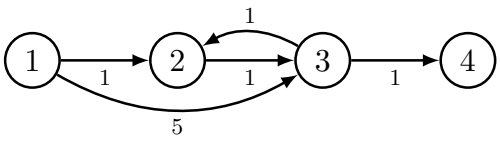
University of Naples "Federico II"

Dep. of Mathematics and Applications "R. Caccioppoli"

The Shortest Path Tour Problem and its variants

PhD Thesis

DANIELE FERONE



Supervisor: Paola Festa
Professor

Naples, February 2017

Abstract

Scope of this thesis is to provide a treatment of the Shortest Path Tour Problem, and its variants. It presents a deep investigation of two variants of the ***SPTP***, the *Constrained Shortest Path Tour Problem* and *Shortest Path Tour Problem with Time Windows*, respectively. Moreover, a GRASP meta-heuristic is applied to solve further hard combinatorial optimization problems.

Ai miei nonni e...
a mia figlia Chiara!

Ringraziamenti

Non ho mai scritto ringraziamenti nelle mie tesi precedenti perché il percorso non era finito. Questo lavoro, invece, segna davvero la fine di un percorso e l'inizio di uno che sarà completamente nuovo.

Il mio percorso universitario è iniziato nel lontano 2004 e tante cose sono successe durante questi tredici anni. Tante sono state le persone che ho incontrato durante questo percorso e a cui va dedicata una parola.

Inizio con il ringraziare **Fabrizio**, se ho scelto informatica una parte del merito è sicuramente tuo. Dopo il mio primo esame mi hai detto di non accontentarmi… ho provato a fare del mio meglio. Ci siamo conosciuti condividendo l'amore per il teatro e, sebbene sia riuscito a recitare con te, coltivo ancora il desiderio di poterlo rifare sotto la tua guida.

I ringraziamenti successivi vanno ai PdK: **Davide** (bruececkel), **Nino** (gib) e **Francesco** (jobzino). Siete stati i miei compagni di viaggio più stretti durante le lauree. Ora siete tutti lontani per lavoro, ma come dimenticare le pasquette o le nottate di Warrock? Ogni corso che ho potuto frequentare in vostra compagnia è stato più divertente di qualsiasi corso che ho dovuto seguire da solo. Ogni esame in cui abbiamo studiato insieme è stato più un gioco che un lavoro.

Durante il dottorato ho trovato nuovi compagni di viaggio. Un grande ringraziamento va al Team MAT09: **Antonio** (antonap) e **Tommaso** (Supersantos). Mi avete accompagnato per il secondo e terzo anno, lavorare con voi è stato decisamente più divertente che lavorare da solo. Siete stati ottimi colleghi e spero che continueremo ad esserlo a lungo!

I ringraziamenti "universitari" terminano, ovviamente, con il ringraziare la Professoressa **Paola Festa**. È stata mia relatrice per le due tesi di laurea e poi mia tutor di dottorato. Una così lunga collaborazione testimonia la mia stima profonda per te, sia come accademica, ma soprattutto come persona! Sei il modello lavorativo a cui aspiro, grazie per i tuoi insegnamenti e per la stima che ho sempre avvertito. Ho ancora tanto da imparare da te!

Ringrazio la **famiglia Santoro** allargata. Mi siete sempre stati vicini nelle vicende universitarie prima e lavorative poi, facendomi sentire la vostra stima e il vostro affetto, un ringraziamento grande.

Non posso non ricordare i miei **nonni** e le mie **zie**. Molti non hanno potuto vedere la fine del mio percorso, ma so che mi sono vicini. Grazie per il vostro supporto!

In questa lista di persone che mi hanno accompagnato e aiutato non posso non citare **Pasquale**, la mia adorata nipotina **Jolanda** e, soprattutto, la mia sorellina **Marcella**!!! A dispetto della lontananza fisica so, e voglio che lo sappia anche tu, che alla minima necessità siamo disponibili l'uno per l'altra.

Il ringraziamento più grande va sicuramente ai miei **genitori Vincenzo e Rosa**. Tutto ciò che di buono sono riuscito a fare nella vita, compreso questo traguardo, è merito devi vostri insegnamenti, del vostro incoraggiamento e dei vostri sacrifici. Non avrei potuto avere genitori migliori!

I miei genitori sono anche l'esempio a cui aspiro nel mio rapporto con mia **moglie Maria**. La nostra storia è cominciata nello stesso periodo in cui è cominciato il mio viaggio universitario e senza di te non sarebbe stato lo stesso. Mi hai sempre incoraggiato e aiutato nelle difficoltà. Ti amo! Grazie per il tuo supporto e per lo splendido regalo di quest'ultimo anno.

Regalo che ha il nome di **Chiara, mia figlia**. Un ringraziamento enorme a te perché ogni giorno mi regali una gioia infinita! Che la vita ti riservi tantissima gioia!

Spero non aver dimenticato nessuno, ma nel caso non me ne vogliate. Non sono uno scrittore, quindi probabilmente questi ringraziamenti non saranno molto belli, ma sono sentiti.

Voglio bene a tutti voi.

Napoli, Febbraio 2017

Daniele Ferone

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

> Computer science is no more about computers than astronomy is about telescopes.
>
> Edsger Dijkstra

Main scope of this thesis is to provide an exhaustive treatment of the *Shortest Path Tour Problem* (***SPTP***) and its variants. The ***SPTP*** can be seen as a constrained version of the classical Shortest Path Problem and arises in several contexts, such as warehouse management or control of robot motions. In the first case, assume that an order arrives for a certain set of N collections of items stored in a warehouse. A vehicle has to collect at least one item of each collection of the order to ship them to the costumers. In control of robot motions, assume that to manufacture workpieces, a robot has to perform at least one operation selected from a set of N types of operations. In this latter case, operations are associated with nodes of a directed graph and the time needed for a tool change is the distance between two nodes. In both cases, there are precedence constraints to be satisfied that can be well modeled by the ***SPTP***.

