Sapienza Università di Roma
A.A. 2012-2013
Facoltà di Scienze Matematiche Fisiche e Naturali
Dipartimento di Matematica "Guido Castelnuovo" PhD in Mathematics

Improving convergence of combinatorial optimization meta-heuristic algorithms

Candidate
Lorenzo Carvelli

Supervisor
Prof.Giovanni Sebastiani

Contents

Introduction

In this thesis, we focused on algorithms to solve combinatorial optimization problems (COP). Solving a COP consists of finding an element within a finite space, typically exploiting a combinatorial nature, e.g. the space of the permutations of the first N natural numbers. Such an element should minimize (maximize) a given so called *fitness* function. The COP prototype is the Traveling Salesman Problem (TSP), whose solution is an Hamiltonian cycle on a weighted graph with minimal total weight. Although a solution of a COP always exists, finding it may involve a very high computational cost. The study of algorithm computational cost started in the early 1940s with the first introductions of computers. Two different kinds of algorithms can be used to solve a COP problem: exact or heuristic (approximation). A method of the former type consists of a sequence of non ambiguous and computable operations producing a COP solution in a finite time. Unfortunately, it is often not possible to use exact algorithms. This may be the case for instances of a $\mathcal{NP}$-complete COP. In fact, to establish with certainty if any element of the search space is a solution, requires non polynomial computational cost. Alternatively, heuristic algorithms can be applied. Such type of algorithms only guarantee either a solution in an infinite time or a suboptimal solution. Of great importance are the *meta-heuristic* algorithms (MHA). They are general purposes algorithms independent of the particular COP considered. Moreover, they are usually based on advanced techniques. Often, algorithms of this kind are stochastic. Among them, there are Simulated Annealing, Tabu Search, Genetic Algorithms and Ant Colony Optimization (ACO).

The simulated annealing (SA) takes the name from technique involving heating

and controlled cooling of a material to increase the size of its crystals and reduce their defects, both are attributes of the material that depend on its thermodynamic free energy. This notion of slow cooling is implemented in the Simulated Annealing algorithm as a slow decrease in the probability of accepting worse solutions as it explores the solution space ([30],[7]).

Tabu search (TS) enhances the performance of these techniques by using memory structures that describe the visited solutions or user-provided sets of rules [17]. If a potential solution has been previously visited within a certain short-term period or if it has violated a rule, it is marked as "tabu" (forbidden) so that the algorithm does not consider that possibility repeatedly.

The genetic algorithm (GA) mimics the process of natural selection is routinely used to generate useful solutions to optimization and search problems. The GA belongs to the larger class of evolutionary algorithms (EA), which generate solutions to optimization problems using techniques inspired by natural evolution, such as inheritance, mutation, selection, and crossover [18].

The prototype of ACO takes into account the basic mechanisms by which a colony of ants finds the shortest path between nest and food.

A natural issue for MHA is their convergence. Given the stochastic nature of such algorithms, they are studied probabilistically. Anyway, when convergence is theoretically guaranteed, often it is too slow for using the algorithm in practice. One possible way to cope with this problem is the so called *restart* approach. This consists of several independent executions of a basic MHA. The executions are randomly initialized and the best solution is chosen among those produced. Despite the fact that the restart is widely used, very little work has been done to study theoretically this approach. Here, first we studied under which conditions there is an advantage to apply the restart. This was done by looking at the expected time to find a solution by both the basic MHA and the restart one. We provided conditions for the MHA *failure probability* along iterations, which are sufficient to obtain a gain when applying the restart.

Another important issue is the choice of the time to restart the MHA. Usually, the basic MHA is restarted when there are negligible differences in the fitness of

candidate solutions at consecutive iterations. However, this criterion may not be adequate when we want to really find the COP solution while we are not satisfied with suboptimal ones. This is because the criterion above is not related to the restart failure probability. The failure probability when applying the restart is geometrically decreasing towards zero with the number of restarts. The base of such geometric sequence is the failure probability of the involved the MHA at the restart time. Therefore, if the above criterion provides a short restart time, this may result in a slow convergence. In order to increase the speed of convergence one can decrease the base value by choosing a high value of the restart time. Unfortunately, this either may require a very long computation time or may correspond to a low number of restarts. Therefore, a natural problem is to find an "optimal value" of the restart time. This is studied here theoretically. To this aim, we adopted the criterion of minimizing the restart failure probability corresponding to a fixed amount of total computation time. The problem is then to find a minimum of a suitable function of the MHA failure probability curve along time. Whenever the minimum of this function does not exist, there is no advantage to use the restart. In one part of the study, we derived sufficient or necessary conditions for the MHA failure probability in order to have the existence of this minimum. However, these conditions cannot be applied in practice since the MHA failure probability is obviously unknown.

In another part of the study, we propose a new iterative procedure to optimize the restart. It does not rely on the MHA failure probability. Therefore, it can be applied in practice. The procedure consists of either adding new MHA executions or extending the existing ones. The procedure uses a certain version of the MHA failure probability where the optimal solution is replaced by the *best so far* one. We make the hypothesis that the MHA failure probability converges to zero with the number of iterations. Then, we proved that with probability one the restart time of the proposed procedure approaches, as the number of iteration diverges, the optimal value based on the criterion of above.

We applied the proposed restart procedure to several TSP instances with hundreds or thousands of cities with known solution. As MHA we used different versions of an ACO. We compared the results from the restart procedure with those

from the basic ACO, which we implemented in the C language. This was done by considering the failure probability of the two approaches corresponding to the same total computation cost. This comparison showed a significant gain when applying the proposed restart procedure. In fact, the failure probability decreased by one or two orders of magnitude.

The thesis is structured as follow. In the first chapter, we introduce both the computational complexity issue for COP algorithms and the MHA. In the second chapter, we illustrate some theoretical results on ACO convergence. In the third chapter, we study theoretically when there is an advantage to use the restart. Finally, in the last chapter, we describe the proposed restart procedure and we study its convergence. Furthermore, we show some results of the application of the procedure to solve TSP instances in combination with ACO. In the appendix, we include the C code of the used ACO.

Chapter 1

Computational complexity and some approach

An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. Starting from an initial state and initial input, the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input. The definition of the algorithm above is, of course, quite informal. In order to deal with the concept of algorithm with mathematical tools, it was necessary to give a more rigorous definition. This was achieved by inventing a series of mathematical models. One of the most famous is the Turing machine [41]. It represents a sort of ideal computer equipped with a program to run. Compared to an ideal computer, the Turing machine has an easier operation, but with the advantage that its operation is easily described in mathematical terms, using concepts such as set, relation and function. Furthermore, it was shown that the Turing machine is as powerful as the Von Neumann machine, which is the model underlying all real computers. In algorithms theory the study of computational and spatial complexity is very important. We want to know, when the complexity of the problem increasing, how the time to execute the algorithm and the space occupied in

the computer's memory increase. These kind of studies was born in the early 1940s with the confluence of algorithms theory, mathematical logic, and the invention of the stored-program electronic computer. The body of knowledge of computing is frequently described as the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation, and application. The fundamental question underlying all of computing is, What can be (efficiently) automated?

1.1 Computational complexity

A problem is described by giving a general description of all its parameters and a statement of what properties answer or solution is required to satisfy. An instance of a problem is obtained specifying particular values for all the problem parameters. An algorithm is said to solve a problem Π if that algorithm can be applied to any instance I of Π and is guaranteed always to produce a solution for instance I. Each problem has associated with it a fixed encoding scheme which maps problem instances into the strings describing them.

Definition 1.1.1. *The input length for the instance I of a problem Π is the number of symbols in the description of I obtained from the encoding schema for Π.*

Definition 1.1.2. *The time complexity function for an algorithm expresses that its time requirements by giving, for each possible input length, the largest amount of time needed by the algorithm to solve a problem instance of that size*

Definition 1.1.3. *Let us say that $f(n)$ is $\mathcal{O}(g(n))$ whenever there exists a constant c such that $f(n) \leq c \cdot |g(n)|$ for all $n \geq 0$.*

A polynomial time algorithm is defined to be one whose time complexity function in $\mathcal{O}(p(n))$ for some polynomial function $p(n)$, where n is the input length. Any algorithm where time complexity function cannot be bounded with a polynomial function is called on exponential time algorithm. In complexity theory there is wide agreement that a problem has not been well-solved until a polynomial time algorithm

is known for it. So, we shall refer to a problem as formatting if it is so hard that no polynomial time algorithm can possibly solve it. The intractability can be a consequence of two facts. The first, which is the most intuitive, is that the problem is so difficult that an exponential amount of time is needed to discover a solution. The last is that the solution itself is required to be so extensive that it cannot be described with an expression having length bounded by a polynomial function of the input length. For example, if we consider the following variation of *traveling salesman problem* (TSP): for a fixed length $l > 0$, to find all tours with length almost l, we have that the intractability is a consequence of second fact. However, in most cases, the existence of the second type of intractability is apparent from the problem definition because we are asking for more information than we could ever hope to use. So, from now, we shall consider only the first type of intractability. The earliest intractability results for such problems are the classical undecidability result of Alan Turing (cf. [41]) which demonstrated that certain problems are so hard that they are "undecidable" in the sense that no algorithm at all can be given for solving them.

1.2 $\mathcal{NP}$-completeness

In order to introduce the $\mathcal{NP}$-completeness we give the following definitions.

Definition 1.2.1. *A decision problem consists of some formalized question admitting a yes-or-no answer, which may depend on the values of some input parameters.*

Definition 1.2.2. *The class $\mathcal{P}$ is the class of decision problems that can be solved by a polynomial time algorithm.*

Definition 1.2.3. *The class $\mathcal{NP}$ consists of decision problems that can be solved by a nondeterministic polynomial-time algorithm.*

The relation between these two kind of problems can be established by the following theorem; for the proof we refer to [14].

Theorem 1.2.4. *If $\Pi \in \mathcal{NP}$, then there exists a polynomial p such that Π can be solved by a deterministic algorithm having time complexity $\mathcal{O}(2^{p(n)})$.*

The foundations for the theory of $\mathcal{NP}$-completeness is due to Stephen Cook (cf [8]) where it is emphasized the significance of "polynomial time reducibility" for which the required transformation can be executed by a polynomial time algorithm. In order to give a formal definition of "$\mathcal{P}$-reducible", we have to introduce some definitions.

Definition 1.2.5. *A deterministic Turing machine (DTM) is a 6-tuple $T = (\Gamma, \bar{b}, Q, q_0, F, \delta)$ where Γ is tape symbol alphabet, $\bar{b} \notin \Gamma$ is blank symbol, Q is finite set of states no void, q_0 is initial state, $F \subseteq Q$ is final set of states, δ is transition function defined as follow*

$$\delta : (Q \setminus F) \times (\Gamma \cup \{\bar{b}\}) \to Q \times (\Gamma \cup \{\bar{b}\}) \times \{r, l, s\},$$

where we denote by r, l, s the head tape which moves right, left, stationary, respectively.

This definition can be extended to multi-tape Turing machine (MTM) defining the functions $\delta^{(i)}$ in the space $(Q \setminus F) \times (\Gamma_1 \cup \{\bar{b}\}) \times \cdots \times (\Gamma_k \cup \{\bar{b}\})$ to $(Q \setminus F) \times (\Gamma_1 \cup \{\bar{b}\}) \times \cdots \times (\Gamma_k \cup \{\bar{b}\}) \times \{r, l, s\}^k$, where k is a tapes number and $i = 1, \ldots, k$. It proves that MTM and DTM are equivalents.

Definition 1.2.6. *A non-deterministic Turing machine (NDTM) is a 6-tuple $T = (\Gamma, \bar{b}, Q, q_0, F, \delta_N)$ where Γ is tape symbol alphabet, $\bar{b} \notin \Gamma$ is blank symbol, Q is finite set of states no void, q_0 is initial state, $F \subseteq Q$ is final set of states, δ_N is a partial transition function defined as follow*

$$\delta_N : Q \times (\Gamma \cup \bar{b}) \to \mathcal{P}(Q \times (\Gamma \cup \{\bar{b}\}) \times \{r, l, s\}),$$

where we denote by r, l, s the head tape which moves right, left, stationary, respectively.

A query machine is a multi-tape Turing machine with a distinguished tape called the query tape, and three distinguished states called the query state, yes state, and

no state, respectively. If M is a query machine and T is a set of strings, then a T-computation of M is a computation of M in which initially M is in the initial state and has an input string w on its input tape, and each time M assumes the query state there is a string u on the query tape, and the next state M assumes is the yes state if $u \in T$ and the no state if $u \notin T$.

Definition 1.2.7. *A set S of strings is $\mathcal{P}$-reducible ($\mathcal{P}$ for polynomial) to a set T of strings iff there is some query machine M and a polynomial $Q(n)$ such that for each input string w, the T-computation of M with input w halts within $Q(|w|)$ steps ($|w|$ is the length of w) and ends in an accepting state iff $w \in S$.*

Informally, this definition says that if Π can be polynomially reduced to Π', then problem Π' is at least as difficult to solve as problem Π. If we have a polynomial time reduction from one problem to another, this ensures that any polynomial time algorithm for the second problem can be converted into a corresponding polynomial time algorithm for the first problem. Moreover Cook studied on the class $\mathcal{NP}$ of decisions problem that can be solved in polynomial time by nondeterministic computer. He studied satisfiability problem also called SAT that it establishes if the variables of a given boolean formula can be assigned such that the entire logical expression is true. Cook proved that the satisfiability problem has the property that every other problem in $\mathcal{NP}$ can be polynomially reduced to it. If the satisfiability problem can be solved with a polynomial time algorithm, then so can every problem in $\mathcal{NP}$, and if any problem in $\mathcal{NP}$ is intractable, then the satisfiability problem also must be intractable. In this sense, the SAT problem is the hardest problem in $\mathcal{NP}$.

Definition 1.2.8. *A problem Π is $\mathcal{NP}$-complete if and only if for all $\Pi' \in \mathcal{NP}$ holds that Π' is polynomially reducible to Π.*

Definition 1.2.9. *A problem Π is $\mathcal{NP}$-hard if and only if there is an $\mathcal{NP}$-complete problem Π' that is $\mathcal{P}$-reducible to Π.*

It is no difficult to see that if Π belongs to $\mathcal{P}$ and there exist a polynomial time reduction of Π' to Π, then also Π' belongs to $\mathcal{P}$. It implies that if one $\mathcal{NP}$-complete problem can be solved in polynomial time, then each problem in $\mathcal{NP}$ can be solved

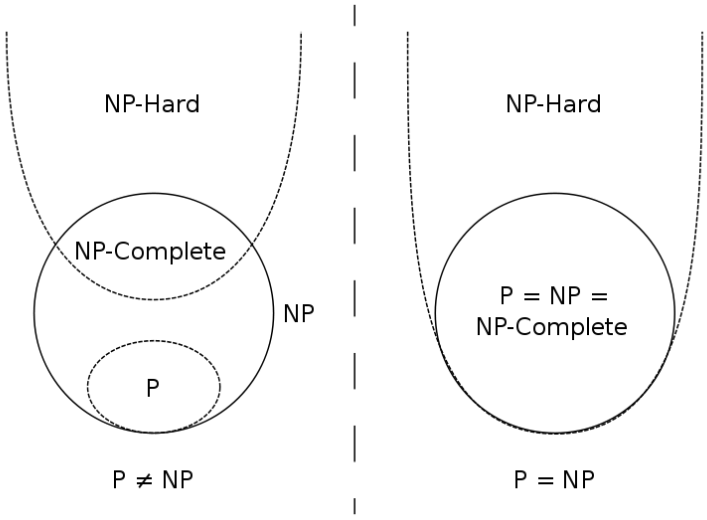

Figure 1.1: In this figure, we illustrate two possibilities in relation with answer of the problem P/NP

in polynomial time. Moreover, if Π belongs to $\mathcal{NP}$, and Π' is $\mathcal{NP}$-complete and there exists a polynomial time reduction of Π' to Π, then also Π is $\mathcal{NP}$-complete.

Theorem 1.2.10. (Cook's theorem) *The satisfiability problem is $\mathcal{NP}$-complete.*

After this result, Richard Karp (cf. [29]) presentend a collection of results proving that indeed the decision problem versions of many well known combinatorial problem, including the TSP are just as "hard" as the SAT problem. Since then a wide variety of other problems have been proved equivalent in difficulty to these problems, and this equivalence class, consisting of the "hardest" problems in $\mathcal{NP}$ has been given a name: the class of $\mathcal{NP}$-complete problems. So the Cook's ideas have provided the means for combining many individual complexity questions into the single question: Are the $\mathcal{NP}$ complete problems intractable? This question is now considered to be one of the foremost open questions of contemporary mathematics and computer science.

