	9214,19	9203,42
8	8781,01	8803,6	8343,9	8246,76	8499,21	8534,89
9	8321,71	8298,49	8040,7	8147,86	8031,99	8168,15
10	8011,41	7847,14	7801,2	7925,96	7824,67	7882,08
11	7790,24	7695,11	7672,21	7681,53	7680,01	7703,82
12	7627,21	7613,97	7589,39	7500,46	7404,47	7547,1
13	7501,59	7514,77	7526,08	7362,64	7275,56	7436,13
14	7396,22	7250,82	7589,57	7258,77	7279,51	7354,98
15	7446,91	7349,59	7402,44	7205,3	7310,53	7342,95
16	7357,37	7315,22	7358,97	7167,97	7653,89	7370,69
17	7354,9	7377,42	7300,93	7269,47	7244,49	7309,44
18	7362,7	7269,48	7354,27	7207,12	7083,48	7255,41
19	7281,99	7320,07	7229,76	7304,74	7067	7240,71
20	7185,52	7160,57	7165,7	7172	7079,39	7152,64
21	7106,46	7220,02	7103,49	7114,42	7117,31	7132,34
22	7077,02	7155,02	7094,32	7078,62	7140,9	7109,17
23	7191,94	7223,74	7072,49	7098,58	7096,96	7136,74
24	7116,64	7189,47	7069,53	7147,86	7098,45	7124,39
25	7070,31	7228,25	7084,73	7107,11	7079,02	7113,89
26	7085,43	7159,73	7082,16	7085,79	7128,28	7108,28
27	7130,26	7193,94	7087,37	7065,39	7175,11	7130,41
28	7126,72	7187,59	7068,11	7112,42	7177,08	7134,38
29	7065,11	7236,38	7084,45	7114,99	7125,36	7125,26
30	7141,16	7315,61	7086,91	7166,19	7143,78	7170,73
31	7232,69	7340,17	7117,39	7068,83	7159,75	7183,77
32	7224,34	7375,17	7157,02	7063,79	7207,43	7205,55
33	7145,14	7502,73	7249,09	7128,41	7124,45	7229,96
34	7081,2	7249,48	7182,83	7173,88	7111,03	7159,68
35	7066,4	7256,7	7159,91	7112,51	7097,07	7138,52
36	7114,43	7389,64	7157,25	7101,74	7081,84	7168,98
37	7173,62	7104,69	7126,17	7064,54	7072,65	7108,33
38	7117,4	7230,37	7201,82	7066,92	7067,39	7136,78
39	7074,59	7180,72	7178,59	7066,38	7092,66	7118,59
40	7111,7	7116,41	7132,18	7126,46	7138	7124,95
41	7064,6	7094,9	7065,51	7131,58	7217,57	7114,83
42	7065,72	7135,7	7131,94	7114,17	7125,07	7114,52
43	7133,33	7265,16	7091,31	7070,65	7073,04	7126,7
44	7114,23	7183,41	7124,41	7065,35	7131,09	7123,7
45	7075,33	7293,16	7096,02	7082,09	7079,45	7125,21
46	7132,32	7247,58	7072,03	7072,48	7069,56	7118,79
47	7184,3	7110,06	7180,17	7209,37	7094,64	7155,71
48	7268,79	7126,26	7179,64	7109,94	7100,07	7156,94
49	7118,81	7115,39	7145,36	7125,38	7090,13	7119,01
50	7191,31	7096,75	7235,02	7131,48	7093,57	7149,62

Meilleures viabilités Tournament

Gén.	Run 1 (519,415)	Run 2 (518,424)	Run 3 (519,424)	Run 4 (518,423)	Run 5 (518,424)	Moy.
1	7273,23	7128,5	7134,21	7130,9	7084,27	7150,22
2	7230,65	7128,5	7134,21	7072,72	7084,27	7130,07
3	7079,17	7069,58	7063,77	7072,72	7084,27	7073,9
4	7066,67	7069,58	7063,77	7068,69	7084,27	7070,6
5	7066,67	7069,58	7063,77	7068,69	7063,16	7066,37
6	7066,33	7069,58	7063,77	7067,53	7063,16	7066,07
7	7066,33	7067,81	7063,77	7067,53	7063,16	7065,72
8	7066,33	7064,3	7063,77	7067,53	7063,16	7065,02
9	7066,33	7064,3	7063,77	7067,12	7063,16	7064,94
10	7065,16	7064,3	7063,77	7067,12	7063,16	7064,7
11	7065,16	7064,3	7063,77	7067,12	7062,98	7064,66
12	7065,16	7064,3	7063,77	7065,04	7062,98	7064,25
13	7065,16	7064,3	7063,77	7064,4	7062,98	7064,12
14	7065,16	7064,3	7063,77	7064,4	7062,98	7064,12
15	7065,16	7064,3	7063,77	7064,4	7062,98	7064,12
16	7065,16	7064,3	7063,77	7063,02	7062,93	7063,84
17	7065,16	7064,3	7063,77	7063,02	7062,93	7063,84
18	7065,16	7064,3	7063,77	7063,02	7062,93	7063,84
19	7065,08	7064,3	7063,35	7063,02	7062,8	7063,71
20	7064,59	7064,3	7063,35	7063,02	7062,8	7063,61
21	7064,59	7064,3	7062,86	7063,02	7062,8	7063,51
22	7064,59	7064,3	7062,86	7063,02	7062,8	7063,51
23	7064,59	7064,3	7062,86	7063,02	7062,8	7063,51
24	7064,59	7064,3	7062,86	7063,02	7062,8	7063,51
25	7064,59	7064,3	7062,86	7063,02	7062,8	7063,51
26	7064,59	7064,3	7062,81	7063,02	7062,8	7063,51
27	7064,59	7064,3	7062,81	7063,02	7062,8	7063,51
28	7064,57	7064,3	7062,81	7063,02	7062,8	7063,5
29	7064,57	7064,3	7062,81	7063,02	7062,8	7063,5
30	7064,57	7064,29	7062,81	7063,02	7062,8	7063,5
31	7064,57	7064,29	7062,81	7063,02	7062,8	7063,5
32	7064,57	7064,29	7062,81	7063,02	7062,8	7063,5
33	7064,57	7064,03	7062,81	7063,02	7062,8	7063,45
34	7064,57	7064,03	7062,81	7063,02	7062,8	7063,45
35	7064,57	7063,65	7062,81	7063,02	7062,8	7063,37
36	7064,57	7063,65	7062,81	7063,02	7062,8	7063,37
37	7064,57	7063,65	7062,81	7062,85	7062,8	7063,34
38	7064,57	7063,65	7062,81	7062,85	7062,8	7063,34
39	7064,57	7063,38	7062,81	7062,85	7062,8	7063,28
40	7064,57	7063,38	7062,81	7062,85	7062,8	7063,28
41	7064,57	7062,95	7062,81	7062,83	7062,8	7063,19
42	7064,57	7062,95	7062,81	7062,83	7062,8	7063,19
43	7064,57	7062,95	7062,81	7062,83	7062,8	7063,19
44	7064,57	7062,95	7062,81	7062,83	7062,8	7063,19
45	7064,57	7062,95	7062,81	7062,83	7062,8	7063,19
46	7064,57	7062,95	7062,81	7062,83	7062,8	7063,19
47	7064,57	7062,83	7062,81	7062,83	7062,8	7063,17
48	7064,57	7062,83	7062,81	7062,83	7062,8	7063,17
49	7064,57	7062,83	7062,81	7062,83	7062,8	7063,17
50	7064,57	7062,8	7062,81	7062,83	7062,8	7063,16

