MASTER IN COMPUTER SCIENCE

Implementation and Applications of Ant Colony Algorithms

DARQUENNES, DENIS

Award date:
2005

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 2004-2005

Implementation and Applications of Ant Colony Algorithms

Denis Darquennes

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

Summary

There are even increasing efforts in searching and developing algorithms that can find solutions to combinatorial optimization problems. In this way, the Ant Colony Optimization Metaheuristic takes inspiration from biology and proposes different versions of still more efficient algorithms. Like other methods, Ant Colony Optimization has been applied to the traditional Traveling Salesman Problem.

The original contribution of this master thesis is to study the possibility of a modification of the basic algorithm of the Ant Colony Optimization family, Ant System, in its application to solve the Traveling Salesman Problem. In this version that we study, the probabilistic decision rule applied by each ant to determine his next destination city, is based on a modified pheromone matrix taking into account not only the last visited city, but also sequences of cities, part of previous already constructed solutions.

This master thesis presents some contribution of biology to the development of new algorithms. It explains the problem of the Traveling Salesman Problem and gives the main existing algorithms used to solve it. Finally, it presents the Ant Colony Optimization Metaheuristic, applies it to the Traveling Salesman Problem and proposes a new adaptation of its basic algorithm, Ant System.

Résumé

De nombreux efforts sont effectués en recherche et développement d'algorithmes pouvant trouver des solutions à des problèmes d'optimisation combinatoire. Dans cette optique, la Métaheuristique des Colonies de Fourmis s'inspire de la biologie et propose différentes versions d'algorithmes toujours plus efficaces. Comme d'autres méthodes, l'Optimisation par Colonies de Fourmis a été appliquée au traditionel Problème du Voyageur de Commerce.

La contribution originale de ce mémoire est d'étudier une modification de l'algorithme de base de la famille des algorithmes issus de l'Optimisation par Colonies de Fourmis, Ant System, dans son application au Problème du Voyageur de Commerce. Dans la version que nous tudions, la règle de décision probabiliste appliquée par chaque fourmis pour déterminer sa prochaine ville de destination, est basée sur une matrice de phéromones modifiée, qui tient compte non seulement de la dernière cité visitée, mais aussi de séquences de cités qui font partie de solutions construites antérieurement.

Ce mémoire présentera d'abord l'apport de certains concepts de la biologie au développement de nouveaux algorithmes . Il parlera ensuite du problème du voyageur de commerce ainsi que des principaux algorithmes existants utilisés pour le résoudre. Finalement il développe la Métaheuristique des Colonies de Fourmis, l'applique au Problème du Voyageur de Commerce et propose une nouvelle adaptation de l'algorithme de base, Ant System.

Preface

The working environment

IRIDIA is the artificial Intelligence research laboratory of the Université Libre de Bruxelles, deeply involved in theoretical and applied research in soft-computing. The major domains of competence are: (i) belief representation and AI techniques for process control and classification, (ii) nature inspired heuristics for the solution of combinatorial and continuous space optimization problems.

For the representation of quantified beliefs, IRIDIA has developed the transferable belief model, based on belief function, and is studying its relations with probability theory, possibility theory and fuzzy sets theory. This model has been applied to problems of diagnosis, decision under uncertainty, aggregation of partially reliable information and approximate reasoning.

For process control and classification, IRIDIA is developing and applying fuzzy sets theory and neural networks to problems of automated control, autonomous robotics, learning and classification encountered in the industrial applications.

For nature inspired heuristics Iridia has proposed the ant colony metaheuristic for combinatorial optimization problems, such as the traveling salesman problem, the quadratic assignment problem, the vehicle routing problem.

In all work of IRIDIA, there is still a close connection between fundamental research on imprecision and uncertainty and the development of soft computing techniques applied to industrial problems.

Overview of the master thesis

This master thesis is divided into six chapters:

Chapter 1 presents a quick introduction to the context problem and the objectives of this work.

Chapter 2 explains first Ant Colony Optimization, which is one contribution of biology in computing science. It presents after a general description of the Ant Colony Metaheuristic.

Chapter 3 first presents the Traveling Salesman Problem as a NP-complete problem. It gives then an overview of the main existing algorithms - not based on the Ant Colony Metaheurisitic - that were used to bring optimal or near-optimal solutions to this problem.

In chapter 4 we apply the Ant Colony Metaheuristic to the Traveling Salesman Problem and give an overview of the main existing algorithms of the Ant Colony Optimization family.

In chapter 5 we first explain the new idea, concerning mainly the pheromone matrix, we want to introduce in the existing basic ACO algorithm, Ant System. Then we present the different procedures that are part of this basic algorithm and the adaptations that have been made to program the new idea.

In chapter 6 we present some experimental results obtained with the new algorithms and discuss a way to improve them.

Acknowledgments

I want here to express my gratitude to Dr. Mauro Birattari and Prof. Marco Dorigo of the Université Libre de Bruxelles, researchers of the Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle (IRIDIA), who helped me to realize my master thesis.

Contents

Glossary

(artificial) ant: is a simple computational agent which constructs a solution to the problem at hand, and may deposit an amount of pheromone $\Delta\tau$ on the arcs it has traversed.

ant colony optimization (ACO): is a particular metaheuristic(*) inspired by the foraging behavior of ants.

approximate (or approximation) algorithm: is an algorithm that typically makes use of heuristics in reducing its computation but produces solutions that are not necessarily optimal.

asymmetric TSP (ATSP): is the case of the Traveling Salesman problem where the distances between the cities are dependent of the direction of traversing the arcs.

exact algorithm: is an algorithm that always produces an optimal solution.

heuristic value: the heuristic value, also called heuristic information, represents a priori information about the problem instance or run-time information provided by a source different from the ants.

intractable: problems that are known not to be solvable in polynomial time are said to be intractable.

memory depth: indicates the length of the sequence of the last cities visited by an ant.

metaheuristic: is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems.

self-organization: is a set of dynamical mechanisms whereby structures appear at the global level of a system from interactions among its lower-level components.

stigmergy: is an indirect interaction between individuals, where one of them modifies the environment and the other responds to the new environment at a later time.

swarm intelligence: is the emergent collective intelligence of groups of simple agents.

symmetric TSP: is the case of the Traveling Salesman problem where the distances between the cities are independent of the direction of traversing the arcs.

tractable: problems that are known to be solvable in polynomial time are said to be tractable.

trail pheromone: is a specific type of pheromone that some ant species use for marking paths on the ground. In algorithmic it encodes a long-term memory about the entire search process and is updated by the ants themselves.

worst-case time complexity: The time complexity function of an algorithm for a given problem Π indicates, for each possible input size n, the maximum time the algorithm needs to find a solution to an instance of that size.

Chapter 1

Introduction

1.1 The existing context

Ant Colony Optimization (ACO) is a population-based approach for solving combinatorial optimization problems that is inspired by the foraging behavior of ants and their inherent ability to find the shortest path from a food source to their nest.

ACO is the result of research on computational intelligence approaches to combinatorial optimization originally conducted by Dr. Marco Dorigo, in collaboration with Alberto Colorni and Vittorio Maniezzo.

The fundamental approach underlying ACO is an iterative process in which a population of simple agents repeatedly construct candidate solutions; this construction process is probabilistically guided by heuristic information on the given problem instance as well as by a shared memory containing experience gathered by the ants in previous iteration.

ACO Algorithm has been applied to a broad range of hard combinatorial problems. Among them, we have the classic Traveling Salesman Problem (TSP), where an individual must find the shortest route by which to visit a given number of destinations.

This problem is one of the most widely studied problems in combinatorial optimization. The problem is easy to state, but hard to solve. The difficulty becomes apparent when one considers the number of possible tours - an astronomical figure even for a relatively small number of cities. For a symmetric problem with n cities there are (n-1)!/2 possible tours, which grows exponentially with n. If n is 20, there are more than 10^{18} tours.

Many algorithmic approaches were developed to find a solution - optimal or near optimal - to this problem. Of course, it plays also an important role in ACO research: the first ACO algorithm, called Ant System, as well as many of the ACO algorithms proposed subsequently, was first tested on the TSP.

1.2 The original contribution

In Ant Colony Optimization, problems are defined in terms of components and states, which are sequences of components. Ant Colony Optimization incrementally generates solutions in the form of paths in the space of such components, adding new components to a state. Memory is kept of all the observed transitions between pairs of solution components and a degree of desirability is associated to each transition depending on the quality of the solutions in which it occurred so far. While a new solution is generated, a component y is included in a state, with a probability that is proportional to the desirability of the transition between the last component included in the state, and y itself. From that point of view, all the states finishing by the same component are identical. Further research (Birattari M., Di Caro G. and Dorigo M. (2002)) maintains that a memory associated with pairs of solution components is only one of the possible representations of the solution generation process that can be adopted for framing information about solutions previously observed.

In this master thesis, we try in a very simple way to distinguish states that are identical in Ant Colony Optimization, using a definition of the desirability of transition based on the new added component and a subsequence of the last components of the states, in place of their last component. By such modification, we hope to obtain better information about solutions previously observed and to improve the quality of the final solution.

The original contribution of the author includes the adaptation of the existing basic Ant System algorithm, mainly through the implementation of the modified memory. The adapted programs were applied on tested files. A discussion of some experimental results is given in Chapter 6.

Chapter 2

Ant Colony Optimization Metaheuristic

In this chapter, we will briefly present some basic biological notions that inspired computer scientists in their search of new algorithms for the resolution of optimization problems. We will then expose the basic elements of the Ant Colony Optimization (ACO) metaheuristic resulting of the application of these ideas in computing science.

2.1 Some contribution of biology in computing science

2.1.1 Social insects cooperation

The social insect metaphor for solving problems has become a hot topic in the last years. This approach emphasizes distributedness, direct or indirect interactions among relatively simple agents, flexibility, and robustness. This is a new sphere of research for developing a new way of achieving a form of artificial intelligence, *swarm intelligence*(Bonabeau, E., Dorigo M., & Theraulaz G (1999)) - the emergent collective intelligence of groups of simple agents. *Swarm intelligence* offers an alternative way of designing intelligent systems, in which autonomy, emergence and distributed functioning, replace control, preprogramming, and centralization.

Insects (ants, wasps and termites) that live in colonies, are able to perform different sophisticated activities like foraging, corpse clustering, larval sorting, nest building, transport cooperation and dividing labor among individuals. They solve these problems in a very flexible and robust way: flexibility allows adaptation to changing environments, while robustness endows the colony with the ability to function even though some individuals may fail to perform their tasks.

Although each individual insect is a complex creature, it is not sufficient to explain the complexity of what social insect colonies can do. The question is to know how to connect individual behavior with the collective performances, or, in other words, to know how cooperation arises.

2.1.2 Self-organization in social insects

Some of the mechanisms underlying cooperation are genetically determined, like for instance, anatomical differences between individuals. But many aspects of the collective activities of social insects are *self-organized.* Theories of self-organization (SO), originally developed in the context of physics and chemistry to describe the emergence of macroscopic patterns out of process and interactions defined at the microscopic level, can be extended to social insects to show that complex collective behavior may emerge from interactions among individuals that exhibit simple behavior: in these cases, there is no need to invoke individual complexity to explain complex collective behavior.

The researches in entomology have shown that self-organization is a major component of a wide range of collective phenomena in social insects and that the models based on it only consider insects like relatively simple interacting entities, having limited cognitive abilities.

If we now consider a social insect colony like a decentralized problem-solving system, comprised of many relatively simple interacting entities, we discover that self-organization provides us with powerful tools to transfer knowledge about social insects to the field of *intelligent system design.* The list of daily problems solved by a colony (finding food, building a nest, efficiently dividing labor among individuals, etc.) have indeed counterparts in engineering and computer science. The modeling of social insects by means of self-organization can help design decentralized, flexible and robust artificial problem-solving devices that self-organize to solve those problems-swarm intelligent systems.

MAIN IDEA

The main idea is to use the self-organizing principles of insect societies to coordinate populations of artificial agents that collaborate to solve computational problems.

Self-organization is a set of dynamical mechanisms whereby structures appear at the global level of a system from interactions among its lower-level components. The rules specifying the interactions among the system's constituent units are executed on the basis of purely local information, without reference to the global pattern, which is an emergent property of the system rather than a property imposed upon the system by an external ordering influence. For example, the emerging structures in the case of foraging in ants include spatiotemporally organized networks of pheromone trails.

Self-organization relies on four basic ingredients:

1. Positive feedback (amplification) is constituted by simple behavioral rules that promote the creation of structures. Examples of positive feedback include recruitment and reinforcement. For instance, recruitment to a food source is a positive feedback that relies on trail laying and trail following in some ant species, or dances in bees. In that last case, it has been shown experimentally that the higher the quality of source food is, the higher the probability for a bee is to dance, so allowing the colony to select the best choice.

2. Negative feedback counterbalances positive feedback and helps to stabilize the collective pattern: it may take the form of saturation, exhaustion, or competition. In the case of foraging, negative feedback stems for the limited number of available foragers, satiation, food source exhaustion, crowding at the food source, or competition between food sources.

3. Self-organization relies on the amplification of fluctuations (random walks, errors, random task-switching). Not only do structures emerge despite randomness, but randomness is often crucial, since it enables the discovery of new solutions, and fluctuations can act as seeds from which structures nucleate and grow. For example, although foragers may get lost in an ant colony, because they follow trails with some level of error, they can find new, unexploited food sources, and recruit nestmates to these food sources.

4. All cases of self-organization rely on multiple interactions. Although a single individual can generate a self-organized structure, the self-organization generally requires a minimal density of mutually tolerant individuals. They should be able to make use of the results of their own activities as well as others' activities: for instance, trail networks can self-organize and be used collectively if individuals use others' pheromone. This does not exclude the existence of individual chemical signatures or individual memory which can efficiently complement or sometimes replace responses to collective marks.

When a given phenomenon is self-organized, it can usually be characterized by a few key properties:

1. The creation of spatiotemporal structures in an initially homogeneous medium. Such structures include nest architectures, foraging trails, or social organization. For example, a characteristic well-organized pattern develops on the combs of honeybee colonies, consisting of three concentric regions: a central brood area, a surrounding rim of pollen, and a large peripheral region of honey.

2. The possible coexistence of several stable states (multistability). Because structures emerge by amplification of random deviations, any such deviation can be amplified, and the system converges to one among several possible stable states, depending on the initial conditions. For example, when two identical food sources are presented at the same distance from the nest to an ant colony that resorts to mass recruitment (based solely on trail-laying and trail-following), both of them represent possible attractors and only one will be massively exploited. Which attractor the colony will converge to depends on random initial events.

3. The existence of bifurcations when some parameters are varied. The behavior of a self-organized system changes dramatically at bifurcations. For example, some species of termite use soil pellets impregnated with pheromone to build pillars. In a first phase, the noncoordination is characterized by a random deposition of pellets. This phase lasts until one of the deposits reaches a critical size. Then the coordination phase starts if the group of builders is sufficiently large: pillars or strips emerge. The accumulation of material reinforces the attractivity of deposits through the diffusing pheromone emitted by the pellets. But if the number of builders is to small, the pheromone disappears between two successive passages by the workers, and the amplification mechanism cannot work; only the noncoordinated phase is observed. Therefore, the transition from the noncoordinated to the coordinated phase doesn't result from a change of behavior by the workers, but is merely the result of an increase in group size.

2.1.3 Stigmergy

Self-organization in social insects often requires interactions among insects: such interactions can be direct or indirect. Direct interactions consist obviously and mainly of visual or chemical contacts, trophallaxis, antennation between individuals. In the second possibility, we speak about indirect interaction between two individuals when one of them modifies the environment and the other responds to the new environment at a later time. Such an interaction is an example of stigmergy.

This concept is easily overlooked, as it does not explain the detailed mechanisms by which individuals coordinate their activities. However, it does provide a general mechanism that relates individual and colony-level behaviors: individual behavior modifies the environment, which in turn modifies the behavior of other individuals.

All these examples share some features. First they show how stigmergy can easily be made operational. That is a promising first step to design groups of artificial agents which solve problems. The second feature is the incremental construction, which is widely used in the context of optimization: a new solution is constructed from previous solutions. Finally, stigmergy is often associated with flexibility: when the environment changes because of an external perturbation, the insects respond appropriately to that perturbation, as if it were a modification of the environment caused by the colony's activities. When it comes to artificial agent, it means that the agents can respond to a perturbation without being reprogrammed to deal with that particular perturbation.

Ant colony optimization (ACO) is one of the most successfull examples of new algorithms based on those biological concepts. It is inspired by the foraging behavior of ant colonies, through their collective trail-laying and trail-following comportment, and targets discrete optimization problems. The next section will discribe it.

2.2 The ACO metaheuristic description

The combinatorial problems are easy to state but very difficult to solve. Many of them are $\mathcal{NP}$-hard, i.e. they cannot be solved to optimality within polynomially bounded computation time. The question of $\mathcal{NP}$ completeness is discussed in section 3.1

2.2.1 The metaheuristic concept

To solve large instances of combinatorial problems, it is possible to use exact algorithms, but without the certainty to obtain the optimal solution within a reasonable short time. Another strategy would then to give up the exact result, and to use approximate methods, providing near-optimal solutions in a relatively short time. Such algorithms are loosely called heuristics and often use some problem-specific knowledge to either build or improve solutions.

Among them, some constitute a particular class called METAHEURISTIC:

METAHEURISTIC
A *metaheuristic* is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems.

The use of metaheuristics has significantly increased the ability of finding very high-quality solutions to hard, practically relevant combinatorial optimization problems in a reasonable time.

As explained in previous section, a particular successful metaheuristic is inspired by the behavior of real ants. She is called Ant Colony Optimization (ACO) and will be the subject of our interest in the next paragraphs.

ACO METAHEURISTIC
The ACO *metaheuristic* is a particular metaheuristic inspired by the behavior of real ants.

In order to apply the ACO metaheuristic to any interesting combinatorial optimization problems, we have to map the considered problem to a representation that can be used by the artificial ants to build a solution.

2.2.2 Problems mapping

What follows is the definition of mapping presented in (Dorigo, M., Stützle T. (2004) Chapter 2)

Let us consider the minimization (respectively maximization) problem ($\mathcal{S}$, f, Ω), where $\mathcal{S}$ is the *set of candidate* solutions, f is the *objective function* which assigns an objective function (cost) value $f(s)$ [1] to each candidate solution $s \in \mathcal{S}$, and Ω [2] is a *set of constraints*. The parameter t indicates that the objective function and the constraints can be time-dependent, as is the case in applications to dynamic problems.

The goal is to find a *globally optimal* feasible solution s^*, that is, a minimum (respectively maximum) cost feasible solution to the minimization (respectively maximization) problem.

[1] f can be dependent in time, when we consider dynamic problems.
[2] Ω can be dependent in time, when we consider dynamic problems.

The combinatorial optimization problem $(\mathcal{S}, f, \Omega)$ is mapped on a problem that can be characterized by the following list of items:

- A finite set $C = \{c_1, c_2, \ldots, c_{N_C}\}$ of *components* is given, where N_C is the number of components.
- The *states* of the problem are defined in terms of sequences $x = \langle c_i, c_j, \ldots, c_h, \ldots \rangle$ of finite length over the elements of C. The set of all possible states is denoted by $\mathcal{X}$. The length of a sequence x, that is, the number of components in the sequence, is expressed by $|x|$. The maximum length of a sequence is bounded by a positive constant $n < +\infty$.
- The set of (candidate) solutions $\mathcal{S}$ is a subset of $\mathcal{X}$ (i.e., $\mathcal{S} \subseteq \mathcal{X}$).
- A set of feasible states $\tilde{\mathcal{X}}$, with $\tilde{\mathcal{X}} \subseteq \mathcal{X}$, defined via a problem-dependent test that verifies that it is not impossible to complete a sequence $x \in \tilde{\mathcal{X}}$ into a solution satisfying the constraints Ω. Note that by this definition, the feasibility of a state $x \in \tilde{\mathcal{X}}$ should be interpreted in a *weak* sense. In fact it does not guarantee that a completion s of x exists such that $s \in \tilde{\mathcal{X}}$.
- A non-empty set $\mathcal{S}^*$ of optimal solutions, with $\mathcal{S}^* \subseteq \tilde{\mathcal{X}}$ and $\mathcal{S}^* \subseteq \mathcal{S}$.
- A *cost* $g(s,t)$ is associated with each candidate solution $s \in \mathcal{X}$. In most cases $g(s,t) \equiv f(s,t)$, $\forall s \in \tilde{\mathcal{X}}$, where $\tilde{\mathcal{X}} \subseteq \mathcal{X}$ is the set of feasible candidate solutions, obtained from $\mathcal{S}$ via the constraints $\Omega(t)$.
- In some cases a cost, or the estimate of a cost, $J(x,t)$ can be associated with states other than candidates solutions. If x_j can be obtained by adding solution components to a state x_i, then $J(x_i, t) \leq J(x_j, t)$. Note that $J(s,t) \equiv g(s,t)$.