1.3 Combinatorial Optimization Problems (COP)

A combinatorial optimization problem (COP) is either a maximization or minimization problem with an associated set of feasible solutions $\mathcal{S}$.

Definition 1.3.1. *An instance of a combinatorial optimization problem is a pair* $(\mathcal{S}, f)$ *where* $\mathcal{S}$ *is the finite set of candidate solutions and* $f : \mathcal{S} \to \mathbb{R}$ *is a function which assigns to every* $s \in \mathcal{S}$ *a value* $f(s)$*, also called objective function value. To goal of a combinatorial optimization problem is to find a solution* $s \in \mathcal{S}$ *with minimal (maximal) objective function, i.e.* $f(s_{opt}) \leq (\geq) f(s) \quad \forall s \in \mathcal{S}$*. Moreover* s_{opt} *is called a globally optimal solution of* $(\mathcal{S}, f)$ *and* $\mathcal{S}_{opt}$ *is the set of all globally optimal solutions.*

Often the set $\mathcal{S}$ naturally arises as a subset of 2^E (the set of all subsets of E), for some finite ground set E. Of course, there is no problem because we can just enumerate all feasible solutions but we seek to do better. Usually, the feasible solutions are described in some concise manner, rather than being explicitly listed. The challenge is to develop algorithms that are provably or practically better than enumerating all feasible solutions. Applications of discrete optimization problems arise in industry and in applied sciences. Besides the applications, discrete optimization has aspects that connect it with other areas of mathematics as well as computer science. Thus research in discrete optimization is driven by mathematics as well as by applications.

In COP, we have to find solutions which are optimal or near-optimal with respect to some goals. Usually, we are not able to solve problems in one step, but we follow some process which guides us through problem solving. Often, the solution process is separated into different steps which are executed one after the other. Commonly used steps are recognizing and defining problems, constructing and solving models, and evaluating and implementing solutions.

1.3.1 Some COP $\mathcal{NP}$ complete

As mentioned above, the first problem that has been shown to be NP complete is

Boolean satisfiability problem: In computer science, satisfiability (often abbreviated SAT) is the problem of determining if there exists an interpretation, which satisfies the formula. In other words, it establishes if the variables of a given boolean formula can be assigned in such a way as to make the formula evaluate to "1". Equally important is to determine whether no such assignments exist, which would imply that the function expressed by the formula is identically "0" for all possible variable assignments.

Since SAT is $\mathcal{NP}$ complete, it is shown that the partition problem is $\mathcal{NP}$ complete (cf. [29]).

Partition problem: Given a collection $\mathcal{C}$ of subsets of a finite set X, is there a subcollection of $\mathcal{C}$ that forms a partition of X?
A corollary of the last result cited is that hamiltonian path problem is $\mathcal{NP}$ complete.

Hamiltonian path problem: Remembering that a Hamiltonian path (Hamiltonian cycle) is a path that visits each vertex exactly once, given a graph G, the problem consists of determining whether G contains a Hamiltonian path.
Moreover, in [29] is shown that also TSP is $\mathcal{NP}$ complete. In order to show this, was given a polynomial time reduction of undirected hamiltonian path to TSP.

Traveling salesman problem (TSP): The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. For this problem, we have two versions: the first is the optimization version and the latter is the decision version. Here we describe both versions. Given a symmetric complete graph $G = (V, E)$, where V is a vertices set (cities) and E is a edge set (distances between each pair of cities) what is the shortest possible route that visits each link

exactly once and returns to the origin node? The parameters of this problem consist of finite set $C = \{c_1, c_2, \cdots, c_m\}$ of cities and, for each pair of cities $(c_i, c_j) \in C \times C$, a distance $d(c_i, c_j)$ between them. A solution is an ordering of $(c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(m)})$ of the given cities that minimizes

$$\left[\sum_{i=1}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)})\right] + d(c_{\pi(m)}, c_{\pi(1)}).$$

The decision version of this problem is: fixed a length l, does it exist a tour with length at most l? There are many variation of this problem. For example, it is possible to consider the tour without the edge that connects the last city with the first city; or it is possible to eliminate the hypothesis of symmetry (existence of one-way streets).

Quadratic assignment problem (QAP): There are a set of n facilities P and a set of n locations L. For each pair of locations, a distance is specified and for each pair of facilities a weight or flow is specified (e.g., the amount of supplies transported between the two facilities). The problem is to assign all facilities to different locations with the goal of minimizing the sum of the distances multiplied by the corresponding flows. In other words, let $c_{i,j}$ be non-negative integer costs with $1 \leq i, j \leq n$ ($c : P \times P \to \mathbb{R}$), and distances $d_{k,l}$, where $1 \leq k, l \leq m$. A solution is a bijective function $f : P \to L$ (“'assignment') such that the cost function

$$\left[\sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} c_{i,j} \cdot d_{f(i),f(j)}\right]$$

is minimized.

Vehicle routing problem (VRT): The road network, used for the transportation of goods, is generally described through a graph $G = (V, E)$, whose arcs represent the road sections and whose vertices correspond to the road junctions and to the depot d and customer locations. The arcs (and consequently the corresponding graphs) can

be directed or undirected, depending on whether they can be traversed in only one direction (for instance, because of the presence of one-way streets, typical of urban or motorway networks) or in both directions, respectively. Each arc is associated with a cost $c_{i,j}$, which generally represents its length, and a travel time, which is possibly dependent on the vehicle type or on the period during which the arc is traversed. In other words, introducing three kind of variables: $x_{i,j}^k$ to represent if the edge (i,j) belongs to the path of vehicle k, $y_{i,j}$ to consider every vehicle, z_i^k to describe if the customer i belongs to the path k, we have to minimize the objective function

$$\sum_{(i,j)\in V} c_{i,j} y_{i,j},$$

with suitable constraints. The constraints are:

- a feasible solution has to verify the maxima capacity $Q : \sum_{i\in V} z_i^k \leq Q$,
- every customer has to be visited by only one vehicle: $\sum_{k\in K} z_i^k = 1$,
- $x_{ij}^k \leq z_i^k \quad \forall (i,j) \in E, \quad \forall k \in K$,
- $y_{ij} = \sum_{k\in K} x_{ij}^k, \quad \forall (i,j) \in E$,
- $\sum_{(i,j)\in\delta(h)} y_{ij} = 2, \quad \forall h \in V \setminus d$,
- $\sum_{(i,j)\in\delta(d)} y_{ij} = 2k$.

A complete description about this problem it can find in [40].

1.4 Solution methods for combinatorial problem

If we consider a problem $\mathcal{NP}$-hard, now, it is not possible to have a polynomial-time algorithm that solves the problem. Therefore we study algorithms of different types. In particular, we have two kind of algorithms.

- Exact algorithms: They reach the optimal solution after a finite number of steps.
- Approximation algorithms: They make research in the search space according to appropriate criteria based on stochastic search and/or local search.

1.4.1 Exact Algorithms

For any optimization problem, way to know the solution is to try all the elements in the search space. This means that for problems $\mathcal{NP}$-hard as TSP, if, for a fixed instance, the space is too large, it is impossible to obtain a exact solution of the problem. For the TSP, the computational cost of a generic algorithm exhaustive search is $\mathcal{O}(n!)$. Nevertheless, research carried out in this area have obtained considerable improvements. In fact, by the dynamic programming, introduced by Bellman in 1960 (cf. [3], [4]), it was born one of the earliest applications of dynamic programming with the Held Karp algorithm [24] that solves the problem in time $\mathcal{O}(n^2 2^n)$. The dynamic programming solution requires exponential space. Using inclusion-exclusion, the problem can be solved in time within a polynomial factor of 2^n and polynomial space (cf. [31], [28]). For some problems in restricted suitable conditions it is possible to find exact algorithm with less time. For example Hwang, Chang and Lee describe in [26] a sub-exponential time $\mathcal{O}(c^{\sqrt{n}\log n})$ exact algorithm with some constant $c > 1$ for the Euclidean TSP. The last one is a special case of the TSP where the cities are points in the euclidean plane and where the distance between two cities is the euclidean distance. This result is based on planar separator structures, and it can not be generalized for all TSP. Other approaches include: various branch-and-bound algorithms, which can be used to process TSPs containing 40-60 cities; progressive improvement algorithms which use techniques reminiscent of linear programming (works well for up to 200 cities); implementations of branch-and-bound and problem-specific cut generation (branch-and-cut) (cf. [2]); this is the method of choice for solving large instances. This approach holds the current record, solving an instance with 85900 cities (see [1]). A survey about exact algorithms for $\mathcal{NP}$-complete problems can be found in [43].

1.4.2 The meta-heuristic algorithms (MHA)

Since many problems $\mathcal{NP}$-hard can not be solved by exact algorithms, we resort to use approximate algorithms to accept even sub-optimal solutions near global optimum. However, using, these kind of algorithm we can not guaranteed to have obtained the optimum. There are different type of an approximate. In fact there are heuristic-constructive algorithms that start by initial point and construct a solution following an appropriate criterion. These algorithms have the advantage of being very fast losing precision compared to local search algorithms which, starting from an initial solution try to improve it moving to solutions neighboring of the current solution and iterating the procedure. An approximation algorithm is always assumed to be polynomial. We also assume that the approximation algorithm delivers a feasible solution to some $\mathcal{NP}$-hard problem that has a set of instance $[I]$.

Definition 1.4.1. *A polynomial algorithm, $\mathcal{A}$, is said to be a δ-approximation algorithm if for every problem instance I with an optimal value $OPT(I)$,*

$$\frac{f_{\mathcal{A}}(I)}{OPT(I)} \leq \delta,$$

where $f_{\mathcal{A}}(I)$ is the value of optimal solution found by $\mathcal{A}$ for the instance I.

We recall that $\delta \geq 1$ for minimization problems and ≤ 1 for maximizations problems. The smallest value of δ is the approximation ratio $R_{\mathcal{A}}$ of the algorithm $\mathcal{A}$. For maximization problems, sometimes $\frac{1}{\delta}$ is considered to be the approximation ratio/factor.

Definition 1.4.2. *The absolute performance ratio, $R_{\mathcal{A}}$, of an approximation algorithm $\mathcal{A}$ is,*

$$R_{\mathcal{A}} = \inf\{r \geq 1 \mid R_{\mathcal{A}}(I) \leq r \quad \textit{forall problem instances } I\},$$

and the asymptotic performance ratio $R_{\mathcal{A}}^{\infty}$ for $\mathcal{A}$ is

$$R_{\mathcal{A}}^{\infty} = \inf\{r \geq 1 \mid \exists n \in \mathbb{N}, R_{\mathcal{A}}(I) \leq r \quad \forall |I| \geq n\},$$

where $|I|$ is the size of the instance I.

The means of $R_{\mathcal{A}}$ is a "worst case" notion, i.e. it suffices to have a single "bad" instance to render the value of δ larger than it is for all other encountered instances. It is the same as the absolute performance ratio, with a lower bound n on the size of problem instances.

1.4.3 Local search algorithms

In computer science, local search is a heuristic method for solving computationally hard optimization problems. Local search can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions. Local search algorithms move from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed. Local search algorithms are widely applied to numerous hard computational problems, including problems from computer science (particularly artificial intelligence), mathematics, operations research, engineering, and bioinformatics. Most problems can be formulated in terms of search space and target in several different manners. For example, for the TSP a solution can be a cycle and the criterion to maximize is a combination of the number of nodes and the length of the cycle. But a solution can also be a path, and being a cycle is part of the target. A local search algorithm starts from a candidate solution and then iteratively moves to a neighbor solution. This is only possible if a neighborhood relation is defined on the search space. As an example, the neighborhood of a vertex cover is another vertex cover only differing by one node.

For boolean satisfiability, the neighbors of a truth assignment are usually the truth assignments only differing from it by the evaluation of a variable. The same problem may have multiple different neighborhoods defined on it; local optimization with neighborhoods that involve changing up to k components of the solution is often referred to as k-opt. Typically, every candidate solution has more than one neighbor solution; the choice of which one to move to is taken using only information about the solutions in the neighborhood of the current one, hence the name local

search. When the choice of the neighbor solution is done by taking the one locally maximizing the criterion, the meta-heuristic takes the name hill climbing. When no improving configurations are present in the neighborhood, local search is stuck at a locally optimal point. This local-optima problem can be cured by using restarts (repeated local search with different initial conditions), or more complex schemes based on iterations, like iterated local search, on memory, like tabu search (TS), on memory-less stochastic modifications, like simulated annealing. Termination of local search can be based on a time bound. Another common choice is to terminate when the best solution found by the algorithm has not been improved in a given number of steps. Local search is an anytime algorithm: it can return a valid solution even if it's interrupted at any time before it ends. Local search algorithms are typically approximation or incomplete algorithms, as the search may stop even if the best solution found by the algorithm is not optimal. This can happen even if termination is due to the impossibility of improving the solution, as the optimal solution can lie far from the neighborhood of the solutions crossed by the algorithms. For specific problems it is possible to design neighborhoods which are very large, possibly exponentially sized. If the best solution within the neighborhood can be found efficiently, such algorithms are referred to as very large-scale neighborhood search algorithms.

Many widely known and high-performance local search algorithms make use of randomized choices in generating or selecting candidate solutions for a given combinatorial problem instance. These algorithms are called stochastic local search (SLS) algorithms, and they constitute one of the most successful and widely used approaches for solving hard combinatorial problems.

1.5 LKH algorithm and its evolutions

The Lin Kernighan heuristic algorithm (LKH) was introduced in [33] by Lin ad Kernighan which extended and generalized a method of local search introduced by Croes in [9] several previous years: 2-opt. Lin and Kernighan suggested the k-opt as the central point of their algorithm Lin Kernighan Heuristic (LKH). A first

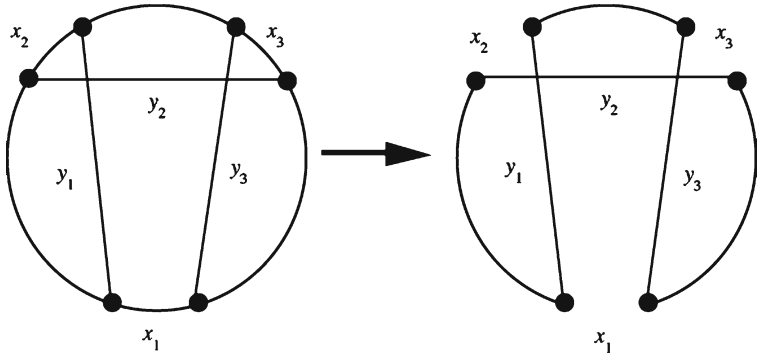

Figure 1.2: A 3-opt move. x_1, x_2, x_3 are replaced by y_1, y_2, y_3

generalization of [9] was proposed by Lin [32] introducing a concept of k-optimality as follows

Definition 1.5.1. *A tour is said to be k- optimal (or simply k-opt) if it is impossible to obtain a tour with smaller cost by replacing any k links by any other set of k links.*

Empirical evidence suggests that iterative improvement algorithms based on k-exchange neighborhoods with $k > 3$ return better tours, but the computation times required for searching these large neighborhoods render this approach ineffective. An idea to overcome this problem is to compose more complex steps from a number of steps in small, simple neighborhoods. The best-known algorithm that exploits this idea is LKH. The logic of this procedure is to understand the optimal k to apply the k-opt. In fact, the authors suppose to have a solution T which has k different arcs to the optimal solution. Lin and Kernighan want to understand this unknown k. In order to do this, they propose the procedure as follow:

1. Generate a random initial solution T

2. (a) Set i=1.

   (b) Select x_i and y_i as the most-out-of place pair at the ith step.This means that x_i and y_i are chose to maximize the improvement.

(c) If it appears that no more gain can be made according to "stopping rule", go to Step 3; otherwise, set $i = i + 1$ and go back to Step 2 (b).

3. If the best improvement is found for $i = k$, exchange $x_1, \ldots, x_k$ with $y_1, \ldots, y_k$ to give a new T and go to Step 2; if no improvement is found, go to Step 4.

4. Repeat from Step 1 if desired.

In [33] there is a detailed description of this procedure applied to the TSP. The k-opt is applied such that there are not repeated path, there is a gain, the gain is maximized. These constraints identify the "stopping rule" mentioned in the Step 2(c).