Viabilités moyennes Rank

Gén.	Run 1 (524,415)	Run 2 (511,423)	Run 3 (518,424)	Run 4 (520,425)	Run 5 (518,423)	Moy.
1	15616,58	16268,73	16236,61	16270,17	16219,81	16122,38
2	12521,01	13081,28	12463,77	13363,9	12530,71	12792,13
3	10889,72	11066,67	10887,85	11444,67	10418,61	10941,5
4	10199,12	10040,75	9899,81	10270,25	9390,06	9960
5	9767,69	9350,49	9202,29	9369,28	8604,63	9258,88
6	9264,8	8906,55	8730,81	8642,1	8393,76	8787,61
7	8672,99	8472,48	8347,36	8112,81	8144,1	8349,95
8	8361,65	8194,3	8143,24	7894,07	7983,64	8115,38
9	7689,78	7686,94	8030,45	7723,58	7744,98	7775,15
10	7463,44	7540,28	7768,88	7597,03	7697,66	7613,46
11	7446,88	7379,03	7651,42	7461,81	7620,67	7511,96
12	7392,79	7560,67	7515,02	7378,93	7422,74	7454,03
13	7330,43	7266,85	7464,15	7249,52	7363,25	7334,84
14	7242,46	7245,15	7364,37	7160,36	7316,95	7265,86
15	7264,14	7171,97	7246,89	7141,47	7188,72	7202,64
16	7214,15	7367,47	7196,34	7216,63	7237,04	7246,32
17	7218,2	7189,44	7202,59	7193,6	7190,75	7198,92
18	7220,09	7199,9	7239,62	7172,61	7231,38	7212,72
19	7191,16	7226,27	7119,23	7209,71	7138,02	7176,88
20	7143,72	7132	7197,39	7201,06	7103,02	7155,44
21	7198,49	7143,5	7144,14	7161,84	7154,21	7160,44
22	7219,87	7177,2	7103,96	7163,17	7182,22	7169,28
23	7190,91	7170,98	7098,82	7134,31	7135,59	7146,12
24	7196,92	7212,41	7093,55	7123,93	7136,26	7152,61
25	7216,32	7118,56	7158,5	7111,32	7142,84	7149,51
26	7174,26	7233,62	7142,51	7181,71	7168,75	7180,17
27	7141,51	7141,83	7069,16	7140,99	7104,65	7119,63
28	7165,41	7118,39	7165,78	7081,29	7111,1	7128,39
29	7261,91	7235,04	7117,29	7089,93	7177,85	7176,4
30	7162,11	7139,11	7075,09	7169,66	7132,07	7135,61
31	7170,66	7083,04	7064,58	7135,96	7128,52	7116,55
32	7075,16	7132,46	7131,65	7135,31	7111,56	7117,23
33	7120,81	7080,98	7067,96	7101,34	7120,18	7098,25
34	7075,51	7070,13	7130,69	7133,95	7084,16	7098,89
35	7173,8	7115,08	7113,84	7132,08	7120,65	7131,09
36	7170,13	7065,58	7082,96	7133,35	7080,06	7106,42
37	7148,61	7088,44	7133,72	7101,6	7066,29	7107,73
38	7184,51	7139,58	7067,59	7265,9	7114,93	7154,5
39	7080,9	7094,84	7062,95	7124,4	7205,87	7113,79
40	7080,61	7084,49	7126,31	7144,01	7092,88	7105,66
41	7076,59	7097,08	7062,9	7114,82	7079,48	7086,18
42	7105,9	7065,11	7217,46	7114,09	7112,99	7123,11
43	7111,8	7276,34	7162,48	7068,06	7165,59	7156,85
44	7137,44	7072,71	7067,51	7118,65	7114,34	7102,13
45	7135,93	7099,47	7109,98	7092,59	7086,67	7104,93
46	7081,07	7114,37	7110,59	7114,08	7137,12	7111,45
47	7065,47	7233,85	7123,76	7179,46	7093,03	7139,11
48	7071,26	7068,61	7099,07	7174,51	7078,28	7098,35
49	7114,3	7064,05	7071,61	7123,95	7160,33	7106,85
50	7068,18	7185,27	7125,14	7179,29	7110,83	7133,74

Meilleures viabilités Rank

Gén.	Run 1 (524,415)	Run 2 (511,423)	Run 3 (518,424)	Run 4 (520,425)	Run 5 (518,423)	Moy.
1	7277,83	7107,59	7075,96	7089,35	7128,03	7135,75
2	7277,83	7107,59	7067,17	7089,35	7110,75	7130,54
3	7164,7	7090,06	7067,17	7089,35	7110,75	7104,41
4	7164,7	7090,06	7063,26	7070,58	7110,75	7099,87
5	7102,6	7090,06	7063,26	7070,58	7110,75	7087,45
6	7102,6	7090,06	7063,26	7068,22	7110,75	7086,98
7	7102,6	7090,06	7063,26	7064,26	7071,46	7078,33
8	7102,6	7068,46	7063,26	7064,26	7071,46	7074,01
9	7102,6	7068,46	7063,26	7064,26	7071,46	7074,01
10	7102,6	7068,46	7063,26	7064,26	7071,46	7074,01
11	7081,25	7068,46	7063,26	7064,26	7071,25	7069,7
12	7070,7	7068,46	7063,26	7064,26	7071,25	7067,59
13	7069,57	7068,46	7063,26	7064,26	7064,43	7066
14	7069,57	7068,46	7063,26	7064,26	7064,43	7066
15	7069,57	7068,46	7063,26	7064,26	7064,11	7065,93
16	7069,57	7068,46	7063,26	7063,2	7064,11	7065,72
17	7069,57	7068,46	7063,26	7063,2	7064,11	7065,72
18	7069,57	7068,46	7063,26	7063,2	7064,11	7065,72
19	7069,57	7068,46	7063,08	7063,2	7063,89	7065,64
20	7069,57	7067,79	7063,08	7063,2	7063,89	7065,5
21	7069,05	7064,1	7063,08	7063,2	7063,89	7064,66
22	7065,47	7064,1	7062,84	7063,2	7063,89	7063,9
23	7065,47	7064,1	7062,84	7063,05	7063,89	7063,87
24	7065,47	7064,1	7062,84	7063,05	7063,89	7063,87
25	7065,47	7064,1	7062,84	7062,88	7063,89	7063,84
26	7065,47	7064,1	7062,84	7062,88	7063,02	7063,66
27	7065,47	7063,86	7062,84	7062,88	7063,02	7063,61
28	7065,31	7063,86	7062,84	7062,88	7063,02	7063,58
29	7065,31	7063,86	7062,83	7062,88	7063,02	7063,58
30	7065,31	7063,86	7062,8	7062,88	7063,02	7063,57
31	7065,31	7063,86	7062,8	7062,88	7063,02	7063,57
32	7065,31	7063,86	7062,8	7062,88	7063,02	7063,57
33	7065,31	7063,86	7062,8	7062,88	7063,02	7063,57
34	7065,31	7063,86	7062,8	7062,88	7063,02	7063,57
35	7065,31	7063,86	7062,8	7062,88	7063,02	7063,57
36	7065,31	7063,86	7062,8	7062,88	7063,02	7063,57
37	7065,31	7063,86	7062,8	7062,88	7063,02	7063,57
38	7065,31	7063,84	7062,8	7062,88	7062,93	7063,55
39	7064,92	7063,84	7062,8	7062,88	7062,93	7063,47
40	7064,92	7063,61	7062,8	7062,88	7062,93	7063,43
41	7064,92	7063,61	7062,8	7062,88	7062,93	7063,43
42	7064,92	7063,61	7062,8	7062,88	7062,93	7063,43
43	7064,92	7063,61	7062,8	7062,88	7062,93	7063,43
44	7064,92	7063,61	7062,8	7062,88	7062,83	7063,41
45	7064,92	7063,61	7062,8	7062,88	7062,83	7063,41
46	7064,92	7063,61	7062,8	7062,88	7062,83	7063,41
47	7064,92	7063,61	7062,8	7062,88	7062,83	7063,41
48	7064,92	7063,61	7062,8	7062,88	7062,83	7063,41
49	7064,92	7063,61	7062,8	7062,88	7062,83	7063,41
50	7064,92	7063,61	7062,8	7062,88	7062,83	7063,41

Viabilités moyennes Sigma

Gén.	Run 1 (511,422)	Run 2 (507,419)	Run 3 (518,430)	Run 4 (522,438)	Run 5 (521,412)	Moy.
1	19455,61	19937,49	21000,51	20857,78	19590,64	20168,41
2	18459,34	18936,97	19691,29	19569,25	19023,45	19136,06
3	16747,15	17733,49	18351,49	18320,34	17863,09	17803,11
4	15624,97	15565,91	17327,2	17075,22	17362,46	16591,15
5	13883,49	13900,72	15553,39	14936,88	16168,27	14888,55
6	13236,17	13181,17	13362,22	14008,79	15465,83	13850,84
7	12298,67	12229,63	12180,69	13612,7	14078,28	12880
8	11553,96	12035,7	11630,54	12341,71	13337,91	12179,97
9	11400,17	11761,99	10774,26	11837,73	12366,72	11628,17
10	11563,69	11194,66	10619,7	11290,87	12072,22	11348,23
11	11092,68	11086,51	10465,65	10983,9	11400,38	11005,82
12	10933,42	11003,56	10421	10143,46	10618,02	10623,89
13	10552,1	10590,17	10136,53	10100,06	10262,41	10328,26
14	10035,3	10014,91	9919,3	10159,76	9640,08	9953,87
15	10226,65	9746,15	9625,5	9920,75	9674,74	9838,76
16	10004,2	9437,01	9395,88	9649,48	9382,2	9573,76
17	9699,63	9316,92	9307,92	9562,21	9173,93	9412,12
18	9362,41	9523,97	9224,32	9607,47	8544,98	9252,63
19	9186,2	9490,1	9059,52	8891,61	8212,4	8967,97
20	8822,8	9200,94	9026,43	9156,19	8111,87	8863,65
21	8499,13	8905,68	8666,92	8668,55	8068,18	8561,69
22	8353,02	8928,94	8621,44	8239,52	8149,35	8458,45
23	8623,87	8991,45	8532,55	8186,44	8031,82	8473,23
24	8201,49	8646,59	8482,19	8002,96	7899,49	8246,54
25	7990,03	8302,62	8440,33	7906,13	7982,35	8124,29
26	7773,43	8384,68	8303,75	8009,97	7876,5	8069,67
27	7725,69	8332,64	8307,77	8010,55	7767,35	8028,8
28	7602,12	8374,62	8279,6	7731,78	7814,71	7960,57
29	7611,29	8397,01	8072,29	7540,87	7759,28	7876,15
30	7600,28	8352,26	8111,08	7454,88	7753,5	7854,4
31	7666,03	8511,7	7957,97	7479,55	7645,21	7852,09
32	7674,71	8344,71	7802,96	7420,8	7686,14	7785,86
33	7604,47	8286,58	7663,18	7318,81	7549,7	7684,55
34	7661,02	8285,52	7541,18	7362,55	7460,16	7662,09
35	7619,78	8429,63	7426,89	7331,27	7392,67	7640,05
36	7747,75	8377,48	7443,63	7349,75	7399,91	7663,71
37	7663,65	8354,7	7516,1	7461,83	7334,95	7666,25
38	7588,8	8272,82	7398,72	7428,8	7288,27	7595,48
39	7491,86	8296,9	7398,82	7326,31	7251,96	7553,17
40	7498,56	8132,37	7282,93	7307,18	7290,14	7502,24
41	7403,58	7889,28	7267,07	7278,13	7344,51	7436,51
42	7343,59	7827,37	7323,32	7251,06	7393,45	7427,76
43	7278,35	7716,3	7546,8	7308,4	7433,75	7456,72
44	7270,25	7656,92	7303,31	7208,15	7311,57	7350,04
45	7286,34	7745,88	7229,05	7205,85	7283,51	7350,13
46	7118,92	7829,31	7221,26	7232,6	7345,38	7349,49
47	7137,33	7870,12	7182,27	7186,15	7374,61	7350,1
48	7114,48	7803,18	7205,4	7198,73	7331,35	7330,63
49	7183,39	7643,09	7226,06	7232,43	7356,8	7328,35
50	7208,16	7477,74	7305,19	7204,89	7364,22	7312,04