Given this formulation, artificial ants build solutions by performing randomized walks on a completely connected graph $G_C = (C, L)$ whose nodes are the components C, and the set of arcs L fully connects the components C. The graph G_C is called *construction graph* and elements of L are called *connections*.

The problem constraints $\Omega(t)$ are implemented in the policy followed by the artificial ants and is the subject of the next section; this choice depends on the combinatorial optimization problem considered.

2.2.3 Example of problem mapping: the graph colouring problem

The graph colouring problem is an example of problem mapped to the ACO logic. This problem can be formulated in the following way. A q-colouring of a graph (Costa, D. and Hertz, A.(1997)) $G = (V, E)$ with vertex set $V = \{v_1, \ldots, v_n\}$ and edge set E is a mapping c: $V \rightarrow \{1, 2, \ldots, q\}$ such that $c(v_i) \neq c(v_j)$ whenever E contains an edge $[i, j]$ linking the vertices v_i and v_j. The minimal number of colours q for which a q-colouring exists is called the *chromatic number* of G and is denoted $\chi(G)$. An optimal colouring is one which uses exactly $\chi(G)$ colours.

Keeping in mind the mapping of a problem as defined in the previous section, and the description of the graph colouring problem $G = (V, E)$, we first consider V as the finite set of components. The states of the problem, elements of $\mathcal{X}$, are defined in terms of sequences of finite length, in which vertices have already been assigned to colours. Defining a *stable set* as a subset of vertices whose elements are pairwise nonadjacent, then a candidate solution s, element of $\mathcal{S}$ of the colouring problem is any partition $s=(V_1, \ldots, V_q)$ of the vertex set V into q stable sets (q not fixed). The objective is then to find an optimal solution $s^* \in \mathcal{S}^*$, which corresponds to a q-coloring of G with q as small as possible.

Considering n the number of vertices of V and m the number of colours,the mathematical formulation of the problem is the following:

Since it is always possible to colour any graph $G=(V,E)$ in $n = \mid V \mid$ colours, we set $m=n$.

We define the boolean variables x_{ij} for vertex i and colour j:

$$x_{ij} = \begin{cases} 1 & \text{if vertex } i \text{ receives colour } j \\ 0 & \text{otherwise} \end{cases}$$

If the admissible set of colours for vertex j is given by:

$$J_i = \{1, \ldots, n\} \quad 1 \leq i \leq n$$

then we have that:

$$\sum_{j \in J_i} x_{ij} = 1 \quad 1 \leq n$$

The objective function to minimize is given by:

$$f(x) = \sum_{k=1}^{n} k.\delta\left(\sum_{l=1}^{n} x_{lk}\right) \quad where\ \delta(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases}$$

This function f(x) adds up the numbers associated with the colours used in the colouring x. In this way an optimal colouring uses necessarily all consecutive colours between 1 and $\chi(G)$.

A last set of constraints expressed by:

$$G_j(x) = \sum_{[v_i,v_k]\in E} x_{ij}.x_{kj} \leq 0 \quad 1 \leq j \leq n$$

avoid edges with both endpoints having the same colour.

2.2.4 The pheromone trail and heuristic value concepts

In ACO algorithms, artificial ants are stochastic constructive procedures that build solutions by moving on a construction graph $G_C = (C, L)$, where the set L fully connects the components C. The problem constraints Ω are built into the ants' constructive heuristic. In most applications, ants construct feasible solutions.

Components $c_i \in L$ and connections $l_{ij} \in L$ can have associated a PHEROMONE TRAIL τ (τ_i if associated with components, τ_{ij} if associated with connections), and a HEURISTIC VALUE η (η_i and η_{ij}, respectively):

PHEROMONE TRAIL

The pheromone trail encodes a long-term memory about the entire ant search process, and is updated by the ants themselves.

HEURISTIC VALUE

The heuristic value, also called heuristic information, represents a priori information about the problem instance or run-time information provided by a source different from the ants.

In many case, this is the cost, or an estimation of the cost, of adding the component or connection to the solution under construction.

The variables storing pheromone trail values contain informations read or written by the ants. These values are used by the ant's heuristic rule to make probabilistic decisions on how to move on the graph. They permit the indirect communication between those artificial agents, and so their cooperation, which is a key design component of ACO algorithm. The ants act concurrently and independently; the good-quality solution they found is then *an emergent property* of their cooperative interaction.

Considering the ACO Metaheuristic from the more general point of view of the Learning Process, we can say :

DISTRIBUTED LEARNING PROCESS
In a way, the ACO Metaheuristic is a distributed learning process in which the single agents, the ants, are not adaptive themselves but, on the contrary, adaptively modify the way the problem is represented and perceived by other ants.

We will now look in details the properties that characterize each artificial agent.

2.2.5 The ants' representation

What follows is the definition of ant's representation presented in (Dorigo, M., Stützle T. (2004) Chapter 2)

Each ant k of the colony has the following properties:

- It exploits the construction graph $G_C = (C, L)$ to search for optimal solutions $s^* \in \mathcal{S}^*$.
- It has a memory $\mathcal{M}^k$ that it can use to store information about the path it followed so far. Memory can be used to (1) build feasible solutions (i.e., implement constraints Ω); (2) compute the heuristic values η; (3) evaluate the solution found; and (4) retrace the path backward.
- It has a *start state* x_s^k and one or more *termination conditions* e^k. Usually, the start state is expressed either as an empty sequence or as a unit length sequence, that is, a single component sequence.
- When in state $x_r = \langle x_{r-1}, i \rangle$, if no termination condition is satisfied, it moves to a node j in its *neighborhood* $\mathcal{N}^k(x_r)$, that is, to a state $\langle x_r, j \rangle \in \mathcal{X}$. If at least one of the termination conditions e^k is satisfied, then the ant stops. When an ant builds a candidate solution, moves to infeasible states are forbidden in most applications, either through the use of the ant's memory, or via appropriately defined heuristic values η.
- It selects a move by applying a probabilistic decision rule. The probabilistic decision rule is a function of (1) the locally available pheromone trails and heuristic values (i.e., pheromone trails and heuristic values associated with components and connections in the neighborhood of the ant's current location on graph G_C); (2) the ant's private memory storing its current state; and (3) the problem constraints.
- When adding a component c_j to the current state, it can update the pheromone trail τ associated with it or with the corresponding connection.
- Once it has built a solution, it can retrace the same path backward and update the pheromone trails of the used components.

2.2.6 The implementation of the metaheuristic

An ACO algorithm is the interplay of three procedures: ConstructAntsSolutions, UpdatePheromones, and DaemonActions.

ConstructAntsSolutions manages a colony of ants that concurrently and asynchronously visit adjacent states of the considered problem by moving through neighbor nodes of the problem's construction graph G_C.

In their moves, ants apply a stochastic local decision policy, using both pheromone trail and heuristic information. In this way, ants incrementally build solutions to the optimization problem.

Once an ant has built a solution, or while the solution is being built, the ant evaluates the (partial) solution that will be used by the UpdatePheromones procedure to decide how much pheromone to deposit.

UpdatePheromone is the process by which the pheromone trails are modified. If the ants deposit pheromone on the components or connection they use, they increase the trails value. On the other hand, the pheromone evaporation contributes to decrease the trails value.

The deposit of new pheromone increases the probability that those components/connections that were either used by many ants or that were used by at least one ant and which produced a very good solution will be used again by future ants.

The pheromone evaporation implements a useful form of *forgetting* by avoiding a too rapid convergence of the algorithm toward a suboptimal region, therefore favoring the exploration of new areas of the search space.

The DeamonActions procedure is used to implement centralized actions which cannot be performed by single ants, being not in possession of the global knowledge. As examples of deamon actions, we have : the activation of a local optimization procedure, or the collection of global information that can be used to decide whether it is useful or not to deposit additional pheromone to bias the search process from a nonlocal perspective.

The ACO metaheuristic is described in pseudo-code in figure 2.1. As said before, the DeamonActions is optional.

```
procedure ACOMetaheuristic
    ScheduleActivities
        ConstructAntsSolutions
        UpdatePheromones
        DaemonActions                % optional
    end-ScheduleActivities
end-procedure
```

Figure 2.1: The pseudo-code of the ACOMetaheuristic procedure

The main procedure of the ACO metaheurisitc manages the scheduling of the three above-discussed components of ACO algorithms via the `ScheduleActivities` construct : (1) management of the ants' activity, (2) pheromone updating, and (3) daemon actions.

The `ScheduleActivities` construct does not specify how these three activities are scheduled and synchronized. The designer is therefore free to specify the way these three procedures should interact, taking into account the characteristics of the considered problem.

Chapter 3

The NP-Complete problems and the Traveling Salesman Problem

In this section, we will first quickly introduce the concepts of combinatorial problem and computational complexity. We will then define a specific combinatorial problem called the " Traveling Salesman Problem " (TSP), his main interests and variants. We will briefly describe the different algorithms used to find optimal or near-optimal solutions to this problem.

3.1 Combinatorial optimization and computational complexity

Combinatorial optimization problems involve finding values for discrete variables such that the optimal solution with respect to a given objective function is found. They can be either maximization or minimization problems which have associated a set of problem instances.

The term *problem* refers to the general problem to be solved, usually having several parameters or variables with unspecified values. The term *instance* refers to a problem with specified values for all the parameters.

An instance of a combinatorial optimization problem Π is a triple $(\mathcal{S}, f, \Omega)$, where $\mathcal{S}$ is the *set of candidate* solutions, f is the *objective function* which assigns an objective function (cost) value $f(s)$ [1] to each candidate solution $s \in \mathcal{S}$, and Ω [2] is a *set of constraints.* The solutions belonging to the set $\tilde{\mathcal{S}} \subseteq \mathcal{S}$ of candidate solutions that satisfy the constraints Ω are called *feasible solutions.* The goal is to find a *globally optimal* feasible solution s^*.

When attacking a combinatorial problem it is useful to know how difficult it is to find an optimal solution. A way of measuring this difficulty is given by the notion of worst-case complexity: a combinatorial optimization problem Π is said to have worst-case time complexity $\mathcal{O}(g(n))$ if the best algorithm known for solving Π finds an optimal solution to any instance of Π having size n in a computation time bounded from above by $const.g(n)$.

In particular, we say that Π is solvable in polynomial time if the maximum amount of computing time necessary to solve any instance of size n of Π is bounded from above by a polynomial in n. If k is the largest exponent of such a polynomial, then the combinatorial optimization problem is said to be solvable in $\mathcal{O}(n^k)$ time.

A POLYNOMIAL TIME ALGORITHM
A polynomial time algorithm is defined to be one whose computation time is $\mathcal{O}(p(n))$ for some polynomial function p, where n is used to denote the size.

EXPONENTIAL TIME ALGORITHM
Any algorithm whose computation time cannot be so bounded is called an exponential time algorithm.

An important theory that characterizes the difficulty of combinatorial problems is that of $\mathcal{NP}$-completeness. This theory classifies combinatorial problem in two main classes: those that are known to be solvable in polynomial time, and those that are not. The first are said to be *tractable*, the latter *intractable.* For the great majority of the combinatorial problems, no polynomial bound on the worst-case solution time could be found so far. The Traveling Salesman Problem (TSP) is an example of such intractable problem. The graph coloring problem is another one.

[1] f can be dependent in time, when we consider dynamic problems.
[2] Ω can be dependent in time, when we consider dynamic problems.

TRACTABLE and INTRACTABLE PROBLEM
Problems that are solvable in polynomial time are said to be tractable. Problems that are not solvable in polynomial time are said to be intractable.

The theory of $\mathcal{NP}$-completeness distinguishes between two classes of problems: the class $\mathcal{P}$ for which an algorithm outputs in polynomial time the correct answer ("yes" or "no"), and the class $\mathcal{NP}$ for which an algorithm exists that verifies for every instance in polynomial time whether the answer "yes" is correct.

A particularly important role is played by procedures called *polynomial time reductions.* Those procedures transform a problem into another one by a polynomial time algorithm. If this last one is solvable in polynomial time, so is the first one too. A problem is $\mathcal{NP}$-hard, if every other problem in $\mathcal{NP}$ can be transformed to it by a polynomial time reduction. Therefore, an $\mathcal{NP}$-hard problem is at least as hard as any of the other problem in $\mathcal{NP}$. However, $\mathcal{NP}$-hard problems do not necessarily belong to $\mathcal{NP}$. An $\mathcal{NP}$-hard problem that is in $\mathcal{NP}$ is said to be $\mathcal{NP}$-complete. The $\mathcal{NP}$-complete problems are the hardest problems in $\mathcal{NP}$: if a polynomial time algorithm could be found for an $\mathcal{NP}$-complete problem, then all problems in the $\mathcal{NP}$-complete class could be solved in polynomial time; but no such algorithm has been found until now. A large number of algorithms have been proved to be $\mathcal{NP}$-complete, including the Traveling Salesman Problem.

For more details on computational complexity, we recommend to consult the reference Garey, M.R., & Johnson, D.S. (1979).

Two classes of algorithms are available for the solution of combinatorial optimization problems: *exact* and *approximate algorithms.* Exact algorithms are guaranteed to find the optimal solution and to prove its optimality for every finite size instance of a combinatorial optimization problem within an instance-dependent run time.

If optimal solutions cannot be efficiently obtained in practice, the only possibility is to trade optimality for efficiency. In other words, the guarantee of finding optimal solutions can be sacrificed for the sake of getting very good solutions in polynomial time. Approximate algorithms, often also loosely called *heuristic methods* or simply *heuristics*, seek to obtain good, that is near-optimal solutions at relatively low computational cost without being able to guarantee the optimality of solutions. Based on the underlying techniques that approximate algorithm use, they can be classified as being either *constructive* or *local search* methods.

A disadvantage of those single-run algorithms is that they either generate only a very limited number of different solutions, or they stop at poor-quality local optima. The fact of restarting the algorithm several times from new starting solutions, often does not produce significant improvements in practice.

Several general approaches, which are nowadays often called metaheuristics, have been proposed which try to bypass these problems. A *metaheuristic* is a set of algorithmic concepts that can be used to define heuristic methods applicable to a wide set of different problems. In particular, the ant colony optimization is a metaheuristic in which a colony of artificial ants cooperate in finding good solutions to difficult discrete optimization problems.

3.2 Interest of the traveling salesman problem

The TSP is an important $\mathcal{NP}$-complete optimization problem; its popularity is due to the fact that TSP is easy to formulate, difficult to solve and has a large number of applications, even if many of them seemingly have nothing to do with traveling routes.

An example of an instance of the TSP is the process planning problem (Helsgaun, K. (2000)), where a number of jobs have to be processed on a single machine. The machine can only process one job at a time. Before a job can be processed the machine must be prepared. Given the processing time of each job and the switch-over time between each pair of jobs, the task is to find an execution sequence of the jobs making the total processing time as short as possible.

Many real-world problems can be formulated as instances of the TSP. Its versatility is illustrated in the following examples of applications areas:

- Computer wiring
- Vehicle routing
- Determination of protein structures by X-ray crystallography
- Route optimization in robotic
- Drilling of printed circuit boards
- Chronological sequencing
- Maximum efficiency or minimum cost in process allocation

3.3 Description of the traveling salesman problem

Intuitively, the traveling salesman problem is the problem faced by a salesman who, starting from his home town, wants to find the shortest possible trip through a given set of customer cities, visiting each city once before finally returning home.

The TSP can be represented by a complete weighted graph $G = (N,A)$ with N being the set of $n = |N|$ nodes (cities), A being the set of arcs fully connecting the nodes. Each arc $(i,j) \in A$ is assigned a weight d_{ij} which represents the distance between cities i and j, with $i,j \in N$.

The traveling salesman problem (TSP) is then the general problem of finding a minimum cost Hamiltonian circuit in this weighted graph, where a Hamiltonian circuit is a closed walk (a tour) visiting each node of G exactly once.

An optimal solution to an instance of the TSP can be represented as a permutation π of the node (city) indices $\{1,2,\ldots,n\}$ such that the length $f(\pi)$ is minimal, where $f(\pi)$ is given by :

$$f(\pi) = \sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} + d_{\pi(n)\pi(1)}.$$

where d_{ij} is the distance between cities i and j, and π is a permutation of $\langle 1,2,\ldots,n\rangle$.

An instance $I_N(D)$ of the TSP problem over N is defined by a distance matrix $D=(d)_{ij}$.

A solution of this problem is a vector π where $j = \pi(k)$ means that city j is visited at step k.

3.4 Different variants of the traveling salesman problem

We may distinguish between symmetric TSPs, where the distances between the cities are independent of the direction of traversing the arcs, that is, $d_{ij} = d_{ji}$ for every pair of nodes, and the asymmetric TSP (ATSP), where at least for one pair of nodes (i,j) we have $d_{ij} \neq d_{ji}$. The factor d_{ij} are used to classify problems.

SYMMETRIC TSP (STSP)

If $d_{ij} = d_{ji}, \forall i, j \in N$, the TSP problem is said to be symmetric.

ASYMMETRIC TSP (ATSP)

If $\exists i, j \in N$: $d_{ij} \neq d_{ji}$, the TSP problem is said to be asymmetric.

Based on the triangle inequality, we can also say that:

METRIC TSP (MTSP)

If the triangle inequality holds ($d_{ik} \leq d_{ij} + d_{jk}, \forall i, j, k \in N$), the problem is said to be metric.

And finally, based on the euclidean distances between points in the plane, we have:

EUCLIDEAN TSP (ETSP)

If d_{ij} are Euclidean distances between points in the plane, the problem is said to be Euclidean. A Euclidean problem is, of course, both symmetric and metric.

3.5 Exact solutions of the traveling salesman problem

The NP-Hardness results indicate that it is rather difficult to solve large instances of TSP to optimality. Nevertheless, there are computer codes that can solve many instances with thousands of vertices within days.

EXACT ALGORITHM

An *exact algorithm* is an algorithm that always produces an optimal solution.

There are essentially two methods useful to solve TSP to optimality: the integer programming approach and the dynamic programming.

3.5.1 Integer programming approaches

The classical integer programming formulation of the TSP (Laburthe, F. (1998)) is the following: define zero-one variables x_{ij} by

$$x_{ij} = \begin{cases} 1 & \text{if the tour traverses arc } (i,j) \\ 0 & \text{otherwise} \end{cases}$$

Let d_{ij} be the weight on arc (i,j). Then the TSP can be expressed as:

$$min \sum_{i,j} d_{ij} x_{ij}$$

$$\forall i, \quad \sum_{j} x_{ij} = 1$$

$$\forall j, \quad \sum_{i} x_{ij} = 1$$

$$\forall S \subset V, \quad S \neq \emptyset, \quad \sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 2$$

The first set of constraints ensures that a tour must come into vertex j exactly once, and the second set of constraints indicates that a tour must leave every vertex i exactly once. There are so two arcs adjacent to each vertex, one in and one out. But this does not prevent non-hamiltonian cycles. Instead of having one tour, the solution could consist of two or more vertex-disjoint cycles (called *sub-tours*). The role of the third set of constraints, called *sub-tour elimination constraints* is to avoid the formation of such solutions.

The formulation without the third set of constraints is an integer programming formulation of the Assignment Problem that can be solved in time $\mathcal{O}(n^3)$, each city being connected with its nearest city such that the total cost of all connection is minimized. A solution of the Assignment Problem is a minimum-weight collection of vertex-disjoint cycle $C_1, \ldots, C_t$ spanning the complete directed graph. If t=1, then an optimal solution of ATSP has been obtained. Otherwise, one can consider two or more subproblems. For example, for an arc $a \in C_i$, one subproblem could require that arc a be in the solution, and a second subproblem could require that arc a not be in the solution. This simple idea gives a basis for branch-and-bound algorithms for ATSP. Other algorithms were also developed, adding more sub-tour elimination constraints. They are called branch and cut and are more efficient for solving the TSP.