Some variants of this algorithm in [27] and [25].

In [38] is proposed the k-opt integrate in ACO algorithm in order to improve the performances.

Chapter 2

ACO algorithms

The Ant Colony Optimization (ACO) paradigm was developed by Dorigo, Maniezzo and Colorni (cf. [11],[12]).It imagines to have a graph and an ant moving randomly from one of these vertices to some other vertex, then to a third, etc., always avoiding still visited vertices. After the visit of the last point, the ant returns to its start position. This defines a tour, which can also be described by a sequence of arcs (i, j), where i and j are vertex indices. The ant releases a substance in the arcs on which it is passed it will attract other ants: the pheromone. So it assumes that if the tours happens to be comparably short, the ant increases its probability to traverse arcs (i, j) that lie on that tour in the future, and it decreases the probability of traversing other arcs. In other words, successful moves are reinforced. Computationally, this is done by an increment of real numbers assigned to the arcs, the so-called *pheromone values*, along the arcs of the tour. Then, the ant starts its walk from a new vertex, applying transition probabilities that are proportional to the current pheromone values. By this mechanism, the average quality of a tour improves over time, which can be used to obtain approximate solutions to the TSP. After first successes on the TSP, it turned out that the range of application of the ACO paradigm was much broader. The paradigm can be extended to the whole area of combinatorial optimization problems (COP) and even beyond this field, which means that ACO algorithms may be considered as meta-heuristic (cf [10]). A survey on different variants of ACO algorithms, their applications and properties is given

in [13].

2.1 The algorithms

Let V and E the set of the nodes (cities) and the set of the arcs, respectively. We denote by $\tau_{i,j}(t)$ the generic element (i,j) of pheromone matrix τ at the time t. The pheromone matrix depends the iteration of the algorithm so that the probability that an ant traverses the arc (i,j) change with the pheromone update rule. In order to solve the TSP, it was introduced a quantity called visibility to reward the shorter paths. That is, we define a function $\eta : V \times V \to \mathbb{R}^+$ (visibility) inversely proportional to the length of the arc (i,j) following the rule $\eta_{i,j} := \eta(i,j) = \frac{1}{d_{i,j}}$, where $d_{i,j}$ is the distance between i to j. Let u denote the current partial path of the ant, and we denote by $\mathcal{F}(u)$ the set of arcs of u that are considered "feasible". The set of "feasible" arcs are the arcs of the best solution found until current time or a variation of this rule. It depends ACO algorithm, which it could be exploit.

Procedure ACO
Initialize pheromone value $\tau_{i,j}$ on the arcs $(i,j) \in E$;
for iteration $t = 1, 2, \ldots$ **do**
  for ant $\sigma = 1, 2, \ldots, s$ **do**
    set i the current position of the ant, equal to the start node of V;
    set u, the current partial path of the ant, equal to the empty list;
    while $(\mathcal{F}(u) \neq \emptyset)$
      select successor node j with probability

$$p_{i,j} = \mathbf{1}_{\{(\mathbf{i},\mathbf{j}) \in \mathcal{F}(\mathbf{u})\}} \cdot \frac{\tau_{\mathbf{i},\mathbf{j}}(\mathbf{t})^{\alpha} \cdot \eta_{\mathbf{i},\mathbf{j}}^{\beta}}{\sum\limits_{(\mathbf{i},\mathbf{r}) \in \mathcal{F}(\mathbf{u})} \tau_{\mathbf{i},\mathbf{r}}(\mathbf{t})^{\alpha} \cdot \eta_{\mathbf{i},\mathbf{j}}^{\beta}}$$

      append arc (i,j) to u and set $i = j$;
    end while
    set $x^{\sigma} = u$;
  end for
  update the pheromone values $\tau_{i,j}$ based on the current solution x^{σ};

end for The parameters $\alpha, \beta \in \mathbb{R}^+$: α represents the weigh given to the pheromone matrix, β represents the weigh given to the visibility. Generally, $\alpha = 1$ and $\beta \in \{1, 2, 3, 4, 5\}$. The last one depends the problem.

2.2 A classification of ACO

The ACO algorithms, having many parameters and functions to be defined, are many and can be classified. Clearly any ACO depends the numbers of ants.

A parameter that gives rise to different algorithms is the rule that indicates which paths are reinforced. This parameter can take the following values:

ac: all the current iteration;

bf: best so far;

bf*: best so far with strict improvements;

bf$_0$: best so far on exchange only;

bf$_0^*$: best so far strict improvements on exchange only;

ib(r)**:** iteration best with r ranked ants.

ac: Each path traversed by some ant in the current iteration is reinforced, for any arc $(i, j) \in E$, following the rule:

$$\tau_{i,j} = (1 - \rho) \cdot \tau_{i,j} + \frac{\rho}{s} \cdot \sum_{\sigma=1}^{s} \mathbf{1}_{\{(i,j) \in x^\sigma\}} \cdot R(x^\sigma),$$

where $\rho \in [0, 1]$ is the so called evaporation rate and R is the reward function.

bf, **bf***, **bf$_0$**, **bf$_0^*$**: Only the best path found up to now in any of the previous iterations (including the current one) by any ant is reinforced. The difference between these variants are that bf and bf$_0$ change the optimum if the current solution of the generic ant σ is '$\leq$' of the previous best path; otherwise for bf$_0$ and bf$_0^*$ the relation is '$<$'. The rule, which reinforces any arc $(i, j) \in E$, is

$$\tau_{i,j} = (1 - \rho) \cdot \tau_{i,j} + \rho \cdot \mathbf{1}_{\{(i,j) \in \hat{x}\}} \cdot R(\hat{x}),$$

where $\hat{x}$ is the best solution found until current time.

(ib)(r): Let $x^1, x^2, \ldots, x^r$ denote the r best paths found until current iteration. Then, for any arc $(i,j) \in E$ the following pheromone update rule is applied:

$$\tau_{i,j} = (1-\rho) \cdot \tau_{i,j} + \rho \cdot \sum_{k=1}^{r} (r-k+1) \cdot \mathbf{1}_{\{(i,j) \in x^k\}} \cdot R(x^k).$$

Clearly these reinforcement rules can be combined.

The value of the reward function $R(x)$ describes the amount of reward that is added to the pheromone value of a reinforced arc. This can be constant (co) or fitness-proportional (fp). For the last case, an example of reward function is $\frac{C}{length(x)}$, where C is a constant.

An other important parameter is the pheromone bound. In fact it could set that any element of pheromone matrix is upper and/or lower bound by τ_{max} and τ_{min}, respectively. These bounds (we denote by b) will be important if ρ is constant.

The classical ACO algorithms, Ant System ([11],[12]) is the variant s-ac-fp-nb. MAX-MIN Ant System (MMAS) in [38] regards the variants s-bf*-fp-b, s-ib(1)-fp-b and s-bf*+ib(1)-fp-b. In [5] there are the cases s-ib(r)-fp-nb and s-bf*+ib(r)-fp-nb, with $r \geq 2$. In theoretical research, a simplified ACO algorithm called GBAS has been designed in order to study convergence properties. The GBAS variant investigated in [19] is s-bf$_0$-co-nb and in [20] the variants are s-bf*-co-nb and s-bf*-co-b. For runtime analysis purposes in [34] there is the study of the variant 1-bf$_0$-co-b and in [21], [23] there is the study of runtime analysis of variants 1-bf*-co-b and 1-bf*-fp-b.

2.3 The construction graph

For treating a given CO problem by ACO we have to define a graph on which the fictitious computational unit called "ant" performs its random walks and to encode each solution $x \in S$ as a path in this graph, such that the trajectories of the ants can be decoded to solutions in S. This graph which obviously has to depend on the problem instance, is called the *construction-graph* (CG). Two different suggestions

for formally defining the CG have been given in the ACO literature: a *pheromone-on-arcs* version and a *pheromone-on-nodes* version.

Definition 2.3.1. *It supposes that $G = (V, E)$ is a directed graph with node set V and arc set E. Let a unique node in $\mathcal{V}$ marked as the so called start node and let W be a set of directed paths $w \in G$, called feasible paths, that satisfy the following conditions*

i) *w starts at the start node of V;*

ii) *w contains each node of V at most once;*

iii) *w is a maximal in W , i.e., it cannot be prolonged to a longer feasible path in W.*

Moreover, let ϕ be a function mapping the set W of feasible paths onto the search space S of the given problem instance: To each feasible path $w \in W$, there corresponds a feasible solution $x = \phi(w) \in S$, and to each feasible solution $x \in S$ corresponds at least one feasible path in W such that $\phi(w) = x$. Then, the graph G, endowed with the function ϕ, is called a construction graph for the problem instance (S, f).

2.4 Convergence

In order to study ACO algorithms theoretically, a way is to use Markov chain theory. In order to do this, it needs to identify the Markov chain to work. If we denote by $\hat{x}(t-1)$ the best solution of ACO until time $t-1$ we can define $X_t = (\tau(t), \hat{x}(t-1))$ which can be proved to be a Markov chain.

Let $X^i(t)$ be the stochastic process modeling the configuration assumed by the i-th of the $|A|$ ants of the colony. Let X^* denote the optimal set for the function f to be maximized (or minimized).

Definition 2.4.1. *Given a function $f : S \to \mathbb{R}$ to be maximized (or minimized), $X(t)$ be the stochastic process and X^* the optimal set for the function f we define*

the failure probability as following:

$$p(t) = \mathbb{P}(X(t) \notin X^*).$$

By definition, the failure probability is a non-increasing function of the number of iterations. The effectiveness of the algorithm can then be translated into the convergence to zero of the failure probability. Although this kind of convergence requirement is very basic, it is not always fulfilled. Therefore, theoretical studies on this kind of convergence are well motivated. Knowing in advance that the failure probability is decreasing to zero makes the user confident that waiting long enough time, there are very good chances that the algorithm will solve the problem.

We can distinguish two kinds of convergence:

convergence in value: when it holds

$$p_v(t) := P\left(\bigcap_{i=1}^{A} \left\{X^i(t) \notin X^*\right\}\right) \to 0;$$

convergence in model: if for some $x^* \in X^*$ we have

$$p_m(x^*, t) := P\left(\bigcap_{i=1}^{A} \left\{X^i(t) = x^*\right\}\right) \to 1\,.$$

The convergence in value is important. This property tells us something about the way in which the algorithm is exploring the configuration space X. It is strongly connected with the strict positivity of the conditional probability to visit at the end of any iteration one point of X^* given that we are not currently in X^*. However, the convergence in model is stronger than the one in value. In the former, the probabilistic model itself evolves towards one that generates only optimal solutions. Not all algorithms converging in value are also converging in model. For example, this is the case for the algorithm that explores the configuration space in a uniform and independent way, known as Random Search (RS). In fact, $p_v(t) = \left(1 - \frac{|X^*|}{|X|}\right)^{|A|t} \to 0$, while it holds $p_m(x^*, t) = \left(\frac{1}{|X|}\right)^{|A|}$.

In the following, we will only cope with the case where the ACO algorithm does not use the visibility matrix η_{ij}. In this case, the configuration of each ant of the colony is built by a path over the construction graph [19]:

$$p(j|s^p) = \frac{\tau_{ij}^{\alpha}}{\sum_{j \in N(s^p)} \tau_{ij}^{\alpha}}, \quad \forall\, j \in N(s^p) \tag{2.1}$$

where (i, j) is the arc by which we continue the partial path s^p, and τ_{ij} is its current pheromone value.

By simply imposing some constraints on the pheromone matrix, it is possible to have an ACO algorithm that converges in value. In fact, for an ACO algorithm with $0 < \tau_{min} \leq \tau_{ij} \leq \tau_{max}$, given any $\epsilon > 0$, we have that, for t large enough, it holds

$$p_v(t) \leq \epsilon\,.$$

that is, by definition,

$$\lim_{t \to \infty} p_v(t) = 0\,.$$

Because of the bounds on the pheromone matrix, every choice made by the rule in Eq. (2.1) has a probability larger or equal to

$$p_{min} = \frac{\tau_{min}^{\alpha}}{(D_{max} - 1)\tau_{max}^{\alpha} + \tau_{min}^{\alpha}} > 0\,,$$

where D_{max}, is the maximum value of the degree of the nodes of the construction graph. It then follows that any configuration of the whole space X, including an optimal one x^*, can be visited with a probability larger or equal $\hat{p} = (p_{min})^{L_{max}} > 0$, where $L_{max} = \max_{x \in X} L(x)$, and $L(x)$ is the length of the path by which we have built the configuration x. From this, it follows that

$$p_v(t) \leq (1 - \hat{p})^{|A|t}\,, \tag{2.2}$$

i.e. there is convergence in value.

Different kinds of ACO algorithms are such that suitable bounds on the pheromone matrix values hold. Among them, the Max Min Ant System (MMAS),

where lower and upper bounds on the pheromone matrix values are imposed explicitly when updating recursively them along iterations

$$\tau_{ij}(t+1) = \min\{\tau_{max}, \max\{(1-\rho)\tau_{ij}(t) + \rho\, 1\{(i,j) \in x_b\}/L(\bar{x}), \tau_{min}\}\},$$

where $0 < \rho < 1$, and $1(\cdot)$ is the indicator function [39],[38]. We notice that the pheromone is reinforced only on arcs belonging to one configuration x_b, e.g. the best one (w.r.t the objective function f) that we have visited so far, i.e. $x_b = \arg_{i=1,\ldots,A,\ k=1,\ldots,t} \max f(x^i(k))$. We could use another kind of update where no bound on the pheromone are explicitly imposed:

$$\tau_{ij}(t+1) = (1-\rho)\tau_{ij}(t) + \rho\, 1\{(i,j) \in x_b\}/L(x_b)\,.$$

In this case, after t iterations, the pheromone values are bounded from above by

$$(1-\rho)^t \tau_0 + \rho \sum_{i=1}^{t} (1-\rho)^{t-i}/L_{min}\,.$$

The above quantity converges from below to $1/L_{min}$. In this case, we do not have in general any guarantee that also a lower bound holds. However, not having a lower bound sometimes can be a positive condition. In fact, as seen above, when we have both lower and upper bounds, convergence in value holds. When a lower bound holds, however, we cannot have convergence in model because at any iteration we have a lower bound for the conditional probability of reaching any configuration given any other. We will see now that if we impose the weaker condition that the lower bound of the pheromone matrices at time t, $\tau_{min}(t)$ (it always exists since there is a finite number of edges) goes to zero slowly enough, convergence in value still holds. This is stated by the following theorem [13].

Theorem 2.4.2. *Given an ACO algorithm with pheromone values having constant upper bound τ_{max} and lower bound $\tau_{min}(t)$*

$$\tau_{min}(t) = \Omega\left(\frac{1}{\ln(t+1)}\right),$$

then we have

$$p_v(t) \to 0\,.$$

In fact, similarly to Eq. (2.2), one can prove that

$$p_v(t) \leq \prod_{k=1}^{t} \left(1 - (p_{min}(k))^{L_{max}}\right)^A,$$

where

$$p_{min}(t) \geq \frac{\tau_{min}^{\alpha}(t)}{(D_{max}-1)\tau_{max}^{\alpha} + \tau_{min}^{\alpha}(t)} \geq \frac{\tau_{min}^{\alpha}(t)}{D_{max}\tau_{max}^{\alpha}}.$$

By combining the two inequalities of above, we have

$$p_v(t) \leq \prod_{k=1}^{t} \left(1 - \left(\frac{\tau_{min}^{\alpha}(t)}{D_{max}\tau_{max}^{\alpha}}\right)^{L_{max}}\right)^A = \prod_{k=1}^{t} \left(1 - K\left(\tau_{min}(t)\right)^{\alpha L_{max}}\right)^A.$$

By the following lemma, it is sufficient to show that

$$\sum_{t=1}^{\infty} \left(\tau_{min}(t)\right)^{\alpha L_{max}} \to +\infty.$$

Lemma 2.4.3. *Let $\{a_n\}_{n \in I\!N}$ be a sequence of real numbers converging to zero such that $0 \leq a_n < 1$, $\forall\, n$. Then, it follows that $\sum_n a_n \to +\infty \Rightarrow \prod_n (1-a_n)^k \to 0$, $\forall k \geq 1$.*

Since $\tau_{min}(t) = \Omega\left(\frac{1}{\ln(t+1)}\right)$, then the terms of the series $\sum_{t=1}^{\infty} \left(\tau_{min}(t)\right)^{\alpha L_{max}}$ are eventually bounded from below by $\left(\frac{C}{\ln(t+1)}\right)^{\alpha L_{max}}$. Then, this series is diverging because $\sum_{t=1}^{\infty} \left(\frac{C}{\ln(t+1)}\right)^{\alpha L_{max}}$ is infinite. Finally, we have

$$\lim_{t \to \infty} p_v(t) = 0.$$

In order to obtain ACO convergence avoiding to bound the pheromone matrix, it needs to have pheromone evaporation depending to the time-dependent evaporation factor (tdev). In particular $\rho = \rho(t)$ has to converge to zero "slowly". This kind of problem is studied in [20], for a ACO variant called GBAS (Graph based Ant System). In [20] the pheromone update rule is as follows:

$$\tau_{kl}(t+1) = \begin{cases} (1-\rho(t))\tau_{kl}(t) + \frac{\rho(t)}{L(\hat{x}(t))} & \text{if } (k,l) \in \hat{x}(t)), \\ (1-\rho(t))\tau_{kl}(t) & \text{otherwise.} \end{cases}$$

where $0 < \rho(t) < 1$ $(t = 1, 2, \dots)$ and $\hat{x}(t)$ is the best so far solution until time t.