Meilleures viabilités Sigma

Gén.	Run 1 (511,422)	Run 2 (507,419)	Run 3 (518,430)	Run 4 (522,438)	Run 5 (521,412)	Moy.
1	7121,88	7157,45	7181,99	7234,23	7165,26	7172,16
2	7121,88	7157,45	7181,99	7234,23	7104,64	7160,04
3	7121,88	7157,45	7067,67	7146,17	7104,64	7119,56
4	7121,88	7157,45	7067,67	7146,17	7104,64	7119,56
5	7121,88	7157,45	7063,47	7146,17	7104,64	7118,72
6	7121,88	7157,45	7063,47	7117,75	7067,58	7105,62
7	7112,97	7157,45	7063,47	7077,12	7067,58	7095,72
8	7081,52	7157,45	7063,47	7077,12	7067,58	7089,43
9	7081,52	7157,45	7063,47	7077,12	7067,27	7089,37
10	7081,52	7101,23	7063,47	7077,12	7067,27	7078,12
11	7081,52	7101,23	7063,47	7077,12	7067,27	7078,12
12	7081,52	7101,23	7063,47	7075,67	7067,27	7077,83
13	7081,52	7101,23	7063,47	7075,67	7067,27	7077,83
14	7068,75	7101,23	7063,47	7075,67	7066,81	7075,19
15	7067,3	7101,23	7063,47	7075,67	7065,95	7074,73
16	7067,3	7071,76	7063,47	7075,67	7065,95	7068,83
17	7067,3	7071,76	7063,47	7075,67	7065,95	7068,83
18	7067,3	7071,76	7063,47	7075,67	7065,95	7068,83
19	7067,3	7070,96	7063,47	7075,67	7065,95	7068,67
20	7067,3	7070,96	7063,47	7075,67	7065,95	7068,67
21	7067,3	7070,96	7063,47	7075,67	7065,95	7068,67
22	7067,3	7070,96	7063,47	7075,67	7065,95	7068,67
23	7067,3	7070,49	7063,47	7067,21	7065,95	7066,88
24	7067,3	7070,49	7063,47	7067,21	7065,95	7066,88
25	7063,72	7070,49	7063,47	7067,21	7065,95	7066,17
26	7063,72	7067,09	7063,47	7067,21	7065,95	7065,49
27	7063,72	7066,8	7063,47	7067,21	7065,95	7065,43
28	7063,72	7066,8	7063,47	7067,21	7065,95	7065,43
29	7063,72	7066,8	7063,47	7067,21	7065,95	7065,43
30	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
31	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
32	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
33	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
34	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
35	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
36	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
37	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
38	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
39	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
40	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
41	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
42	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
43	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
44	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
45	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
46	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
47	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
48	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
49	7063,72	7066,11	7063,47	7067,21	7065,95	7065,29
50	7063,72	7065,46	7063,47	7067,21	7065,95	7065,16

Viabilité moyenne Tournament et Elitisme

Gén.	Run 1 (518,424)	Run 2 (518,423)	Run 3 (511,425)	Run 4 (520,415)	Run 5 (511,432)	Moy.
1	17000,01	18655,43	17666,99	17409,61	17865,03	17719,41
2	14282,61	15438,71	14269,69	14055,53	14827,55	14574,82
3	12219,46	13929,91	12722,56	11874,59	12408,22	12630,95
4	10337,27	11644,91	10671,15	10115,27	10076,55	10569,03
5	8791,58	9883,54	8963,57	8765,76	9279,32	9136,75
6	7906,71	9056,26	7991,38	8316,11	7963,3	8246,75
7	7746,84	8308,22	7924,83	7590,11	7816,23	7877,25
8	7416,45	7624,02	7307,52	7519,56	7520,1	7477,53
9	7292,92	7408,14	7271,19	7244,71	7209,11	7285,21
10	7188,2	7270	7117,15	7111,02	7155,41	7168,35
11	7233,19	7239,82	7101,39	7094,49	7085,48	7150,87
12	7131	7380,45	7142,22	7146,77	7085,52	7177,19
13	7101,03	7182,95	7085,25	7091,03	7167,06	7125,46
14	7108,71	7126,21	7113,3	7143,6	7086,08	7115,58
15	7070,11	7250,34	7224,55	7167,36	7148,6	7172,19
16	7127,08	7111,16	7143,74	7261,76	7226,15	7173,98
17	7069,04	7160,57	7162,24	7251,39	7163,96	7161,44
18	7067,67	7242,67	7113,08	7246,77	7179,08	7169,85
19	7178,71	7079,46	7158,84	7189,05	7212,21	7163,66
20	7072,92	7078,28	7145,27	7147,1	7210,19	7130,75
21	7126,6	7078,28	7123,12	7085,15	7080,88	7098,8
22	7206,71	7065,34	7073,09	7065,47	7179,92	7118,11
23	7286,59	7124,04	7123,42	7070,03	7132,99	7147,41
24	7238,52	7069,14	7114,24	7070,18	7112,28	7120,87
25	7245,97	7124,03	7079,86	7112,11	7096,6	7131,71
26	7267,18	7077,22	7067,38	7112,9	7111,36	7127,21
27	7175,3	7142,73	7113,34	7111,88	7161,78	7141
28	7201,74	7143,81	7182,17	7176,93	7124,89	7165,91
29	7128,14	7174,39	7063,7	7136,92	7130,64	7126,76
30	7093,55	7160,98	7120,55	7071,5	7153,04	7119,92
31	7122,43	7068,38	7079,94	7071,5	7204,48	7109,34
32	7068,64	7161,62	7094,5	7085,19	7136,46	7109,28
33	7064,06	7129,62	7094,8	7120,31	7238,78	7129,51
34	7109,78	7149,34	7106,18	7065,97	7197,78	7125,81
35	7101,59	7143,57	7089,21	7136,88	7173,44	7128,94
36	7078,5	7078,54	7139,05	7134,61	7212,37	7128,61
37	7161,53	7129,02	7135,54	7120,68	7211,88	7151,73
38	7221,85	7163,4	7131,26	7224,77	7236,68	7195,59
39	7129,72	7175,66	7118,16	7139,76	7302,31	7173,12
40	7078,44	7072,08	7121,88	7125,4	7099,23	7099,41
41	7131,51	7088,37	7179,58	7128,6	7235,64	7152,74
42	7143,7	7142,45	7109,59	7242,4	7236,6	7174,95
43	7145,31	7121,81	7132,51	7147,71	7245,22	7158,51
44	7120,58	7125,01	7195,9	7118,05	7223,13	7156,53
45	7072	7088,24	7185,76	7118,5	7170,66	7127,03
46	7138,18	7146,7	7093,33	7155,99	7093,33	7125,51
47	7210,48	7269,75	7155,26	7152,9	7064,88	7170,65
48	7354,85	7180,1	7177,74	7102,08	7080,56	7179,07
49	7272,19	7082,97	7212,89	7077,31	7097,62	7148,6
50	7124,1	7130,26	7065,81	7160,72	7123,02	7120,78