The rest of Chapter 1 is devoted to the formal definition of the ***SPTP***, proved to be polynomially solvable. Given both the practical and theoretical interests related to an in-depth study of the problem, in this work two different new variants of the ***SPTP*** are presented: the *Constrained Shortest Path Tour Problem* (***CSPTP***) and the *Shortest Path Tour Problem with Time Windows* (***SPTPTW***). Both variants are deeply investigated in Chapter 2. Unlike the ***SPTP***, we have proved that these two problems are computationally intractable. To exactly solve them, we have proposed two Branch & Bound (B&B) algorithms and a dynamic programming approach. Given the intractability of the problems, those exact methods are not able to tackle large size instances in reasonable running times. Due to this, the use of an heuristic algorithm has been explored for the ***CSPTP***. In particular, a GRASP meta-heuristic has been applied to solve the problem.

The results obtained by our GRASP tailored for the ***CSPTP*** encouraged the use of this

technique to approximatively solve other complex Combinatorial Optimization Problems (COPs) arising in several practical contexts. The results obtained by these studies are presented in Chapter 3.

Nowadays, the vast majority of practical interest problems can be modeled through hard COPs. Hence, the refinement of these meta-heuristics and their proper implementation are increasingly needed. Furthermore, in order to handle more realistic scenarios, it is needed to introduce stochasticity: one of the primary components of the real world. It is in this spirit that in Chapter 3 we propose an extension of the GRASP meta-heuristic, which can be used to solve non deterministic problems.

Conclusions and final remarks of the thesis are summarized in Chapter 4.

1.1 The Shortest Path Problem

The *Shortest Path Problem* (***SPP***) is a classical combinatorial problem that arises as subproblem when solving many optimization problems. Since it contains the most important ingredients of network flows problems, it has been widely studied and there exists a great number of algorithms to solve it [6, 9, 11, 21, 38].

Several variants of ***SPP*** appear in a wide variety of contexts and practical application settings. In particular, the *Shortest Path Tour Problem* is a constrained version of the classical Shortest Path Problem.

In the rest of this document, the following notation is used. Let $G = (V, A, C)$ a weighted directed graph, where Given $G = (V, A, C)$ be a directed graph, where

- $V = \{1, \ldots, n\}$ is a set of n nodes;
- $A = \{(i,j) \in V \times V \mid i,j \in V \wedge i \neq j\}$ is a set of m arcs;
- $C : A \to \mathbb{R}^+ \cup \{0\}$ is a function that assigns a nonnegative length c_{ij} to each arc $(i,j) \in A$;
- for each node $i \in V$, let $FS(i) = \{j \in V \mid (i,j) \in A\}$ and $BS(i) = \{j \in V \mid (j,i) \in A\}$ be the *forward star* and *backward star* of node i, respectively;
- a path $P = \{i_1, \ldots, i_k\}$ is a sequence of k nodes $i_j \in V$, such that $(i_j, i_{j+1}) \in A$, for all $j = 1, \ldots, k-1$;
- the length (or cost) $C(P)$ of any path P is defined as the sum of the lengths of the arcs on the path, i.e. $C(P) = \sum_{j=1}^{k-1} c_{jj+1}$.

Let $G = (V, A, C)$ a directed weighted graph, and let $s, d \in V$, $s \neq d$, be two different nodes of the graph, the *single-origin single-destination Shortest Path Problem* consists in

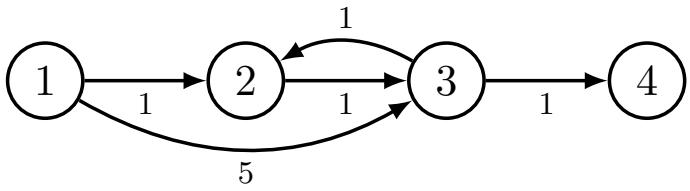


Figure 1.1: *A **SPTP** instance on a small graph G, where $T_1 = \{1\}$, $T_2 = \{3\}$, $T_3 = \{2\}$, $T_4 = \{4\}$.*

finding a directed path from s to d that minimizes the cost of the path. Formally, the ***SPP*** admits the following 0-1 integer programming formulation:

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{1.1a}$$

s.t.

$$\sum_{j\in FS(i)} x_{ij} - \sum_{j\in BS(i)} x_{ji} = \begin{cases} 1, & i = s; \\ -1, & i = d; \\ 0, & \text{otherwise}; \end{cases} \tag{1.1b}$$

$$x_{ij} \in \{0, 1\} \qquad \forall\ (i,j) \in A. \tag{1.1c}$$

1.2 The Shortest Path Tour Problem

The Shortest Path Tour Problem (***SPTP***) was first defined by Bertsekas in his book devoted to Dynamic Programming and Optimal Control [7], and later formalized and studied by Festa [26].

Definition 1.1. *Given $G = (V, A, C)$ be a directed weighted graph, the shortest path tour problem (**SPTP**) consists in finding a shortest path from a source node s to a destination node d, $s, d \in V$, $s \neq d$, by ensuring that at least one node of each node disjoint subsets $T_1, \ldots, T_N$, is crossed according to the sequence wherewith the subsets are ordered. Any intermediate nodes between visits to the subsets T_h, $h = 1, \ldots, N$ are allowed.*

In Figure 1.1, an example instance of ***SPTP*** is depicted. Let the source node $s = 1$ and the destination node $d = 4$, it is easy to verify that the shortest path between them is the path $\Pi = (1, 2, 3, 4)$, whose cost is $C(\Pi) = 3$. Conversely, given the sets $T_1 = \{1\}$, $T_2 = \{3\}$, $T_3 = \{2\}$, $T_4 = \{4\}$, the best shortest path tour from s to d is the path $\Pi = (1, 2, 3, 2, 3, 4)$, whose cost is $C(\Pi) = 5$ and is not simple.

Festa [26] proved that ***SPTP*** is polynomially solvable, since it can be reduced to a classical ***SPP*** on an opportunely expanded graph, and Festa et al. [27] proposed a dynamic programming algorithm to solve the problem.

Chapter 2

Shortest Path Tour Problem variants

The Constrained Shortest Path Tour Problem (***CSPTP***), first presented in Ferone et al. [19], is a variant of the ***SPTP***. In particular, in ***CSPTP*** it is forbidden to cross an arc twice. Respect to the Definition 1.1, the problem can be defined as follows.