Theorem 2.4.4. *Let, in the algorithm GBAS-tdev,*

$$\rho(t) \leq 1 - \frac{\log(t)}{\log(t+1)} \qquad \textit{eventually,}$$

and it holds

$$\sum_{t=1}^{\infty} \rho(t) = \infty.$$

Then, the algorithm converges in model.

Several convergence results of different types have been shown for different ACO variants. Some can be found in [19], [22], [36].

Chapter 3

The restart and some new theoretical results

Trying to find the optimal solution of an instance of a $\mathcal{NP}$-hard problem by using a meta-heuristic algorithms (MHA) may be unsatisfactory. This is typical when the instance dimension is high. In fact, also in the case when the failure probability converges theoretically to zero, the expected time to find the optimal solution can be very large. Hence, in practice the failure probability remains high. Based on the stochastic nature of a MHA, a very simple approach has been proposed known as restart. This consists of the execution of a certain number of independent runs of the underlying algorithm. Each independent run has the same temporal length T. The different restarts are obtained by changing typically the starting point and or some other characteristics of the underlying algorithm. The restart has been widely applied with success in combination with several algorithms to solve instances of different $\mathcal{NP}$-hard problems. For example, in [35],[37],[38] a restart for ACO algorithms is proposed to solve the TSP. In [15] and [16], a method is proposed to apply the restart at some intermediate stages to generate more occurrences of a rare event. When applying the restart, the failure probability decreases to zero geometrically with the number of restarts. Therefore, one is interested to use the highest number of restarts that is compatible with the computational resource available. Having a finite computation time t available, the number of restarts can

be increased by decreasing T. On the other side, the base of the geometric sequence of above is the failure probability p_T of the underlying algorithm at the restart time T. Hence, we wish to execute each restart for a sufficiently long T in order to make p_T small. Therefore, it is natural that the choice of T optimized the trade off between those to requirements. Usually, the criterion to find a value of T is based on the presence of negligible fluctuation in the solution values along time.

Despite the huge literature on the application of the restart, a few theoretical works on this approach are available ([42]). In this thesis, we study theoretically the restart. In this chapter, we first illustrate two examples of the application of the restart. This is done to provide an intuitive idea about the conditions when applying the restart is convenient in terms of the failure probability. After that, we proceed by studing the expected value of the time needed to find the optimal solution. We give sufficient conditions in terms of $p(t)$ (cf. [6]) ensuring that the above expected value is lower than when applying the underlying algorithm. Moreover, we cope with the problem of finding an optimal value for T. This is done by adopting the following criterion. Given the finite computation time t available, as criterion to find an optimal value for the restart time T, we choose to minimize the restart failure probability at the time t. As it will be clear later, this is strongly related to the problem of minimizing the function $p(T)^{\frac{1}{T}}$. We provide necessary and sufficient conditions in terms of $p(t)$ so that such optimal value exists. We illustrate some numerical example when the restart is convenient, assuming to know the optimal restart time. Finally, we describe a related theoretical work.

Let $X(t)$ be a stochastic process corresponding to the state of underlying algorithm at the time (iteration) t and let f be the function to maximize (minimize). We denote by f_m the optimal value of the function f. Let $Y(t)$ the value of the function f evaluated at the state at the time t of the process corresponding to the "best so far solution", i.e. $Y(t) = \max\{f(X(i)) : \quad i \in [1,t] \cap \mathbb{N}\}$. In the case the function f should be minimized, the max is replaced by min.

3.1 Two examples

We describe now two examples. In the first example, due to the sub-exponential decay to zero of the failure probability $p(t)$ of the underlying algorithm, the restart is successful. Instead, in the second example the underlying failure probability goes to zero more than exponentially. Then, the restart is not successful.

Example 3.1.1. If $p(t) = \frac{c}{t^\alpha}$, where $\alpha > 0$ and $0 < c < 1$, we are going to show that, for any t sufficiently large, there exists $T < t$ such that the failure probability $p(t)$ of the algorithm after t iterations is larger than $p(T)^{\lfloor \frac{t}{T} \rfloor}$, that is the failure probability of the algorithm restarted $\lfloor \frac{t}{T} \rfloor$ times with restart time equal to T. To this aim, it will be sufficient to have that

$$p(t) > p(T)^{(\frac{t}{T}-1)}.$$

To show this, we compute the derivate of $p(T)^{\frac{1}{T}}$:

$$\begin{aligned} \frac{d}{dT}\left(p(T)^{\frac{1}{T}}\right) &= \left(\frac{c}{T^\alpha}\right)^{\frac{1}{T}} \left[-\frac{1}{T^2}\ln\left(\frac{c}{T^\alpha}\right) + \frac{1}{T}\left(-\alpha\frac{1}{T}\right)\right] \\ &= -\left(\frac{c}{T^\alpha}\right)^{\frac{1}{T}} \frac{1}{T^2}\left(\ln\left(\frac{c}{T^\alpha}\right) + \alpha\right). \end{aligned}$$

This derivate vanishes when $\ln\left(\frac{c}{T^\alpha}\right) + \alpha = 0$ and hence for $T = ec^{\frac{1}{\alpha}}$. Of course we will choose a restart time equal to $\bar{T} = \left\lceil ec^{\frac{1}{\alpha}} \right\rceil = \beta ec^{\frac{1}{\alpha}}$, where $\beta \geq 1$. After having calculated $p(\bar{T}) = \frac{1}{(\beta e)^\alpha}$, we consider the quantity

$$p(\bar{T})^{t/\bar{T}-1} = \left(\frac{1}{(\beta e)^\alpha}\right)^{t/(\beta e \sqrt[\alpha]{c})-1} < \frac{c}{t^\alpha} = p(t),$$

where the last inequality is true for t sufficiently large. Therefore, in this case there is an advantage to consider the process with restart.

Example 3.1.2. Here, we choose $p(t) = c^{t^\alpha}$, where $c < 1$ and $\alpha > 1$. In this case, it holds

$$p(T)^{\lfloor \frac{t}{T} \rfloor} \geq p(T)^{\frac{t}{T}} = c^{T^{\alpha-1}t} \geq c^{t^\alpha} = p(t) \quad \forall\, T \leq t.$$

Therefore, in this case the choice $T = t$ minimizes the restart failure probability, which unfortunatly is then equal to the underlying failure probability. Hence there is no advantage to use the restart.

3.2 Restart expected value

We will now study the restart in terms of the expected value of the first time τ_T we find the optimal solution. Let τ_∞ denote the analogous time for the underlying algorithm i.e. $\tau_\infty = \inf\{t : \quad Y(t) = f_m\}$. The expected value of τ_∞ (τ_T) can be expressed in terms of the survival function

$$E[\tau_\infty] = \sum_{k=0}^{\infty} P(\tau_\infty > k). \tag{3.1}$$

We notice that the event $\{\tau_\infty > t\}$ is equal to the event $\{Y(t) \neq f_m\}$ for $t \geq 1$. Then, the equation (3.1) becomes

$$E[\tau_\infty] = \sum_{k=0}^{\infty} p(k),$$

where $p(k) = P(Y(k) \neq f_m)$ and we defined $p(0) = 1$. The same reasoning is valid for τ_T so that $\mathbb{P}(\tau_T > t)$ can be expressed as

$$p(T)^{\lfloor \frac{t}{T} \rfloor} p\left(t - \left\lfloor \frac{t}{T} \right\rfloor T\right).$$

We notice that the power term appearing above gives the probability that the restart did not find the optimal value in the first $\left\lfloor \frac{t}{T} \right\rfloor$ restarts already completed, and the other term corresponds to the current restart. Moreover we can write $t = mT + r$ where $m = m(t) = \left\lfloor \frac{t}{T} \right\rfloor$ and $r = r(t) \in \{0, 1, \ldots, T-1\}$. Hence, we can write

$$E[\tau_T] = \sum_{t=1}^{\infty} p_T(t),$$

where

$$p_T(t) = p(T)^m p\,(r) \tag{3.2}$$

is the restart failure probability at the time t. In general, we notice that a necessary condition for $E[\tau_\infty]$ to be finite is that the non increasing sequence $p(t)$ goes to zero as t diverges. Instead, to have that the expected value of τ_T is finite is just enough that $\exists T$ such that $p_T < 1$. In fact, we have

$$E[\tau_T] \leq \sum_{t=1}^{\infty} p(T)^{\frac{t-1}{T}},$$

and the series on the r.h.s. is convergent. Here, there is an example of a case where $p(t)$ is not going to zero as t goes to infinity.

Example 3.2.1. The algorithm involved is known as (1+1)EA with single flip proposed for maximizing pseudo-boolean functions [23]. This algorithm, at each iteration, first inverts the bit of a single component, randomly chosen among the n of the binary string. Then, the proposed flip is accepted if it corresponds to an increase of the function f. For this algorithm, the probability $p(t)$ has a positive lower-bound in the case when there are more than one local maximum of the objective function f. In fact, the algorithm will converge to one of the local maxima because it is of "hill climbing" type. Moreover, the candidate solution belongs to the hamming neighborhood of size one of the current one, Therefore, if the initial point belongs to the "bacine of attraction" R of a certain local maximum different from the global one, the algorithm will converge to that local maximum with probability one. Therefore, in this case, it holds

$$p(k) \geq P(X(0) \in R).$$

If $P(X(0) \in R) > 0$ we have that $p(k)$ does not go to zero.

A sufficient condition to have $E[\tau_T] < E[\tau_\infty]$ is obviuosly

$$p(T)^m p\,(r) < p(t), \tag{3.3}$$

for any $t > T$. However, the above condition reduces to identity for $t \leq T$. In the following, we provide some sufficient conditions for Eq. (3.3) when $t > T$, and therefore also for $E[\tau_T] < E[\tau_\infty]$.

Proposition 3.2.2. *If it exists $T > 0$ such that for any $t > 0$, it holds $\frac{p(t)}{p(t-1)} > p(T)^{\frac{1}{T}}$, it follows*

$$E[\tau_T] < E[\tau_\infty].$$

Proof. For any $t > T$ we can write $t = mT+r$, so that using the convention $p(0) = 1$, we get

$$p(t) = p(mT + r) = \prod_{i=1}^{r} \frac{p(i)}{p(i-1)} \cdot \prod_{i=r+1}^{mT+r} \frac{p(i)}{p(i-1)} > p(r)p(T)^m,$$

that is (3.3). □

Proposition 3.2.3. *If it exists $T > 0$ for which it holds*

$$\frac{p(t)}{p(t-1)} > \frac{p(r)}{p(r-1)} \qquad \forall t > T, r > 0 \tag{3.4}$$

$$\frac{p(mT+1)}{p(mT)} \geq p(1) \quad \forall m \in \mathbb{N}, \tag{3.5}$$

then we have

$$E[\tau_T] < E[\tau_\infty].$$

Proof. To obtain the thesis it is sufficient to show that it holds

$$p(mT + r) > p(T)^m p(r) \qquad \forall m \in \mathbb{N} \setminus \{0\}, \forall r \in \{0, 1, \ldots, T-1\}. \tag{3.6}$$

The last inequality is true. In fact, assuming $p(0) = 1$, we can write

$$p(mT + r) = \prod_{i=1}^{mT+r} \frac{p(i)}{p(i-1)} = \prod_{i=1}^{T} \frac{p(i)}{p(i-1)} \cdots \prod_{i=(m-1)T+1}^{mT} \frac{p(i)}{p(i-1)} \cdot \prod_{i=mT+1}^{mT+r} \frac{p(i)}{p(i-1)}. \tag{3.7}$$

Using the hypothesis, we have

$$\prod_{i=nT+1}^{(n+1)T} \frac{p(i)}{p(i-1)} > \prod_{i=1}^{T} \frac{p(i)}{p(i-1)} = p(T),$$

where $n = 1, 2, \ldots, m-1$. The similar result holds for the last product of (3.7). In fact, we have

$$\prod_{i=mT+1}^{mT+r} \frac{p(i)}{p(i-1)} > \prod_{i=1}^{r} \frac{p(i)}{p(i-1)} = p(r).$$

Hence, since (3.7) we obtain (3.6), that implies the thesis. □

3.3 Restart failure probability

From the equation (3.2), we see that the failure probability of the restart algorithm at the finite computational time available t is geometrically decreasing to zero with base equal to $p(T)^{\frac{1}{T}}$. Therefore, a criterion for choosing an optimal value for the restart time T could consist of minimizing the function $g(k) := p(k)^{\frac{1}{k}}$, where $k \in \mathbb{N}$. Moreover, the existence of a finite number of local minima of the function g is strongly related to the successfull application of the restart, as shown in the next three propositions. In order to prove the next result, we introduce the following definition. We notice that it assumes $p(t) > 0$ for every t in order to give a sense to what follows.

Definition 3.3.1. *We say that the restart is convenient if* $\exists T \in \mathbb{N}$ *such that* $p_T(t)/p(t) = o(1)$.

Proposition 3.3.2. *The Definition 3.3.1 implies that*

$$\exists T \in \mathbb{N} \quad \text{and } a < 1 \quad \text{such that} \quad \frac{p(T)^{\frac{t}{T}}}{p(t)} < a^t \quad \text{for any } t \text{ large enough.} \tag{3.8}$$

Proof. The hypothesis implies that

$$\exists T \in \mathbb{N} \text{ and } a < 1 \text{ such that} \quad p_T(t)/p(t) = o(a^t). \tag{3.9}$$

By using the expression of $p_T(t)$ in (3.2), we can write $p_T(t) = p(T)^{\frac{t}{T}} \cdot f(t)$, where

$$f(t) = p(r) \cdot p(T)^{-\frac{r}{T}},$$

and $r = t - \left\lfloor \frac{t}{T} \right\rfloor T$. The (3.9) can be written as $\frac{p(T)^{\frac{t}{T}}}{p(t)} \cdot f(t) < c \cdot a^t$ eventually. Since the function $f(t)$ is bounded from below and above, the thesis easily follows. □

Proposition 3.3.3. *The condition (3.8) is equivalent to*

$$\lim_{t \to \infty} \inf_{s > t} g(s) > \inf_t g(t). \tag{3.10}$$

Proof. (3.8) $\Rightarrow$(3.10)
From (3.8), it follows that

$$g(t) = p(t)^{\frac{1}{t}} > \frac{1}{a} \cdot p(T)^{\frac{1}{T}} = \frac{1}{a} \cdot g(T) > \inf_t g(t).$$

(3.8) $\Leftarrow$ (3.10)
From (3.10), we have $g(t) > c$ eventually, where $c = \beta \cdot \inf_t g(t)$, for a suitable $\beta > 1$. Condition (3.10) also implies that $\min_t g(t)$ exists and can be reached only for a finite number of values of t. Hence, we obtain

$$p(t)^{\frac{1}{t}} = g(t) > \beta \cdot \min_t g(t),$$

for t large enough and a suitable $\beta > 1$, which implies that

$$\frac{1}{\beta^t} > \frac{(\min_t g(t))^t}{p(t)}.$$

The proof is completed by setting $a = \frac{1}{\beta}$ and $T = \min(\arg\min_t g(t))$. □

We notice that for any choice of T such that $g(T) < \lim_{t\to\infty} \inf_{s>t} g(s)$, we could use the same scheme applied in the second implication of the above proposition, to get that the restart is convenient for such T.