3.5.2 Dynamic programming

The dynamic programming (Laburthe, F. (1998)) is a general technique for exact resolution of combinatorial optimization problems, and consisting to explicitly enumerate the all set of solutions of the problem. This technique needs a recurrent formulation of the TSP problem. Calling an hamiltonian chain every path containing only once every vertex, if $V = \{0, \ldots, n\}$, for $\in \{1, \ldots, n\}$ and $S \subseteq \{1, \ldots, n\}, x \notin S$, we write $f(S,x)$ the length of the smallest hamiltonian chain starting from 0, visiting all vertices of S and finishing in x. f can be calculated with the recurrent function:

$$f(S, x) = min_{y \in S}(f(S - \{y\}, y) + d(y, x))$$

and the value of the optimal tour length is $f(\{1, \ldots, n\})$. The calculation of the optimal tour needs to store $n2^n$ values of f: 2^n parts of $\{1, \ldots, n\}$ for the first argument and all the values of $\{1, \ldots, n\}$ for the second argument.

The interest of the dynamic programming lies in the rapidity of the calculations for one part, and in the possibility of integration of new constraints (for instance time windows). The disadvantage comes from the memory size which is necessary for the calculations. For this last reason, this method is limited to small problems of at most 15 nodes.

3.6 Heuristic solutions of the traveling salesman problem

Exact Algorithms cannot be relied upon for applications requiring very fast solutions or ones that involve huge problem instances. Although approximate algorithms forfeit the guarantee of optimality, with good heuristics they can normally produce solutions close to optimal. In the case of the TSP, the heuristics can be roughly partitioned into two classes (Nilsson, CH.): *construction heuristics* and *improvement heuristics.*

APPROXIMATE ALGORITHM

An *approximate* (or *approximation*) *algorithm* is an algorithm that typically makes use of heuristics in reducing its computation but produces solutions that are not necessarily optimal.

CONSTRUCTION HEURISTICS

Approximate algorithms based on construction heuristics build a tour from scratch and stop when one is produced.

IMPROVEMENT HEURISTICS

Approximate algorithms based on improvement heuristics start from a tour and iteratively improve it by changing some parts of it at each iteration.

When evaluating the empirical performance of heuristics, we are often not allowed the luxury of comparing to the precise optimal tour length, since for large instances we typically do not know the optimal tour length. As a consequence, when studying large instances it has become the practice to compare heuristic results to something we can compute; the lower bound on the optimal tour length due to Held and Karp, noted ($\mathcal{HK}_b$). In case of the TSP, this bound is the solution to the linear programming relaxation of the integer programming formulation of this problem. The *excess over Held-Karp lower bound* is given by:

$$\frac{\mathcal{H}(I_{\mathcal{N}}(D)) - \mathcal{HK}_b(I_{\mathcal{N}}(D))}{\mathcal{HK}_b(I_{\mathcal{N}}(D))}.100\%$$

where $I_{\mathcal{N}}(D)$ is an instance of the TSP problem on a set $\mathcal{N}$ of n cities, with D being the matrix of distances between those cities.

We will first consider the heuristic methods coming under the tour construction.

3.6.1 Tour construction

The algorithms based on tour construction stop when a solution is found and never try to improve it. For each algorithm, the time complexity is given.

Nearest neighbor

This is the simplest and most straightforward TSP heuristic. The key of this algorithm is to always visit the nearest city.

Nearest Neighbor, $\mathcal{O}(n^2)$

1. Select a random city.
2. Find the nearest unvisited city and go there.
3. If there are unvisited cities left, repeat step 2.
4. Return to the first city.

The Nearest Neighbor algorithm will often keep its tours within 25 % of the Held-Karp lower bound.

Greedy heuristic

The Greedy heuristic gradually constructs a tour by repeatedly selecting the shortest edge and adding it to the tour as long as it doesn't create a cycle with less than N edges, or increases the degree of any node to more than 2.

Greedy, $\mathcal{O}(n^2 log_2(n))$

1. Sort all edges.
2. Select the shortest edge and add it to our tour if it doesn't violate any of the above constraints.
3. If we don't have N edges in the tour, repeat step 2.

The Greedy algorithm normally keeps within 15-20% of the Held-Karp lower bound.

Insertion heuristics

The basics of insertion heuristics is to start with a tour of a subset of all cities, and then inserting the rest by some heuristic. The initial subtour is often a triangle or the convex hull. One can also start with a single edge as subtour.

Nearest Insertion, $\mathcal{O}(n^2)$

1. Select the shortest edge, and make a subtour of it.
2. Select a city not in the subtour, having the shortest distance to any of the cities in the subtour.
3. Find an edge in the subtour such that the cost of inserting the selected city between the edge's cities will be minimal.
4. Repeat steps 2 and 3 until no more cities remain.

Convex Hull, $\mathcal{O}(n^2 log_2(n))$

1. Find the convex hull of our set of cities, and make it our initial subtour.
2. For each city not in the subtour, find its cheapest insertion (as in step 3 of the Nearest Insertion). Then choose the city with the least cost/increase ratio, and insert it.
3. Repeat step 2 until no more cities remain.

For big instances, the insertion heuristic normally keeps within 29% of the Held-Karp lower bound.

Clarke-Wright or savings algorithm

Clarke-Wright, $\mathcal{O}(n^2 log_2(n))$

The Clarke-Wright savings heuristic (Johnson, D.S., McGeoch, L.A. (1997)) is derived from a more general vehicle routing algorithm due to Clarcke and Wright. In terms of the TSP, we start with a pseudo-tour in which an arbitrarily chosen city is the hub and the salesman return to the hub after each visit to another city. For each pair of non-hub cities, let the *savings* be the amount by which the tour would be shortened if the salesman went directly from one city to the other, bypassing the hub. The next step proceeds analogously to the the Greedy algorithm, going through the non-hub city pairs in non-increasing order of savings, performing the bypass so long as it does not create a cycle of non-hub vertices or cause a non-hub vertex to become adjacent to more than two other non-hub vertices. The construction process terminates when only two non-hub cities remain connected to the hub, in which case we have a true tour.

For big instances, the savings algorithm normally keeps within 12% of the Held-Karp lower bound.

Christofides

The Christofides heuristic extends the Double Minimum Spanning Tree algorithm (complexity in $\mathcal{O}(n^2 log_2(n))$) with a worst-case ratio of 2 (i.e. a tour with twice the length of the optimal tour). This new extended algorithm has a worst-case ratio of 3/2.

Christofides Algorithm, worst-case ratio 3/2, $\mathcal{O}(n^3)$.

1. Build a minimal spanning tree from the set of all cities.
2. Create a minimum-weight matching on the set of nodes having an odd degree. Add the minimal spanning tree together with the minimum-weight matching.
3. Create an Euler cycle from the combined graph, and traverse it taking shortcuts to avoid visited nodes.

The Christofides' algorithm tends to place itself around 10% above the Held-Karp lower bound.

3.6.2 Tour improvement

Once a tour has been generated by some tour construction heuristic, it is possible to improve it by some local searches methods. Among them we mainly find 2-opt and 3-opt. Their performances are somewhat linked to the construction heuristic used.

2-opt and 3-opt

The 2-opt algorithm (see figure 3.1) removes two edges from the tour (edges (t_4, t_3) and (t_2, t_1)), and reconnects the two paths created (edges (t_4, t_1) and (t_3, t_2)). There is only one way to reconnect the two paths so that we still have a valid tour. This is done only if the new tour will be shorter and stop if no 2-opt improvement can be found. The tour is now 2-optimal.

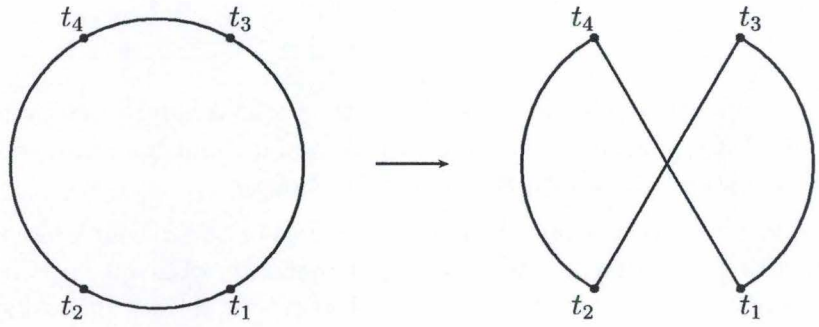

Figure 3.1 A 2-opt move

The 3-opt algorithm (see figure 3.2) works in a similar fashion, but three edges (x_1, x_2 and x_3) are removed instead of two. This means that there are two ways of reconnecting the three paths into a valid tour (for instance y_1, y_2 and y_3). A 3-opt move can be seen as two or three 2-opt moves. The search is finished when no more 3-opt moves can improve the tour.

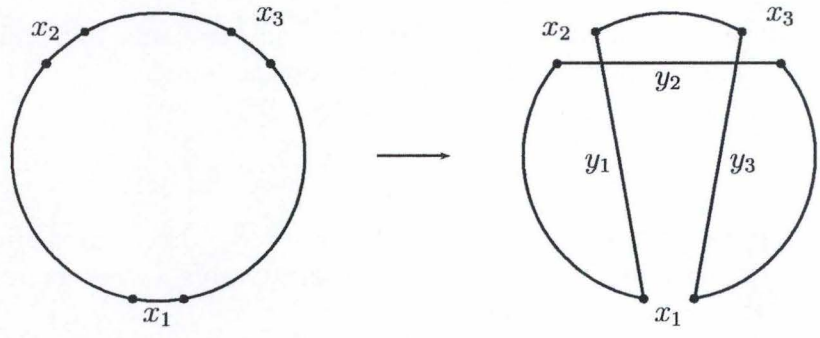

Figure 3.2 A 3-opt move

The 2-opt and 3-opt algorithms are a special case of the k-opt algorithm, where in each step k links of the current tour are replaced by k links in such a way that a shorter tour is achieved. The k-opt algorithm is based on the concept *k-optimality*:

K-OPTIMAL

A tour is said to be *k-optimal* (or simply k-opt) if it is impossible to obtain a shorter tour by replacing any k of its links by any other set of k links.

Running the 2-opt heuristic will often result in a tour with a length less than 5% above the Held-Karp bound. The improvements of a 3-opt heuristic will usually give a tour about 3% above the Held-Karp bound.

About the complexity of these k-opt algorithms, one has to notice that a move can take up to $\mathcal{O}(n)$ to perform. A naive implementation of 2-opt runs in $\mathcal{O}(n^2)$, this involves selecting an edge (c_1, c_2) and searching for another edge (c_3, c_4), completing a move only if $dist(c_1, c_2) + dist(c_3, c_4) > dist(c_2, c_3) + dist(c_1, c_4)$.

The search can be pruned if $dist(c_1, c_2) > dist(c_2, c_3)$ does not hold. This means that a large piece of the search can be cut by keeping a list of each city's closest neighbors. This extra information will of course take extra time to calculate ($\mathcal{O}(n^2 log_2 n)$). Reducing the number of neighbors in the lists will allow to put this idea in practice.

By keeping the m nearest neighbors of each city, it is possible to improve the complexity to $\mathcal{O}(mn)$. The calculation of the nearest neighbors for each city is a static information for each problem instance and needs to be done only once. It can be reused for any subsequent runs on that particular problem.

Finally, a 4-opt algorithms or higher will take more and more time and will only yield a small improvement on the 2-opt and 3-opt heuristics.

Lin-Kernighan

The Lin-Kernighan algorithm (LK) is a variable k-opt algorithm. The main idea is to decide at each step which k is the most suitable to reduce at maximum the length of the current tour.

Those k-opt moves are seen as a sequence of 2-opt moves. Every 2-opt move always deletes one of the edge added by the previous move. The algorithm is described below:

Let T be the current tour. At each iteration step, the algorithm attempts to find two sets of links, $X = \{x_1, \ldots, x_r\}$ and $Y = \{y_1, \ldots, y_r\}$, such that, if the links of X are deleted from T and replaced by the links of Y, the result is a better tour. This interchange of links is a r-opt move. The two sets X and Y are constructed element by element. Initially, X and Y are empty. In step i a pair of links, x_i and y_i, are added to X and Y respectively.
In order to achieve a sufficient efficient algorithm, only links that fulfill the following criteria may enter X and Y.

1. The sequential exchange criterion (see figure 3.3): x_i and y_i must share an endpoint, and so must y_i and x_{i+1}. If t_1 denotes one of the two endpoints of x_1, we have in general that: $x_i = (t_{2i-1}, t_{2i}), y_i = (t_{2i}, t_{2i+1})$ and $x_{i+1} = (t_{2i+1}, t_{2i+2})$ for $i \geq 1$.

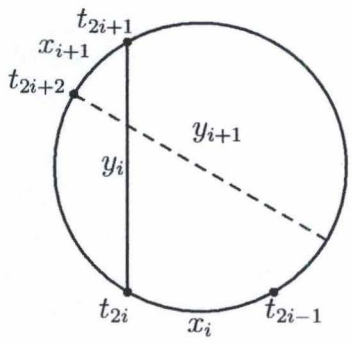

Figure 3.3 Restricting the choice of x_i, y_i, x_{i+1} and y_{i+1}.

2. The feasibility criterion: It is required that $x_i = (t_{2i-1}, t_{2i})$ is chosen so that, if t_{2i} is joined to t_1, the resulting configuration is a tour. This criterion is used for $i \geq 3$ and guarantees that it is possible to close up a tour.

3. The positive gain criterion: It is required that y_i is always chosen so that the gain, G_i, from the proposed set of exchanges is positive. If we suppose $g_i = c(x_i) - c(y_i)$ is the gain from exchanging x_i with y_i, then G_i is the sum $g_1 + g_2 + \ldots + g_i$.

4. The disjunctivity criterion: It is required that the sets X and Y are disjoint.

So the basic algorithm limits its search by using the following four rules:

1. Only sequential exchanges are allowed.
2. The provisional gain must be positive.
3. The tour can be 'closed'.
4. A previously broken link must not be added, and a previously added link must not be broken.

In order to limit or to direct the search even more, additional rules were introduced:

1. The search for a link to enter the tour, $y_i = (t_{2i}, t_{2i+1})$, is limited to the five nearest neighbors to t_{2i}.
2. Th search for improvements is stopped if the current tour is the same as previous solution tour.
3. When link $y_i (i \geq 2)$ is to be chosen, each possible choice is given the priority $c(x_{i+1}) - c(y_i)$.

The two first rules save running time (30 to 50 percent), but sometimes at the expense of not achieving the best possible solutions. If the algorithm has a choice of alternatives, the last rule permits to give priorities to these alternatives, by ranking the links to be added to Y. The priority for y_i is the length of the next link to be broken, x_{i+1}, if y_i is included in the tour, minus the length of y_i. By maximizing the quantity $c(x_{i+1}) - c(y_i)$, the algorithm aims at breaking a long link and including a short link.

The time complexity of LK is $\mathcal{O}(n^{2.2})$, making it slower than a simple 2-opt implementation. This algorithm is considered to be one of the most effective methods for generating optimal or near-optimal solutions for the TSP.

Tabu-search

A neighborhood-search algorithm searches among the neighbors of a candidate solution to find a better one. Such process can easily get stuck in a local optimum. The use of tabu-search can avoid this by allowing moves with negative gain if no positive one can be found. By allowing negative gain we may end up running in circles, as one move may counteract the previous. To avoid this, the tabu-search keeps a tabu-list containing illegal moves. After moving to a neighboring solution the move will be put on the tabu-list and will thus never be applied again unless it improves the best tour or the tabu has been pruned from the list.

There are several ways to implement the tabu list. One involves adding the two edges being removed by a 2-opt move to the list. Another way is to add the shortest edge removed by a 2-opt move, and then making any move involving this edge tabu.

Most implementations for the TSP in tabu-search will take $\mathcal{O}(n^3)$, making it far slower than a 2-opt local search. Given that we use 2-opt moves, the length of the tours will be slightly better than that of a standard 2-opt search.

Simulated annealing

Simulated Annealing (SA) has been successfully adapted to give approximate solutions for the TSP. SA is basically a randomized local search algorithm allowing moves with negative gain. An implementation of SA for the TSP uses 2-opt moves to find neighboring solutions. The resulting tours are comparable to those of a normal 2-opt algorithm. Better results can be obtained by incorporating neighborhood lists, so that the algorithm can compete with the LK algorithm.

Genetic Algorithms

Genetic Algorithms (GA) work in a way similar to nature. An evolutionary process takes place within a population of candidate solutions. A basic Genetic Algorithm starts out with a randomly generated population of candidate solutions. Some (or all) candidates are then mated to produce offspring and some go through a mutating process. Each candidate has a fitness value telling us how good they are. By selecting the most fit candidates for mating and mutation the overall fitness of the population will increase.

Applying GA to the TSP involves implementing a crossover routine, a mutation routine and a measure of fitness. Some implementations have shown good results, even better than the best of several LK runs, but running time is an issue.

3.7 Synthesis of the different algorithms

The following table present a synthesis of the different algorithms previously presented. For 2-opt and 3-opt, m represents the nearest neighbors of each city. In the following table, $\mathcal{HK}$ means Held-Karp lower bound.

Algo.	**Complexity**	**Sol. quality**
Near. Neighbor	$\mathcal{O}(n^2)$	25% $\mathcal{HK}$
Greedy	$\mathcal{O}(n^2 log_2(n))$	15%-20% $\mathcal{HK}$
Insertion	$\mathcal{O}(n^2 log_2(n))$	29% $\mathcal{HK}$
Christofides	$\mathcal{O}(n^3)$	10% $\mathcal{HK}$
2-opt 3-opt	$\mathcal{O}(mn)$	3% $\mathcal{HK}$
Saving Algo.	$\mathcal{O}(n^2 log_2(n))$	12% $\mathcal{HK}$
Sim. Annealing	$\mathcal{O}(n^2)$	3% $\mathcal{HK}$
Lin-Kernighan	$\mathcal{O}(n^{2.2})$	318 cities in 1 sec; optimal solution for 7397 cities
Tabu Search	$\mathcal{O}(n^3)$	3% $\mathcal{HK}$

Chapter 4

Ant Colony Optimization and the Traveling Salesman Problem

4.1 Application of the ACO algorithms to the TSP

ACO can be applied to the TSP in a straightforward way.

- *Construction graph:* The construction graph is identical to the problem graph: the set of components C is identical to the set of nodes (i.e., C=N), the connections correspond to the set of arcs (i.e., L=A), and each connection has a weight which corresponds to the distance d_{ij} between nodes i and j. The states of the problem are the set of all possible partial tours.
- *Constraints:* The only constraint in the TSP is that all cities have to be visited and that each city is visited at most once. This constraint is enforced if an ant at each construction step chooses the next city only among those it has not visited yet (i.e., the feasible neighborhood $\mathcal{N}_i^k$ of an ant k in city i, where k is the ant's identifier, comprises all cities that are still unvisited).

- *Pheromone trails and heuristic information:* The pheromone trails τ_{ij} in the TSP refer to the desirability of visiting city j directly after i. The heuristic information η_{ij} is typically inversely proportional to the distance between cities i and j, a straightforward choice being $\eta_{ij} = 1/d_{ij}$.

- *Solution construction:* Each ant is initially placed in a randomly chosen start city and at each step iteratively adds one unvisited city to its partial tour. The solution construction terminates once all cities have been visited.

Tours are constructed by applying the following simple constructive procedure to each ant: after having chosen a start city at which the ant is positioned, (1) use pheromone and heuristic values to probabilistically construct a tour by iteratively adding cities that the ant has not visited yet, until all cities have been visited; and (2) go back to the initial city.

4.2 Ant system and its direct successors

The first ACO algorithm, Ant System (AS), was developed by Professor Dorigo in 1992 (Dorigo, 1992). This algorithm was introduced using the TSP as an example application. AS achieved encouraging initial results, but was found to be inferior to state-of-the-art algorithms for the TSP. The importance of AS therefore mainly lies in the inspiration it provided for a number of extensions that significantly improved performance and are currently among the most successful ACO algorithms. In fact most of these extensions are direct extensions of AS in the sense that they keep the same solution construction procedure as well as the same pheromone evaporation procedure. These extensions include elitist AS, rank-based AS, and $\mathcal{MAX} - \mathcal{MIN}$ AS. The main differences between AS and these extensions are the way the pheromone update is performed, as well as some additional details in the management of the pheromone trails. A few other ACO algorithms that more substantially modify the features of AS were also developed; those algorithms are the Ant Colony System (ACS), the Approximate Nondeterministic Tree Search and the Hyper-Cube Framework for ACO. Only the ACS will be briefly presented; for the others, we invite the reader to consult the reference (Dorigo, M., Stützle T. (2004) Chapter 3).

Those algorithms are presented in the order of increasing complexity in the modifications they introduce with respect to AS.