Definition 2.1. *The **Constrained** Shortest Path Tour Problem (**CSPTP**) consists in finding a shortest path from a source node s to a destination node d, $s, d \in V$, $s \neq d$, by ensuring that at least one node of each node disjoint subsets $T_1, \ldots, T_N$, is crossed according to the sequence wherewith the subsets are ordered **and each arc** $(i, j) \in A$ **is crossed at most once**. Any intermediate nodes between visits to the subsets T_h, $h = 1, \ldots, N$ are allowed.*

Although the definition is only slightly different from the definition of the ***SPTP***, the resulting problem has a huge difference in theoretical properties. Indeed, the ***CSPTP*** is a **NP**-hard problem, and it is not possible to optimally solve big instances in reasonable computation times.

2.1 Complexity

To prove the hardness of the ***CSPTP***, it will be proved that the *Hamiltonian Path problem* (***HP***) is polynomially Karp-reducible to the ***CSPTP***.

Let $G = (V, A, C)$ a graph, and let s and d, $s, d \in V$, $s \neq d$, be the origin and the destination node, respectively. Then, the ***HP*** consists in finding in G a minimum cost Hamiltonian path from s to d. In Karp [42], it was proven that the ***HP*** is **NP**-complete.

To prove that ***HP*** is polynomially Karp-reducible to the ***CSPTP*** (***HP*** $<_m^p$ ***CSPTP***), we need to define a polynomially computable function $f(\mathcal{I}_{HP})$ that transforms any instance $\mathcal{I}_{HP}$ of the ***HP*** in an instance $\mathcal{I}_{CSPTP}$ of the ***CSPTP***. In other words, we need to design a polynomial-time Karp-reduction algorithm that takes as input an instance $\mathcal{I}_{HP}$ of the ***HP*** and builds a directed graph containing a feasible path tour P_T if and only if there exists a feasible Hamiltonian path in $\mathcal{I}_{HP}$.

Given any ***HP*** instance $\mathcal{I}_{HP}$

$$\langle G = (V, A, C), s, d \rangle ,$$

Algorithm 2.1 is the pseudo-code of the reduction algorithm that outputs a ***CSPTP*** instance

$$\langle G' = (V', A', C'), s^-, d^+, \{T_h\}_{h=1,\dots,n+1} \rangle$$

by performing the following operations:

- for each node $i \in V$,
  - insert in V' nodes i^- and i^+;
  - insert in A' arc (i^-, i^+) with cost 0;
- for each arc $(i,j) \in A$ and for each $k = 2, \dots, n$,
  - insert in V' node ij^k;
  - insert in T_k node ij^k;
  - insert in A' arc (i^+, ij^k) with cost c_{ij} and arc (ij^k, j^-) with cost 0;
- set $T_1 = \{s^-\}$ and $T_{n+1} = \{d^+\}$.

Algorithm 2.1: *Polynomial reduction algorithm from the* ***HP*** *to the* ***CSPTP***.

```
1  Function HamPath-to-CSPTP(V, A, C, s, d)
2      V' ← A' ← ∅ ;
3      n ← |V| ;
4      for i ← 2 to n do
5          T_i ← ∅ ;
6      for i ∈ V do
7          V' ← V' ∪ {i⁻, i⁺} ;
8          A' ← A' ∪ {(i⁻, i⁺)} ;
9          c'(i⁻, i⁺) ← 0 ;
10     for (i,j) ∈ A do
11         for k ← 2 to n do
12             V' ← V' ∪ {ij^k} ;
13             A' ← A' ∪ {(i⁺, ij^k), (ij^k, j⁻)};
14             c'(i⁺, ij^k) ← c(i,j);
15             c'(ij^k, j⁻) ← 0 ;
16             T_k ← T_k ∪ {ij^k} ;
17     T_1 ← {s⁻};
18     T_{n+1} ← {d⁺};
19     return (G = (V', A', C'), s⁻, d⁺, {T_h}_{h=1,...,n+1});
```

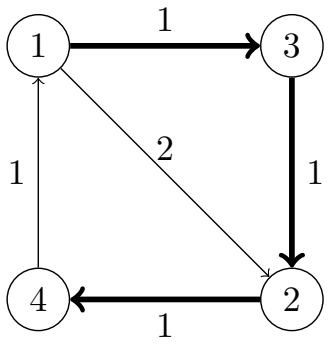


Figure 2.1: *A small **HP** instance: bold line style indicates a Hamiltonian Path from node 1 to node 4.*

Summarizing, the set of nodes V' can be defined as follows:

$$V' = \bigcup_{i \in V} \{\, i^-, i^+ \,\} \cup \bigcup_{(i,j) \in A}^{k=2,\dots,n} \left\{\, ij^k \,\right\},$$

while the set A' can be defined as:

$$A' = \bigcup_{i \in V} \{\, (i^-, i^+) \,\} \cup \bigcup_{(i,j) \in A}^{k=2,\dots,n} \left\{\, (i^+, ij^k), (ij^k, j^-) \,\right\}.$$

A small instance $\mathcal{I}_{HP}$ is depicted in Figure 2.1. Figure 2.2 shows the corresponding instance $\mathcal{I}_{CSPTP}$ as returned by algorithm in Algorithm 2.1, whose computational complexity is $O(n \cdot m)$.

Lemma 2.2. *There exists a feasible path $P = i_1, i_2, \dots, i_k$, $k \leq n$, in*

$$\langle G = (V, A, C), s, d \rangle,$$

if and only if in

$$\langle G' = (V', A', C'), s^-, d^+, \{T_h\}_{h=1,\dots,n+1} \rangle$$

there exists a feasible path tour P' from i_l^- to i_k^+, such that

$$P' = \left\{ \bigoplus_{l=1}^{k-1} \left(i_l^-, i_l^+, i_l i_{l+1}{}^{l+1} \right), i_k^-, i_k^+ \right\}.$$

Proof. Suppose that there exists in G a feasible path $P = \{i_1, i_2, \dots, i_k\}$, $k \leq n$. Then, by construction there exists in A' an arc (i_l^-, i_l^+), for each $l = 1, \dots, k$.

Moreover, for each arc (i_l, i_{l+1}) in P, there exist arcs $(i_l^+, i_l i_{l+1}{}^q)$ and $(i_l i_{l+1}{}^q, i_{l+1}^-)$ for each $q = 2, \dots, n$.