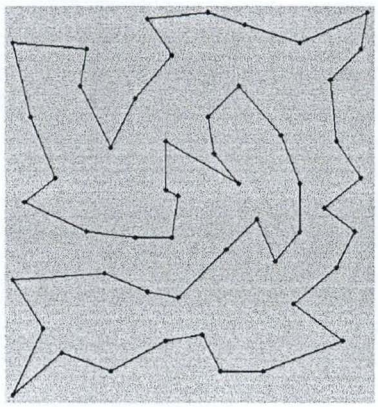
Viabilité moyenne Random et Elitisme

Gén.	Run 1	Run 2	Run 3	Run 4	Run 5	Moy.
1	19924,24	19396,47	20605,13	18495,93	19010,86	19486,53
2	19568,76	17356,11	20114,11	17753,91	18133,39	18585,26
3	18635,49	16585,84	19664,02	16571,85	15603,79	17412,2
4	16122,39	15391,64	18197,04	15720,42	14238,7	15934,04
5	15426,41	13618,73	16672,19	15871,38	14141,89	15146,12
6	14382,54	13405,64	15646,28	15404,65	13024,68	14372,76
7	13419,58	12651,18	15275,54	14377,08	13095,82	13763,84
8	12094,45	12604,63	14324,09	13330,92	12738,87	13018,59
9	11583,65	12700,25	14307,82	12866,1	12616,04	12814,77
10	10695,9	11158,04	12884,57	12302,43	12790,77	11966,34
11	10417,47	10915,22	11876,69	12028,16	13341,09	11715,73
12	9789,28	10238,65	11910,14	12038,35	13826,71	11560,63
13	8984,41	10236,57	11951,35	12133,28	12911,33	11243,39
14	8787,36	10238,74	11336,62	11915,52	12887,22	11033,09
15	8627,65	10708,21	10695,44	11948,49	13523,13	11100,59
16	8218,01	10816,94	10214,03	11967,51	12644,81	10772,26
17	8410,26	10307,11	9816,3	11296,83	12408,38	10447,78
18	8627,86	10361,6	9442,36	10852,38	11881,21	10233,08
19	8407,06	9537,75	9335,27	10546,31	12096,33	9984,54
20	8321,84	9028,78	8498,9	10577,22	11071,31	9499,61
21	8540,68	9885,2	8182,4	10427,61	10929,96	9593,17
22	8215,42	9750,95	7956,98	10192,27	11196,25	9462,38
23	7887,36	9311,55	8096,11	9306,64	10740,65	9068,46
24	7759,62	9660,38	7831,68	9283,36	10168,83	8940,77
25	7756,43	9266,82	7420,43	8817,08	9932,13	8638,58
26	7682,39	9061,64	7298,43	8397,78	9226,92	8333,43
27	7570,19	9117,15	7972,46	7907,8	9276,21	8368,76
28	7622,22	9123,33	8674,54	7831,5	8718,87	8394,09
29	7668,66	8728,4	8286,78	7878	9116,15	8335,6
30	7617,15	8787,61	8017,84	7904,37	8773,19	8220,03
31	7649,96	7944,85	7977,74	7380,68	8270,95	7844,83
32	7772,54	7900,57	8235,91	7314,58	8167,32	7878,18
33	7604,94	7932,19	8074,02	7485,17	7926,08	7804,48
34	7608,99	8086,5	7851,64	7763,79	7727,79	7807,74
35	7548,62	8022,1	7727,67	7790,09	7613,94	7740,48
36	7478,15	7838,98	7462,66	8132,34	7617,21	7705,87
37	7572,95	8244,21	7754,5	8486,36	7592,71	7930,14
38	7506,31	8320,66	7705,52	7784,4	7631,08	7789,6
39	7547,7	7912,51	7891,1	7518,93	7563,72	7686,79
40	7704,34	7616,93	7676,57	7481,14	7742,82	7644,36
41	7730,54	7604,94	7859,3	7605,53	7499,84	7660,03
42	7824,41	7650,05	7944,71	7658,81	7320,39	7679,67
43	7933,56	8107,06	7859,35	7504,65	7270,51	7735,02
44	7907,46	7977,1	7934,72	7381,56	7293,12	7698,79
45	8020,4	7808,98	7923,22	7681,36	7274,26	7741,64
46	7754,69	7612,93	7856,3	7624,24	7193,25	7608,28
47	7730,66	7645,2	7814,05	7505,48	7480,29	7635,14
48	7491,27	7649,79	7889,04	7355,52	7559,17	7588,96
49	7748,06	7504,11	7690,06	7421,09	7419,91	7556,64
50	7673,19	7565,66	7557,8	7526,56	7612,78	7587,2

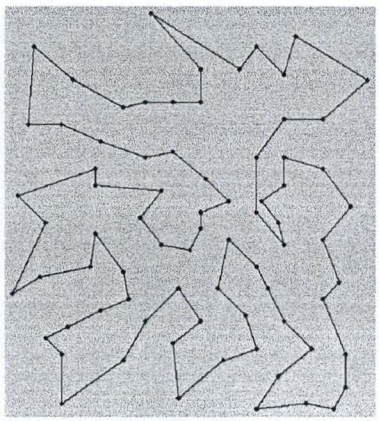
Annexe C

Meilleurs trajets obtenus dans diverses expérimentations du TSP

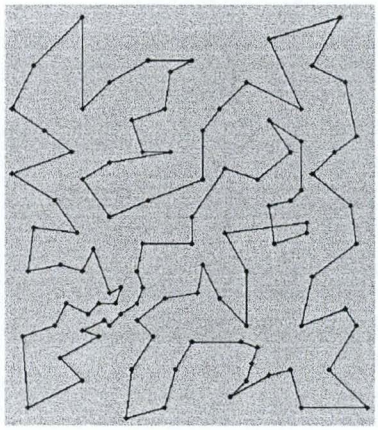
Problèmes TSP "eil"



eil51

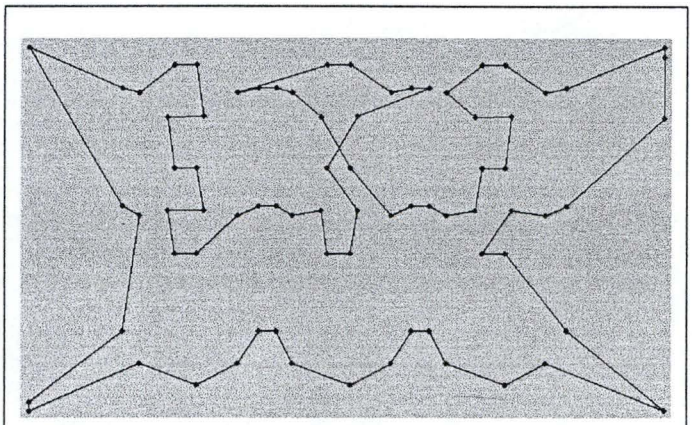


eil76

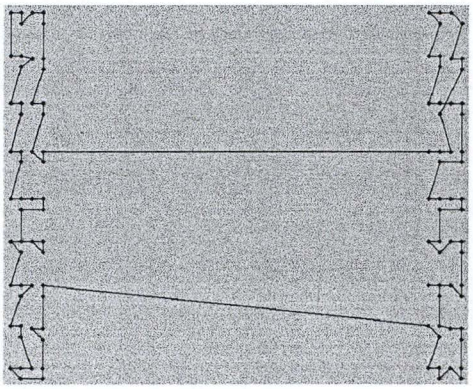


eil101

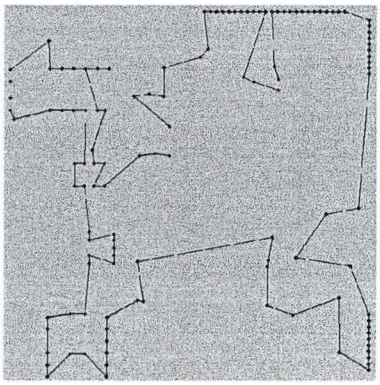
Problèmes TSP "pr"