4.2.1 The ant system

In the early 1991, three different versions of AS (Dorigo, M., Maniezzo, V., Colorni, A. (1991a)) were developed: they were called *ant-density*, *ant-quantity* and *ant-cycle*.Whereas in the ant-density and ant-quantity versions the ants updated the pheromone directly after a move from one city to an adjacent city, in the ant-cycle version the pheromone update was only done after all the ants had constructed the tours and the amount of pheromone deposited by each ant was set to be a function of the tour quality. Due to their inferior performance the ant-density and ant-quantity versions were abandoned and the actual AS algorithm only refers to the ant-cycle version.

The two main phases of the AS algorithm are the ants' solution construction and the pheromone update. The initialization of the pheromone trails is made by a value slightly higher than the expected amount of pheromone deposited by the ants in one iteration; a rough estimation of this value is obtained by setting, $\forall(i,j), \tau_{ij} = \tau_0 = m/C^{nn}$, where m is the number of ants, and C^{nn} is the length of a tour generated by the nearest-neighbor heuristic.

The reason for this choice is that if the initial pheromone values τ_0's are too low, then the search is quickly biased by the first tours generated by the ants, which in general leads toward the exploration of inferior zones of the search space. On the other hand, if the initial pheromone values are too high, then many iterations are lost waiting until pheromone evaporation reduces enough pheromone evaporation, so that pheromone added by ants can start to bias the search.

Tour construction

In AS, m artificial ants concurrently build a tour of the TSP. At each construction step, ant k applies a probabilistic action choice rule, called *random proportional rule*, to decide which city to visit next. In particular, the probability with which ant k currently at city i, chooses to go to city j is

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i{}^k} [\tau_{ij}]^\alpha [\eta_{ij}]^\beta}, \qquad \text{if } j \in \mathcal{N}_i^k, \tag{4.1}$$

where $\eta_{ij} = 1/d_{ij}$ is a heuristic that is available a priori, α and β are two parameters which determine the relative influence of the pheromone and the heuristic information, and $\mathcal{N}_i^k$ is the feasible neighborhood of ant k when being at city i, that is, the set of cities that ant k has not visited yet. By this probabilistic rule, the probability of choosing a particular arc (i, j) increases with the value of the associated pheromone trail τ_{ij} and of the heuristic information value η_{ij}.

The discussion about the values of the parameters α and β is the following: if $\alpha = 0$, the closest cities are more likely to be selected; if $\beta = 0$, only the pheromone is at work, without any heuristic bias. This generally leads to rather poor results and, in particular, for $\alpha > 1$ it leads to the rapid emergence of a *stagnation* situation, that is, a situation in which all the ants follow the same path and construct the same tour, which, in general, is strongly suboptimal.

Each ant k maintains a memory $\mathcal{M}^k$ which contains the cities already visited, in the order they were visited. This memory is used to define the feasible neighborhood $\mathcal{N}_i^k$ in the construction rule given by equation (4.1). This memory also allows ant k both to compute the length of the tour T^k it generated and to retrace the path to deposit pheromone.

Update of pheromone trails

After all the ants have constructed their tours, the pheromone trails are updated. First the pheromone values on all arcs are lowered by a constant factor, after what pheromone values are added on the arcs the ants have crossed in their tours. Pheromone evaporation is implemented by

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} \tag{4.2}$$

where $0 < \rho \leq 1$ is the pheromone evaporation rate. Evaporation avoids unlimited accumulation of the pheromone trails and enables the algorithm to forget bad decisions previously taken. After evaporation, all ants deposit pheromone on the arcs they have crossed in their tour:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k, \qquad \forall (i,j) \in L, \tag{4.3}$$

where $\Delta\tau_{ij}^k$ is the amount of pheromone ant k deposits on the arcs it has visited. It is defined as follows:

$$\Delta\tau_{ij}^{k} = \begin{cases} 1/C^{k}, & \text{if arc } (i,j) \text{ belongs to } T^{k}; \\ 0, & \text{otherwise;} \end{cases} \tag{4.4}$$

where k, the length of the tour T^k built by the k-th ant, is computed as the sum of the lengths of the arcs belonging to T^k. By means of equation (4.4), the better an ant's tour is, the more pheromone the arcs belonging to this tour receive. In general, arcs that are used by many ants and which are part of short tours, receive more pheromone and are therefore more likely to be chosen by ants in future iterations of the algorithm.

4.2.2 The elitist ant system

A first improvement on the initial AS, called the *elitist strategy* for Ant System (EAS), was introduced by Dorigo (Dorigo, 1992; Dorigo et al., 1991a, 1996). The idea is now to provide strong additional reinforcement to the arcs belonging to the best tour found since the start of the algorithm; this tour is denoted T^{bs} (*best-so-far* tour) in the following.

Update of pheromone trails

The additional reinforcement of tour T^{bs} is achieved by adding a quantity e/C^{bs} to its arcs, where e is a parameter that defines the weight given to the best-so-far tour T^{bs}, and C^{bs} is its length. The equation for the pheromone deposit is now:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} + e\Delta\tau_{ij}^{bs}, \tag{4.5}$$

where $\Delta\tau_{ij}^{k}$ is defined as in equation(4.4) and $\Delta\tau_{ij}^{bs}$ is defined as follows:

$$\Delta\tau_{ij}^{bs} = \begin{cases} 1/C^{bs}, & \text{if arc } (i,j) \text{ belongs to } T^{bs}; \\ 0, & \text{otherwise;} \end{cases} \tag{4.6}$$

In EAS, the pheromone evaporation stay implemented as it is in AS.

4.2.3 The ranked-based ant system

In the next improved version, called the *rank-based* version of AS (AS_{rank}) (Bullnheimer et al., 1999c), each ant deposits an amount of pheromone that decreases with its rank. Additionally, as in EAS, the best-so-far ant always deposits the largest amount of pheromone in each direction.

Update of pheromone trails

Before updating the pheromone trails, the ants are sorted by increasing tour length and the quantity of pheromone an ant deposits is weighted according to the rank r of the ant. In each iteration only the (w-1) best-ranked ants and the ant that produced the best-so-far tour are allowed to deposit pheromone.

The best-so-far tour gives the strongest feedback, with weight w; the r-th best ant of the current iteration contributes to pheromone updating with the value $1/C^r$ multiplied by a weight given by $\max\{0, w-r\}$. Thus, the AS_{rank} pheromone update rule is:

$$\tau_{ij} \leftarrow \tau_{ij} + \sum_{r=1}^{w-1}(w-r)\Delta\tau_{ij}^{r} + w\Delta\tau_{ij}^{bs}, \qquad (4.7)$$

where $\Delta\tau_{ij}^{r} = 1/C^{r}$ and $\Delta\tau_{ij}^{bs} = 1/C^{bs}$.

4.2.4 The max-min ant system

The next version, called $\mathcal{MAX} - \mathcal{MIN}$ Ant System ($\mathcal{MM}$AS) (Stützle & Hoos, 1997, 2000; Stützle, 1999), introduces four main modifications with respect to AS.

First, it strongly exploits the best tours found: only either the iteration best-ant, that is, the ant that produced the best tour in the current iteration, or the best-so-far ant is allowed to deposit pheromone. Unfortunately, such a strategy may lead to a stagnation situation in which all the ants follow the same tour, because of the excessive growth of pheromone trails on arcs of a good, although suboptimal, tour.

To counteract this effect, a second modification has been introduced by $\mathcal{MM}$AS: the limitation of the possible range of pheromone trail values to the interval $[\tau_{min}, \tau_{max}]$.

Third, the pheromone trails are initialized to the upper pheromone trail limit, which, together with a small pheromone evaporation rate, increases the exploration of tours at the start of the search.

Finally, in $\mathcal{MM}$AS, pheromone trails are initialized each time the system approaches stagnation or when no improved tour has been generated for a certain number of consecutive iterations.

Update of pheromone trails

After all ants have constructed a tour, pheromones are updated by applying evaporation as in AS, followed by the deposit of new pheromone as follows:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{best}, \qquad (4.8)$$

where $\Delta\tau_{ij}^{best} = 1/C^{best}$. The ant which is allowed to add pheromone may be either the best-so-far, in which case $\Delta\tau_{ij}^{best} = 1/C^{bs}$, or the iteration-best, in which case $\Delta\tau_{ij}^{best} = 1/C^{ib}$, where C^{ib} is the length of the iteration-best tour. In general, in $\mathcal{MM}$AS implementations both the iteration-best and the best-so-far update rules are used, in an alternate way.

Pheromone trail limits

In $\mathcal{MM}$AS, lower and upper limits τ_{min} and τ_{max} on the possible pheromone values on any arc are imposed in order to avoid search stagnation. In particular, the imposed pheromone trail limits have the effect of limiting the probability p_{ij} of selecting a city j when an ant is in city i to the interval $[\tau_{min}, \tau_{max}]$, with $0 < p_{min} \leq p_{ij} \leq p_{max} \leq 1$.

Update of pheromone trails

At the start of the algorithm, the initial pheromone trails are set of the upper pheromone trail limit. This way of initializing the pheromone trails, in combination with a small pheromone evaporation parameter, causes a slow increase in the relative difference in the pheromone trail levels, so that the initial search phase of $\mathcal{MM}$AS is very explorative.

As a further means of increasing the exploration of paths that have only a small probability of being chosen, in $\mathcal{MM}$AS pheromone trails are occasionally reinitialized. Pheromone trail reinitialization is typically triggered when the algorithm approaches the stagnation behavior or if for a given number of algorithm iterations no improved tour is found.

4.2.5 The ant colony system

In this new version, called ACS (Dorigo & Gambardella, 1997a,b), a new mechanism based on idea not included in the original AS is introduced. It differs from this last one in three main points.

First, it exploits the search experience accumulated by the ants more strongly than AS does through the use of a more aggressive action choice rule. *Second*, pheromone evaporation and pheromone deposit take place only on the arcs belonging to the best-so-far tour. *Third*, each time an ant use an arc (i, j) to move from city i to city j, it removes some pheromone from the arc to increase the exploration of alternative paths.

Tour Construction

In ACS, when located at city i, ant k moves to a city j chosen according to the so called *pseudorandom proportional rule*, given by

$$j = \begin{cases} \text{argmax}_{l \in \mathcal{N}_i^k} \{\tau_{il}[\eta_{il}]^\beta\} & \text{if } q \leq q_0; \\ J, & \text{otherwise;} \end{cases} \tag{4.9}$$

where q is a random variable uniformly distributed in $[0, 1]$, $q_0 (0 \leq q_0 \leq 1)$, is a parameter, and J is a random variable selected according to the probability distribution given by equation (4.1) (with $\alpha = 1$).

The ant exploits the learned knowledge with probability q_0, making the best possible move as indicated by the learned pheromone trails and the heuristic information, while with probability $(1 - q_0)$ it performs a biased exploration of the arcs. Tuning the parameter q_0 allows to modulate the degree of exploration and to chose of whether to concentrate the search of the system around the best-so-far solution or to explore other tours.

Global Pheromone Trail Update

In ACS only one ant (the best-so-far ant) is allowed to add pheromone after each iteration. The update in ACS is implemented by the following equation:

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} + \rho\Delta\tau_{ij}^{bs}, \forall(i,j) \in T^{bs}, \tag{4.10}$$

where $\Delta\tau_{ij}^{bs} = 1/C^{bs}$. The main difference between ACS and AS is that the pheromone trail update, both evaporation and new pheromone deposit, only applies to the arcs of T^{bs}, not to all the arcs. As usual, the parameter ρ represents the pheromone evaporation; unlike in AS's equations (4.3) and (4.4), in equation (4.9) the deposited pheromone is discounted by a factor ρ; this results in the new pheromone trail being a weighted average between the old pheromone value and the amount of pheromone deposited.

Local Pheromone Trail Update

In ACS the ants use a local pheromone update rule that they apply immediately after having crossed an arc (i,j) during the tour construction:

$$\tau_{ij} \leftarrow (1-\xi)\tau_{ij} + \xi\tau_0, \tag{4.11}$$

where $\xi, 0 < \xi < 1$, and τ_0 are two parameters. The value for τ_0 is set to be the same as the initial value for the pheromone trails. Experimentally, a good value for ξ was found to be $1/nC^{nn}$, where n is the number of cities in the TSP instance and C^{nn} is the length of a nearest-neighbor tour.

The effect of the local updating rule is that each time an ant uses an arc (i,j) its pheromone trail τ_{ij} is reduced, so that the arcs becomes less desirable for the following ants. This strategy allows an increase in the exploration of arcs that have not been visited yet and has the effect that the algorithm does not show a stagnation behavior.

Finally, in ACS, this local pheromone update rule asks for a tour construction in parallel by the ants, while it did not matter in the algorithms previously explained.

4.2.6 Synthesis of the different algorithms

The following table present a synthesis of the different algorithms previously explained.

Algo.	**Tour Construct.**	**Evaporation**	**Updating**
AS	random proportional rule	all arcs lowered with constant factor ρ	deposit on all arcs visited by all ants
EAS	random proportional rule	all arcs lowered with constant factor ρ	AS with additional reinforcement of best-so-far tour
AS_{rank}	random proportional rule	all arcs lowered with constant factor ρ	ants sorted by increasing tour length; deposit weighted according to the rank of each ant
$\mathcal{MM}$AS	random proportional rule	all arcs lowered with constant factor ρ	deposit only either by the iteration best-ant, or the best-so-far ant; interval $[\tau_{min}, \tau_{max}]$
ACS	pseudorandom proportional rule	only arcs of the best-so-far tour are lowered	deposit only on arcs of the best-so-far tour

4.2.7 Experimental parameters for the different algorithms

The experimental study of the various ACO algorithms for the TSP has identified parameter settings that result in good performance. In the following table, parameter n represents the number of cities in a TSP instance and m is the number of ants.

ACO Algorithm	α	β	ρ	m	τ_0
AS	1	2 to 5	0.5	n	m/C^{nn}
EAS	1	2 to 5	0.5	n	$(e+m)/\rho C^{nn}$
AS_{rank}	1	2 to 5	0.1	n	$0.5r(r-1)/\rho C^{nn}$
$\mathcal{MM}$AS	1	2 to 5	0.02	n	$1/\rho C^{nn}$
ACS	-	2 to 5	0.1	10	$1/nC^{nn}$

In EAS, the parameter e (which defines the weight given to the best-so-far tour - see formula 4.5) is set to $e = n$. In AS_{rank}, the number of ants that deposit pheromones is w=6 and parameter r is the rank (see formula 4.7).

Chapter 5

The Effect of Memory Depth

As explained in the introduction, there are theoretical reasons to hope an improvement of the solution obtained with the classical version of Ant Algorithm. In this work we tried to observe this effect by modifying in a very simple way the pheromone matrix τ. The first section exposes the reason of the modifications to the pheromone matrix; the next section concentrates on the adaptation of the pheromone matrix; finally the last section explores the idea of a serialization of the algorithms developed with the new pheromone matrix.

5.1 The modified pheromone matrix TAU

In the existing versions of the AS algorithm, the pheromone values are stored in a symmetric n^2 matrix. For each connection (i, j) between city i and city j, a number τ_{ij} corresponding to the pheromone trail associated with that connection is stored in this matrix. The symmetry of our test problem (TSP) implies that $\forall i, j, \tau_{ij} = \tau_{ji}$ and explains the symmetric property of the pheromone matrix. As expected, with the pheromone matrix so defined, AS works well and produces near-optimal solutions to TSP problem.

This construction of the τ matrix is clearly overgeneralizing. It doesn't take into account, in the solution under construction, of the sequences already used by the ants as they were constructing previous solutions. Considering pheromone values now associated to sequences of visited cities, we hope to observe a general improvement of the quality of solutions, better than those obtained with AS.

The choice of the city to move to will now take into account not only the current city i where the ant stands, but also the previous cities where the ant stood before arriving at city i. Working this way, the decision for the next movement will be seen as a continuation of previous sequences, which are part of solutions constructed before. Each pheromone value will now be attached to a certain sequence of n already visited cities, and stored in a new pheromone matrix called *Tau*. This remembrance of the most recent already visited cities is called the *memory depth* and is at the base of a possible improvement of the resulting solutions issued from the algorithm. Only the global memory of the learning process has been modify, in a sense of taking into account not only the last visited city, but also the cities already visited just before.

MAIN IDEA

In traditional ACO, each pheromone value in the global indirect memory is associated to a pair current city (where an ant stands) - next city (possibly to be visited by this ant). In the modified version of the pheromone matrix, each pheromone value of the global indirect memory is still associated to a pair where the first element is now a sequence of visited cities (the last one of this sequence being the current city where an ant stands), and the second element is the next city (possibly to be visited by this ant).

In this way we hope that the entire process will converge to a more precise near optimal solution than this one found with the classical version of AS algorithm.

5.1.1 The classical and the modified pheromone matrix

We will here describe both the actual and new forms of the pheromone matrix presented before. It is clear that this new matrix will have an influence on some procedures of the AS algorithm. The detailed description of the AS algorithm, and his adaptations are the subject of the next section.

The classical pheromone matrix. ACO is characterized as being a distributed, stochastic search method based on the indirect communication of a colony of artificial ants, mediated by artificial pheromone trails. The pheromone trails in ACO serve as distributed numerical information, used by the ants to probabilistically construct solutions to the problem under consideration, and so constitutes an indirect form of memory of previous performance. The ants modify the pheromone trails during the algorithm's execution to reflect their search experience. This memory is a key element in the global learning process of the ants, working to construct a solution.

The goal of this work is to modify the pheromone matrix in such a way that it will make more efficient the search process of the ants. We expect finally to obtain a better solution, than with the actual version of AS algorithm. This one is based on a pheromone matrix described hereafter (figure5.1):

	1	2	3	...	j		n
1	-	$\tau_{1,2}$	$\tau_{1,3}$	...	$\tau_{1,j}$		$\tau_{1,n}$
2	$\tau_{2,1}$	-	$\tau_{2,3}$	...	$\tau_{2,j}$		$\tau_{2,n}$
⋮	⋮		$\ddots$				⋮
⋮	⋮			$\ddots$			⋮
i	$\tau_{i,1}$				- $\tau_{i,j}$		$\tau_{i,n}$
⋮	⋮				$\ddots$		⋮
⋮	⋮					$\ddots$	⋮
n	$\tau_{n,1}$	$\tau_{n,2}$	...		$\tau_{n,j}$	$\tau_{n,n-1}$	-

Figure 5.1: The classical pheromone matrix, for a set of n cities. There are no pheromone values on the diagonal.

This is a square matrix of size n^2, where n is the maximum number of cities in the considered instance problem. There are of course no pheromone values on the diagonal of this matrix. This matrix is also symmetric, i.e. that $\tau_{ij} = \tau_{ji}$ $\forall i, j$ with $i \neq j$.

The new pheromone matrix. The previous description of the pheromone matrix is possibly overgeneralizing. Each pheromone value is only based on the city where the ant stands, and the city where this ant eventually wants to move to. It doesn't consider the ordered sequence of the previous already visited cities standing before the last visited city by the ant. If we consider for instance a sequence of three last visited cities, the new pheromone matrix, called TAU, to be considered is (figure 5.2):

	1	2	...	i	...	j	...	k	...	m	...	n
⋮												
i				x		$\tau_{i,j}$						
⋮												
ik				x		$\tau_{ik,j}$		x				
⋮												
ikm				x		$\tau_{ikm,j}$		x		x		
⋮												

Figure 5.2: The new pheromone matrix TAU for a set of n cities, considering sequences constituted by the three last visited cities. "x" indicates forbidden pheromone values.

This is no more a square matrix. On the top of this table we find the indeces of the cities to be visited; each index defines a column of the matrix; on the left side of the table we find sequences of cities index. We first consider the sequences constituted of one city, then all possible ordered sequences constituted from couple of cities, and finally all possible ordered sequences constituted from three cities. Each previous sequence (from 1,2 or 3 cities) defines a line of the matrix. It will be necessary in the implementation to calculate the number of a line, knowing the index of each city being part of a sequence.

The first block of the matrix is constituted by the pheromones $\tau_{i,j}$ associated to a couple of cities, city i that the ant is just leaving and city j that the ant is moving to; at city i corresponds a line of the matrix and at city j a column of the matrix; this block is used by the algorithm during the first iteration, when the partial solution of each ant comprises only one city, the departure city. In this block, there are no pheromone values in case index i = index j.

The second block of the matrix is constituted by the pheromones $\tau_{ik,j}$ associated to three cities, city k that the ant is just leaving, city i that was visited just before city k and city j that the ant is moving to; at the sequence of two cities i and k corresponds a line of the matrix and at city j corresponds a column of the matrix; this block is used by the algorithm during the second iteration, when the partial solution of each ant comprises only two cities, the initial departure city i of each ant and the first city k just visited after. In this block, there are no pheromone values in case index i = index j or index k = index j.