Therefore, there must exist also arcs $(i_l^+, i_l i_{l+1}{}^{l+1})$ and $(i_l i_{l+1}{}^{l+1}, i_{l+1}^-)$. Conversely, sup-

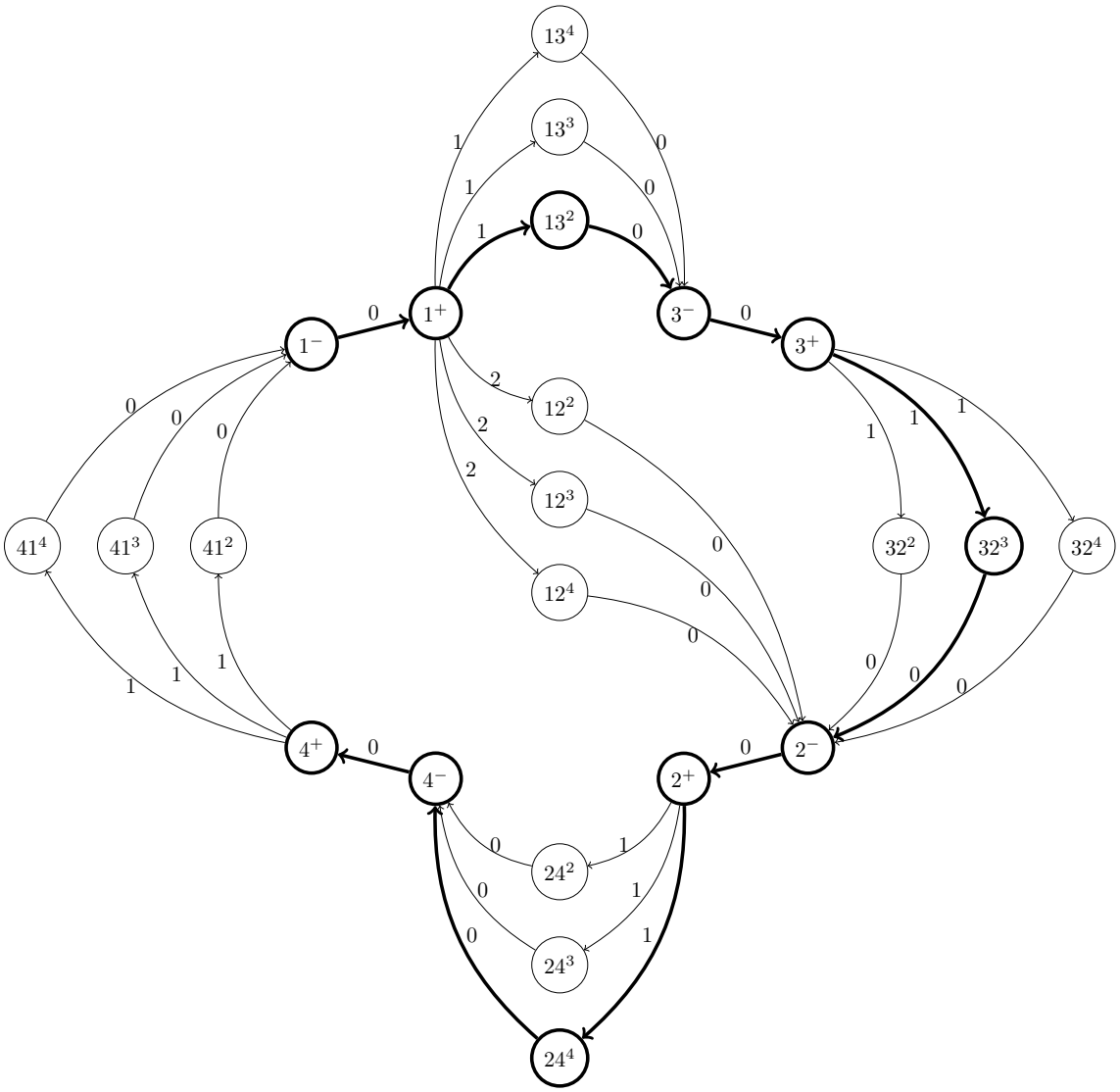


Figure 2.2: *The **CSPTP** instance corresponding to the small **HP** instance depicted in Figure 2.1.*

pose that there exists in G' the path P', whereas path P is not present in G. This last situation occurs if either at least one node $i_l \notin V$ or at least one arc $(i_l, i_{l+1}) \notin A$.

If a node $i_l \notin V$, then nodes i_l^- and i_l^+ would not be in V', which is not true. Similarly, if an arc $(i_l, i_{l+1}) \notin A$, then arcs $(i_l^+, i_l {i_{l+1}}^{l+1})$ and $(i_l {i_{l+1}}^{l+1}, i_{l+1}^-)$ would not be in A' and this contradicts the hypothesis of existence of path P'. ∎

Theorem 2.3. *Function $f(\mathcal{I}_{HP})$ computed by Algorithm 2.1 is a polynomially computable function that transforms any instance $\mathcal{I}_{HP}$ of **HP** in an instance $\mathcal{I}_{CSPTP}$ of the **CSPTP**.*

Proof. $f(\mathcal{I}_{HP})$ is polynomially computable given the polynomial complexity of Algorithm 2.1.

To show that $f(\mathcal{I}_{HP})$ is a reduction function, we must prove that there exists in G a Hamiltonian path P from s to d with length $L(P)$ if and only if there exists in G' a constrained path tour P' from s^- to d^+ with length $L(P') = L(P)$.

⇒ By hypothesis, there exists in G a Hamiltonian path $P = \{i_1, i_2, \cdots, i_n\}$, where $i_1 = s$ and $i_n = d$. We have already shown in lemma 2.2 that there exists in G' a path

$$P' = \left\{ \bigoplus_{l=1}^{n-1} \left(i_l^-, i_l^+, i_l {i_{l+1}}^{l+1} \right) i_n^-, i_n^+ \right\},$$

where $i_1^- = s^-$ and $i_n^+ = d^+$.