We have seen that a necessary condition for the restart to be convenient accordingly to the definition 3.3.1 is the fact that $g(t)$ admits a finite number of global minima. Moreover, as said before, a criterion to find an optimal value for the restart time could be based on the minimization of the function g.

Proposition 3.3.4. *1. Condition (3.10) holds under the hypotheses*

- **B1)** *There exist $a \in (0,1)$ and $t_a \in \mathbb{N}$ such that $\forall t \geq t_a$ we have $p(t) > a^t$.*
- **B2)** *It exists $i \in \mathbb{N}$ such that $p(i)^{\frac{1}{i}} < a$.*

*2. Condition (3.10) holds under the hypotheses **B2)** and*

- **C1)** *There exist $a \in (0,1)$ and $t_a \in \mathbb{N}$ such that $\forall t \geq t_a$ we have $\frac{p(t+1)}{p(t)} > a$.*

3. *If* $\forall a \in (0,1)$ *it exists* $t_a \in \mathbb{N}$ *such that* $\forall t \geq t_a$ *we have* $\dfrac{p(t+1)}{p(t)} > a$*, then, it follows* $\lim_{t\to\infty} g(t) = 1$*. Moreover, condition (3.10) holds.*

Proof. **1**: **B1-B2** $\Rightarrow$ **(3.10)**
Since **B1** holds, there exist $a \in (0,1)$ and $t_a \in \mathbb{N}$ such that $\forall t \geq t_a$ we have $p(t) > a^t$. Moreover, because of **B2**, it exists $i \in \mathbb{N}$ such that $g(i) < a$. Therefore, $g(t)$ admits minimum which is equal to $\min_{t\in\mathbb{N}\cap[1,t_a-1]} g(t)$. Furthermore the minimum value of $g(t)$ is reached at most in $t_a - 1$ points.

2: **C1-B2** $\Rightarrow$ **(3.10)**
Since **C1** holds, there exist $a \in (0,1)$ and $t_a \in \mathbb{N}$ such that $\forall t \geq t_a$ we have $\frac{p(t+1)}{p(t)} > a$. Moreover, we remark that condition **B2** also holds when replacing a by suitable number c such that $c < a$. For $t \geq t_a$ we have $p(t+1) > ap(t)$, which implies $p(t) > a^{t-t_a}p(t_a) = ka^t$. Since $a > c$, the exponential sequence ka^t will be smaller than c^t after a certain time t_c. For $t \geq max(t_a, t_c)$ condition **B1** is fulfilled. By using the implication **B1-B2** $\Rightarrow$ **(3.10)**, we get thesis.

3 By the hypothesis, it follows that , for any b such that $b < 1$, we have

$$\exists t_b > 0 \quad \text{such that} \quad \forall t \geq t_b \quad p(t+1) > bp(t). \tag{3.11}$$

By applying (3.11) recursively, we get

$$p(t) > b^{t-t_b}p(t_b) = cb^t \quad \forall t \geq t_b. \tag{3.12}$$

The above inequality is equivalent to

$$p_t > a^t \quad \forall t \geq t_a.$$

Therefore, for $t \geq t_a$ we have $p(t)^{\frac{1}{t}} = g(t) > a$. Hence, it holds

$$\forall \varepsilon > 0 \quad \exists t_\varepsilon > 0 \quad \text{such that} \quad 1 - \varepsilon < p(t)^{1/t} < 1 \quad \forall t > t_\varepsilon,$$

which is the limit definition. Let t_1 be such that $p(t_1) < 1$ so that we have $g(t_1) < 1$. By choosing ε such that $g(t_1) < 1 - \varepsilon$, by the limit condition we have $g(t) > g(t_1)$

for $t > t_\varepsilon$, with obviously $t_1 < t_\varepsilon$. Therefore, as in the proof of **1**, the sequence $g(t)$ admits a minimum value equal to $\min_{t\in[1,t_\varepsilon]} g(t)$. □

3.4 Some numerical simulations

We will illustrate now a simulation study where restarting the MMAS ACO algorithm is successful. We want to maximize the following pseudo-Boolean function

$$f(x) = \left| \sum_{i=1}^{N} x_i - \frac{N-1}{2} \right| , \tag{3.13}$$

with respect to all binary strings of length N. In Fig. 1, the function considered is plotted as function of the number of 1s in the case $N = 20$.

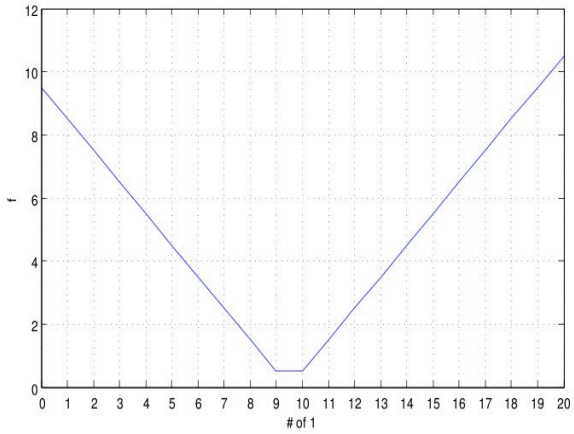

Figure 1. The plot of the considered pseudo-Boolean function versus the number of 1s of the binary string.

This function has two local maxima but only one of them is a global maximum. The presence of the pheromone bounds τ_{min} and τ_{max} ensures convergence in value of the MMAS algorithm. However, if the algorithm visits a configuration with few 1s it takes a very long time in average to move towards the global maximum. Therefore, we expect that in this case the restart will be successfull. By using as construction graph, the chain graph [19], and by setting the initial values of the pheromone matrix

equal to 0.5, the pheromone matrix coincides with the matrix of the probability of transitions [23]. The initial string was chosen uniformly in the state space $\{0,1\}^N$. The algoritms were implemented in Matlab. The values of the MMAS parameters were $\rho = 0,01$, $\tau_{min} = 0.1$, $\tau_{max} = 0.9$. We used one thousands runs of the algorithm each with 20.000 iterations. Based on these simulations, we estimated the failure probability. In Fig. 2, the estimation $\hat{g}(t)$ of $g(t)$ is plotted versus the iteration numbers t. A minimum of this function is clearly visible and located at iteration 2876.

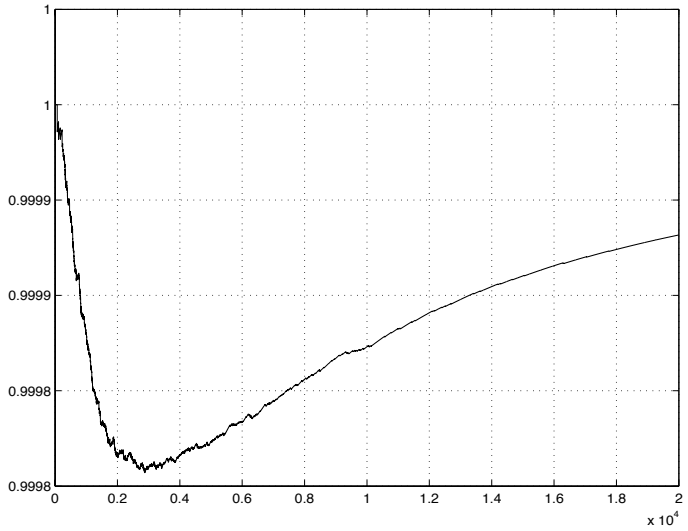

Figure 2. The estimated function $\hat{g}(t)$.

Finally, in Fig. 3 we show the estimated failure probability $\hat{p}(t)$ for the MMAS algorithm with chain graph to maximize the pseudo-Boolean function of Fig. 1 (continuous line). On the same figure, the estimated failure probability of the restart algorithm with $T = 2876$ is plotted (dashed line). As seen, there is a clear advantage to use the restart MMAS algorithm when compared to the stardard MMAS.

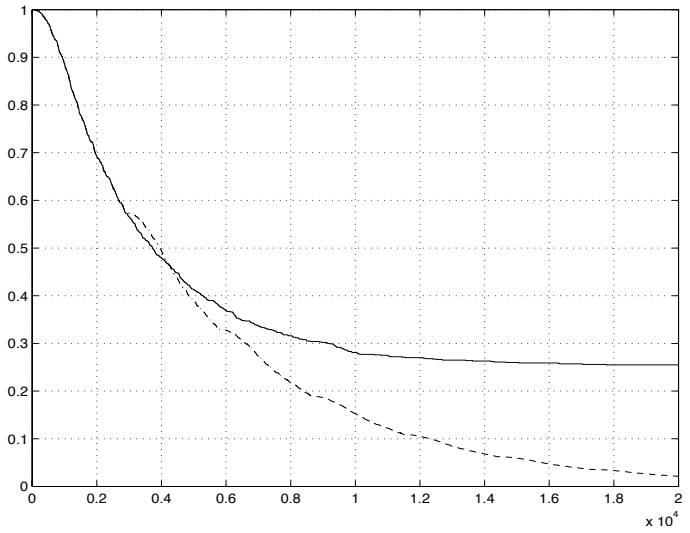

Figure 3. The estimated failure probability for the standard MMAS (continuous line) and the restarted MMAS (dashed line).

3.5 Related work

In [42] theoretical conditions to apply the restart are studied, though for a simple problem: when do I click my browser's reload button if a web page takes too long to download? If the random variable t denote the completion time of a job, the authors ask if one would restart at time T, when it holds

$$\mathbb{E}[t] < \mathbb{E}[t - \tau \mid t > T].$$

Let $f(s)$ be probability density function of t and $F(s)$ its distribution. The authors assume that $F(s)$ is a continuous probability distribution function defined over $[0, \infty)$, such that $F(s) > 0$ if $s > 0$. After assuming c time units for each restart, the authors introduce the random variable t_T to denote the completion time when an unbounded number of restarts is allowed. Let $f_T(s)$ and $F_T(s)$ be the density and the distribution of t_T. It is possible to compute the distribution for the restart

$$F_T(s) = \begin{cases} 1 - (1 - F(T))^k (1 - F(s - k(T + c))) & \text{if } k(T + c) \leq s < k(T + c) + T, \\ 1 - (1 - F(T))^{k+1} & \text{if } k(T + c) + \tau \leq s < (k + 1)(T + c), \end{cases}$$

and density

$$f_T(s) = \begin{cases} (1 - F(T))^k f(s - k(T + c)) & \text{if } k(T + c) \leq t < k(T + c) + T, \\ 0 & \text{if } k(T + c) + \tau \leq s < (k + 1)(T + c). \end{cases}$$

The expected value of t_T^n is computed in the next result

Theorem 3.5.1. *The moments* $\mathbb{E}[t_T^n] = \int_0^\infty s^n s f_T(s) ds$, $n = 1, 2, \ldots$, *of the completion time with unbounded number of restarts, restart interval length* $T > 0$, *and time* c *consumed by a restart, can be expressed as:*

$$\mathbb{E}[t_T^n] = \frac{M_n(T)}{F(T)} + \frac{1 - F(T)}{F(T)} \sum_{l=0}^{n-1} \binom{n}{l} (T + c)^{n-l} \mathbb{E}[t_T^l],$$

where $\mathbb{E}[t_T^0] = 1$.

The next result describes a condition to obtain the optimal time to apply the restart.

Theorem 3.5.2. *The optimal restart time* $T^* > 0$ *that minimized the expected completion time* $\mathbb{E}[t_T]$ *is such that:*

$$\frac{1 - F(T^*)}{f(T^*)} \mathbb{E}[t_{T^*}] + c.$$

That is, if $c = 0$, *the inverse of the hazard rate at* T^* *equals the expected completion time under unbounded restarts.*

Chapter 4

A new restart procedure and its convergence

As seen in the last chapter, the restart procedure could be optimized by choosing a value σ for the restart time that minimizes the function $g(t) := p(t)^{\frac{1}{t}}$, where $p(t)$ is the failure probability of the underlying algorithm. This has also been illustrated by a numerical example. However, this approach cannot be used in practice. This is because the failure probability could be computed theoretically or estimated from a suitable large simulation sample only if we know the combinatorial optimization problems (COP) solution. In this chapter, we describe a new iterative procedure to optimize the restart which does not use such a knowledge. This property together with the convergence's results, allow us to apply this procedure in practice. Every iteration, the procedure provides an estimation of $\hat{\sigma}$ which converges to σ as the number of steps of the procedure grows with probability one. Moreover, we show some numerical results where the proposed restart procedure (RP) is applied to ACO for solving several TSP instances.

4.1 The procedure

The RP starts by executing r_0 replications of the underlying algorithm until time T_0. Then, at the end of iteration k, based on the criterion described later in this section,

the RP either increases the number of replications from r_k to r_{k+1} by executing $r_{k+1} - r_k$ replications of the underlying algorithm until time T_k, or it continues the execution of the r_k replications until time T_{k+1}. Let $Y_i(t)$ be the value of the best solution found by i-th replication until time t i.e. $Y_i(t) = \min(f(X_i(s)), s = 1, ..., t)$, where f is the function to minimize. Each $Y_i(t)$ is an independent realization of the same process. We can always think at the RP in the following way. Given the infinite matrix $\mathbf{Y}$ with generic element $Y_i(j)$ where $i, j = 1, 2, \ldots,$ at iteration k, a realization of the RP consists an increasing sequence $\{Y_{A_k}\}_{k\in\mathbb{N}}$ of finite matrices, where $A_k := \{(i,t) : i = 1, \ldots, r_k \quad t = 1, \ldots, T_k\}$. The matrix Y_{A_k} corresponds to the first r_k rows and T_k columns of $\mathbf{Y}$. Let $\tilde{Y}_k$ denote the minimum value of this matrix at the end of iteration k: $\tilde{Y}_k = \min Y_{A_k} = \min\limits_{(i,t)\in A_k} Y_i(t)$. We estimate the failure probability sequence by means of the empirical frequency

$$\hat{p}_k(t) = \begin{cases} \frac{1}{r_k} \sum_{i=1}^{r_k} 1_{\{Y_i(t) > \tilde{Y}_k\}} & t = 1, \ldots, T_k, \\ 0 & \text{otherwise.} \end{cases}$$

We denote by $\hat{\sigma}_k$ the smallest value of the time where the minimum of $\hat{p}_k(t)^{\frac{1}{t}}$ is reached in the interval $[1, T_k] \cap \mathbb{N}$. Let λ be a number in $(0, 1)$. If $\hat{\sigma}_k < \lambda \cdot T_k$, then the RP increases the number of replications by means of the rule such that $r_{k+1} := f_r(r_k) > r_k$. Otherwise, the RP increases the restart time accordingly to $T_{k+1} := f_T(T_k) > T_k$. We assume that both f_r and f_T are functions such that for any fixed $x > 0$ it holds $f_r^{(k)}(x), f_T^{(k)}(x) \to \infty$, where the power k denotes the consecutive application of the function k times. These are the two basic requirements to choose f_r and f_T. Apart from this, there is complete freedom to choose the functions. An example of these functions is provided in section 4.3. Therefore, the recursive formula for (r_k, T_k) is

$$(r_{k+1}, T_{k+1}) = \begin{cases} (f_r(r_k), T_k) & \text{if } \hat{\sigma}_k < \lambda \cdot T_k, \\ (r_k, f_T(T_k)) & \text{otherwise.} \end{cases}$$

Below, there is the pseudo code for RP:

$r = r_0$;

$T = T_0$;
for replication $i = 1, 2, \ldots, r$ **do**
    execute algorithm $\mathcal{A}$ until time T;
    save $\mathcal{A}_i(T)$;
end for
save Y_{A_0};
compute $\hat{\sigma}_0$ from Y_{A_0};
for iteration $k = 1, 2, \ldots$ **do**
    if $\hat{\sigma}_{k-1} > \lambda \cdot T_{k-1}$ **then**
        $T_k = f_T(T_{k-1})$;
        $r_k = r_{k-1}$;
        for replication $i = 1, 2, \ldots, r_k$ **do**
            continue the execution of $\mathcal{A}_i$ until T_k;
            save $\mathcal{A}_i(T_k)$;
        end for
    else then
        $r_k = f_r(r_{k-1})$;
        $T_k = T_{k-1}$;
        for replication $i = r_{k-1} + 1, r_{k-1} + 2, \ldots, r_k$ **do**
            execute $\mathcal{A}_i$ until T_k;
            save $\mathcal{A}_i(T_k)$;
        end for
    end if
    save Y_{A_k};
    compute $\hat{\sigma}_k$ from Y_{A_k};
end for

4.2 RP convergence

We denote by f_m the value of the solution of the optimization problem. Moreover, for simplicity, we introduce the function $g_k(t) := \hat{p}_k(t)^{\frac{1}{t}}$, whose domain is $\mathbb{N} \cap [1, T_k]$. In order to derive the following results, we assume that

1. $p(t) \underset{t\to\infty}{\longrightarrow} 0$,
2. $g(t)$ admits only one point of minimum t_m and that it is strictly monotone decreasing for $t \leq t_m$,
3. $p(1) < 1$.