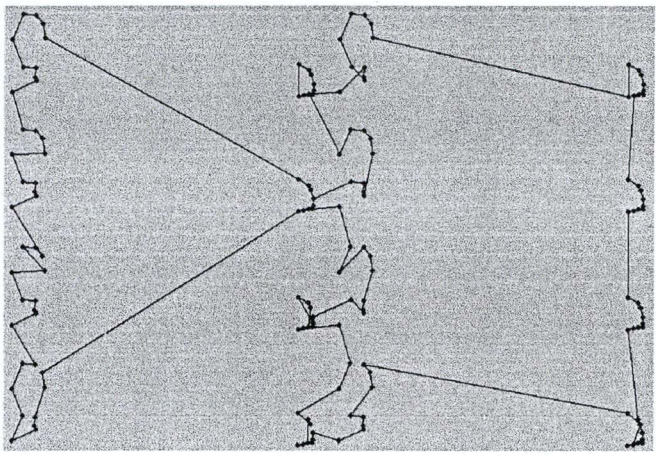
pr76



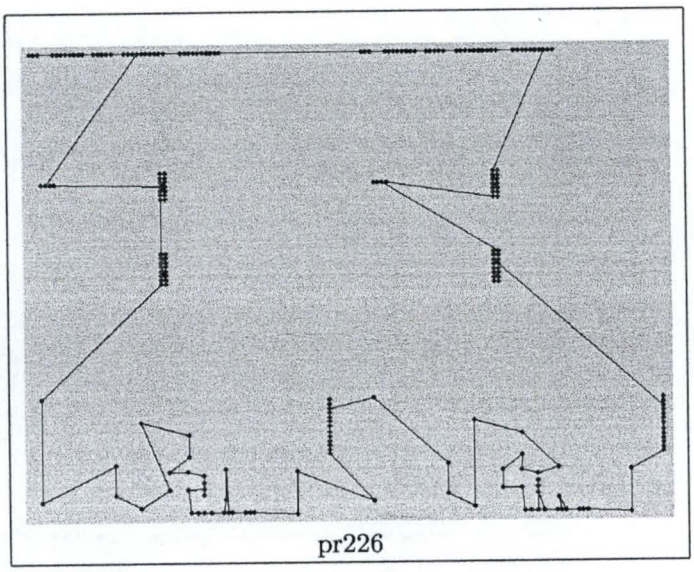
pr107



pr124

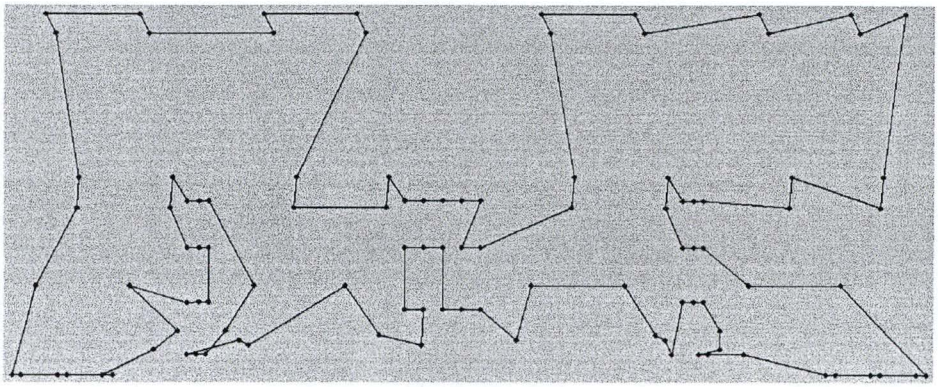


pr152

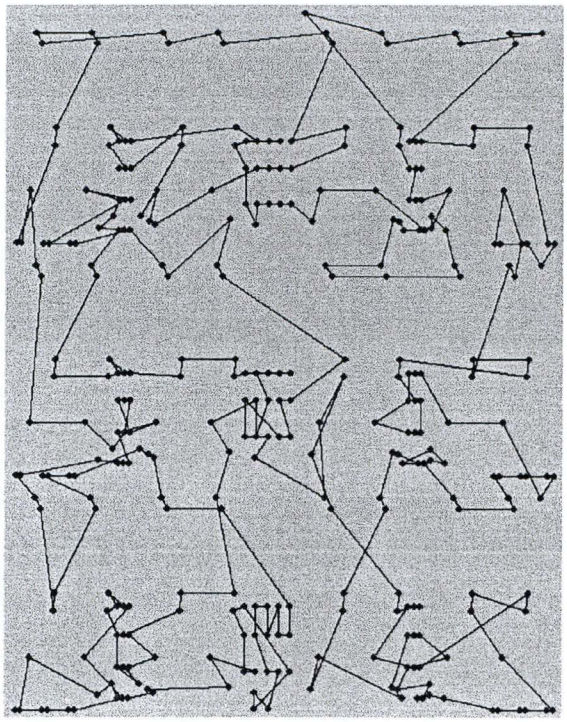


pr226

Problèmes TSP "lin"



lin105



lin318

Annexe D

Apprentissage évolutionnaire de réseaux neuronaux CLP-contraints

Evolutionary Training of CLP-Constrained Neural Networks

Joost N. Kok[1], Elena Marchiori[1,2], Massimo Marchiori[3], Claudio Rossi[4]

[1] *Dept. of Computer Science, University of Leiden*
P.O. Box 9512, 2300 RA Leiden, The Netherlands
joost@cs.leidenuniv.nl

[2] *CWI*
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
elena@cwi.nl

[3] *Dept. of Pure and Applied Mathematics, University of Padova*
via Belzoni 7, 35131 Padova, Italy
max@hilbert.math.unipd.it

[4] *Dip. di Matematica Applicata e Informatica, Università di Ca' Foscari*
via Torino 155, 30173 Mestre-Venezia, Italy
crossi@moo.dsi.unive.it

Abstract

The paper is concerned with the integration of constraint logic programming systems (CLP) with systems based on genetic algorithms (GA). The resulting framework is tailored for applications that require a first phase in which a number of constraints need to be generated, and a second phase in which an optimal solution satisfying these constraints is produced. The first phase is carried by the CLP and the second one by the GA. We present a specific framework where ECL^{i}PS^e (ECRC Common Logic Programming System) and GENOCOP (GEnetic algorithm for Numerical Optimization for COnstrained Problems) are integrated in a framework called CoCo (COmputational intelligence plus COnstraint logic programming). The CoCo system is applied to the training problem for neural networks. We consider constrained networks, e.g. neural networks with shared weights, constraints on the weights - for example domain constraints for hardware implementation - etc. Then ECL^{i}PS^e is used to generate the chromosome representation together with other constraints which ensure, in most cases, that each network is specified by exactly one chromosome. Thus the problem becomes a constrained optimization problem, where the optimization criterion is to optimize the error of the network, and GENOCOP is used to find an optimal solution.

Note: The work of the second author was partially supported by SION, a department of the NWO, the National Foundation for Scientific Research. This work has been carried out while the third author was visiting CWI, Amsterdam, and the fourth author was visiting Leiden University.

1 Introduction

There are two major paradigms for problem solving that play a distinguished role in Computing Science and Artificial Intelligence: Reasoning and Search [16]. Although every automated reasoning process involves search aspects (e.g. how to traverse the derivation tree of a logic program), the two paradigms and the corresponding research communities are rather disjoint. As a consequence, high level paradigms for reasoning generally incorporate rather inefficient search techniques, due to the exponential size of the search space they use. In this paper we try to go beyond the borders of each individual paradigm by applying them in combination. More specifically, the paper is concerned with the integration of constraint logic programming systems (CLP) with systems based on genetic algorithms (GA). The resulting system is tailored for applications that require a first phase in which a number of constraints need to be generated, and a second phase in which an optimal solution satisfying these constraints is produced. The first phase is carried by a suitable CLP and the second one by a suitable GA.

In order to test the adequacy of such an integration, we have combined ECL^{i}PS^e (ECRC Common Logic Programming System) and GENOCOP [14] (GEnetic algorithm for Numerical Optimization for COnstrained Problems) in a framework called CoCo (COmputational intelligence plus COnstraint logic programming). We have applied the CoCo system to solve the training problem for constrained neural networks. Neural networks have been used in many applications, for example in planning, control, content-addressable memory, optimization, constraint satisfaction, and classification (see e.g. [5]). They are being promoted for their robustness, massive parallelism, and ability to learn. The training of a neural network is a major design step: Roughly, one has to find a set of weights that minimizes the neural network's error on an initial set of input-output examples called the training set. The standard method for that problem is a local gradient search method known as the back-propagation algorithm [15]. Since the problem's error surface is highly dimensional and usually it contains many local minima, this method can get stuck in local minima. Moreover, it needs gradient information. Alternative approaches based on genetic algorithms have been proposed, which apply for instance to the following types of neural networks.

1. recurrent networks;
2. networks with non-differentiable error criteria (for example due to non-differentiable transfer functions, error measures that use absolute values, bonuses for small weights etc.).

For this kind of neural networks the standard back-propagation learning rule is not in general applicable. Nevertheless, this type of networks can be very useful in applications (for example in applications based on time sequences [6]). Genetic algorithms usually avoid local minima by searching several regions simultaneously. They act on a population of trial solutions, and use information on some performance value describing the 'quality' of a set of weights. Thus they do not use gradient information, and do not require restrictions on the network topology. However, the drawback of genetic algorithms is that they seem to have difficulties in the fine tuning of the parameters [9]. Moreover, there is the "competing conventions problem" [4, 17]: when one chooses a representation (in this case for a neural network), then it can be the case that the same individual has more than one representation. This enlarges (in an artificial way) the search space and affects the convergence speed of the algorithms. Therefore, the programmer has to find a clever representation such that this does not happen.

In this paper we introduce an automatic tool for dealing with the competing conventions problem, based on the following novel approach. We consider constrained networks, e.g. neural networks with shared weights, constraints on the weights - for example domain constraints for hardware implementation - etc. Moreover, we introduce other constraints which ensure that in most cases each network is specified by exactly one chromosome. Thus the problem becomes a constrained optimization problem. The optimization criterion is to optimize the error of the network (usually this error includes a sum over all the training patterns of the network), and the constraints specify the domain constraints on the weights and some constraints for avoiding the competing conventions problem. More precisely, a constraint logic program in ECL^{i}PS^e is used, which generates constraints on the weights such that each network has in most cases a unique chromosome representation. These constraints and the optimization function are given as input to GENOCOP, which searches for an optimal solution that satisfies the constraints. The advantages of this approach are:

1. we do not have to find a special representation;
2. we can easily integrate through the constraint logic program the constraints for the competing conventions problem with other constraints (for example shared weights, domain constraints etc.);
3. the CLP system checks for the satisfiability of the constraints.

So we consider constrained optimization problems for which a set of relatively simple constraints, generated by a suitable constraint logic program, restricts the search space, and for which the optimization criterion can be rather complex (e.g. a non-differentiable, non-linear function). For this class of optimization problems, genetic algorithms are valuable search tools. Since the weights are real numbers, and they are constrained, a natural choice is to employ a system that can handle constraints and where the data are encoded using real numbers, instead of the original bit-encoding. The GENOCOP system satisfies both these requirements. We have used ECL^{i}PS^e as constraint logic programming system. A nice feature of ECL^{i}PS^e is the possibility to structure programs by means of modules. We have used modules for describing various kinds of constraints that one can impose on the weights of a neural network, depending for instance on the form of the activation function, or on the symmetry of the problem. Moreover, modules have been used to specify various kinds of chromosome representations.

The paper is organized as follows. In the next section the integration of ECL^{i}PS^e and GENOCOP is discussed. Section 3 introduces the new approach for the training of CLP-constrained neural networks. In Section 4 some computational results and evaluation are given. Section 5 discusses related work on the integration of Constraint Logic Programming and Computational Intelligence. Finally, Section 6 contains some conclusions and future work.