P' is a feasible constrained path tour from s^- to d^+. In fact, let us suppose that P' is not feasible. This can happen if at least one of the following cases occurs: 1) P' crosses some arcs more than once; 2) P' does not involve any node in some node subsets T_i, $i = 1, \ldots, n+1$; 3) P' involves at least a node for each T_i, , $i = 1, \ldots, n+1$, but not successively and sequentially.

Suppose that P' crosses some arcs twice. Since only nodes of type i^- are such that $|FS(i^-)| > 1$, if some arc is involved at least twice, it must be some arc of type (i^-, i^+). Nevertheless, if this is the case, then necessarily node i must be involved by P at least twice and this contradicts the hypothesis of P as Hamiltonian path.

Finally, cases 2) and 3) can not ever occur by construction. In fact, path P' starts at $s^- \in T_1$ and ends in $d^+ \in T_{n+1}$. Then, it involves successively and sequentially all nodes $i_l {i_{l+1}}^{l+1}$, for each $l = 1, \ldots, n-1$, and each node $i_l {i_{l+1}}^{l+1}$ belongs to T_{l+1}.

⇐ By hypothesis, there exists in G' a feasible constrained path tour from s^- to d^+.

Remember that by construction, it holds that

- for each node $i^- \in V'$, $FS(i^-) = \{i^+\}$;
- for each node $i^+ \in V'$, $FS(i^+) = \{i j^k \mid k = 2, \ldots, n\}$;
- for each node $i j^k \in V'$, $FS(i j^k) = \{j^-\}$.

Therefore, path P' must be necessarily as follows

$$P' = \left\{ \bigoplus_{l=1}^{n-1} \left(i_l^-, i_l^+, i_l i_{l+1}{}^{l+1} \right) i_n^-, i_n^+ \right\},$$

where $i_1^- = s^-$ and $i_n^+ = d^+$.

In fact, if for some $k = 2, \ldots, n$, $k \neq l+1$, P' contains a sub-path

$$i_l, i_{l+1}, i_l i_{l+1}{}^k,$$

then P' would not be feasible, because it would violate the constraint of successively and sequentially passing through at least one node of the node subsets T_i. Finally, if P' involves a smaller number of nodes, then for some subset T_i, no node in T_i would be crossed. Similarly, if P' involves a higher number of nodes, then P' would cross at least one arc more than once.

From Lemma 2.2, it follows that there exists in G a path $P = \{i_1, i_2, \ldots, i_n\}$, such that $i_1 = s$ and $i_n = d$.

P must be Hamiltonian. In fact, let us suppose that P is not Hamiltonian. Since P visits exactly n nodes, there must be i_j and $i_k, j, k \in \{1, \ldots, n \mid j \neq k\}$, such that $i_j = i_k$. But this implies that P' crosses arcs (i_j^-, i_j^+) and (i_k^-, i_k^+) such that $(i_j^-, i_j^+) \equiv (i_k^-, i_k^+)$ and this contradicts the hypothesis of feasibility of the constrained path tour P'.

The Hamiltonian path P in G and the constrained path tour P' in G' have the same length by construction and by the definition of cost functions C and C', respectively. ∎

Corollary 2.4. ***CSPTP*** *is* ***NP****-hard.*

2.2 Solution approaches

2.2.1 An exact method

The first method that has been proposed to solve ***CSPTP*** is based on the B&B technique. To design it, the ***CSPTP*** has been reduced to the *Path Avoiding Forbidden Pairs Problem (****PAFPP****).*

Given a graph $G = (V, A)$ and a set of pairs of nodes

$$F = \{ (a_1, b_1), \ldots, (a_k, b_k) \},$$

where for all $i = 1, \ldots, k$, $a_i \neq b_i$ and $(a_i, b_i) \in (V \times V)$, the ***PAFPP*** consists in finding the shortest path P from a given node s to a given node d, with the constraint that P must contain at most one node of each pair in F.

The ***PAFPP*** admits the following 0-1 integer programming formulation:

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{2.1a}$$

s.t.

$$\sum_{j\in FS(i)} x_{ij} - \sum_{j\in BS(i)} x_{ji} = \begin{cases} 1, & i = s; \\ -1, & i = d; \\ 0, & \text{otherwise}; \end{cases} \tag{2.1b}$$

$$\sum_{j\in BS(a)} x_{ja} + \sum_{j\in BS(b)} x_{jb} \leq 1 \qquad \forall\ (a,b) \in F \tag{2.1c}$$

$$x_{ij} \in \{\, 0, 1 \,\} \qquad \forall\ (i,j) \in A. \tag{2.1d}$$

The objective function (2.1a) minimizes the total length of a path. Constraints (2.1b) represent the flow balance constraint at each node, while constraints (2.1c) guarantee that no forbidden pair is violated.

Any ***CSPTP*** instance

$$\langle G = (V, A, C), s, d, N, \{\, T_h \,\}_{h=1,\dots,N} \rangle$$

can be polynomially transformed into a ***PAFPP*** instance

$$\left\langle G' = (V', A', C'), s, d' = d + (N-1)\cdot m, F = \left\{ (a_1, b_1), \dots, (a_p, b_p) \right\} \right\rangle,$$

where $p = \frac{m(N-2)(N-1)}{2}$ and G' is a multi-stage graph with N stages, each of them replicating G.

The nodes in V' can be classified in *actual* nodes (from 1 to $n \cdot N$) and *dummy* nodes (from $N \cdot n + 1$ to $N \cdot (n + m)$). Each arc of A' connects an actual node to a dummy node (or vice-versa). The actual nodes are used to replicate the paths, dummy nodes are used to preserve feasibility.

For each arc $(v, w) \in A$ and for each $h = 1, \dots, N$, an arc from the actual node $v+n\cdot(h-1)$ to the dummy node i is inserted in A'. In addition, if $w \in T_{h+1}$ arc $(i, w + h \cdot n)$ is inserted in A', while arc $(i, h + n \cdot (k-1))$ is added in A' if $w \notin T_{h+1}$. In practice, for each of the N stages, each arc $(v, w) \in A$ is represented in A' by a pair of arcs: one connects v to a dummy node, and the other one connects the dummy node to w. For each stage h and for each node w, if $w \in T_{h+1}$, then those arcs originate in stage h and end in stage $h + 1$; otherwise their tail node and head node are in the same stage.