Remark 4.2.1. We notice that, by the assumption on f_r and f_T ,the probability that both sequences r_k and T_k are bounded is zero.

Lemma 4.2.2. *Let $p(t)$ be as above. Let (r_k, T_k) be the sequence of random variables which describes RP. Then*

1. $\mathbb{P}(r_k \to \infty) = 1$.
2. *It holds*
$$\mathbb{P}\left(\left\{\exists k : \quad \tilde{Y}_k = f_m\right\}\right) = 1. \tag{4.1}$$

Proof. **1** If $\mathbb{P}(r_k \to \infty) < 1$, then with positive probability the following three conditions hold for a certain positive integer r:

i) $r_k = r$ eventually;

ii) T_k diverges (for Remark 4.2.1);

iii) $\hat{\sigma}_k \geq \lambda T_k$ eventually (from ii and the definition of RP).

However, since $p(t) = o(1)$, with probability 1, the underlying r copies of the algorithm will have all reached the optimum after a certain time t_0. Therefore, since ii), it follows that, for all h large enough, we will have $\hat{p}_h(t) = 0$ for $t_0 \leq t \leq T_h$.

Hence, eventually $\hat{\sigma}_h$ will not change which is a contradiction with iii). Therefore $\mathbb{P}(r_k \to \infty) = 1$.

2 Remark 4.2.1, by looking at the RP in terms of a increasing sequence of finite matrices extracted by $\mathbf{Y}$, means that $\mathbb{P}(A) = 1$, where $A = \{\text{the first row or the first column of } \cup_k A_k \text{ is infinite}\}$. Then, we have

$$\mathbb{P}(\{\exists k : \quad \tilde{Y}_k = f_m\}) \geq \mathbb{P}(A \cap B), \tag{4.2}$$

where $B = \{\exists t : \quad Y_1(t) = f_m\} \cap \{\exists i : \quad Y_i(1) = f_m\}$. The event B has probability one since it is an intersection between events with probability one. In fact, $\mathbb{P}(\{\exists t : \quad Y_1(t) = f_m\}) = 1$ from the hypothesis $p(t) \to 0$. Moreover, by the strong law of large numbers $\mathbb{P}(\{\exists i : \quad Y_i(1) = f_m\}) = 1$. Hence, by (4.2)

$$\mathbb{P}(\{\exists k : \quad \tilde{Y}_k = f_m\}) = 1.$$

This proves (4.1). □

Lemma 4.2.3. *If, for every $t \in \mathbb{N}$, we have*

$$\mathbb{P}\left(\left\{\sup_k T_k < t\right\} \cup \left\{\lim_{k\to\infty} \hat{p}_k(t) = p(t)\right\}\right) = 1, \tag{4.3}$$

then it holds

$$\mathbb{P}\left(\sup_k T_k > \frac{t_m}{\lambda}\right) = 1.$$

Proof. Assume that the thesis is not true. Then, it exists a real number M such that $0 < M \leq \frac{t_m}{\lambda}$ and $\mathbb{P}(\{\sup_k T_k = M\}) > 0$. When $\left\{\sup_k T_k = M\right\}$ happens, by both (4.3) and the continuous mapping, we have the convergence $g_k(t) \to g(t)$, for any $t \leq M$. This means that, for any $\varepsilon > 0$ eventually it holds $\left\{\bigcap_{t=1}^{M} |g_k(t) - g(t)| < \varepsilon\right\}$. This implies that $g_k(M) < g(M) + \varepsilon$ and $g_k(t) > g(t) - \varepsilon$ for any $1 \leq t < M$. The first inequality can be rewritten as $-g_k(M) > -g(M) - \varepsilon$. Summing side by side the first and the last inequalities, we obtain

$$g_k(t) - g_k(M) > g(t) - g(M) - 2\varepsilon.$$

Since g is strictly decreasing until $t_m \geq \min(M, t_m) := \tilde{M}$, the r.h.s. of the last inequality is larger than $g(\tilde{M}-1)-g(\tilde{M})-2\delta$. By taking $\varepsilon = \frac{1}{2}\min_{t\in[2,\tilde{M}]\cap\mathbb{N}}(g(t-1)-g(t))$ we get $g_k(t) - g_k(\tilde{M}) > 0$ for $t < \tilde{M}$. Therefore, $\hat{\sigma}_k \to \arg\min_{t\in[0,M]\cap\mathbb{N}} g(t) = \tilde{M})$.

Hence, with a positive probability, we get eventually $\hat{\sigma}_k = \tilde{M}$. If $M \leq t_m$, we have eventually $\hat{\sigma}_k = M$. Since $\hat{\sigma}_k \leq T_k \leq \sup_k T_k = M$, we have eventually $T_k = M$. For one of such k, it holds $\frac{\hat{\sigma}_k}{T_k} = 1 > \lambda$, so that, by the definition of the RP, at the following iteration we have $T_{k+1} > T_k = M = \sup_k T_k$, which is a contradiction. In the other case $t_m < M \leq \frac{t_m}{\lambda}$, for an infinite number of values of k, we have $t_m = \hat{\sigma}_k \leq T_k \leq \sup_k T_k = M$. For any of these values of k, we get $\frac{\hat{\sigma}_k}{T_k} = \frac{t_m}{T_k} \geq \frac{t_m}{M} \geq \lambda$. As a consequence, given the current value T_k, as before, the following one T_{k+1} will be $f_T(T_k)$. By the property of the function f_T, after a certain number of iteration k, we get $T_k > M = \sup_k T_k$, which is a contradiction.

□

Theorem 4.2.4. *If we define* $T := \left\lfloor \frac{t_m}{\lambda} \right\rfloor$ *it holds*

$$\mathbb{P}\left(\sup_k T_k > \frac{t_m}{\lambda}\right) = 1,$$

and

$$\mathbb{P}\left(\bigcap_{t=1}^{T} \lim_{k\to\infty} \hat{p}_k(t) = p(t)\right) = 1.$$

Proof. We now show that

$$\mathbb{P}\left(\left\{\sup_k T_k < t\right\} \cup \left\{\lim_{k\to\infty} \hat{p}_k(t) = p(t)\right\}\right) = 1. \tag{4.4}$$

In fact, if the event $\{\sup_k T_k \geq t\}$ happens, then we can eventually compute $\hat{p}_k(t)$. Therefore, by the first statement of Lemma 4.2.2 and using the strong law of large numbers, we get

$$\mathbb{P}\left(\lim_{k\to\infty} \frac{1}{r_k}\sum_{i=1}^{r_k} 1_{\{Y_i(t)>f_m\}} = p(t)\right) = 1. \tag{4.5}$$

Hence, using (4.1), we obtain

$$\mathbb{P}\left(\lim_{k\to\infty}\frac{1}{r_k}\sum_{i=1}^{r_k}1_{\{Y_i(t)>\tilde{Y}_k\}}=p(t)\right)=1. \tag{4.6}$$

It then follows that,

$$\mathbb{P}\left(\sup_k T_k\geq t\right)=\mathbb{P}\left(\left\{\sup_k T_k\geq t\right\}\cap\left\{\lim_{k\to\infty}\hat{p}_k(t)=p(t)\right\}\right).$$

Since

$$\begin{aligned}\mathbb{P}\quad&\left(\left\{\sup_k T_k<t\right\}\cup\left\{\lim_{k\to\infty}\hat{p}_k(t)=p(t)\right\}\right)=\\ \mathbb{P}\quad&\left(\left\{\sup_k T_k<t\right\}\right)+\mathbb{P}\left(\left\{\sup_k T_k\geq t\right\}\cap\left\{\lim_{k\to\infty}\hat{p}_k(t)=p(t)\right\}\right),\end{aligned}$$

it follows (4.4), which is the hypothesis of Lemma 4.2.3, so that it holds

$$\mathbb{P}\left(\sup_k T_k>\frac{t_m}{\lambda}\right)=1.$$

Therefore, for any $t=1,2,\dots T$, with probability one we can eventually compute $\hat{p}_k(t)$ and (4.6) holds. Therefore we get

$$\mathbb{P}\left(\bigcap_{t=1}^{T}\lim_{k\to\infty}\hat{p}_k(t)=p(t)\right)=1,$$

that is the thesis.

□

4.3 Application of the RP to TSP instances

Here, we assess the performances of the RP and we compare them with those of the underlying algorithm. We consider several TSP instances with hundreds or thousands of cities for which the optimal solution is known. This information can be used to estimate the failure probability of the RP and of the underlying algorithm. However, obviously, this information has not be used when applying the underlying

algorithm and the RP. In order to compare the results from the two algorithms with the same computational effort, we consider for the RP a pseudo-time t, defined as follows. To this aim, we may think that the replications of RP are executed in parallel. When adding new replications (r_k increase), we stop the execution of the existing ones. Instead, when increasing the time T_k, we just continue the execution of them. The first T_1 instants of the pseudo-time correspond to the first T_1 iterations of the first replication. The following T_1 pseudo-time instants correspond to the analogous of the second replication and so on. At the end of the k-th RP iteration, we have produced r_k executions (replications) for T_k times and the final pseudo-time instant is $t = r_k \cdot T_k$. At the $(k+1)$-th iteration, we have a certain (r_{k+1}, T_{k+1}), with either $r_{k+1} > r_k$ and $T_{k+1} = T_k$ or $r_{k+1} = r_k$ and $T_{k+1} > T_k$. In the first case, the pseudo-time instant $t = T_k \cdot r_k + 1$ corresponds to the first iteration time of the $r_k + 1$ replication. The pseudo-time is increased until the end of that replication. We proceed in the same way until the end of r_{k+1} replication. In the second case, the pseudo-time instant $t = T_k \cdot r_k + 1$ corresponds to the iteration time $T_k + 1$ of the first replication. The pseudo-time is then increased until the iteration time T_{k+1} of that replication. Then, the following pseudo-time instant $t = T_k \cdot r_k + (T_{k+1} - T_k) + 1$ corresponds to the iteration time $T_k + 1$ of the second replication, and so on.

We denote by $\tilde{Y}(t)$ $(t = 1, 2, \dots)$, the process describing the best so far solution of the RP corresponding to the pseudo-time instant t. Hence, based on a set of m replications of the RP, we can estimate the failure probability $p_{\text{RP}}(t)$ by using the classical estimator

$$\hat{p}_{\text{RP}}(t) = \frac{1}{m} \sum_{i=1}^{m} 1_{\{\tilde{Y}_i(t) \neq f_m\}}. \tag{4.7}$$

By the law of large numbers this estimator converges to the failure probability of the RP $p_{\text{RP}}(t)$.

We notice now that $\hat{p}_{\text{RP}}(t)$ conditioned to a sequence $\{(r_k, T_k)\}_k$ can be expressed as a function of the failure probability $p(t)$ of the underlying algorithm as follow

$$\begin{cases} p(T_k)^{\left\lfloor \frac{t}{T_k} \right\rfloor} p\left(t - T_k \left\lfloor \frac{t}{T_k} \right\rfloor\right), & \text{if } r_{k+1} > r_k, \\ p(T_{k+1})^{\Delta_{t,k}} \cdot p\left(t - T_k r_k - \Delta_{t,k}(T_{k+1} - T_k) + T_k\right) \cdot p(T_k)^{r_{k+1} - \Delta_{t,k} - 1}, & \text{if } T_{k+1} > T_k, \end{cases} \tag{4.8}$$

where $\Delta_{t,k} = \left\lfloor \frac{t-T_k r_k}{T_{k+1}-T_k} \right\rfloor$, and $t = T_k \cdot r_k + 1, T_k \cdot r_k + 2, \ldots, T_{k+1} \cdot r_{k+1}$. In the second case $r_{k+1} = r_k$, the first factor describes the $\Delta_{t,k}$ replications which have been already extended until time T_{k+1}. The second factor represents the replication which is currently being extended. The last one accounts for the $r_{k+1} - \Delta_{t,k} - 1$ replications that have not yet been extended.

Replacing $p(t)$ in the last formula by the classical estimator $\hat{p}(t)$ in (4.7) based on a set of replications of the underlying algorithm, we can estimate the failure probability of the RP conditioned to a sequence $\{r_k, T_k\}_k$. Since $\hat{p}(t)$ is a consistent estimator of $p(t)$, by the continuous mapping the same will be true for the above estimator of $p_{\mathrm{RP}}(t)$.

Below, we describe some results of the application of the RP to different TSP instances studied in [38]. The underlying algorithm used here in the RP and as term of comparison is the ACO proposed in [38]. It is a particular ACO known as MMAS, combined with different local search procedures. Moreover, we compare them with those from the underlying algorithm. The RP setting is as follows: $T_{k+1} = f_T(T_k) := q(T_k)T_k$ where

$$q(T_k) = c_2 + C \cdot \sqrt{\frac{\left|\left(\bar{Y}(r_k, T_k) - \bar{Y}(r_k, \hat{\sigma}_k)\right)\right|}{\left(\bar{Y}(r_k, T_k) + \bar{Y}(r_k, \hat{\sigma}_k)\right)/2}}, \tag{4.9}$$

where $\bar{Y}(r, s) = \dfrac{1}{r}\displaystyle\sum_{i=1}^{r} Y_i(s)$, and c_2 and C are constant larger than one and zero, respectively. The square root factor appearing in the last formula is not essential, but we have empirical evidence that it accelerates the RP convergence. Moreover, $r_{k+1} = f_r(r_k) := c_1 \cdot r_k$, where $c_1 > 1$. Finally, we set $\lambda = \frac{2}{3}$.

We consider now an instance with 1291 cities (d1291). For this instance, the underlying algorithm has a high failure probability (f.p.) after some tens of thousands of iterations. Instead, the RP has a significantly lower f.p., as shown in the Figure 4.1. We remark that, until the value 10^4 for the time or pseudo-time, the f.p. the underlying algorithm is lower than the one of RP. This is due to the fact that the RP is still searching for the optimal value of the restart time. After that, the T_k value for RP is approaching the optimal restart time t_m. In fact, the value

of t_m and the final value of $\hat{\sigma}_k$ are 650 and 652, respectively. Consequently, the RP overcomes the underlying algorithm and gains up to three orders of magnitude in terms of the f.p. at pseudo-time 70000.

For the same instance above, in Figure 4.2, we compare the f.p. curves of RP computed by the estimators in (4.7) and (4.8). As one can see, there is a very good agreement between the two estimates. This is an evidence that the mathematical model for the RP is adequate. Furthermore, we have seen empirically that the variance of the estimator (4.8) is lower than the other one. Therefore, we have used the first one to compute the f.p. values for the RP applied to the several TSP instances which are shown in the Table 1. Of course, the estimator (4.8), uses the information about the estimate of $\hat{p}(t)$. This last requires additional independent replications of the underlying algorithm. However, the estimate of $\hat{p}(t)$ was necessary in any case for the comparison of the performances of the RP with those of the underlying algorithm.

Finally in Figure 4.3, we compare the f.p. curve of RP computed by the estimator in (4.7) (blue curve) and the curve obtained applying the restart periodically at the optimal restart time (red curve). We remark that the RP curve has the same behavior shifted by the pseudo time which takes to learn the optimal restart time than the red curve. We notice that for all instances of Table 4.3, curves similar to those as in Figure 4.1, 4.2 and 4.3 , were obtained.

By looking at the results in Table 4.3, it is evident the advantage of using the RP instead of the underlying algorithm. In fact, for all instances, the f.p. of the RP is two or three orders of magnitude lower than the one of the underlying algorithm. We can then claim the success of the RP to optimize the computation resource available with respect to the f.p. Therefore, given a certain computation resource, by applying the RP, we are far more confident that the result obtained is a solution of the COP instance analyzed.