2 CoCo: Integrating ECL^{i}PS^e and GENOCOP

In this section we describe the platform that is used for performing our experiments on the evolutionary training of constrained neural networks. First, we motivate the need for a framework based on two different systems. Next, we briefly discuss the two systems used in the application of this paper, ECL^{i}PS^e and GENOCOP, and their integration.

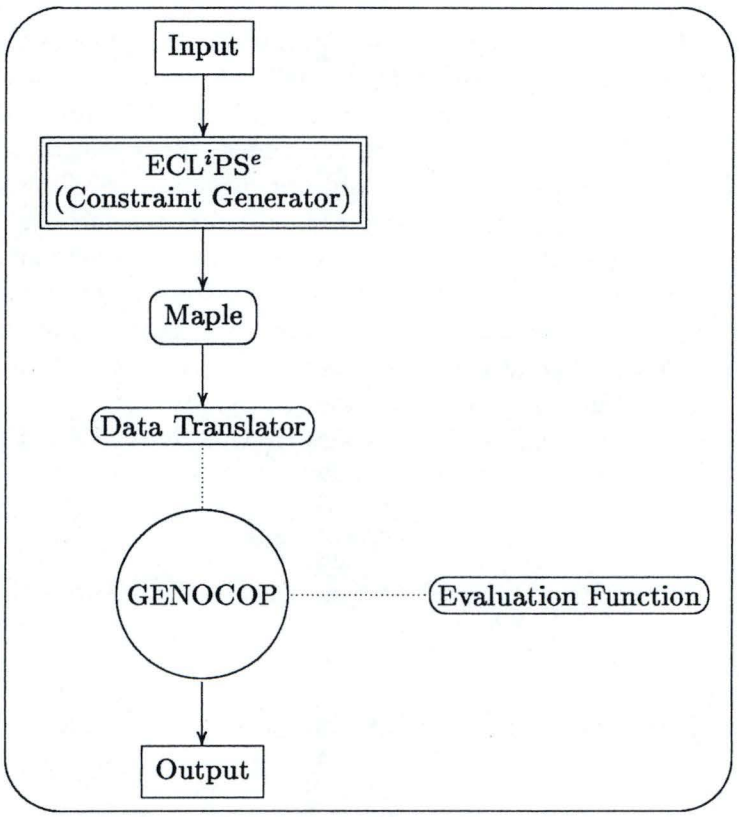


Figure 1: The CoCo Framework

We consider applications that require a first phase in which a number of constraints need to be generated, and a second phase in which an optimal solution, w.r.t. a given objective function, satisfying these constraints is produced. Examples of such applications include design of VLSDC, finance, medical engineering and many others (cf. e.g. [8]). These applications have been tackled using different methods, for instance approximation methods, like genetic algorithms and neural networks, or exact methods like those of operational research or constraint logic programming. Recent works have shown the advantages of an integrated use of exact and approximation methods. For instance, constraint logic programming and neural networks have been integrated in a system called PROCLANN [12], which exploits the elegant formalism of CLP and the efficiency of constraint solvers based on stochastic systems, like GENET [18].

The specific integration we consider for our application, called CoCo (COmputational Intelligence plus COnstraint Logic Programming), employs the CLP system ECL^{i}PS^e for generating constraints, and the GA system GENOCOP for the optimization phase. We describe briefly the two systems. The reader is referred to the rich documentation on ECL^{i}PS^e available from ECRC at the ftp address ftp.ecrc.de/pub/ECRC_tech_reports/reports, and to the book by Michalewicz [14] for GENOCOP. ECL^{i}PS^e (ECRC Common Logic Programming System) is a Prolog based system built to facilitate the integration of various programming extensions. In particular, it includes a number of libraries, like the one for treating lists, and a constraint solver over the rationals, available in the library `r.pl`. The solver uses a combination of the Simplex algorithm and Gaussian elimination to solve a system of arith-

metic constraints consisting of linear equalities and inequalities. We have used the rational constraint solver in CoCo. Moreover, ECL^{i}PS^e allows to structure programs by means of modules. We have used modules for describing various kinds of constraints for our application. GENOCOP (GEnetic algorithm for Numerical Optimization for COnstrained Problems) is an evolutionary program which can handle a large class of optimization problems with linear constraints and any objective function. It uses a floating point representation for chromosomes, and special 'genetic' operators which guarantee that all the chromosomes remain within the constrained solution space. This is possible because only a set of linear constraints, including domain constraints, equalities and inequalities, is allowed. The relevance of this approach is that it provides a general and problem independent way to handle constraints in optimization problems. Hence the system can be easily integrated in our framework.

We conclude this section with issues on the integration of ECL^{i}PS^e and GENOCOP. In order to pass the set of constraints produced by the ECL^{i}PS^e program to GENOCOP, we have designed an ECL^{i}PS^e program that gets the constraints and write them into a file, together with other information about the system, like the number of variables, the number of 'slack variables' (these are variables introduced by the constraint solver when using the Simplex algorithm and Gaussian elimination), and their names. In this first stage of the integration we use files for the exchange of information between ECL^{i}PS^e and GENOCOP, but in the future we plan to use ECL^{i}PS^e's External Language Interface to incorporate GENOCOP into ECL^{i}PS^e. One of the difficulties we encountered was that the rational constraint solver of ECL^{i}PS^e is based on the Simplex Algorithm, hence is tailored towards equalities (i.e. every inequality is replaced by an equality using 'slack variables'). However, the GENOCOP system is based on inequalities. We solved this problem by using the Maple system to remove the slack variables, and to re-introduce the inequalities. However, a more elegant approach we intend to investigate in the future, is the use of an interval constraint solver.

Next, we modified GENOCOP in order to allow it to read the constraints in a different format. This has been done replacing the GENOCOP function which reads the data from the file with a new one that reads the equations in the ECL^{i}PS^e form.

Finally, the evaluation function, which must be provided separately as a C function and is part of the GENOCOP source code, was also modified in order to avoid the need of the re-compilation each time the parameters of the network are modified.

3 Training a Constrained Neural Network

We have introduced the CoCo framework, which integrates ECL^{i}PS^e and GENOCOP. In this section we apply CoCo to a specific problem, namely the training of a constrained neural network. We consider feedforward neural networks (FFNN) (cf. [5]), where there are only connections from nodes of one layer to nodes of the successive layer. For the generalization of the results to recurrent neural networks, the reader is referred to [10].

The program is composed by several modules. At top level, there are three main modules (see Figure 2):

1. The first **Start Module** asks the user the representation of the FFNN, and builds up the corresponding data that will be used in the sequel.

2. The second **Constraining Module** produces constraints on the weights in order to avoid the competing conventions problem.

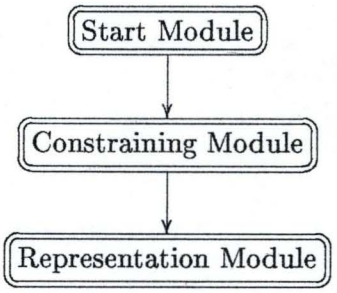


Figure 2: The Constraint Generator System (Top Level)

3. The third **Representation Module** is responsible for the translation of these constraints into the chosen genetic representation.

We now analyze the behaviour of the three components in more detail.

3.1 The Start Module

The user has to give as input the (constrained) network. As far as the network is concerned, instead of giving the whole graph, a compact and user-friendly codification can be used: the input is a list of integers specifying the number of nodes occurring in the network layers (the i-th element of the string corresponds to the i-th layer). So, for instance, the string $[5, 4, 7]$ represents a network with 5 nodes in its first layer, 4 in the second and 7 in the third. Note that this compact representation of the network graph is possible since we employ layered feedforward neural networks.

Furthermore, the user can provide the constraints on the network weights. Since we are dealing with a constraint programming language, the constraints can be very complex, due to the power of the ECL^{i}PS^e rational constraint system. Each node is assigned a unique indexing number, counting progressively from 1 till the last node. The order is from left to right, and from the first layer of the network (the input one) to the last one (the output). The weight from the i-th node to the j-th one of the next layer is indicated with $WiTj$. Then, every constraint can be expressed using these variables, the ECL^{i}PS^e operations for rational constraints (e.g. sum, difference, multiplication etc.), and the ordering relations (equality, disequality, >, <, <=, >=).

From this input, the data structures for the corresponding network are built. All this work is done by the Start module.

Observe that using constraints of the form $WiTj = 0$ we can eliminate arcs: hence, the system is also able to cope with *partially connected* layered feedforward neural networks.

3.2 The Constraining Module

In the second module, the effective computation of the constraints solving the competing conventions problem is performed. As mentioned in the introduction, this problem consists in the fact that many structurally different networks can represent the same functional mapping (see [17, 4]). This lack of a unique representation, as well known, leads to serious drawbacks with evolutionary training algorithms (see e.g. [1]).

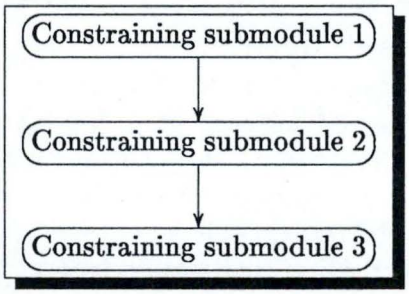


Figure 3: The Constraining Submodules Chain

The first problem is due to genetic recombination: if two functionally similar but structurally different parents are recombined via a crossover operator, the produced offspring is very likely to be completely inappropriate, since the two different encodings clash.