The multi-stage graph G' associated to the original graph G shown in Figure 1.1 is depicted in Figure 2.3. The dotted nodes indicate the dummy nodes, whereas pair of dummy nodes, belonging to the same level, represent forbidden pairs. The optimal path

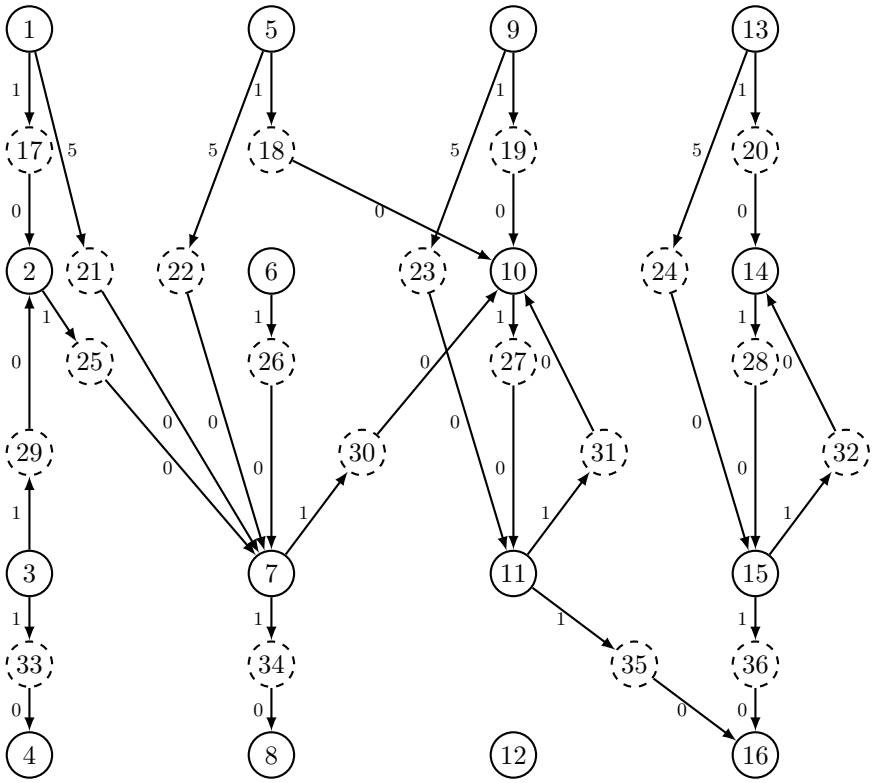


Figure 2.3: *Multistage graph G' associated to the graph G represented in Figure 1.1.*

P' in G' is $P' = \{1, 21, 7, 30, 10, 27, 11, 35, 16\}$, whose length is 8. The corresponding optimal solution in G is $P = \{1, 3, 2, 3, 4\}$ with the same length.

To obtain a lower bound, a relaxation of the ***PAFPP*** is built by deleting constraints (2.1c). The resulting problem is a classical ***SPP***, that can be solved by using any of the well-known methods to address this kind of problem.

The B&B pseudo-code is reported in Algorithm 2.2. To begin, the algorithm expands the graph to obtain the ***PAFPP*** instance, solves it with Dijkstra's algorithm, and if the solution is feasible it is returned as optimal (lines 2 to 5).

Otherwise, it finds the violated nodes and creates the first two subproblems (lines 6 to 10). Until the queue Q of the open subproblems is empty (line 11), a node is extracted from the queue (line 12), the backward stars of all violated nodes are removed (line 13) and Dijkstra algorithm solves the corresponding ***SPP*** (line 14). If the solution is feasible and better than the incumbent, the latter is updated (lines 15 to 17).

Conversely, if the solution is unfeasible, and the solution cost of the relaxed problem is better than the incumbent cost, the branching nodes are generated (lines 18 to 22). The backward stars of the two nodes that violate the constraints (2.1c) are added one each to the two subproblems. Finally, the graph is restored (line 23), in order to solve the open subproblems.

The strategy for selecting the next sub-problem to investigate determines how the

Algorithm 2.2: *Branch & Bound for the* ***CSPTP***.

```
1  Function B&B(G = (V, A, C), s, d, {T_i}_{i=1,...,N})
2      (G' = (V', A', C', s', d'), F) ← Expand(G, s, d, {T_i}_{i=1,...,N});
3      P ← Dijkstra(G');
4      if P is feasible then
5          return (P, C'(P));
6      P* ← Nil; C'(P*) ← +∞;
7      (v_1, v_2) ← Find(P);
8      S_1 ← {BS(v_1)};
9      S_2 ← {BS(v_2)};
10     Q ← {S_1, S_2};
11     while Q ≠ ∅ do
12         S ← Pop(Q);
13         A' ← A' \ S;
14         P ← Dijkstra(G');
15         if P is feasible then
16             if C'(P) < C'(P*) then
17                 P* ← P; C'(P*) ← C'(P);
18         else if C'(P) < C'(P*) then
19             (v_1, v_2) ← Find(P);
20             S_1 ← S ∪ {BS(v_1)};
21             S_2 ← S ∪ {BS(v_2)};
22             Q ← Q ∪ {S_1, S_2};
23         A' ← A' ∪ S;
24     return (P*, C'(P*))
```

B&B algorithm should proceed through the search tree and can have a significant effect on the behaviour of the algorithm. To choose the next node for exploration, the proposed approach adopts both a depth first search (DF, for short) strategy, where the node with the largest level in the search tree is chosen, and a best bound first strategy (BF, for short), that chooses a node with the best relaxed objective function value.

Algorithm complexity

Theorem 2.5. *Let* $n = |V|$ *and* $m = |A|$*, the complexity of each iteration of the B&B approach is* $O(((N-1)(n+m)-1)^2 + 2m(N-1) + (2m(N-1))\log((n+m)\cdot(N-1)))$.

Proof. The algorithm manages an expanded graph $G' = (V', A')$, such that $|V'| = (N-1)(n+m)$ and $|A'| = 2m(N-1)$. At each iteration, the algorithm has to find a violated constrained pair, and it requires $O(((N-1)(n+m)-1)^2)$ to scan twice the path. After, it removes all violated arcs with a complexity of $O(2m(N-1))$ and, finally, executes Dijkstra's algorithm, whose complexity is $O((2m(N-1))\log((n+m)\cdot(N-1)))$. ∎

2.2.2 A GRASP

Owing the computational intractability of the ***CSPTP***, a Greedy Randomized Adaptive Search Procedure (GRASP) has been designed to find optimal or near-optimal feasible solutions.