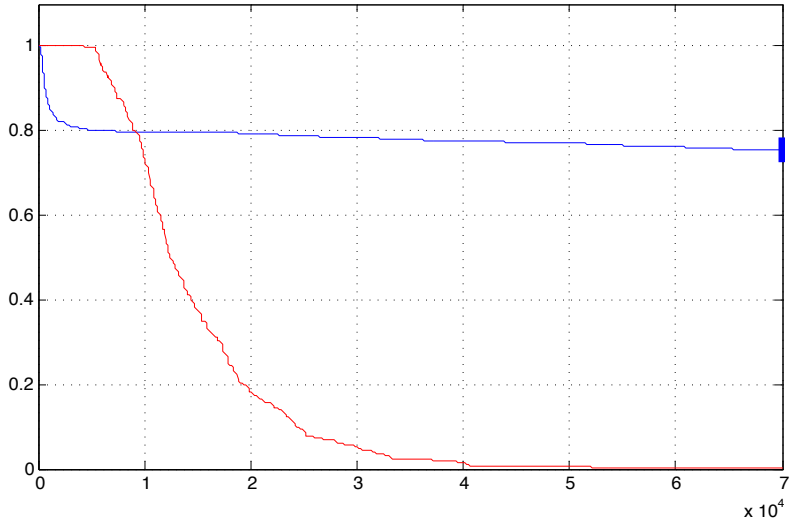

Figure 4.1: Failure probability versus pseudo-time and time for the RP (red line) and the underlying algorithm (blue line) respectively. The considered TSP instance (d1291) has 1291 cities. The f.p. curves of the RP and the underlying algorithm have been computed by the classical estimator in (4.7) based on 300 and 1600 replications, respectively. The vertical segment shows the 99% level confidence interval.

Instance	**ACO algorithm**	**T**	**ACO f.p.**	**RP f.p.**
eil51	MMAS -2.5 opt	100000	0.88	$5.3 \cdot 10^{-2}$
lin318	MMAS-2opt	84000	0.73	$1.7 \cdot 10^{-3}$
att532	MMAS-3opt	57000	0.54	$2.0 \cdot 10^{-3}$
pcb1173	MMAS-3opt	150000	0.75	$5.2 \cdot 10^{-2}$
d1291	MMAS-3opt	70000	0.76	$3.1 \cdot 10^{-3}$
rat783	MMAS-3opt	25000	0.10	$2.2 \cdot 10^{-2}$

Table 4.1: Results of the application of the RP and the underlying algorithm to TSP instances with known optimal solutions. The f.p. values are computed at the time T reported in the third column (pseudo-time for the RP).

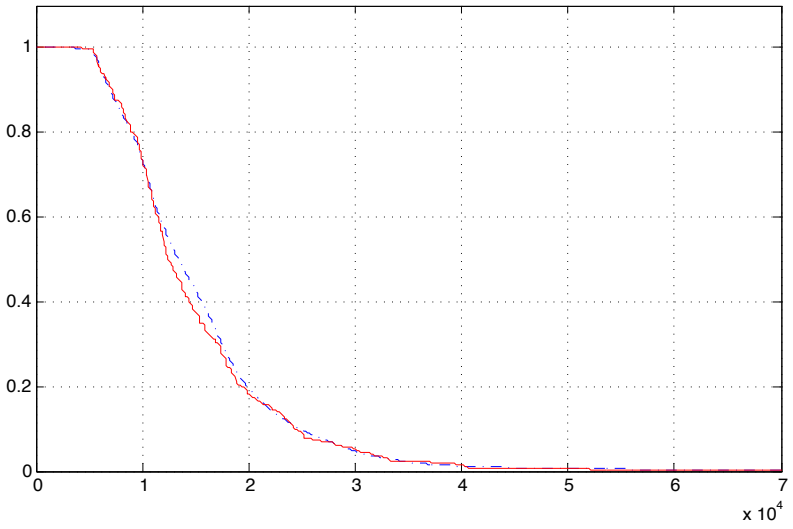

Figure 4.2: Failure probability for the RP versus pseudo-time for the TSP instance d1291. The continuous and dashed-dotted lines correspond to the estimators (4.7) and (4.8), respectively.

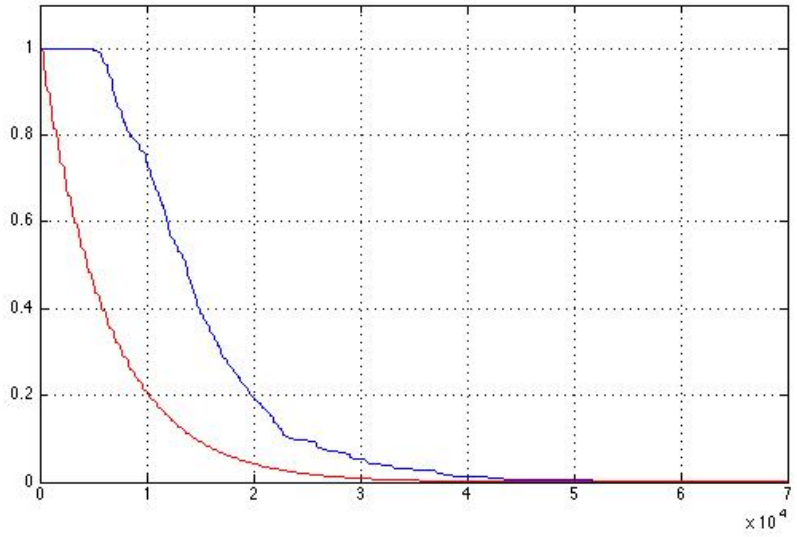

Figure 4.3: In this figure, using ACO as underlying algorithm, we compare f.p. of the RP that appears in Figure 1 (blue curve) with the one obtained applying the restart periodically at the optimal restart time (red curve).

Conclusions

For a given COP, when the search space has high cardinality, it is useful to apply stochastic algorithms, which exploit the space using the regularity of the problem. In these cases the problem could be hard so that it could happen that the algorithm stays in sub-optimal solution too much time with positive probability. Hence, it could be useful to improve the algorithm's performance. A way to do this, it is to apply the restart, which consists of periodic initializations of the algorithm. Although this technique is often applied in practice, there are few theoretical works which describe when, applying the restart, there is a gain. In this thesis, we have studied, from the theoretical side, sufficient conditions to have gain applying the restart. This gain is defined in two ways: the first is based on the failure probability, the second is based on the first hitting time to optimal solutions. In the first case, we say that the restart is convenient if the ratio between the failure probabilities of restart and underlying algorithm goes to zero. We show that this condition holds if the failure probability of the underlying algorithm is sub-exponential. The second theoretical approach consists of comparing the expected value of the first hitting time in optimal solutions between the underlying and restart algorithm. We have found sufficient conditions for having mean time to reach optimal solutions is lower for the restart algorithm than the one underlying. The sufficient condition is the existence of a time T such that, for $t > T$ the failure probability of the underlying algorithm decreases slower than for $t \leq T$. This conditions have limited utility, because one needs to know the failure probability of the underlying algorithm. In fact, for almost every problems the failure probability is unknown. Hence, we have proposed a new procedure, which does not use the failure probability of the underlying algorithm.

The iterative procedure executes a certain number of replication until a fixed time. These replications are used to estimate the failure probability of the underlying algorithm. In order to do this, it is computed the minimum value of the replications executed $\tilde{Y}_k$. Hence, it is computed for any time, the failure probability respect of $\tilde{Y}_k$, so that we obtain $g_k(t)$ that is an estimate of $g(t)$. Therefore, we have compute the position of the minimum of $\hat{g}_k(t)$ which is $\hat{\sigma}_k$. If $\hat{\sigma}_k$ close to the end of time window, the time of underlying algorithm is increased, otherwise the number of replication is increased. This choice is made because if the estimate of $g(t)$ continues to decrease, we extend the time, otherwise we improve the estimate of $g(t)$ increasing the number of replications. In order to establish if $\hat{\sigma}_k$ close to the end of time window, we set a parameter $\lambda \in (0, 1)$. For small λ we have less confidence than for λ near to one. In this way, the procedure overcomes the problem that the failure probability of underlying algorithm is unknown and, after a suitable numbers of iterations, the procedure applies the restart, using a time upper and close to the optimal restart time. In other words, the procedure, after a suitable numbers of iterations of underlying algorithm, that is used to understand the optimal restart time, the procedure works like if the failure probability is known. This fact, is shown in practice for some TSP instances (Table 1) and is guaranteed by the theory. In fact, the theorem, that we have proved, tell us that if the $p(t)$ goes to zero and $g(t)$ admits only one minimum t_m, and until t_m, $g(t)$ is strictly decreasing function, then $\hat{p}_k(t)$ converges almost certainly to $p(t)$. Hence, $\hat{g}_k(t)$ converges almost certainly to $g(t)$, so that the procedure finds the minimum of $g(t)$. In the last part of thesis, we have shown the results obtained applying the restart procedure for several TSP instances with hundred and thousand of cities. The procedure proposed can be improved in terms of preserve the performance, decreasing the computational cost. In order to do this, we can follow two approaches. The first is to take λ which depends from the iteration number of the procedure. In this way, the first iterations of the restart procedure, have an estimate of $g(t)$ that is not good, so that we could take small λ. Increasing the iterations of the procedure the estimate of $g(t)$ is improved, so that we can take λ larger. Hence, if we denote by k the k-th iteration of restart procedure, we could choose $\lambda = \lambda_k$, which is increasing with k. The second improvement is to

reduce the time window when the estimate of $g(t)$ is good enough. In this way, the last replications are executed for less time than the previous replications.

Bibliography

[1] D. L. Applegate, R. M. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem*. Princeton University Press, 2006.

[2] F. Barahona and L. Ladányi. Branch-and-cut based on the volume algorithm: Steiner trees in graphs and max cut. *IBM Research Report RC22221*, 2001.

[3] R. Bellman. Combinatorial processes and dynamic programming. *American Mathematical Society*, Combinatorial Analysis, Proceedings of Symposia in Applied Mathematics(10):217–249, 1960.

[4] R. Bellman. Dynamic programming treatment of the travelling salesman problem. *J. Assoc. Comput. Mach*, 9:61–63, 1962.

[5] B. Bullnheimer, R.F. Hartl, and C. Strauss. A new rank-based version of the ant system: a computational study. *Central European Journal for Operations Research and Economics*, 7(1):25–38, 1999.

[6] L. Carvelli and G. Sebastiani. *Some issues of ACO algorithm convergence*, chapter 4. INTECH, 2011.

[7] V. Cerny. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[8] S.A. Cook. The complexity of theorem-proving procedures. *Proc. 3rd Ann. ACM. Symp. on Theory of Computing*, 1971.

[9] A. Croes. A method for solving traveling salesman problem. *Opns. Res.*, 5:791–812, 1958.

[10] M. Dorigo and G. Di Caro. *The Ant Colony Optimization metaheuristic.* New Ideas in Optimization, 1999.

[11] M. Dorigo, V. Maniezzo, and A. Colorni. Positive feedback as a search strategy. *Technical Report, Politecnico di Milano*, 1991.

[12] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: Optimization by a colony of cooperating ants. *IEEE Trans. on System, Man, and Cybernetics*, 26, 1996.

[13] M. Dorigo and T. Stützle. *Ant Colony Optimization.* MIT Press, 2004.

[14] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman, San Francisco, CA,, 1979.

[15] M.J.J. Garvels and D.P. Kroese. A comparison of restart implementations. *Proceeding of the 1998 Winter Simulation Conference*, pages 601–609, 1998.

[16] M.J.J. Garvels and D.P. Kroese. On the entrance distribution in restart simulation. *RESIM '99*, 1999.

[17] F. Glover. Tabu search-part 1. *ORSA Journal of Computing*, 1(2):190–206, 1989.

[18] D.E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3:493–530, 1989.

[19] W.J. Gutjahr. A graph based ant system and its convergence. *Fut. Gener. Comput. Sys*, 16:873–888, 2000.

[20] W.J. Gutjahr. Aco algorithms with guaranteed convergence to the optimal solution. *Information Processing Letters*, 82:145–152, 2002.

[21] W.J. Gutjahr. First steps to the runtime complexity analysis of ant colony optimization. *Computer and Operations Reasearch*, 35:2711–2727, 2008.

[22] W.J. Gutjhar. A generalized convergence result for the graph-based ant system. *Probability in the engineering and informational sciences*, 17:545–569, 2003.

[23] W.J. Gutjhar and G. Sebastiani. Runtime analysis of ant colony optimization with best-so-far reinforcement'. *Meth. Comput. Appl. Prob.*, 10:409–433, 2008.

[24] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied Mathematics*, 10(2):196–210, 1962.

[25] K. Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130, 2000.

[26] R.Z. Hwang, R.C. Chang, and R.C.T. Lee. The searching over separators strategy to solve some np-hard problems in subexponential time. *Algoritmica*, 9:398–423, 1993.

[27] D. S. Johnson, L. A. McGeoch, and E. E. Rothberg. Asymptotic experimental analysis for the held-karp traveling salesman bound. *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 341–350, 1996.

[28] R. M. Karp. Dynamic programming meets the principle of inclusion and exclusion. *per. Res. Lett.*, 1(2):49–51, 1982.

[29] R.M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

[30] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[31] S. Kohn, A. Gottlieb, and M. Kohn. A generating function approach to the traveling salesman problem. *ACM Annual Conference ACM Press*, pages 294–300, 1977.

[32] S. Lin. Computer solutions of the traveling salesman problem. *The bell system technical journal*, pages 2245–2269, December 1965.

[33] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Oper. Res.*, 21:498–516, 1973.

[34] F. Neumann and C. Witt. Runtime analysis of a simple ant colony optimization algorithm. *Proc. ISAAC 06*, Springer LNCS 4288:618–624, 2006.

[35] S. Olivera, S. Hussin, T. Stützle, A. Roli, and M. Dorigo. A detailed analysis population-based ant colony optimization algorithm for the tsp and the qap. Technical Report TR/IRIDIA/2011-006, IRIDIA, February 2011.

[36] G. Sebastiani and G.L. Torrisi. An extended ant colony algorithm and its convergence analysis. *Methodology and Computing in Applied Probability*, 7:249–263, 2005.

[37] T. Stützle. Parallelization strategies for ant colony optimization. *Proceedings of PPSN-V, Fifth International Conference on Parallel Problem Solving from Nature, Springer-Verlag*, 1498:722–731, 1998.

[38] T. Stützle and H.H. Hoos. Max-min ant system. *Fut. Gener. Comput. Sys.*, 16:889–914, 2000.

[39] Stützle T. and Hoos H.H. The max-min ant system and local search for the travelling salesman problem. pages 309–314, 1997.

[40] P. Toth and D. Vigo, editors. *The vehicle routing problem*. SIAM Monograph on Discrete Mathematics and Applications, 2002.

[41] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math Soc.*, 42, 1936.

[42] A.P.A. Van Moorsel and K. Wolter. Analysis and algorithms for restart. *IEEE Computer Society: Proceedings of the 1st International Conference on the Quantitative Evaluation of Systems*, pages 195–204, 2004.

[43] G.J. Woeginger. Exact algorithms for np-hard problems. *OPTIMA*, 68, 2002.