The second crucial problem is, of course, the explosion of the search space: with a domain much larger than the effective problem requires, the efficiency of the genetic system badly degrades (the chance of successful crossover recombinations, the prime factor to consider in genetic algorithms, decreases by far).

The importance of the above two points is substantiated by the fact the competing conventions problem affects *exponentially* the size of the domain: if a network has n hidden neurons, there are always $n!$, and in most of the cases at least $n!2^n$, different networks which are functionally the same.

3.2.1 The Constraining Submodules Chain

The idea then is to produce a suitable set of constraints that in most of the cases eliminates the competing conventions problem.

Abstractly, we can see the competing conventions problem as the existence of some transformations that take a network and give a structurally different but functionally equivalent one (*ccp-transformations* for short).

There are two main classes of ccp-transformations (see [17, 4, 1, 10]). The first class is the permutation of k nodes belonging to the same hidden layer. This affects the search space with a maximal total complexity of $n!$ (n is the number of hidden neurons).

The second class is present if the activation function is odd: given some subset of hidden nodes, flip the sign of all the weights incoming or outcoming from these nodes. This affects the search space with a maximal total complexity of 2^n.

The approach we have developed is to generate constraints by a chain of submodules, called *constraining submodules* (see Figure 3):

- A first submodule has the task to produce constraints preventing the problems arising from the first class of ccp-transformations. The idea is that for every layer but for the last two there must be at least a neuron with all of its output weights totally ordered. That is to say, if such neuron has output weights $o_1, \ldots, o_m$, there must be a permutation π of $\{1, \ldots, m\}$ such that $o_{\pi(1)} < o_{\pi(2)} < \ldots < o_{\pi(m)}$. This ordering chain will be a produced constraint. Such total ordering prevents the neurons of the successive layer to be permuted, and so repeating the argument layer by layer ensures that the first class of ccp-transformations cannot be applied. Note that a strict ordering

on some weights is generated. However, one can replace $<$ by $\leq$ in those cases where the resulting constraints are not satisfiable.

Note that at first sight it could seem that it suffices to take as π the identity permutation and to select a fixed neuron for each layer (e.g. the leftmost one). However, where the real power and flexibility of constraint logic programming plays a role, is that we have also to ensure that the produced constraints are compatible with the set of those constraints on the network weights already present: the *input constraints* provided by the user (since we are dealing with *constrained networks*), and also the constraints that can be possibly imposed *afterwards*, to get rid of other possibilities of competing conventions problems (e.g. the next two submodules). This is dealt with in a completely elegant and transparent way by integrating sorting techniques and usage of backtracking in the constraint logic program. We will come back to this point later on.

- A subsequent submodule faces the second class of ccp-transformations. The imposed requirement is that for every layer there must be a neuron with two output weights o_i and o_j that are constrained by a strict inequality $o_i < o_j$. This prevents flipping on that neuron since then it should hold $-o_i < -o_j$, that is $o_i > o_j$, a contradiction.

- Finally, a third submodule further cuts the search space by imposing ordering conditions that avoid flipping of arcs coming from the input nodes: it is imposed that there is an ordering of the input nodes such that for every input node there is an outcoming arc whose weight is strictly less than all the outcoming arcs of the successive input node.

The advantages of such implementation, besides the clarity of the program, are several.

First, it allows easy control over the constraint generations: if the user is aware of some particular competing conventions problems due to the particular activation function chosen, or due to symmetries in the data, etc., (s)he can safely add a submodule performing this task. This is also the case if (s)he knows that some cases will *not* occur, in which case some submodules which are not needed and that will maybe restrict too much the search space can be safely removed.

Second, modules are not stand-alone objects increasingly pruning the search space, but they flexibly interact. Indeed, suppose the first submodule has produced its constraints, and the second submodule is activated. It may be the case that this second submodule cannot find a consistent set of constraint, because of the constraints produced by the first submodule. In this case, the failure makes the execution *backtrack* into the first submodule, where a different constraint is generated, and the execution of the second submodule starts again. The flexibility of backtracking so allows automatic re-setting of the constraints produced by a submodule in response to the requests of the next submodules.

In this case, i.e. if the second module finitely fails, the execution is re-started from a suitable state, among the intermediate states obtained during the execution of the first module, and from a suitable point in the second submodule. This flexible technique allows the automatic re-setting of the constraints produced by a submodule in response to the requests of the next submodules.

Furthermore, we plan to implement a technique, called forward-tracking, for dealing with over-constrained information. Suppose that, for the given constrained network, the set of the constraints produced by all the submodules is inconsistent. A naïve implementation would in this case yield failure, thus being of no help. It may however be the case that the execution of the first k submodules succeeds, and that the inconsistency arises from some

of the requirements imposed by the $k+1$-th submodule. In this case we can still derive some information in order to prune the search space. The idea is to consider the constraints produced at a suitable point afterwards the execution of the k-th module, and to re-start the execution from some point forwards, in the $k+1$-th submodule. This amounts to impose a precedence among the submodules: submodules occurring in the chain in earlier positions have precedence on those coming afterwards, in the sense that they are executed before, and their results still hold in case some submodule with lower precedence yields failure. Moreover, there are also precedences *inside* a single submodule. This are deduced by exploiting the particular structure of the algorithms used by the submodules. All of them produce constraints on the network 'layer-by-layer', and so a kind of hierarchical precedence *over layers* can be introduced: if the submodule managed to produce constraints for k layers, and then yields a global inconsistency, then the execution is resumed in a forward point by considering only the set of constraints that have been produced till the k-th layer. The reader is referred to [13] for a formal definition and operational semantics of this mechanism.

Finally, let us spend some words on the way each of the two available constraining submodules is implemented. When a constraint is added, the consistency of the obtained store is automatically checked by the constraint solver of ECL^{i}PS^e. It is however important that the store is kept to a reasonable size, to avoid the introduction of too much overhead in the system. The submodules are implemented in such a way that only the constraints on the weights are mantained in the store, that is there is *no* extra information, e.g. on the structure of the network, present on the store.

3.3 The Representation Module

The choice of a suitable representation of the data is of fundamental importance for the successful application of genetic techniques. This is in general a creative task: in the field of evolutionary training, there are some studies that introduced some suitable effective representation for this application (e.g. [17, 19]), but of course it is still questionable what representation is the best one. Therefore, we have neatly separated the building of the constraint part from an actual genetic representation, devoting a module to the first task (as we have previously seen), and a stand-alone module that translates the constraints produced by the first module to constraints that employ a particular representation of the genes.

Thus, the choice of the called submodule is actually parametric: depending on the chosen representation, it is automatically activated the submodule corresponding to the desired representation. So far, we have implemented submodules for the Yoon *et al.* representation ([19]), and for another representation which progressively encodes the output weights of every node layer by layer, from left to right.

4 Computational Results and Evaluation

In this section we give some results of experiments done with the CoCo system. We did two series of experiments: one series of experiments on a standard dataset about Iris flowers [3], and a second series of experiments with a real-life data set that is used to assess how the air pollution affects on children's Peak Expiratory Flow. We did runs with and without the "competing conventions" constraints, put constraints on the weights, and used several non-differentiable error criteria. The results show that the addition of the "competing conventions"

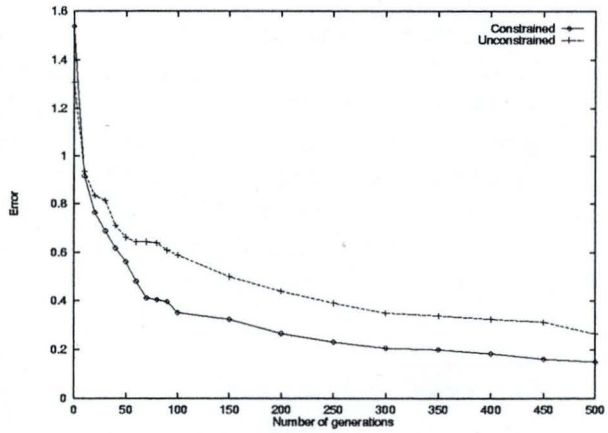


Figure 4: Iris: typical run w.r.t. squared sum of errors

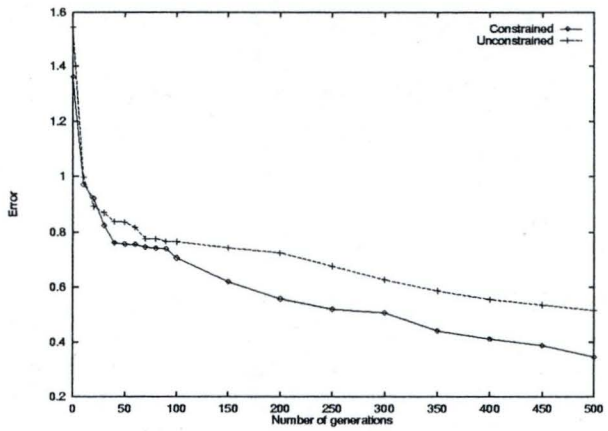


Figure 5: Iris: typical run w.r.t. the sum over the absolute values of errors

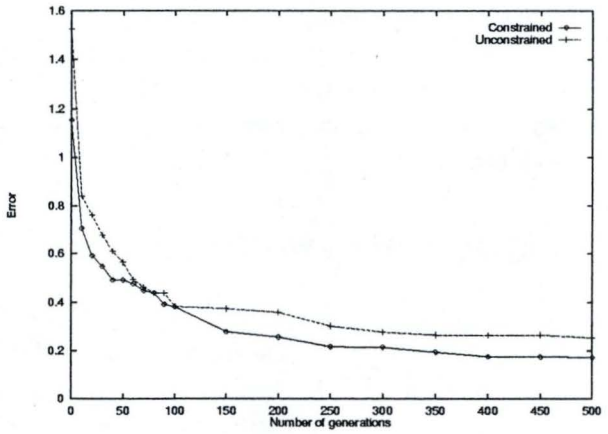


Figure 6: Iris: typical run w.r.t. the sum over the absolute values of the cube of the errors

constraints helps in the convergence. It is difficult to compare results with other methods: our method works for cases where standard learning with back-propagation does not work.