Finally, the third block is constituted by the pheromones $\tau_{ikm,j}$ associated to three cities, city k that the ant is just leaving, city i that was visited just before city k and city j that the ant is moving to; at the sequence of three cities i, k and m corresponds a line of the matrix and at city j corresponds a column of the matrix; this block is used by the algorithm for all iterations since the third, when the partial solution of each ant comprises at least three cities, the last visited city m by an ant, the city k just visited before city m and the city i just visited before city k. In this block, there are no pheromone values in case index i = index j or index k = index j or index m = index j.

It is clear that the size of the new matrix Tau will increase more rapidly than with the classical matrix τ. In the new matrix, the number of lines will be proportional to n^{γ}, where n is the total number of cities to be visited and γ is the memory depth, i.e. the maximum size of the sequences of cities.

PERSONNAL CONTRIBUTION
To implement the new pheromone matrix, the procedures for memory allocation and initialization of the matrix have been modified, taking into account the new dimension of the matrix.

5.1.2 Working in serialization

Another possible improvement should be to realize a serialization of the previous algorithms, in order of their increasing *memory depth*. The pheromone values corresponding to the best result found for a given value of memory depth *n-1* will initialize the pheromone matrix used for the next memory depth *n*. In this way, we hope to converge more quickly to a near optimal solution of the TSP.

PERSONNAL CONTRIBUTION

The basic version of the AS algorithm was implemented in series with the modified AS algorithm of memory depth 2, followed by the same algorithm with memory depth 3.

5.2 Construction of a solution in ACO algorithms

In this section, we will consider the different steps of the ACO algorithms, to construct a solution (Dorigo, M., Stützle T. (2004) Chapter 3). In fact tours are constructed by applying the following simple constructive procedure to each ant:

1. choose, according to some criterion, a start city at which the ant is positioned;
2. use pheromone and heuristic values to probabilistically construct a tour by iteratively adding cities that the ant has not visited yet, until all cities have been visited;
3. go back to the initial city;

After all ants have completed their tour, they may deposit pheromone on the tour they have followed. In some cases, before adding pheromone, the tours constructed by the ants may be improved by the application of a local search procedure. Such procedure will not be applied in the present work.

This high-level description applies to most of the published ACO algorithms for the TSP, described in chapter 4, section 4.2, with an exception for the Ant Colony System in subsection 4.2.5, where the pheromone evaporation is interleaved with tour construction.

When applied to the TSP and to any other static combinatorial optimization problem, most ACO algorithms employ a more specific algorithm scheme than the general one of the ACO metaheuristic given in figure 2.1, subsection 2.2.6. This new algorithm's scheme is shown in figure 5.3 hereafter.

```
procedure ACOMetaheuristicStatic
    Set parameters, initialize pheromone trails
    while(termination condition not met)do
        ConstructAntsSolutions
        ApplyLocalSearch              % optional
        UpdatePheromones
    end
end
```

Figure 5.3: The pseudo-code of the ACO Metaheuristic Static procedure

After initializing the parameters and the pheromone trails, these ACO algorithms iterate through a main loop, consisting in three steps: first all of the ants' tours are constructed; then an optional phase takes place in which the ants' tours are improved by the application of some local search algorithm; finally the pheromone trail are updated, involving pheromone evaporation and the update of the pheromone trails by the ants to reflect their search experience.

5.2.1 Implementing AS algorithm for the TSP

Data structures.

The first step to implement the AS algorithm for the TSP is to define the basic data structures. These must allow storing the data about the TSP instance and the pheromone trails, and representing artificial ants.

Figure 5.4 gives a general outline of the main data structures that are used for the implementation of the AS algorithm, which includes the data for the problem representation and the data for the representation of the ants.

```
% Representation of problem data
integer      dist[n][n]          % distance matrix
integer      nn_listdist[n][n]   % matrix with nearest neighbor lists of depth nn
real         pheromone[n][n]     % pheromone matrix
choice_info  dist[n][n]          % combined pheromone and heuristic information
```

```
% Representation of ants
structure single_ant
begin
   integer     tour_length       % the ant's tour length
   integer     tour[n+1]         % ant's memory storing (partial) tours
   integer     pheromone[n][n]   % visited cities
end
single_ant[m]  % structure of type single_ant
```

Figure 5.4: Main data structures for the implementation of the AS algorithm for the TSP.

Problem representation.

A symmetric TSP instance is given as the coordinates of a number of n points. All the intercity distances are precomputed and stored in a symmetric *distance matrix* with n^2 entries. Although for symmetric TSPs we only need to store $n(n\text{-}1)/2$ distinct distances, it is more efficient to use an n^2 matrix to avoid performing additional operations to check whether, when accessing a generic distance $d(i,j)$, entry (i,j) or entry (j,i) of the matrix should be used. For a reason of code efficiency, the distances are stored as integers.

Nearest-Neighbor Lists. In addition to the distance matrix, we also store for each city a list of its nearest neighbors. Let d_i be the list of the distances from a city i to all cities j, with $j = 1, \ldots n$ and $i \neq j$ (we assume here that the value d_{ii} is assigned a value larger than d_{max}, where d_{max} is the maximum distance between any two cities).

The nearest-neighbor list of a city i is obtained by sorting the list d_i according to nondecreasing distances, obtaining a sorted list d'_i. The position r of a city j in city i's nearest-neighbor list $nn\_list[i]$ is the index of the distance d_{ij} in the sorted list d'_i, that is, $nn\_list[i][r]$ gives the identifier of the r-th nearest city to city i (i.e., $nn\_list[i][r] = j$).

Nearest-neighbor lists for all cities can be constructed in $\mathcal{O}(n^2 logn)$. An enormous speedup is obtained for the solution construction in ACO algorithms, if the nearest-neighbor list is cut off after a constant number nn of nearest neighbors, where typically nn is a small value ranging between 15 and 40. An ant located in city i firstly chooses the next city among the nn nearest neighbors of i; in case the ant has already visited all the nearest neighbors, then it makes its selection among the remaining cities. This reduces the complexity of making the choice of the next city to $\mathcal{O}(1)$, unless the ant has already visited all the cities in $nn\_list[i]$. A disadvantage to the use of truncated nearest-neighbor lists is that it can make impossible to find the optimal solution.

Pheromone Trails. It is also necessary to store for each connection (i,j) a number τ_{ij} corresponding to the pheromone trail associated with that connection. As in the case for the distance matrix, it is more convenient to use some redundancy and to store the pheromones in a symmetric n^2 matrix.

PERSONNAL CONTRIBUTION
In this work, the pheromone matrix has been redefined. Each pheromone value in the new matrix is associated on the one hand to a sequence of the immediate last visited cities, and on the other hand to the next city to be visited.

Combining Pheromone and Heuristic Information. When constructing a tour, an ant located on city i chooses the next city j with a probability which is proportional to the value of $[\tau_{ij}^{\alpha}][\eta_{ij}^{\beta}]$. Because these very same values need to be computed by each of m ants, computation times may be significantly reduced by using an additional matrix *choice_info*, where each entry $choice\_info[i][j]$ stores the value $[\tau_{ij}^{\alpha}][\eta_{ij}^{\beta}]$. As in the case of the pheromone and the distance matrices, a matrix is more convenient to store those values.

PERSONNAL CONTRIBUTION
In this work, for implementation facilities, the *choice_info* matrix is not used. Each ant always consults the new pheromone matrix and calculate the value $[\tau_{ij}^{\alpha}][\eta_{ij}^{\beta}]$.

Representing ants.

An ant is a single computational agent which constructs a solution to the problem at hand, and may deposit an amount of pheromone $\Delta\tau$ on the arcs it has traversed. To do so, an ant must be able to:

1. store the partial solution it has constructed so far; this can be satisfied by storing, for each ant, the partial tour in an array of length $n + 1$, where at position $n + 1$ the first city is repeated. This choice makes easier some of the other procedures like the computation of the tour length.

2. determine the feasible neighborhood at each city; the knowledge of the partial tour at each step is sufficient to allow the ant to determine, by scanning of the partial tour, whether a city j is in its feasible neighborhood or not; this involves an operation of worst-case complexity $\mathcal{O}(n)$ for each city i, resulting in a high computational overhead. The simplest way around this problem is to associate with each ant an additional array *visited* whose values are set to $visited[j] = 1$ if city i has already been visited by the ant, and $visited[j] = 0$ otherwise. This array is updated by the ant while it builds a solution.

3. compute and store in the tour_length variable the objective function value of the solution it generates; the computation is done by summing the length of the n arcs in the ant's tour.

An ant is then represented by a structure that comprises one variable *tour_length* to store the ant's objective function value, one $(n+1)$-dimensional array *tour* to store the ant's tour, and one n-dimensional array *visited* to store the visited cities.

The algorithm.

The main tasks to be considered in an ACO algorithm are the solution construction, the management of the pheromone trails, and the additional techniques such as local search. In addition, the data structures and parameters need to be initialized and some statistics about the run need to be maintained. Figure 5.3 gives a high-level view of the algorithm.

Data Initialization In the data initialization, (1) the instance has to be read; (2) the distance matrix has to be computed; (3) the nearest-neighbor lists for all cities have to be computed; (4) the pheromone matrix and the *choice_info* matrix have to be initialized; (5) the ants have to be initialized; (6) the algorithm's parameters must be initialized; and (7) some variables that keep track of statistical information, such as the used CPU time, the number of iterations, or the best solution found so far, have to be initialized. Figure 5.5 shows a possible organization of these tasks into several data initialization procedures.

```
procedure InitializeData
   ReadInstance
   ComputeDistances
   ComputeNearestNeighborLists
   ComputeChoiceInformation
   InitializeAnts
   InitializeParameters
   InitializeStatistics
end-procedure
```

Figure 5.5: Procedure to initialize the algorithm.

Solution Construction The tour construction is managed by the procedure ConstructSolutions, shown in figure 5.6

```
procedure ConstructSolutions
   for k = 1 to m do
      for i = 1 to n do
         ant[k].visited[i] ← false
      end-for
   end-for
   step ← 1
   for k = 1 to m do
      r ← random{1,...,n}
      ant[k].tour[step] ← r
      ant[k].visited[r] ← true
   end-for
   while (step < n) do
      step ← step + 1
      for k = 1 to m do
```

```
        NeighborListAsDecisionRule(k,step)
      end-for
    end-while
    for k = 1 to m do
      ant[k].tour[n + 1] ← ant[k].tour[1]
      ant[k].tour_length ← ComputeTourLength(k)
    end-for
end-procedure
```

Figure 5.6: Pseudo-code for the solution construction procedure for AS.

The solution construction requires the following phases:

1. The ant's memory must be initialized, by marking all the cities as unvisited, that is, by setting all the entries of the array *ants.visited* to *false* for all the ants;

2. Each ant has to be assigned a random initial city; the function random returns a number chosen according to a uniform distribution over the set $\{1, \ldots, n\}$;

3. Each ant constructs a complete tour; at each construction step the ants apply the AS action choice rule. The procedure NeighborListASDecisionRule implements the action choice rule and takes as parameters the ant identifier and the current construction step index. This procedure exploits candidate lists and is discussed below;

4. Finally, the ants move back to the initial city and the tour length of each ant's tour is computed.

The solution construction of all ants is synchronized in such a way that the ants build solutions in parallel.

Action Choice Rule with Candidate Lists Figure 5.7 shows the pseudo-code for the action choice rule with candidate list. In the action choice rule, an ant located at city i probabilistically chooses to move to an unvisited city j based on the pheromone trails τ_{ij}^{α} and the heuristic information η_{ij}^{β} [see equation (3.1)].

```
procedure NeighborListASDecisionRule(k,i)
  input    k    % ant identifier
  input    i    % counter for construction step
  c ← ant[k].tour[i - 1]
  sum_probabilities ← 0.0
  for j = 1 to nn do
    if ant[k].visited[nn_list[c][j]] then
      selection_probability[j] ← 0.0
    else
      selection_probability[j] ← choice_info[c][nn_list[c][j]]
      sum_probabilities ← sum_probabilities + selection_probability[j]
    end-if
  end-for
  if (sum_probabilities = 0.0) then
    ChooseBestNext(k,i)
  else
    r ← random[0,sum_probabilities]
    j ← 1
    p ← selection_probability[j]
    while (p < r) do
      j ← j + 1
      p ← p + selection_probability[j]
    end-while
    ant[k].tour[i] ← nn_list[c][j]
    ant[k].visited[nn_list[c][j]] ← true
  end-if
end-procedure
```

Figure 5.7: AS with candidate lists: pseudo-code for the action choice rule.

The procedure works as follows:

1. the current city c of ant k is determined;

2. when choosing the next city, one needs to identify the appropriate city index from the candidate list of the current city c; this is the object of the first for loop;

3. to deal with the situation in which all the cities in the candidate list have already been visited by ant k (characterized by the variable *sum_probabilities* still at 0.0), the procedure ChooseBestNext is used to identify the city with maximum value $[\tau_{ij}^{\alpha}][\eta_{ij}^{\beta}]$ as the next to move to; this process is the object of the test if-then-else.

Figure 5.8 shows the pseudo-code for the procedure ChooseBestNext.

```
procedure ChooseBestNext(k,i)
  input    k    % ant identifier
  input    i    % counter for construction step
  v ← 0.0
  c ← ant[k].tour[i - 1]
  for j = 1 to n do
    if not ant[k].visited[j] then
      if choice_info[c][j] > v then
        nc ← j    % city with maximal τ^α η^β
        v ← choice_info[c][j]
      end-if
    end-if
  end-for
    ant[k].tour[i] ← nc
    ant[k].visited[nc] ← true
end-procedure
```

Figure 5.8: AS: pseudo-code for the procedure ChooseBestNext.

By using candidate lists the computation time necessary for the ants to construct solutions can be significantly reduced, because the ants choose from a smaller set of cities. In fact the computation time is reduced only if the procedure ChooseBestNext does not need to be applied too often.

PERSONNAL CONTRIBUTION
In this work, according to the new pheromone matrix TAU, both procedures NeighborListASDecisionRule and ChooseBestNext have been adapted mainly in the programming of the action choice rule, i.e. in the calculation of the $\tau^{\alpha}\eta^{\beta}$.

The Pheromone Update

The last step in an iteration of AS is the pheromone update, implemented by the procedure ASPheromoneUpdate (see figure 5.9), which comprises two pheromone update procedures: the pheromone evaporation and the pheromone deposit.

```
procedure ASPheromoneUpdate
   Evaporate
   for k = 1 to m do
       DepositPheromones(k)
   end-for
   ComputeChoiceInformation
end-procedure
```

Figure 5.9: AS: Management of the pheromone updates.

The procedure Evaporate decreases the value of the pheromone trails on all the arcs (i,j) by a constant factor ρ. The procedure DepositPheromone adds pheromone to the arcs belonging to the tours constructed by the ants. Finally the procedure ComputeChoiceInformation computes the matrix *choice_info* to be used in the next algorithm iteration.

PERSONNAL CONTRIBUTION
In this work, according to the new pheromone matrix TAU, both procedures Evaporate and DepositPheromone have been modified. The modification doesn't concern the evaporation or updating process. It concerns the way the number of the line in the matrix Tau is computed, based on a sequence of indeces of cities.

5.3 Modifications of the existing AS algorithm

As explained in the previous subsections, the procedures used for the management of the pheromone matrix or for the calculation of a solution based on the information of that matrix have been adapted. The experimental setup and the results obtained are the object of the next chapter.

PERSONNAL CONTRIBUTION

All procedures of the basic AS algorithm concerned by the new pheromone matrix have been adapted. Those procedures concern mainly the memory allocation, the initialization of the matrix, the construction of a solution, mainly through the procedure implementing the action choice rule, the evaporation and the updating of the pheromones.

Chapter 6

Experimental Results

In this chapter, we present the results obtained with the new algorithms applied on tests files of 50 cities. We discuss those results and propose some ways to improve the new algorithms.

6.1 Available software package

The ACO family algorithms for the TSP is available as a software package freely available subject to the GNU General Public Licence. This software package called ACOTSP was developed, in his Version 1.0, by Thomas Stützle in connection with the book "Ant Colony Optimization" [Dorigo, M., Stützle T. (2004)] and is available from http://www.aco-metaheuristic.org/aco-code. The software was developed in ANSI C under Linux, using the GNU 2.95.3 gcc compiler. The software is distributed as a gzipped tar file.

This software package provides an implementation of various Ant Colony Optimization (ACO) algorithms for the symmetric Traveling Salesman Problem (TSP). The ACO algorithms implemented are Ant System (AS), MAX-MIN Ant System, Rank-based version of Ant System, Best-Worst Ant System, and Ant Colony System. It was developed to have one common code for the various known ACO algorithms that were at some point applied to the TSP in the literature.

The contents of the package is the following:

1. The main control routines, main: acotsp.c;
2. The procedures to implement the ants behaviour: ants.c and ants.h;
3. The input, output and statistics routines: InOut.c and InOut.h;
4. The procedures specific to the TSP: TSP.c and TSP.h;
5. The local search procedures: ls.c and ls.h;
6. The additional useful and helping procedures: utilities.c and utilities.h;
7. The command line parser: parse.c and parse.h;
8. The time measurement: timer.c and timer.h;

Some problem instances from TSPLIB are also available. For this work, we produced our own test files. The coordinates of the cities were randomly generated using a random generator as described in "Numerical Recipes in C". They were stored in set of 100 test files.

6.2 Specific parameters and command line

Given the large number of ACO algorithms available in the package, also the number of command line options is relatively large. We give hereafter those that were useful for our work, i.e. those that were used in the execution of our new versions of the AS algorithms. They are given in their short and long options.

```
-r, –tries          # number of independent trials
-s, –tours          # number of steps in each trial
-t, –time           # maximum time for each trial
-i, –tsplibfile     # inputfile (TSPLIB format necessary)
-a, –alpha          # alpha (influence of pheromone trails)
-b, –beta           # beta (influence of heuristic information)
-e, –rho            # rho (pheromone trail evaporation)
-u, –as             # apply basic Ant System if selected
```

PERSONNAL CONTRIBUTION

For our work, a new parameter was added to introduce the concept of memory depth. The parsing procedures were also modified.

All those options take some default values; the parameter "-i, –tsplibfile" is the only mandatory option, because the program may not work without an input file.

For each test file, it is possible to execute several times the same algorithm; at each trial, the pheromone matrix is reinitialized but not the nearest neighbor list. In this work we decide to execute only one run for each test file.

6.3 Experimental settings

The original AS algorithm has been adapted for the cases where the memory depth is equal to 2 and 3. The table 6.1 hereafter presents the evolution of the number of lines and the number of pheromones values to be stored, of the matrix Tau, in function of the total number of cities and the memory depth.

Nbr. cities	**Mem. depth**	**Nbr. lines**(10^6)	**Nbr. val.**(10^6)
25	2	0,0006	0,016
25	3	0,014	0,361
25	4	0,318	7,951
50	2	0,025	0,125
50	3	0,120	6,005
50	4	5,6	282,3
100	2	0,01	500,0
100	3	0,98	2450,5
100	4	95,1	4754,5

Table 6.1: Evolution of the number of pheromone values to be stored, in function of the number of cities and the memory depth.

Very simply, we decided to store the new matrix Tau fully in memory. Following the previous table, as the number of pheromone values grows rapidly with the total number of cities and the memory depth, we decided to adapt the AS algorithm only for the case where the memory depth is equal to 2 or 3. Even in this situation, the first tests evolved slowly when the size of the test files was bigger or equal to 50 cities, especially for a memory depth equal to 3. So we came to the conclusion to fix the number of cities in each test file to 25, and this for both algorithm with memory depth equal to 2 and 3.

The structure of each test file comprises mainly the name of the file, the total number of the cities, the metric used and the list of the coordinates of the nodes. Those coordinates are generated using a procedure producing random numbers uniformly distributed in [0,1] (see Press, W.H., Teukolsky S.A., Numerical recipes in C). The final integer coordinates range from 0 to 100. The list of the coordinates finish by an EOF mark. As explained before, we fixed the total number of cities to 25 and the metric used is the euclidean one.

At the beginning of the execution of an algorithm, the name of the test file is given as a parameter. The parsing of the command line includes the analysis of the test file. It produces the calculation of the distance matrix, which characterizes this instance of the problem.

Our experimental process is driven by a main script program called "Param" written in PERL (see annexe). This program first prepares the directories and the tables to store the results produced during the execution of an algorithm; it also prepares the tables which register the parameter used during each run. It makes then a call to another program called "random.c", which generates randomly 100 test files, using the random procedure previously presented.