Appendix A

ACO implementation code

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define n_points 194
#define new_dim 18
#define num_sub_pb 7
#define T 5000
#define TLOC 1000
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)
#define MAXFLDS 3 /* maximum possible number of fields */
#define MAXFLDSIZE 10000 /* longest possible field + 1 = 31 byte field */
```

```
22 #define bufsize 1024
23 #define nAnts 25
24
25
26 void edit_matrix_distance(float Cities[n_points][2],double M[n_points][n_points])
27 {
28         int i,j;
29         for(i = 0; i < n_points; i++){
30                 for(j = 0; j < n_points; j++){
31
32                         M[i][j]=sqrt((Cities[i][0]-Cities[j][0])*
33                                 (Cities[i][0]-Cities[j][0])+(Cities[i][1]-Cities[j][1])*
34                                 (Cities[i][1]-Cities[j][1]));
35                 }
36         }
37         return;
38 }
39
40 void set_diff(int set[n_points],int setNew[n_points], int elem,int lung )
41 {
42         int i,k=0;
43         for (i=0;i<n_points;i++)
44         {
45                 if(set[i]!=elem)
46                 {
47                         setNew[k]=set[i];
48                         k++;
49                 }
50
51         }
52         return;
53 }
54
```

```
55
56 void set_diff_local(int set[new_dim],int setNew[new_dim], int elem,int lung )
57 {
58         int i,k=0;
59         for (i=0;i<new_dim;i++)
60         {
61                 if(set[i]!=elem)
62                 {
63                         setNew[k]=set[i];
64                         k++;
65                 }
66
67         }
68         return;
69 }
70
71 float minimo(float a,float b){
72         if (a<b)
73                 return a;
74         else
75                 return b;
76 }
77
78 float massimo(float a,float b){
79         if (a>b)
80                 return a;
81         else
82                 return b;
83 }
84
85
86 int pos_massima(double v[n_points],int lung,double a)
87 { int j,i=0;
```

```
88          double w[lung],mass=-1;
89          int pos=-1;
90          for(j=0;j<lung;j++){
91                  w[j]=v[j];
92          }
93
94          while(mass<a)
95          {
96                  mass=w[i];
97                  pos=i;
98                  i++;
99          }
100
101         return pos;
102 }
103
104
105 float ran1(long *idum)
106 {
107         int j;
108         long k;
109         static long iy=0;
110         static long iv[NTAB];
111         float temp;
112
113
114         if (*idum <= 0 || !iy) {
115                 if (-(*idum) < 1) *idum=1;
116                 else *idum = -(*idum);
117                 for (j=NTAB+7;j>=0;j--) {
118                         k=(*idum)/IQ;
119                         *idum=IA*(*idum-k*IQ)-IR*k;
120                         if (*idum < 0) *idum += IM;
```

```
121                         if (j < NTAB) iv[j] = *idum;
122                 }
123                 iy=iv[0];
124
125         }
126         k=(*idum)/IQ;
127         *idum=IA*(*idum-k*IQ)-IR*k;
128         if (*idum < 0) *idum += IM;
129         j=iy/NDIV;
130         iy=iv[j];
131         iv[j] = *idum;
132         if ((temp=AM*iy) > RNMX) return RNMX;
133         else return temp;
134
135 }
136
137 void opt_3(int Pat[n_points],int NewPat[n_points], int cit[3])
138 {
139         float dist;
140         int i,j,pos_0,pos_1,pos_2;
141         pos_0=cit[0];
142         pos_1=cit[1];
143         pos_2=cit[2];
144         for(i=0;i<pos_0;i++)
145                 NewPat[i]=Pat[i];
146         j=0;
147         for(i=pos_0;i<pos_1;i++){
148                 NewPat[i]=Pat[pos_1-1-j];
149                 j++;
150         }
151         j=0;
152         for (i=pos_1;i<pos_2;i++){
153                 NewPat[i]=Pat[pos_2-1-j];
```

```
154                     j++;
155             }
156             for (i=pos_2;i<n_points;i++){
157                     NewPat[i]=Pat[i];
158             }
159     }
160
161     void opt_3_2(int Pat[n_points],int NewPat[n_points], int cit[3])
162     {
163             float dist;
164
165             int i,j,pos_0,pos_1,pos_2;
166             pos_0=cit[0];
167             pos_1=cit[1];
168             pos_2=cit[2];
169             for(i=0;i<pos_0;i++)
170                     NewPat[i]=Pat[i];
171             j=0;
172             for(i=pos_0;i<pos_0+pos_2-pos_1;i++){
173                     NewPat[i]=Pat[pos_1+j];
174                     j++;
175             }
176             j=0;
177             for (i=pos_0+pos_2-pos_1;i<pos_2;i++){
178                     NewPat[i]=Pat[pos_0+j];
179                     j++;
180             }
181             j=0;
182             for (i=pos_2;i<n_points;i++){
183                     NewPat[i]=Pat[pos_2+j];
184                     j++;
185             }
186     }
```

```
187
188
189
190
191
192
193 double verify_opt(double M[n_points][n_points],int Pat[n_points],double value_dist)
194
195 {
196         int iOp,jOp,hOp,bestCit2[2],bestCit3[3],kOp ,flag=0,i,j,pos_0,pos_1,pos_2,pos_3,NewPat[n_points];
197         double dist1,dist2,gain1,gain2,gain3,gain2bis,gain1bis,gain3bis,gain=0;
198
199         for(iOp=0;iOp<n_points-5;iOp++)
200         {
201                 for(jOp=iOp+2;jOp<n_points-3;jOp++)
202                 {
203                     pos_0=iOp+1;
204                         pos_1=jOp+1;
205                         gain1=M[Pat[pos_0-1]][Pat[pos_0]]-M[Pat[pos_1-1]][Pat[pos_0-1]];
206                         gain1bis=M[Pat[pos_0-1]][Pat[pos_0]]-M[Pat[pos_1]][Pat[pos_0-1]];
207                         if((gain1>0)||(gain1bis>0))
208                         {
209
210                                 for(kOp=jOp+2;kOp<n_points-1;kOp++)
211                                 {
212                                         pos_2=kOp+1;
213
214                                         gain2=gain1+M[Pat[pos_1-1]][Pat[pos_1]]-
215                                                 M[Pat[pos_1]][Pat[pos_2]];
216
217                                         if ((gain1>0) && (gain2>0))
218                                         {
219                                                 gain3=gain2+M[Pat[pos_2]][Pat[pos_2-1]]-
```

```
220                                                     M[Pat[pos_2-1]][Pat[pos_0]];
221
222                                             if(gain3>gain)
223
224                                             {
225                                                     flag=2;
226                                                     bestCit3[0]=pos_0;
227                                                     bestCit3[1]=pos_1;
228                                                     bestCit3[2]=pos_2;
229                                                     gain=gain3;
230                                             }
231                                     }
232                                     gain2bis=gain1bis+M[Pat[pos_1-1]][Pat[pos_1]]-
233                                                     M[Pat[pos_1-1]][Pat[pos_2]];
234                                     if ((gain1bis>0) && (gain2bis>0))
235                                     {
236                                             gain3bis=gain2bis+M[Pat[pos_2]][Pat[pos_2-1]]-
237                                                             M[Pat[pos_2-1]][Pat[pos_0]];
238                                             if(gain3bis>gain)
239
240                                             {
241
242                                                     flag=3;
243                                                     bestCit3[0]=pos_0;
244                                                     bestCit3[1]=pos_1;
245                                                     bestCit3[2]=pos_2;
246                                                     gain=gain3bis;
247                                             }
248                                     }
249                             }
250                     }
251             }
252     }
```

```
253         if(flag==2)
254         {
255                 opt_3(Pat,NewPat,bestCit3);
256                 for(i=0;i<n_points;i++)
257                         Pat[i]=NewPat[i];
258         }
259         else if(flag==3)
260         {
261
262                 opt_3_2(Pat,NewPat,bestCit3);
263                 for(i=0;i<n_points;i++)
264                         Pat[i]=NewPat[i];
265         }
266
267         value_dist=value_dist-gain;
268         return value_dist;
269 }
270
271
272
273
274
275 void tsp(double M[n_points][n_points], double sol[T],int bestPat[n_points],long h )
276 {
277         double N[n_points][n_points];
278         int i,jj,iDim,bestPatPart[n_points],bestPat2[n_points],iAnt,cont2=0,j,iLoc;
279         int iTime,w,k,setExpl[n_points],pat[n_points],ncities,ipat,lSetExpl,div,contator[n_points];
280         double fMinAnt,taumin,taumax,rho,tau[n_points][n_points],dist,appo[n_points],somma;
281         double fMin,fMinBis,fMin2,fMin3,dim,a,solLocal,solLocal2,fMinLocal,solLocalOpt;
282         long *idum;
283         int par,par1,new_dim_2;
284         int bestPatLocal[new_dim],bestPatLocalOpt[new_dim],bestPatLocalAppo[new_dim];
285         idum=&h;
```

```
286         rho=0.26;
287         taumax=10000;
288         taumin=0;
289
290         for(i=0;i<n_points;i++)
291         {
292                 for (j=0;j<n_points;j++)
293
294                         N[i][j]=M[i][j]*M[i][j]*M[i][j]*M[i][j]*M[i][j]*M[i][j];
295         }
296         for (i=0;i<n_points;i++)
297         {
298                 for (j=0;j<n_points;j++)
299                 {
300                         tau[i][j]=taumax;
301                 }
302         }
303         fMin=1000000000;
304         par1=0;
305         for (iTime=0;iTime<T;iTime++)
306         {
307                 if (iTime<=25)
308                         par=30;
309                 if(iTime>25 && iTime<=75)
310                         par=5;
311                 else if(iTime>75 && iTime<=125)
312                         par=3;
313                 else if(iTime>125 && iTime<=250)
314                         par=2;
315                 else if(iTime>250)
316                         par=1;
317                 if(iTime==26 || iTime==76 || iTime==126 || iTime==251)
318                         par1=0;
```

```
319             if( iTime==251)
320                     par1=0;
321             fMinAnt=1000000000;
322             for (iAnt=0;iAnt<nAnts;iAnt++)
323             {
324                     for (i=0;i<n_points;i++)
325                     {
326                             setExpl[i]=i;
327                             contator[i]=0;
328                     }
329                     ncities=1;
330                     i=0;
331                     lSetExpl=n_points-1;
332                     set_diff(setExpl,setExpl,i,lSetExpl);
333                     ipat=0;
334                     pat[ipat]=i;
335                     dist=0;
336                     while(ncities<n_points-1)
337                     {
338
339                             for(j=0;j<lSetExpl;j++)
340                             {
341                                     appo[j]=0;
342                             }
343                             ipat++;
344                             somma=0;
345                             for(j=0;j<lSetExpl;j++){
346                                     somma=somma+tau[i][setExpl[j]]/(N[i][setExpl[j]]);
347                             }
348
349                             for(j=0;j<lSetExpl;j++){
350                                     appo[j]=tau[i][setExpl[j]]/(N[i][setExpl[j]]*somma);
351                             }
```

```
352
353                     for(j=1;j<lSetExpl;j++)
354                     {
355                             appo[j]=appo[j]+appo[j-1];
356                     }
357                     a=ran1(idum);
358                     w=pos_massima(appo,lSetExpl,a);
359                     w=setExpl[w];
360                     pat[ipat]=w;
361                     dist=dist+M[i][w];
362                     i=w;
363                     set_diff(setExpl,setExpl,i,lSetExpl);
364                     lSetExpl--;
365                     ncities++;
366             }
367             dist=dist+M[setExpl[0]][pat[0]]+M[setExpl[0]][pat[n_points-2]];
368             pat[ipat+1]=setExpl[0];
369             dist=verify_opt(M,pat,dist);
370             if(dist<fMinAnt)
371             {
372                     fMinAnt=dist;
373                     for(i=0;i<n_points;i++)
374                             bestPatPart[i]=pat[i];
375             }
376     }
377     if(par-par1>1) {
378             for (i=0;i<n_points;i++)
379             {
380                     for (j=0;j<n_points;j++)
381                     {
382                             tau[i][j]=(1-rho)*tau[i][j];
383                     }
384             }
```

```
385                     for(k=1;k<n_points;k++)
386                     {
387                             tau[bestPatPart[k]][bestPatPart[k-1]]=
388                             tau[bestPatPart[k]][bestPatPart[k-1]]+ 1/fMinAnt;
389                             tau[bestPatPart[k-1]][bestPatPart[k]]=
390                             tau[bestPatPart[k-1]][bestPatPart[k]]+ 1/fMinAnt;
391                     }
392                     tau[bestPatPart[0]][bestPatPart[n_points-1]]=
393                     tau[bestPatPart[0]][bestPatPart[n_points-1]]+ 1/fMinAnt;
394                     tau[bestPatPart[n_points-1]][bestPatPart[0]]=
395                     tau[bestPatPart[n_points-1]][bestPatPart[0]]+1/fMinAnt;
396                     for (i=0;i<n_points;i++)
397                     {
398                             for (j=0;j<n_points;j++)
399                             {
400                                     tau[i][j]=minimo(taumax,massimo(tau[i][j],taumin));
401                             }
402                     }
403                     par1++;
404
405             }
406             if(fMinAnt<fMin)
407             {
408                     fMin=fMinAnt;
409                     for(i=0;i<n_points;i++)
410                             bestPat[i]=bestPatPart[i];
411             }
412
413                     for (i=0;i<n_points;i++)
414                     {
415                             for (j=0;j<n_points;j++)
416                             {
417                                     tau[i][j]=(1-rho)*tau[i][j];
```

```
418                         }
419                 }
420                 for(k=1;k<n_points;k++)
421                 {
422
423                         tau[bestPat[k]][bestPat[k-1]]=
424                         tau[bestPat[k]][bestPat[k-1]]+ 1/fMin;
425
426                         tau[bestPat[k-1]][bestPat[k]]=
427                         tau[bestPat[k-1]][bestPat[k]]+ 1/fMin;
428
429                 }
430                 tau[bestPat[0]][bestPat[n_points-1]]=
431                 tau[bestPat[0]][bestPat[n_points-1]]+ 1/fMin;
432
433                 tau[bestPat[n_points-1]][bestPat[0]]=
434                 tau[bestPat[n_points-1]][bestPat[0]]+1/fMin;
435
436
437                 for (i=0;i<n_points;i++)
438                 {
439                         for (j=0;j<n_points;j++)
440                         {
441                                 tau[i][j]=minimo(taumax,massimo(tau[i][j],taumin));
442                         }
443                 }
444
445         }
446
447         sol[iTime]=fMin;
448         taumax=1/(rho*fMin);
449         taumin=taumax/(2*n_points);
450 }
```

```
451         printf("\n MinimoAco=%f\n",fMin);
452         return;
453 }
454 int main( )
455 {
456         FILE *fp;
457         int fld,iRep,nRep=1,i;
458         float s[n_points][2];
459         double sol[T];
460         int idx=0,bestPat[n_points];
461         int idx2;
462         char c;
463         char *tok;
464         char ca[10000];
465         char*p;
466         double M[n_points][n_points];
467
468         char arr[MAXFLDS]={0};
469         fp = fopen("q2.txt", "r");
470         if (fp == NULL)
471                 printf("File doesn't exist\n");
472         else {
473                 while (c != EOF) {
474                         c = getc(fp); /* get one character from the file*/
475                         fgets (ca, 10000, fp);
476                         idx2=0;
477                         tok = strtok(ca,";");
478                         while (tok != NULL)
479                         {
480                                 if(idx2==0)
481                                 {
482
483                                 }
```

```
484                                     else if(idx2==1)
485                                             s[idx][0]=atof(tok);
486
487                                     else if(idx2==2)
488                                             s[idx][1]=atof(tok);
489
490                                     idx2++;
491
492
493                                     tok = strtok(NULL, ";");
494
495
496                             }
497                             idx++;
498                     }
499                     }
500     fclose(fp);
501         FILE * pFile;
502         FILE * pFilenew;
503         pFile = fopen ("qt194simple10bis.txt","w+");
504         pFilenew = fopen ("solqt194simple10bis.txt","w+");
505         int contatore=0;
506         for (i=0;i<n_points;i++){
507                 contatore++;
508         }
509         edit_matrix_distance(s,M);
510         for (iRep=0;iRep<nRep;iRep++){
511                 tsp(M,sol,bestPat,-(iRep+23));
512                 for (i=0;i<T;i++)
513                         fprintf(pFile,"%f ",sol[i]);
514                 for(i=0;i<n_points;i++)
515                         fprintf(pFilenew,"%d ",bestPat[i]);
516                 }
```

```
517         return 0;
518 }
```

Appendix B

ACO for pseudo boolean functions

```
function risultato=funzione_4(x)
risultato=abs(sum(x)-(length(x)-1)/2);
end

nRep=1000;
T=20000;
N=20;
A=zeros(T,1);
B=A;
contt(T,1)=0;
for w=1:nRep
tau=1/2*ones(N,1);
taumin=.3*ones(N,1);
taumax=.7*ones(N,1);
rho=rho_1(T);
nAntsNumbers=1;
nFcn= @funzione_4;
fMax=-1000000;
iTime=1;
while iTime <= T
    for iAntNumber=1:nAntsNumbers
```

```
22            v=unifrnd(0,1,N,1);
23            sol=heaviside(v-tau);
24           fMaxNew=nFcn(sol);
25       if fMaxNew>fMax
26        fMax=fMaxNew;
27       massimo=sol;
28       end
29           end
30   for i=1:length(tau)
31       tau(i)=(1-rho(iTime)).*tau(i) + rho(iTime).*(1-massimo(i));
32       end
33       tau=max(taumin,tau);
34       tau=min(taumax,tau);
35       x=sum(abs(massimo-ones(N,1)));
36  massi(w,iTime)=nFcn(massimo);
37
38       if x == 0
39       contt(iTime)=contt(iTime)+1;
40       iTime= 2*T;
41       end
42       iTime=iTime+1;
43      end
44
45  end
```