Next we describe our datasets and experiments in more detail. The Iris problem is a classic pattern recognition problem. We are given four parameters describing an Iris flower: the sepal length, the sepal width, the petal length and the petal width.

Each flower is in one of the following three classes: Iris Setosa, Iris Versicolour, and Iris Virginica. The dataset contains 150 examples, 50 for each class (it can be obtained via ftp at ftp://ics.uci.edu/pub/machine-learning-databases/iris/). We used a feedforward neural network with four input units, four hidden units and three output units. The inputs encode the four parameters, and for the output gives one of the three classes. These classes are coded by a so-called one-out-of-three coding scheme. Each output node represents one class, and for a particular input all the three output nodes are required to be zero, except for the one with the right associated class, which has to be one.

Figure 4 shows a typical run. We see that the addition of the "competing conventions"-constraints improves the convergence. Figures 5, 6 show runs with different error-criteria. Instead of the squared sum of errors (Fig. 4), we took the sum over the absolute values of errors (Fig. 5), and the sum over the absolute values of the cube of the errors (Fig. 6).

The second dataset is used for a study of possible correlations between air pollution and children's breathing. It consists of 497 samples, and is available by contacting the authors of the paper. In the experiment we used a feedforward network with 26 inputs, 15 hidden nodes and one output node. The features given as input to the network are data concerning the air pollution estimated by recording ambient levels of pollution and meteorological conditions at a fixed site on a given day (of a four month winter period) and children's pulmonary function measured with their Peak Expiratory Flow (PEF). After a training period, the network should predict the PEF of the evening of a a given day, given the morning, afternoon and evening PEF of the two days before, the morning and afternoon PEF and the data about the air pollution of that day. A typical run of the system is shown in Figure 7. We also see here that the "competing conventions"-constraints improve the convergence.

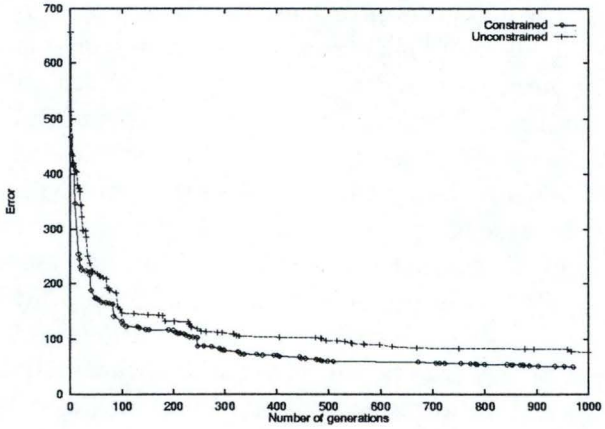


Figure 7: The Peak Expiratory Flow example

5 Related Work

We have illustrated in the previous sections how the proposed integration performs on our specific application. We discuss in the follow some related work that integrates constraint logic programming and computational intelligence.

We are aware of three related papers. A study on the possibility of integrating novel search techniques like genetic algorithms in CLP was done in the CHIC Esprit project [2]. In [11], the author has considered the effect of the combination of local search, genetic algorithms and finite domain solver. In particular, constrained genetic algorithms are proposed, where the creation of a new chromosome is supervised by a constraint solver. Experiments on various instances of the traveling salesman problems have been done, using Prolog and Chip's finite domains. This approach is orthogonal to our, because there CLP is used inside the genetic algorithm, while we use GENOCOP to deal also with constraints, and use a CLP system to produce these constraints.

Another paper on the integration of CLP and GA is [7]. Their approach is a kind of dual of the one in [11], since they use a genetic algorithm for labeling in CLP over finite integer domain. They design a genetic algorithm where variables are labeled with an integer domain, hence a chromosome encodes an area of the search space, which can contain none or many solutions. As a consequence, the genetic operators they define are rather complex, because they apply to data structures instead of values. Moreover, they report the lack of a self adaptive parameter tuning feature, which is essential to achieve a self contained optimization predicate for constraint logic programming over finite integer domains.

Finally, in [12] a programming language called PROCLANN that integrates CLP and neural networks is proposed. Their language is a committed-choice logic programming language with a stochastic constraint solver. The GENET system [18] is used as constraint solver, which translates dynamically a binary constraint problem over finite domain into a suitable neural network, and simulates the network convergence procedure.

6 Conclusion

This paper advocates the integration of constraint logic programming systems (CLP) and evolutionary systems (ES), for applications that require a first phase in which a number of constraints need to be generated, and a second phase in which an optimal solution satisfying these constraints is produced. The first phase is carried by the CLP and the second one by the ES. We have presented a specific framework where ECL^{i}PS^e and GENOCOP are integrated in the CoCo framework. We have applied this framework to the training problem for constrained neural networks. To the best of our knowledge, the idea of constraining a neural network by means of a CLP is original. Standard methods for training neural networks based on back-propagation do not apply in the presence of constraints, and alternative methods based on genetic algorithms are problem dependent, while our framework can be applied to neural networks with any set of constraints and any optimization criterion. Moreover, the results show that training neural networks using our constraint-based approach yields significant improvements.

We think that this approach (integration of CLP with ES) is also useful for many other applications with difficult optimization criteria. For our kind of applications we would need a form of Constrained optimization Logic Programming (CoLP), in which the result of a

program is not only a set of constraints, but also an optimization criterion. This optimization criterion should be build during the execution of the program, and it can depend for example on what kind of constraints we add to the store.

Acknowledgments

We would like to thank Maarten Lamers for providing the PEF dataset.

References

[1] J. Branke. Evolutionary algorithms for neural network design and training. In *Proc. of the 1st Nordic Workshop on Genetic Algorithms and its Applications.* Vaasa, Finland, 1995.

[2] A. Chamard, A. Fischler, D. Guinaudeau, and A. Guillaud. Chic lessons on CLP methodology. Technical report, ECRC, 1995. Available via ftp at ftp.ecrc.de/pub/ECRC_tech_reports/reports/.

[3] R.A. Fisher. The use of multiple measurements in taxonomic problems. *Annual Eugenics*, 7(II):179–188, 1936.

[4] P.J.B. Hancock. *Coding Strategies for Genetic Algorithms and Neural Nets.* PhD thesis, Dept. of Computing Science and Mathematics, University of Stirling, 1992.

[5] S. Haykin. *Neural Networks, A Comprehensive Foundation.* Macmillan, 1994.

[6] J. Hertz, A. Krogh, and R.G. Palmer. *Introduction to the Theory of Neural Computation.* Addison-Wesley, 1991.

[7] A. Ruiz-Andino Illera and J.J. Ruz Ortiz. Labelling in CLP(FD) with evolutionary programming. In M. Alpuente and M.I. Sessa, editors, *Proc. of the GULP-PRODE'95 Joint Conference on Declarative Programming*, pages 569–580. Poligraph Press, 1995.

[8] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *J. Logic Programming*, 19,20:503–581, 1994.

[9] H. Kitano. Empirical studies on the speed of convergence of neural network training using genetic algorithms. In *Proc. AAAI*, pages 789–795, 1990.

[10] J.N. Kok, E. Marchiori, M. Marchiori, and C. Rossi. Constraining of weights using regularities. In M. Verleysen, editor, *Proc. of the 4th European Symposium on Artificial Neural Networks.* D facto, 1996. To appear.

[11] V. Küchenhoff. Novel search and constraints - an integration. Technical Report IR-LP-92-16i, ECRC, 1992.

[12] J.H.M. Lee and V.W.L. Tam. Towards the integration of artificial neural networks and constraint logic programming. Technical Report CS-TR-94-14, The Chinese University of Hong Kong, 1994.

[13] E. Marchiori, M. Marchiori, and J.N. Kok. From failure to success with forward-tracking. Submitted, 1996.

[14] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer, 1994.

[15] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[16] H.A. Simon. Search and reasoning in artificial intelligence. *Artificial Intelligence*, 21:7–29, 1983.

[17] D. Thierens, J. Suykens, J. Vanderwalle, and B. De Moor. Genetic weight optimization of a feedforward neural network controller. In *Proc. of the Conference on Artificial Neural Nets and Genetic Algorithms*, pages 658–663. Springer-Verlag, 1993.

[18] E.P.K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[19] B. Yoon, D.J. Holmes, G. Langholz, and A. Kandel. Efficient genetic algorithms for training layered feedforward neural networks. *Information Sciences*, 76:67–85, 1994.