The program "Param" is now ready to process each execution script we want to submit to our different algorithms. Each script may redefine every parameter to be used by the algorithms. When a parameter is not redefined in the script, the algorithm uses the default value for this parameter. A command line is build with all the parameters and is given to a procedure called "Exec". To write our different scripts, we focused on the following parameters: the exponents α and β used in the probabilistic decision rule (see equation 4.1) and the evaporation parameter ρ. Although it is possible to execute an algorithm more than one time with the same test file, we decided to limit this number of independent trial to 1. Finally the first tests we made showed us that the results obtained didn't evolve significantly after an execution time of 15 seconds. We fixed then the execution time to this value for all algorithm.

The "Exec" procedure (available in the annexe) is written in PERL and executes two embedded loops. The first loop covers the different algorithms by increasing order of memory depth. The first algorithm to be executed will be the classical Ant System; the second one will be our version with memory depth equal to 1 and the third one our version with memory depth equal to 3. For each algorithm, the second loop executes the command line received from the calling program "Param", and this for the 100 generated test files. The results are stored in dedicated directories. The "Exec program" also process those directories to produce two registering tables. Those tables register, for each algorithm, for each test file and for each script, in one table the values of the parameters, and in the other table the results corresponding to this set of parameters.

When all the scripts registered in the program "Param" have been processed, a last call to a procedure called "commands3.r" is made. This procedure is written in R and is available in the annexe. It process the table produced by the program "Exec", registering the results by algorithm, by test file, by script. For each algorithm, the program "commands3.r" calculate the mean of the results over the 100 test files.

The results generated by an algorithm consist in the length of the best-so-far tour build by the ants. Only an improvement in the obtained tour length is registered, with its corresponding CPU time and its corresponding number of iterations. An iteration is counted when all the ants of the colony have constructed one tour at the end of the procedure "ConstructSolutions" (see section 5.2.1). The presentation of the final results in the graphic can be made with the CPU time or the iteration number in abscissa. We chose to work by parity of iterations.

We give hereafter a resume of the experimental settings used for our tests:

1. Three algorithms are tested: the classic Ant System, its new version with a memory depth of 2 and its version with a memory depth of 3;
2. 100 test files are randomly generated;
3. This set of test files is the same used by each algorithm;
4. For each algorithm, the number of trial for a given test file is fixed to 1;
5. Each test file comprises the euclidean coordinates of 25 cities, randomly generated in a uniform distribution ranging from 0 to 100;

6. Our scripts modify the value of the parameters α, β and ρ;

7. The maximum execution time is fixed to 15 seconds.

6.4 Results of the experiments

6.4.1 Results by algorithm

Different tests have been made. We always have modified one parameter at a time, keeping the others unchanged.

Impact if we modify α. In a first step, we modified the value of α from 0.5 to 2, keeping $\beta(= 2)$ and $\rho(= 0.5)$ constant (see figure 6.1 to 6.4). We wanted to see how the new algorithms (AS2 and AS3) would react if we put more or minder emphasis to the new pheromones.

If $\alpha = 0.5$, we see that AS2 reaches a better result than the classical Ant System (AS1), but this last one is out of its optimal range of parameters, where α must be equal to 1. So with $\alpha = 0.5$, AS2 gives a better result than AS1, but this result is not better than this obtained by AS1 when it works in its normal range of parameters. As3 gives a bad quality result.

If $\alpha = 1$, we see that AS1 and AS2 reach a similar result. But AS2 doesn't provide a better result than AS1. The quality of the solution of AS3 stays bad.

If $\alpha = 1.5$, we see that AS2 reaches a good quality result, but needs for that a bigger amount of iterations. The quality of the solution for AS1 is still decreasing, because we are still far from the optimal range of its parameters. The quality of the solution of AS3 is still decreasing.

As a conclusion, we can say that AS2, for every studied values of α, gives similar results as AS1, when this algorithm is in its optimal conditions. But never gives AS2 a better result than AS1. AS3 seems to stay worst than AS1 and AS2 for every value of α. It comes quickly to stagnation and reaches a bad quality solution, in regards of the result of the two previous algorithms.

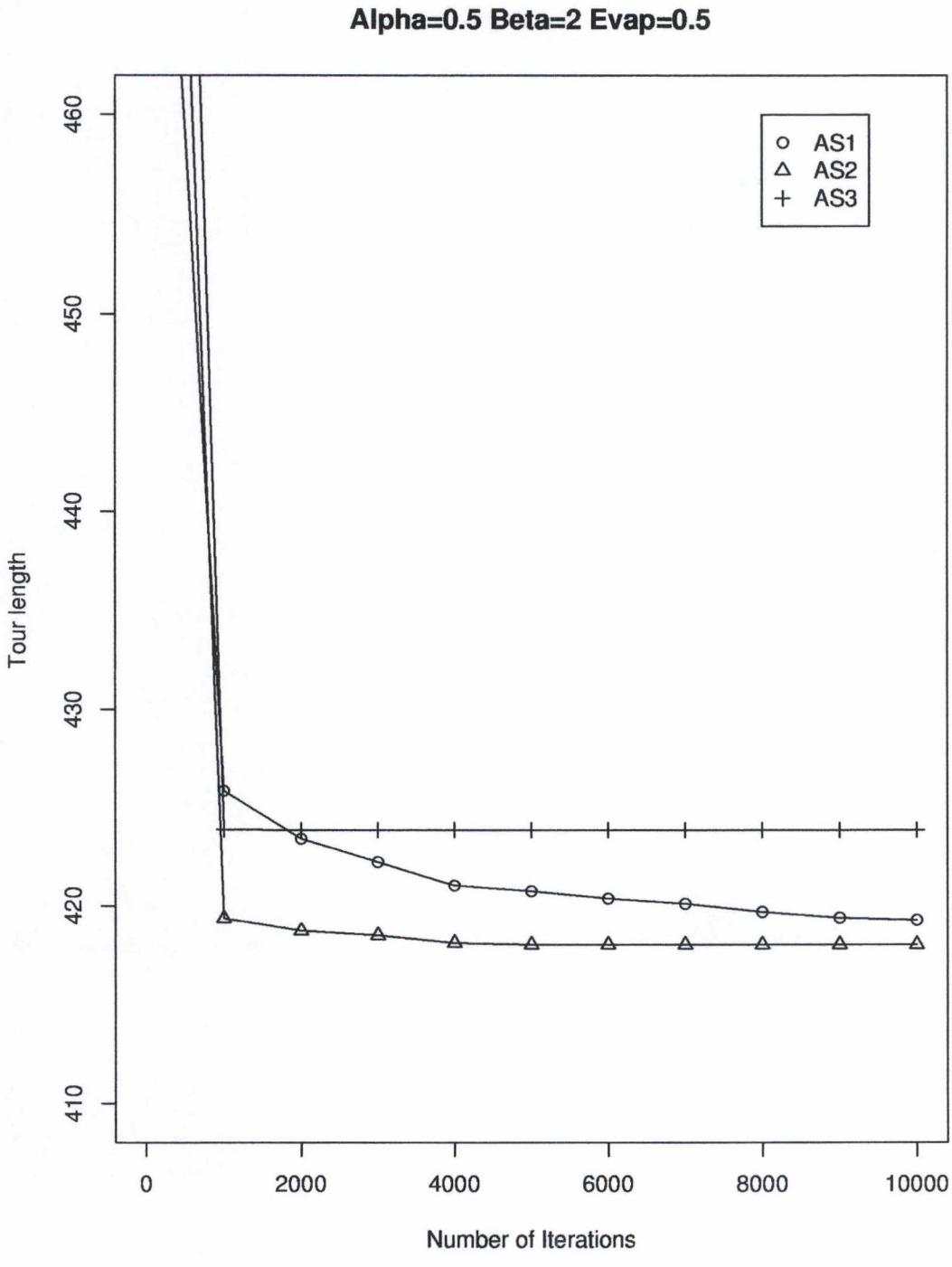

Figure 6.1: Best tour length versus number of iterations, for $\alpha = 0.5$, $\beta = 2$ and $\rho = 0.5$

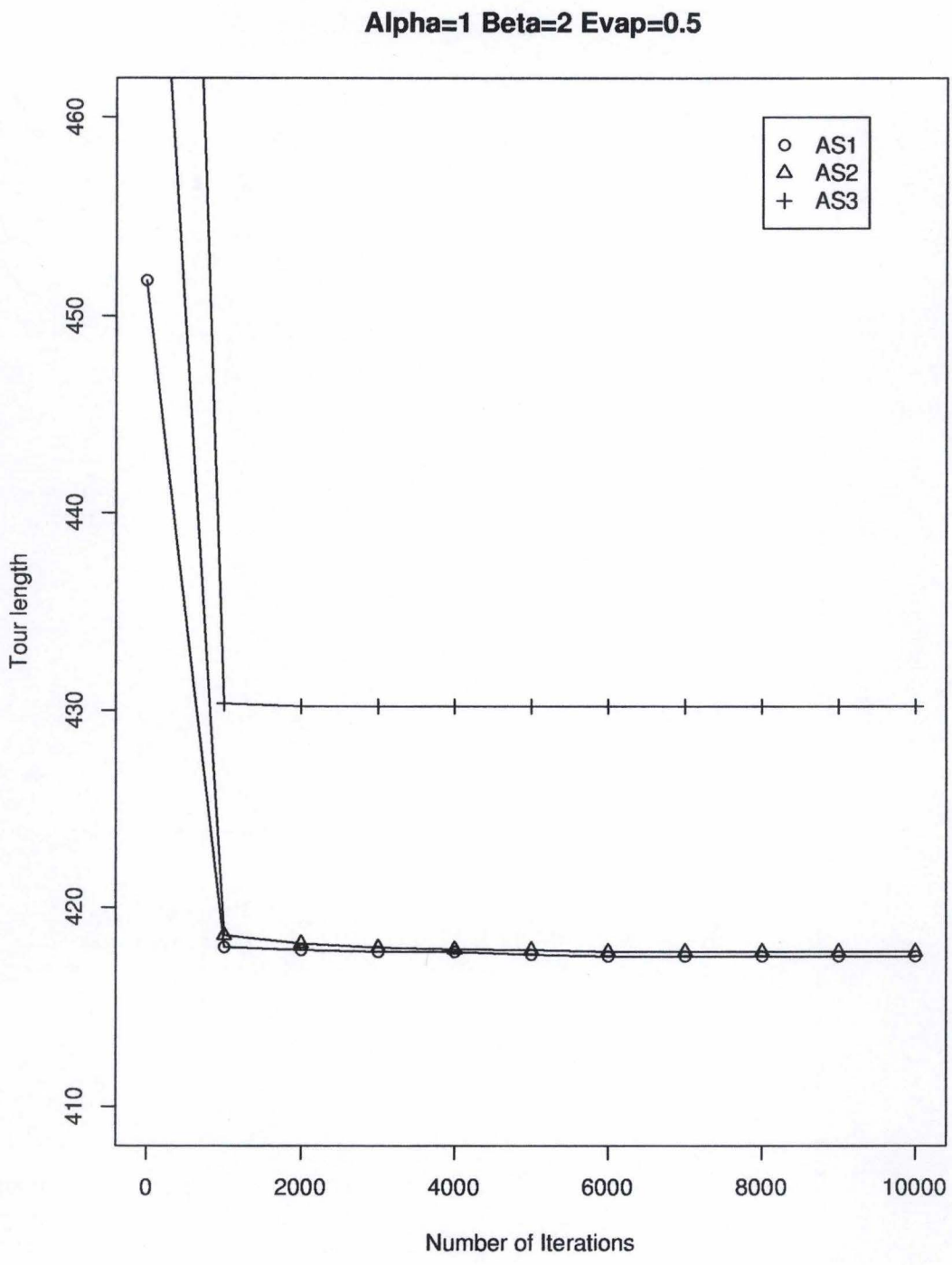

Figure 6.2: Best tour length versus number of iterations, for $\alpha = 1$, $\beta = 2$ and $\rho = 0.5$

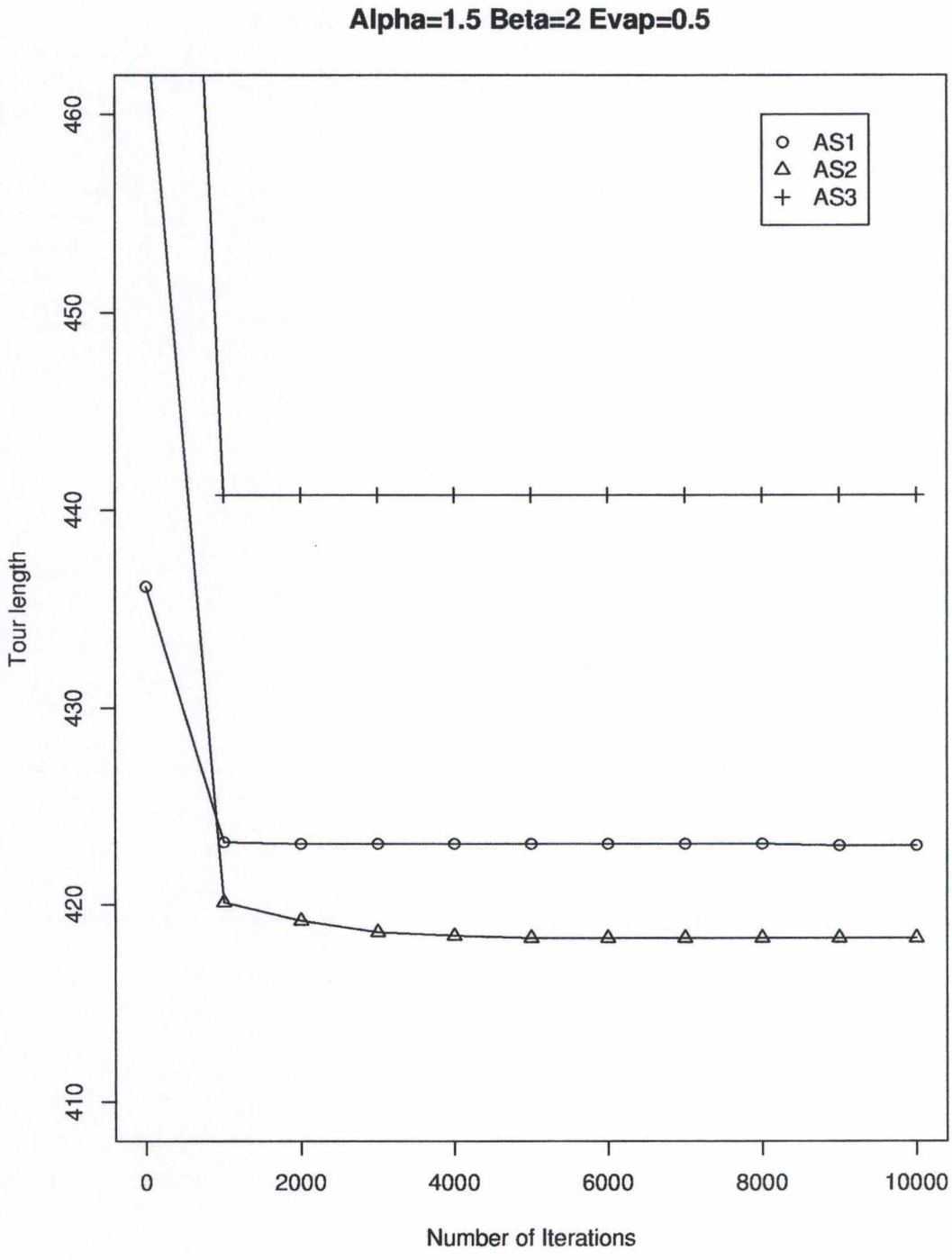

Figure 6.3: Best tour length versus number of iterations, for $\alpha = 1.5$, $\beta = 2$ and $\rho = 0.5$

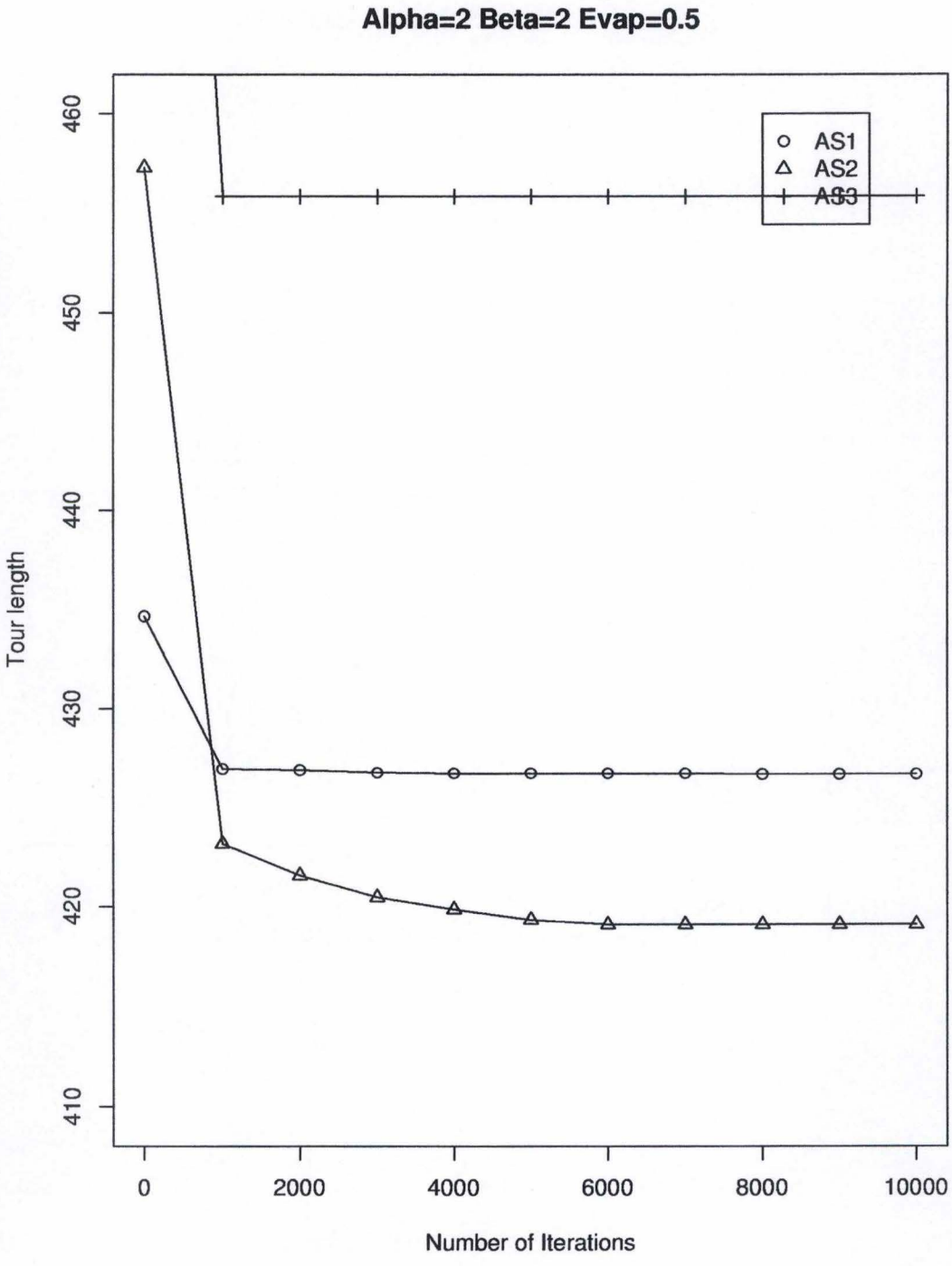

Figure 6.4: Best tour length versus number of iterations, for $\alpha = 2$, $\beta = 2$ and $\rho = 0{,}5$

Impact if we modify β. In a second time (see figure 6.5 to 6.8), we considered the situation where the values of the parameters are the following: $\alpha = 2$ and $\rho = 0.5$, the value of β being modified from 2 to 5, giving more emphasis to the heuristic. We wanted to see if it was possible to improve the results obtained with AS2, and possibly to reach a better solution than this obtained with AS1, when it works in its normal range of parameter values.

For every values of β we have tested (from 2 to 5), we observe that AS2 gives a better solution than AS1 and AS3. AS1 and AS3 come quickly to stagnation (after maximum 2000 iterations); AS2 reaches later its best result (after 6000 iterations). We also observed that the higher the value of β we have, the more the three algorithms converge. But never AS2 gives a better result than AS1 in its optimal range of parameters. As a conclusion, the fact to increase the value of β doesn't improve the quality of results provided by AS2; AS3 stay bad for every values but converges to the results of AS1 and AS2 as β increases.