GRASP is an iterative multi-start meta-heuristic for difficult combinatorial optimization problems [15, 16]. It has been applied to a large set of problems [29, 30, 31], ranging from scheduling and routing to drawing and turbine balancing.

Each GRASP iteration is characterized by two main phases: a construction phase and a local search phase.

The construction phase starts with an empty solution, and iteratively adds one element at a time until a complete solution is obtained. At each iteration, an element is randomly selected from a *restricted candidate list (RCL)*, whose elements are among the best ordered, according to some greedy function that measures the (myopic) benefit of selecting each element.

Once a feasible solution is obtained, the local search procedure attempts to improve it by producing a locally optimal solution with respect to some suitably defined neighborhood structure. Construction and local search phases are repeatedly applied. The best solution found is returned as approximation of the optimal one. Algorithm 2.3 depicts the pseudocode of a generic GRASP heuristic for a minimization problem.

Construction phase

The GRASP construction phase relies on an adaptive greedy function, a construction mechanism for the RCL, and a probabilistic selection criterion. The greedy function takes

Algorithm 2.3: *Pseudo-code of a generic GRASP.*

```
1 Algorithm GRASP(MaxIterations)
2   for i = 1,...,MaxIterations do
3     Build a greedy randomized solution x ;
4     x ← LocalSearch(x) ;
5     if i = 1 then
6       x* ← x;
7     else if x is better than x* then
8       x* ← x;
9   return x* ;
```

into account the contribution to the objective function achieved by selecting a particular element. The construction phase for the ***CSPTP*** is described in Algorithm 2.5. It starts with an empty chain of paths and ends with a complete solution given by a chain of paths from s to d.

At a generic iteration, the choice of the next path to be added is determined by ordering all candidate paths (i.e. those that can be added to the solution) in a candidate list CL, with respect to a greedy function related to the length of the candidate paths, computed by `DijkstraVariant` function, that applies Dijkstra's algorithm taking into account arc repetitions.

The greedy function is adaptive because the benefits associated with every candidate path are updated at each iteration of the construction phase to reflect the changes involved by the selection of the previous element in terms of number of times each arc is used. The probabilistic component is characterized by randomly choosing one of the best candidates in the RCL, but not necessarily the top candidate.

If the number N of subsets of the problem instance to be solved is strictly less than 4, a feasible solution P_T is a chain made of either a shortest simple path from s to d ($N = 2$) or of two simple paths ($N = 3$). In both cases, for building P_T there is no gain in using the RCL mechanism, since it can be constructed by simply invoking the DijkstraVariant function, as done by algorithm in Algorithm 2.4.

If $N \geq 4$, index i is selected at random in $[2, N-1]$ (line 4 of Algorithm 2.5). Path $P_i = \{s_i, \ldots, d_i\}$ is computed from $s_i \in T_i$ to $d_i \in T_{i+1}$, path P_T is initialized with a chain made of only P_i. Then, the partial chain P_T is iteratively augmented both toward the origin s and the destination d. In fact, starting from $j = i+2$ and until a complete feasible solution is obtained, at each iteration of the loop **while** (line 14) two more paths are added in a greedy randomized adaptive fashion: a path from a node in T_{i-1} to s_i and a path from d_{j-1} to a node in T_j.

Algorithm 2.5 uses a $n \times n$ matrix K such that k_{ij} is the number of times arc (i,j) has been involved in the partial solution P_T. Procedures Increase and Decrease update K

Algorithm 2.4: *Construction of a Greedy Randomized Solution when* $N \in \{2, 3\}$.

```
1  Function SimpleConstruction(V, A, s, d, N, {T_i}_{i=1,...,N}, C, K)
2      if N = 2 then
3          return DijkstraVariant(V, A, s, d, C, K);
4      else
5          P_T ← Nil ;
6          min ← +∞ ;
7          for v ∈ T_2 do
8              (l_1, P_1) ← DijkstraVariant(V, A, s, v, C, K);
9              if l_1 < +∞ then
10                 Increase(P_1, K);
11                 (l_2, P_2) ← DijkstraVariant(V, A, v, d, C, K);
12                 if l_1 + l_2 < min then
13                     P_T ← P_1 ⊕ P_2;
14                     min ← l_1 + l_2;
15                 Decrease(P_1, v, K);
16         return (min, P_T);
```

to reflect the choices made by the construction algorithm.

Local search

At each GRASP iteration, once obtained a path tour P_T from the construction phase, a local search procedure is applied starting from P_T in the attempt of improving it by producing a locally optimal solution with respect to a suitably defined neighborhood structure.

For the ***CSPTP***, given two solutions $P_T = P_1 \oplus \cdots \oplus P_{N-1}$ and $R_T = R_1 \oplus \cdots \oplus R_{N-1}$ their symmetric difference set is defined as follows:

$$\Delta(P_T, R_T) = \{\, i = 1, \ldots, N-1 \mid P_i \neq R_i \,\}.$$

Coherently, the distance between P_T and R_T can be defined as:

$$d(P_T, R_T) = |\Delta(P_T, R_T)|.$$

The pseudo-code of the local search procedure we have designed for the ***CSPTP*** is reported in Algorithm 2.6. It takes as input the GRASP path tour P_T and outputs a local optimal feasible tour with the respect to the neighborhood $\mathcal{N}(P_T)$, defined as follows:

$$\mathcal{N}(P_T) = \{\, R_T \mid d(P_T, R_T) \leq 2 \,\}.$$

The **while** loop (lines 3 to 20) stops as soon as any improving solution in the neighborhood of the current solution cannot be found. At each iteration, given the current