Finally, we didn't observe that evaporation had a significant impact on the quality of the final results (No graphics included).

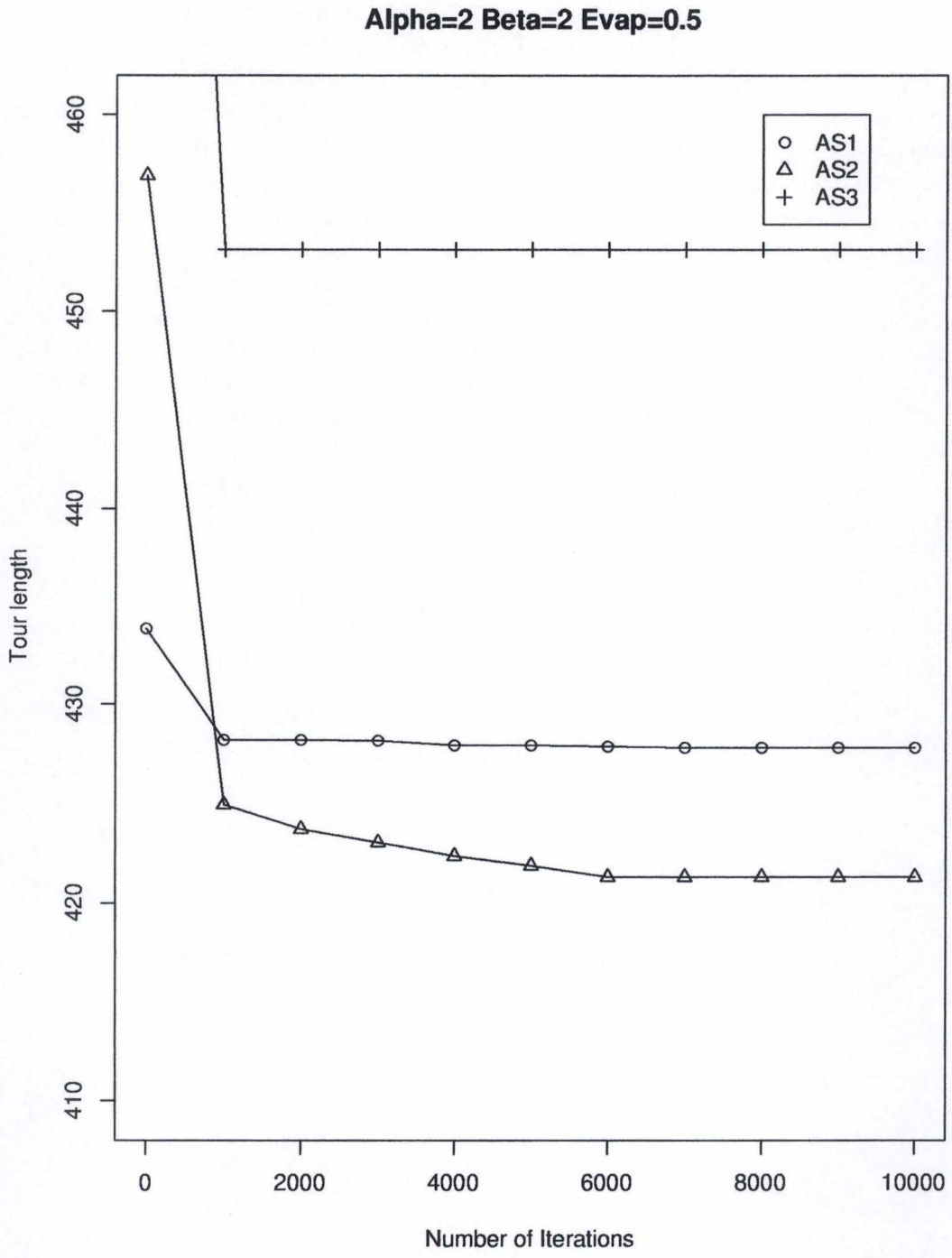

Figure 6.5: Best tour length versus number of iterations, for $\alpha = 2$, $\beta = 2$ and $\rho = 0{,}5$

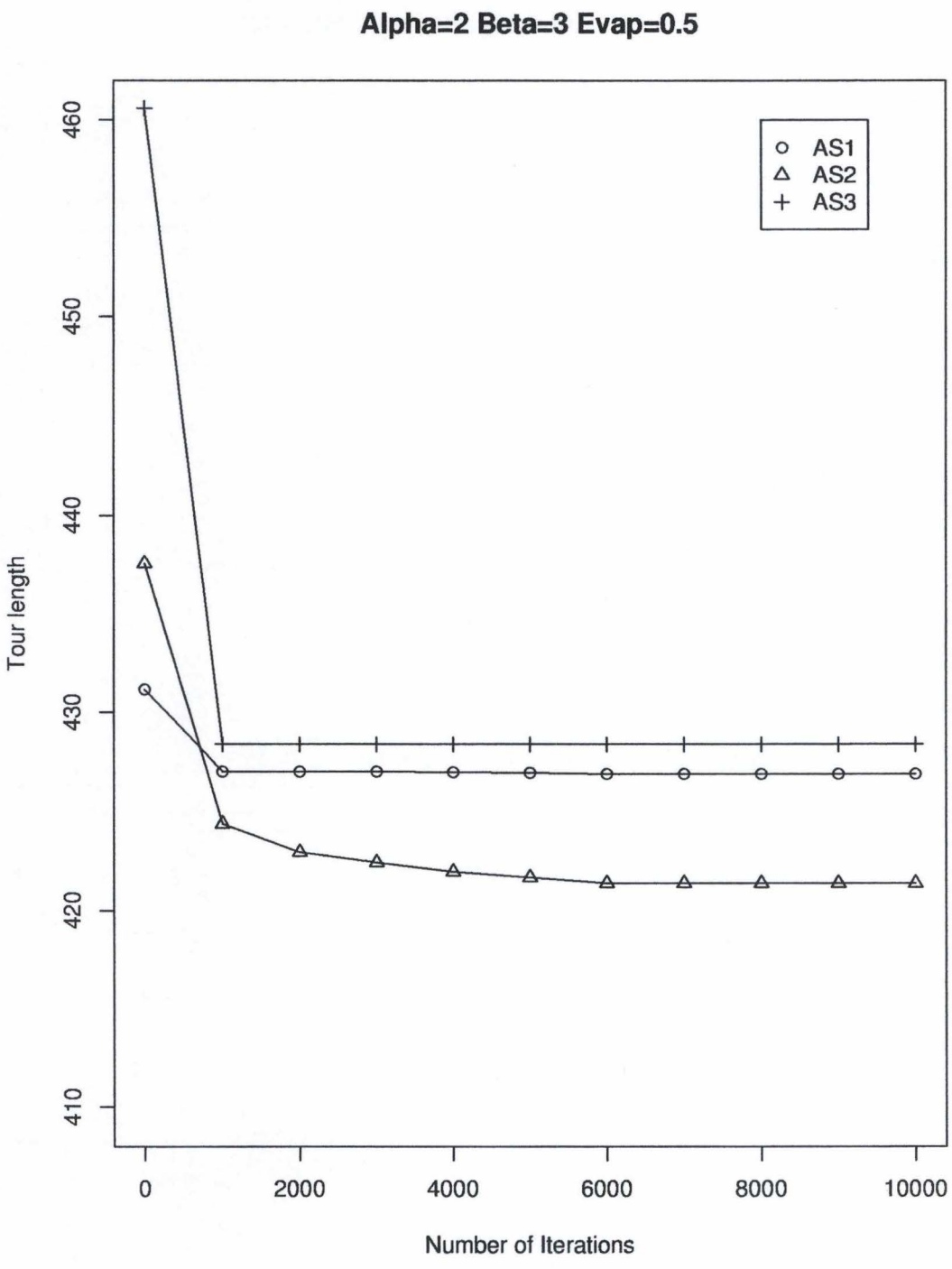

Figure 6.6: Best tour length versus number of iterations, for $\alpha = 2$, $\beta = 3$ and $\rho = 0{,}5$

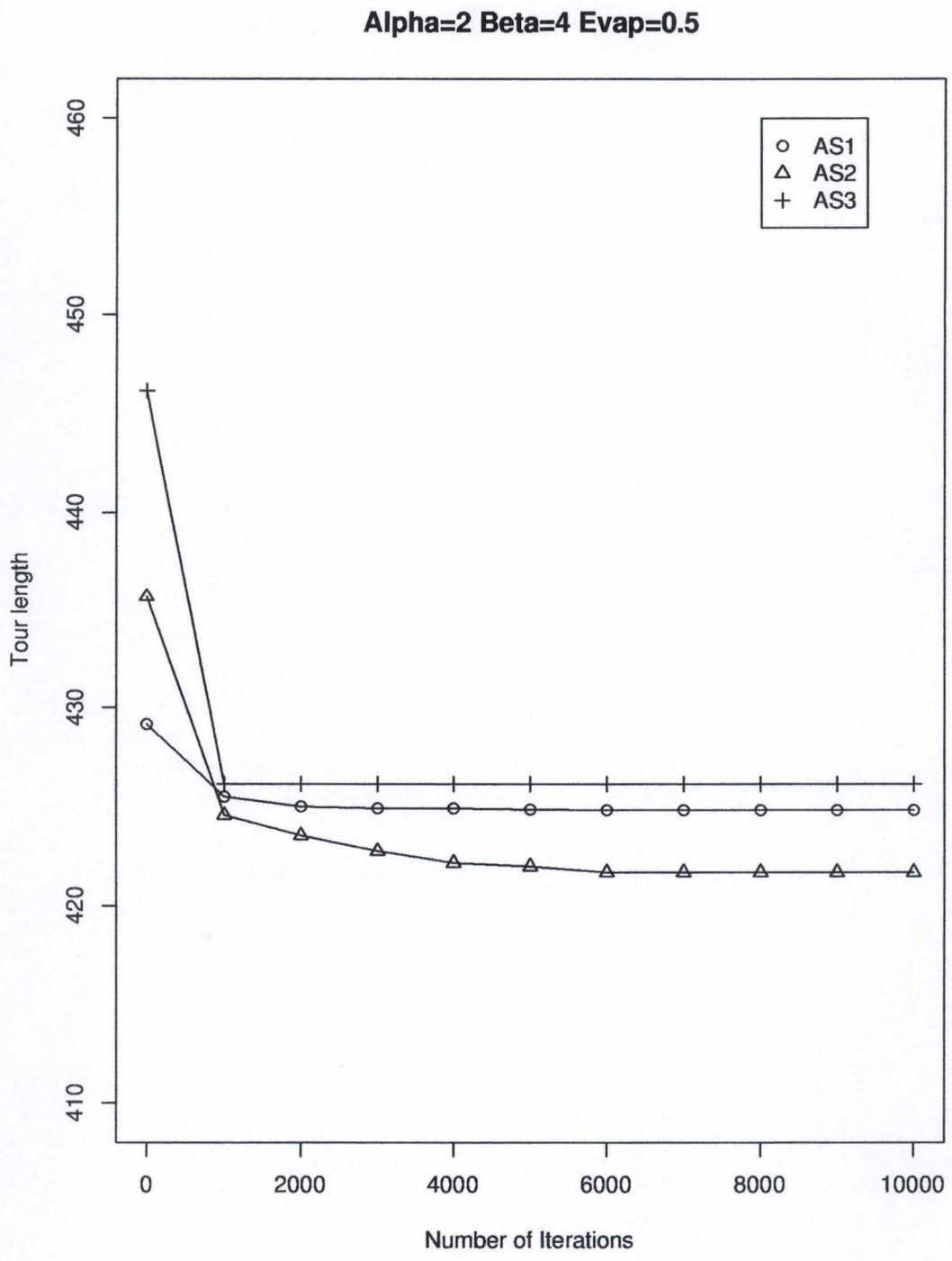

Figure 6.7: Best tour length versus number of iterations, for $\alpha = 2$, $\beta = 4$ and $\rho = 0{,}5$

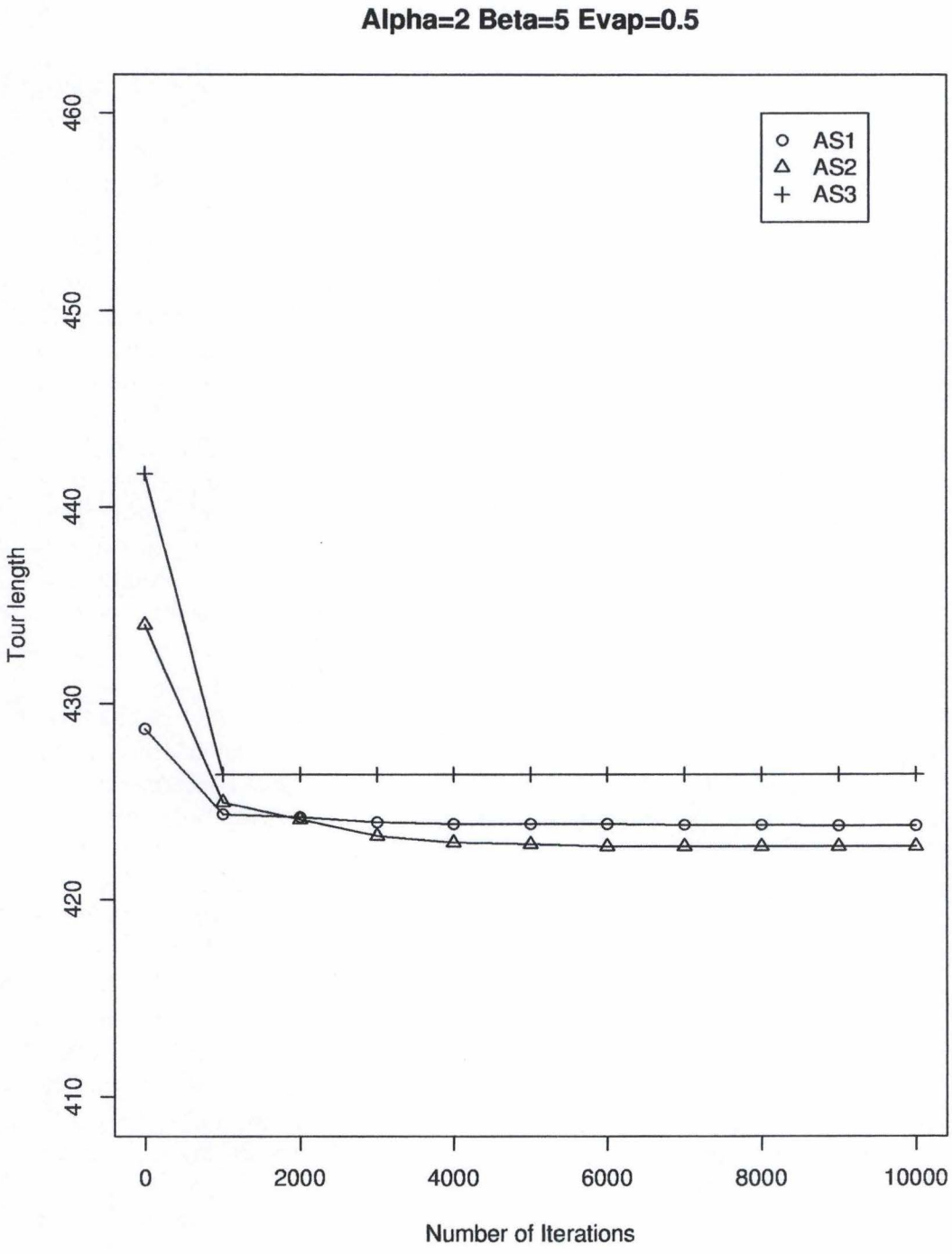

Figure 6.8: Best tour length versus number of iterations, for $\alpha = 2$, $\beta = 5$ and $\rho = 0{,}5$

6.5 Results with the three algorithms in cascade

The three algorithms, the basic version corresponding to a memory depth equals to 1, and the two new versions (with memory depth equal to 2 and 3) were programmed in a serial way, in increasing order of the memory depth. At the end of the execution of one algorithm, the last pheromone matrix corresponding to the best final solution serves to initialize the corresponding bloc of the pheromone matrix of the next algorithm, for the following value of memory depth. So the pheromone matrix obtained at the end of Ant System will initialize the corresponding part of the Tau matrix for the algorithm with a memory depth equal to 2. The other part of the same matrix, dedicated to the sequences of two cities, will be initialized as usual, with a tour length build by one ant. Again for the algorithm with a memory depth equal to 3, the part of the Tau matrix dedicated to sequences with one city will be initialized with the pheromone values of Ant System, corresponding to the best tour found with this algorithm. The part of the Tau matrix dedicated to sequences constituted of two cities will be initialized with the corresponding pheromones of the Tau matrix for the algorithm with memory depth of 2. Finally, the pheromones dedicated to sequences of three cities will be initialized with a tour length build by one ant.

The experimental setup is slightly modified in regard of this one used for the three algorithms. The number of cities for each test file (25) and the total number of generated test files (100) stay the same. The main difference is that we have now a fourth algorithm, called "ASChained" constructed with the three previous one. Each script written in "Param" will also be executed for this new algorithm. The execution for the three first algorithm (AS1, AS2 and AS3) follows the same plan as explained in section 6.3. The new algorithm follows also the same plan; in fact the adaptation mainly concerns the programs "Exec" and "commands3.r", where some loops have to include this new version.

By this way, we hope to observe a improvement in the quality of the final solution. Unfortunately, this is not the case and we didn't observe such effect, as we can see in figure 6.3. In fact, this way of initializing the pheromone matrix of each algorithm is not the more adapted. It would be correct to implement the serialization so that the transition between each algorithm happens at a certain iteration and would be visible on the graph. On our graph ASChained is only the result of AS3 initialized with the pheromone matrix of AS1 and AS2 at the end of their run.

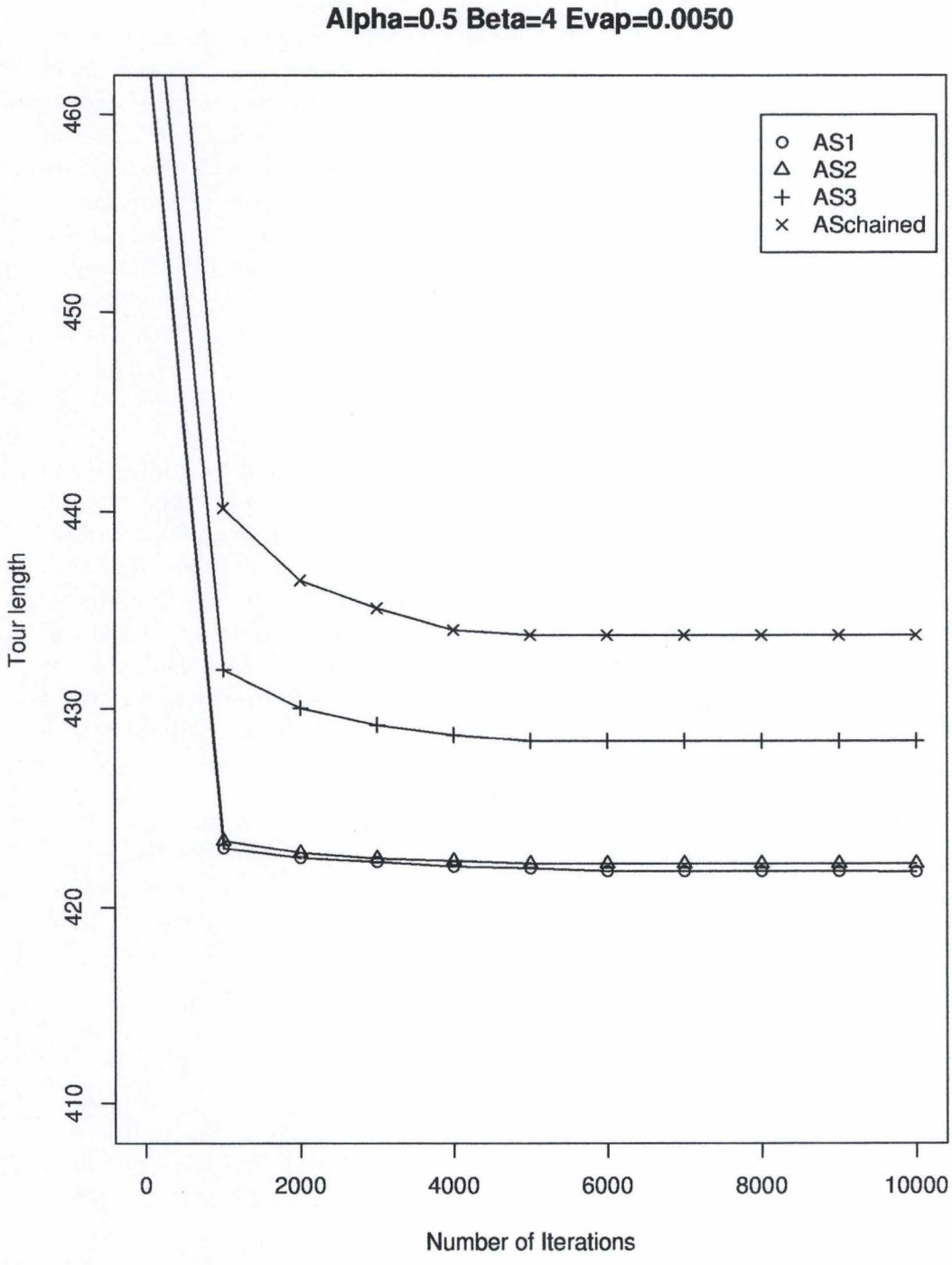

Figure 6.9: Serialization of the algorithms AS1, AS2 and AS3, giving AS-Chained

6.6 Conclusion of the experiments

As a conclusion for the set of experiments we have realized, in the experimental conditions we have used, it is not possible to observe any improvement in the final results obtained with AS2 and AS3, related to the new pheromones values based on the couple "sequence of last visited cities-cities". The second algorithm AS2 provides some better results than AS1, always when this last one is out of its best running conditions. The Third algorithm AS3 provides always bad results, in every experimental situation. This could indicates a problem with the way we implement this algorithm. A possible solution is discussed in the next section.

6.7 Possible improvements

It is obvious that the algorithm with memory depth equal to 3 provides in general bad results. A possibility to improve it is to replace the way we calculate the number of the line of the TAU matrix, making intensive calls to modulo operator, with the following formula (in the case of a memory depth equal to 3): $N^2(v_1 - 1) + N(v_2 - 1) + v_3$ where $v_1, v_2 and v_3$ are the number of the three last visited cities, in the order of their classification in the test file, and N is the total number of cities in the test file. This formula permits to calculate the number of all possible combination (with repetition of three cities. The size of the matrix will increase because of the combinations with repeated cities; the total number of lines will be of $N^3 + N^2 + N$ and the number of columns will be equal to N.

Chapter 7

Conclusion

The ACO metaheuristic was inspired by the foraging behavior of real ants and is characterized as being a distributed, stochastic search method based on the indirect communication of a colony of artificial ants, mediated by artificial pheromone trails.

The ACO metaheurisitc can be applied to any combinatorial optimization problem for which a solution construction can be conceived. An interesting case is the TSP problem, not only for its applications but also because it constitutes a test bed problem for new algorithmic ideas, in particular ACO.

In the ACO approach, one key element is the indirect form of memory of previous performance. This role is played by the pheromone matrix. The first basic algorithm was Ant System (AS) and has been introduced using the TSP as an example application. AS achieved encouraging initial results, but was found to be inferior to state-of-the-art algorithms for the TSP. His importance is more in the inspiration it provides for a number of extensions that significantly improved performance.

In this work, we tried in a very simple way to define the memory in another way than those envisaged by the formal definition of Ant Colony Optimization. In the version we have developed, we try to make a difference between states that are identical from the point of view of Ant Colony Optimization, associating the memory with pairs of "sequence of components - component". We didn't observe within the limits of our experimental conditions any improvement.

Bibliography

Bonabeau, E., Dorigo M., & Theraulaz G. (1999). "Chapter 1 Introduction", *Swarm Intelligence, From Natural to Artificial Systems* Santa Fe Institute, Studies in the Sciences of Complexity, Oxford University Press

Birattari, M., Di Caro G., Dorigo M. (2002). Toward the formal foundation of Ant Programming. *Ant Algorithms. Third International workshop, ANTS 2002.* LNCS 2463, pp. 188-201, Springer Verlag, Berlin, Germany.

Bullnheimer, B., Hartl, R., & Strauss, C. (1999c). A new rank-based version of the Ant System: A computational study. *Central European Journal for Operations Research and Economics,* 7(1), 25-38.

Costa, D. & Hertz, A. (1997). Ants and colour graphs. *Journal of the Operational Research Society,* (1997)48, 295-305

Dorigo, M. (1992). *Optimization, Learning and Natural Algorithms* [in Italian]. PhD Thesis, Dipartimento di Elettronica, Politecnico di Milano, Milan.

Dorigo, M., & Gambardella, L. M. (1997a). Ant colonies for the traveling salesman problem. *BioSystems, 43*(2), 73-81.

Dorigo, M., & Gambardella, L. M. (1997b). Ant Colony System: A cooperative learning approach to the traveling salesman problem. *IEEE Transaction on Evolutionary Computation, 6*(4), 317-365.

Dorigo, M., Maniezzo, V., & Colorni, A.(1991a). Positive feedback as a search strategy. Technical report 91-016, Dipartimento di Elettronica, Politecnico di Milano, Milan.

Dorigo, M., Maniezzo, V., & Colorni, A.(1996). Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics-Part B, 26*(1), 29-41.

Dorigo, M., Stützle T. (2004), "Chapter 1 From Real to Artificial Ants", "Chapter 2 The Ant Colony Optimization Metaheuristic", "Chapter 3 Ant Colony Optimization Algorithms for the Traveling Salesman Problem", *Ant Colony Optimization*, Massachusetts Institute of Technology, Cambridge Massachusetts, Chapter 1, 1-24, Chapter 2, 26-62, Chapter 3, 65-117

Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of $\mathcal{NP}$-Completeness.* San Francisco, Freeman.

Helsgaun, K. (2000). An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. Department of Computer Science, Roskilde University *European Journal of Operational Research, 2000,* 1-31.

Johnson, D.S., McGeoch, L.A. (1997), "The Traveling Salesman Problem: A Case Study in Local Optimization", in *Local Search in Combinatorial Optimization* John Wiley and Sons, London, 1997, pp 215-310.

Laburthe, F. (1998). "Chapitre 4 Le Voyageur de commerce (TSP)", de *Contraintes et Algorithmes en Optimisation Combinatoire.* Thèse Doctorale en Informatique, Université Paris VII - Denis Diderot, U.F.R. d'Informatique (04/09/1998), pages 1 à 16

Nilsson, Ch. Linkping University. Heuristics for the Traveling Salesman Problem

Press, W. H., Teukolsky, S. A., Vetterling, W. T., *Numerical Recipes in C: The art of Scientific Computing*, Second Edition, Cambridge University Press

Stützle, T. (1999). *Local Search Algorithms for Combinatorial Problems: Analysis, Improvements and New Applications*, vol. 220 of *DISKI.* Sankt Augustin, Germany, Infix.

Stützle, T., & Hoos, H. H. (1997). The MAX-MIN Ant System and local search for the traveling salesman problem. In T. Bäck, Z. Michalewicz, & X., Yao (Eds.), *Proceedings of the 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)* (pp. 309-314). Piscataway, NJ, IEEE Press.

Stützle, T., & Hoos, H. H. (2000). $\mathcal{MAX} - \mathcal{MIN}$ Ant System. *Future Generation Computer Systems, 16*(8), 889-914.

Annexes

The reader will also find hereafter, first the program " Param ", written in Perl, fixing the different parameters for the execution of the ants' algorithms. It permits to program the different scripts of execution. The second program " Exec ", written in Perl and called by the " Param " program, constructs the tables with the results provided by the execution of the ants' algorithms. Finally the third program, " commands3.r ", written with the R software and called by the " Param " program, uses the data stored in the tables generated by the " commands3.r " program to produce final graphics (CPU Time or Iteration versus Tour Length).

Program Param : execution scripts

```
#!/usr/bin/perl

use Exec;

system 'rm -f /home/ddarquennes/ACOTSPV10sp1/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp2/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp3/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp4/resultats/*';
system 'rm -f /home/ddarquennes/resultats/*';

system 'rm -f /home/ddarquennes/ACOTSPV10sp1/F*.tsp';
system 'rm -f /home/ddarquennes/ACOTSPV10sp2/F*.tsp';
system 'rm -f /home/ddarquennes/ACOTSPV10sp3/F*.tsp';
system 'rm -f /home/ddarquennes/ACOTSPV10sp4/F*.tsp';
system 'rm -f /home/ddarquennes/F*.tsp';

system 'rm -f /home/ddarquennes/instance0603.dat';
system 'rm -f /home/ddarquennes/values0603.dat';
system 'rm -f /home/ddarquennes/datas0603.dat';
system 'rm -f /home/ddarquennes/resfin0603.dat';

system "gcc -o random random.c";

system "nice -19 ./random";

system 'cp /home/ddarquennes/F* /home/ddarquennes/ACOTSPV10sp1';
system 'cp /home/ddarquennes/F* /home/ddarquennes/ACOTSPV10sp2';
system 'cp /home/ddarquennes/F* /home/ddarquennes/ACOTSPV10sp3';
system 'cp /home/ddarquennes/F* /home/ddarquennes/ACOTSPV10sp4';

open(RESULT, ">>values0603.dat");
open(INST, ">>instance0603.dat");

$param = 1;

$titre[1] = "Sp ";
$titre[2] = " I_nam ";
$titre[3] = " Param ";
$titre[4] = "  Nc ";
$titre[5] = " MTr ";
$titre[6] = " MTo ";
$titre[7] = " Opt";
$titre[8] = " Maxim_Time ";
$titre[9] = "NAn ";
```

```
$titre[10] = " NNe ";
$titre[11] = "  Alpha  ";
$titre[12] = "  Beta  ";
$titre[13] = "   Rho  ";
$titre[14] = "   Q_0  ";
$titre[15] = "  Br_up  ";
$titre[16] = " Ls ";
$titre[17] = " Rr ";
$titre[18] = " Elit";
$titre[19] = "  Nn ";
$titre[20] = " Dlb";
$titre[21] = " As ";
$titre[22] = " Ea ";
$titre[23] = " Ra ";
$titre[24] = " Mm ";
$titre[25] = " Bw ";
$titre[26] = " Ac ";
for($L = 1 ; $L < 27 ; $L++){
  print INST $titre[$L];
}
print INST "\n";

$res[1] = "Sp  ";
$res[2] = "  I_nam  ";
$res[3] = " Param  ";
$res[4] = "Last  ";
$res[5] = "  Value  ";
$res[6] = "  Iter  ";
$res[7] = "  Time";
for($L = 1 ; $L < 8 ; $L++){
  print RESULT $res[$L];
}
print RESULT "\n";

$tpar[1] = " -r 1 ";          # number of independent trials
$tpar[2] = " -s 100 ";        # number of steps in each trial
$tpar[4] = " -t 15 ";         # maximum time for each trial
$tpar[5] = " -o 1 ";          # stop if tour better or equal optimum is found
$tpar[6] = " -m 25 ";         # number of ants
$tpar[7] = " -g 20 ";         # nearest neighbours in tour construction
$tpar[8] = " -a 0.5 ";        # alpha (influence of pheromone trails)
$tpar[9] = " -b 4 ";          # beta (influence of heuristic information)
$tpar[10] = " -e 0.5 ";       # rho: pheromone trail evaporation
$tpar[11] = " -q 0.0 ";       # q_0: prob. of best choice in tour construction
$tpar[12] = " -c 100 ";       # number of elitist ants
$tpar[13] = " -f 6 ";         # number of ranks in rank-based Ant System
```

```
$tpar[14] = " -k 20 ";          # No.of nearest neighbours for local search
$tpar[15] = " -l 0 ";           # 0: no local search 1: 2-opt 2: 2.5-opt 3: 3-opt
$tpar[16] = " -d 1 ";           # 1 use don't look bits in local search
$tpar[17] = " --as ";           # apply basic Ant System
$tpar[18] = " --eas ";          # apply elitist Ant System
$tpar[19] = " --ras ";          # apply rank-based version of Ant System
$tpar[20] = " --mmas ";         # apply MAX-MIN Ant System
$tpar[21] = " --bwas ";         # apply best-worst Ant System
$tpar[22] = " --acs ";          # apply Ant Colony System

#1
$tpar[10] = " -e 0.0025 ";
$com1 = $tpar[1].$tpar[2];
$com2 = $tpar[4].$tpar[8].$tpar[9].$tpar[10].$tpar[15].$tpar[17];
$param =  execut($com1,$com2,$param);
system 'rm -f /home/ddarquennes/ACOTSPV10sp1/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp2/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp3/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp4/resultats/*';
system 'rm -f /home/ddarquennes/resultats/*';

#2
$tpar[10] = " -e 0.0050 ";
$com1 = $tpar[1].$tpar[2];
$com2 = $tpar[4].$tpar[8].$tpar[9].$tpar[10].$tpar[15].$tpar[17];
$param =  execut($com1,$com2,$param);
system 'rm -f /home/ddarquennes/ACOTSPV10sp1/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp2/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp3/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp4/resultats/*';
system 'rm -f /home/ddarquennes/resultats/*';

#3
$tpar[10] = " -e 0.0075 ";
$com1 = $tpar[1].$tpar[2];
$com2 = $tpar[4].$tpar[8].$tpar[9].$tpar[10].$tpar[15].$tpar[17];
$param =  execut($com1,$com2,$param);
system 'rm -f /home/ddarquennes/ACOTSPV10sp1/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp2/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp3/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp4/resultats/*';
system 'rm -f /home/ddarquennes/resultats/*';

#4
$tpar[10] = " -e 0.01 ";
$com1 = $tpar[1].$tpar[2];
```

```
$com2 = $tpar[4].$tpar[8].$tpar[9].$tpar[10].$tpar[15].$tpar[17];
$param =  execut($com1,$com2,$param);
system 'rm -f /home/ddarquennes/ACOTSPV10sp1/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp2/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp3/resultats/*';
system 'rm -f /home/ddarquennes/ACOTSPV10sp4/resultats/*';
system 'rm -f /home/ddarquennes/resultats/*';


system 'R < commands3.r --save';

#fin
#system 'rm -f /home/ddarquennes/ACOTSPV10sp1/F*.tsp';
#system 'rm -f /home/ddarquennes/ACOTSPV10sp2/F*.tsp';
#system 'rm -f /home/ddarquennes/ACOTSPV10sp3/F*.tsp';
#system 'rm -f /home/ddarquennes/ACOTSPV10sp4/F*.tsp';
#system 'rm -f /home/ddarquennes/F*.tsp';
```

Exec program : construction of data tables

```
package Exec;
require Exporter;

our @ISA    = qw(Exporter);
our @EXPORT = qw(execut);

#### definition fonction

sub execut { # debut sous-programme

  $com1  = shift(@_);
  $com2  = shift(@_);
  $param = shift(@_);

  open(RESULT, ">>values0603.dat");
  open(INST, ">>instance0603.dat");

  for($j = 1; $j < 5; $j++) {
    $part1 = "cd ACOTSPV10sp";
    $part2 = "; for fichier in \$(ls F*.tsp); do echo \"test avec sp \" ";
    $part3 = "; nice -19 ./acotsp ";
    $part4 = "; done";
    $commande1 = $part1.$j.$part2.$j.$part3.$com1." -p ".$j.$com2." -i \$fichier ".$part4;

    system $commande1;

    $part11 = "cp /home/ddarquennes/ACOTSPV10sp";
    $part12 = "/resultats/* /home/ddarquennes/resultats";

    $commande2 = $part11.$j.$part12;

    system $commande2;

    $rep = "/home/ddarquennes/resultats";
    opendir DH, $rep or die "Impossible d'ouvrir $rep : $!";
    foreach $fichiers (readdir DH) { # begin foreach
      if($fichiers ne "." and $fichiers ne "..") { # if
        $fichiers =  $rep."/".$fichiers;

        open(TITI, $fichiers) or die "impossible d'ouvrir $fichiers : $!";
        $nbl = 0;
        LIGNE: while($ligne = <TITI>) { # LIGNE
          $nbl++;
        } # LIGNE
        close TITI;
```

```
        open(TOTO, $fichiers) or die "impossible d'ouvrir $fichiers : $!";
        select INST;
        $i = 1;
        $nbli = 0;
        LIGNE1: while($ligne = <TOTO>) { # LIGNE1
          $nbli++;
          last LIGNE1 if $i > 25;
          $der = rindex($ligne, "\t");
          substr($ligne, 0, $der + 1) = " ";
          $dern = rindex($ligne, "\n");
          substr($ligne, $dern, $dern + 1) = "  ";
          if ($i == 2){
            $tab[$i] = $ligne."   ".$param."    ";
            } else {
            $tab[$i] = $ligne;
            }
          if ($i == 1){ $Sp = $ligne;}
          if ($i == 2){ $Inst = $ligne;}
          print $tab[$i];
          $i++;
          } # E LIGNE1
        print "\n";
        select RESULT;
        LIGNE2: while($ligne = <TOTO>) { # LIGNE2
          if ($i < 26) {
            $i++;
            next LIGNE2;
          }
          $nbli++;
          $_ = $ligne;
          s/[a-z]+/  /;
          s/[a-z]+/  /;
          s/[a-z]+/  /;
          if($nbli == $nbl){
                    } else {
            $last = 0;
          }
          $_ = $Sp."   ".$Inst."    ".$param."    ".$last."     ".$_;
          print "$_";
          $i++;
        } # E LIGNE2
      } # end if
      close TOTO;
    } # end foreach
  } # end for
  $param++;
  return $param;
} # end sous-programme $last = 1;
```

Commands3.r program : final graphics

```
Val <- read.table("values0603.dat", header=TRUE)

u <- subset(Val, Val$Last==1)
write.table(u, file="datas0603.dat", quote=TRUE, row.names=FALSE, col.names=TRUE)
Val1 <- read.table("datas0603.dat", header=TRUE)
resfin <- aggregate(Val1[,5:7], list(Sp=Val1$Sp, Param=Val1$Param), median)
resfin
write.table(resfin, append = TRUE, file="resfin0603.dat", quote=FALSE, col.names=TRUE)


nbpa  <- 4
# assign("pa", c(0.0025, 0.0050, 0.0075, 0.01))   # number of tested parameters
nbsp <- 4                                          # number of memory depth types
nbfi <- 100                                        # number of test files

cnt  <- 0

pa   <- 1:nbpa                                     # vector for number of parameters
sp   <- 1:nbsp                                     # vector for number of memory depth type
su   <- seq(0.05, 15.05, by=5)                     # vector for number of subdivision abscisse
fi   <- 1:nbfi                                     # vector for number of test files

t    <- length(pa)*nbsp*length(su)                 # max number of final results
z    <- seq(1, t, by=1)                            # size of vector for final-results
o    <- 1:length(su)
abs  <- numeric(length(o))                         # vector of abscisses


sure <- numeric(length(fi))                        # vector of sub-results
fire <- numeric(length(z))                         # vector of final-results


for(i in 1:length(pa)) {
  for(k in 1:length(sp)) {
    for(n in 1:length(su)) {
      for(m in 1:length(fi)) {
        j <- subset(Val, Val$Param == i & Val$Sp == k & Val$I_nam == m)
        sure[m] <- min(j$Value[j$Time <= su[n]])
      }
      cnt <<- cnt + 1
      fire[cnt] <- mean(sure)
    }
  }
}
```

```
opar <- par()
pdf()

cnt <- 0

for(i in 1:length(pa)) {
  plot(su, abs, type="n", xlab="CPU Time", ylab="Tour length", xlim=c(0, 15), ylim=c(400,
      700))
  legend(10, 700, legend=c("Algo 1", "Algo 2", "Algo 3", "Algo 4"), pch=1:4)
  switch(i, title("Alpha=0.5 Beta=4 Evap=0.0025"), title("Alpha=0.5 Beta=4 Evap=0.0050"),
          title("Alpha=0.5 Beta=4 Evap=0.0075"), title("Alpha=0.5 Beta=4 Evap=0.01"))
  for(k in factor(sp)) {
    for(n in 1:length(su)) {
      cnt <<- cnt + 1
      abs[n] <- fire[cnt]
    }
    switch(k, points(su, abs, pch=1), points(su, abs, pch=2), points(su, abs, pch=3), points(su,
            abs, pch=4))
    switch(k, lines(su, abs, lty=1), lines(su, abs, lty=1), lines(su, abs, lty=1), lines(su, abs,
            lty=1))
  }
}

write.table(fire, file="resf0603.dat", quote=FALSE, row.names=FALSE, col.names=TRUE)
```