Algorithm 2.5: *Construction of a Greedy Randomized Solution for the* ***CSPTP***.

```
1  Function Construction(V, A, s, d, N, {T_i}_{i=1,...,N}, C, K, a)
2      if N < 4 then
3          return SimpleConstruction(V, A, s, d, N, {T_i}_{i=1,...,N}, C, K) ;
4      i ← Rand(2, N-2);   P_T ← Nil;   CL ← ∅;
5      for v ∈ T_i do
6          for w ∈ T_{i+1} do
7              (l, P) ← DijkstraVariant(V, A, v, w, C, K);
8              CL ← CL ∪ { (l, P, v, w) };
9      RCL ← MakeRCL(a);   (l, P, v, w) ← Select(RCL) ;
10     P_i ← P; P_T ← P_T ⊕ P_i;
11     s_i ← v; d_i ← w;
12     Increase(P_i, K) ;
13     j ← i + 2; i ← i − 1 ;
14     while (i > 0) ∨ (j < N) do
15         CL ← ∅;
16         if i > 1 then
17             for v ∈ T_i do
18                 (l, P) ← DijkstraVariant(V, A, v, s_{i+1}, C, K) ;
19                 CL ← CL ∪ { (l, P, v, s_{i+1}) } ;
20             else if i = 1 then
21                 (l, P) ← DijkstraVariant
22             V, A, s, s_2, C, K ;
23             CL ← CL ∪ { (l, P, s, s_2) } ;
24         RCL ← makeRCL(a);   (l, P, v, w) ← Select(RCL) ;
25         P_i ← P; P_T ← P_i ⊕ P_T ;
26         s_i ← v; d_i ← w ;
27         Increase(P_i, K) ;
28         CL ← ∅ ;
29         if j < N then
30             for v ∈ T_j do
31                 (l, P) ← DijkstraVariant(V, A, d_{j−1}, v, C, K) ;
32                 CL ← CL ∪ { (l, p, d_{j−1}, v) } ;
33             else if j = N then
34                 (l, P) ← DijkstraVariant(V, A, d_{j−1}, d, C, K) ;
35                 CL ← CL ∪ { (l, P, d_{j−1}, d) } ;
36             RCL ← makeRCL(a);   (l, P, v, w) ← Select(RCL) ;
37             P_j ← P; P_T ← P_T ⊕ P_j ;
38             s_j ← v; d_j ← w ;
39             Increase(P_j, K) ;
40             j ← j + 1; i ← i − 1 ;
41         return P_T;
```

Algorithm 2.6: *Local Search for the* ***CSPTP***.

```
1  Function LocalSearch(pt, V, A, s, d, N, {T_i}_{i=1,...,N}, C, K)
2      flag ← true;
3      while flag = true do
4          flag ← false;
5          for i ← 2 to N − 1 do
6              Decrease(P_{i−1}, K) ;
7              Decrease(P_i, K) ;
8              min ← L(P_{i−1}) + L(P_i) ;
9              for v ∈ T_i do
10                 (l′, P′) ← DijkstraVariant(V, A, s_{i−1}, v, C, K) ;
11                 Increase(P′, K) ;
12                 (l″, P″) ← DijkstraVariant(V, A, v, d_i, C, K) ;
13                 Decrease(P′, K) ;
14                 if l′ + l″ < min then
15                     min ← l′ + l″ ;
16                     P_{i−1} ← P′ ;
17                     P_i ← P″ ;
18                     flag ← true ;
19             Increase(P_{i−1}, K);
20             Increase(P_i, K);
21     return P_T ;
```

solution $P_T = \bigoplus_{i=1}^{N-1} P_i$, iteratively for $i = 2, \ldots, N-1$ it checks whether $P_{i-1} \oplus P_i$ can be substituted by any shorter $P' \oplus P''$, where P' originates in s_{i-1} and ends in some node v in T_i and P'' originates in v and ends in d_i (lines 9 to 18).

The neighborhood is explored with a best improvement strategy.

2.2.3 Experimental results

To analyze the performance of the proposed algorithms, a large amount of computational experiments were performed. The algorithms have been coded in *C++ language* and run on a *Intel Core i7 Quad core @ 2.67 GHz* processor, under the *Linux* (*Ubuntu 11.10*) operating system.

The objective of the computational study has been to compare the running times achieved by the algorithms as a function of the parameter N, when applied on several different networks, with different densities and number of nodes. All test problems have been pseudo-randomly generated by using a generator proposed by Festa and Pallottino [28]. For a detailed description of how such instances are generated, the reader is referred to Festa [26].

B&B has been implemented using two different strategies to build the branching tree: *depth first* (`BBdf`) and *best bound first* (`BBbf`). The criterion used to stop `GRASP` has been `MaxIterations` = 100.

For each problem family, ten different instances have been generated and the mean running time (in seconds) has been computed and stored. The quality of the solutions determined by `GRASP` has been evaluated by computing the mean relative error $\epsilon = \frac{z'-z^*}{z^*}$, where z' is the objective function value corresponding to the suboptimal solution and z^* is the optimal value.

In the following, we use γ to denote the number of nodes belong to some T_i.

Complete graphs

A first set of experiments involves complete graphs with the goal of analyzing how running times of B&B and `GRASP` vary depending on the number n of nodes. The number of sets T_i is $N = .25n$ and $\gamma = .40n$ nodes belong to some T_i.

Looking at the results reported in Table 2.1, it is evident that `GRASP` is a robust heuristic, able to find an optimal solution in very limited running times compared with those required by B&B. Furthermore, for most instances `BBdf` slightly outperforms `BBbf`.

On complete graphs with more than 260 nodes only `GRASP` could be tested, because both B&B implementations failed to solve the problem. Figure 2.4 plots the running times required by `GRASP` to solve instances on complete graphs with $n \in \{300, 350, 400, 450, 500\}$ and $N = .25n$. Looking at the results, it emerges that in a reasonable running time, medium-large size problem instances can be managed.

Table 2.1: *Complete graphs:* $N = .25n$ *and* $\gamma = .40n$.

	BBbf	BBdf	GRASP	
n	Time	Time	Time	ϵ
100	1.09	1	1.27	0
150	4.2	4.2	4.72	0
200	14.8	15	11.57	0
250	50.36	32	21.59	0
252	54	65.6	23.55	0
254	213.9	122.5	24.45	0
256	255.9	225.6	24.04	0
258	197.1	186.7	24.20	0
260	371.6	370.5	25.82	0

In order to assess the behaviour of the algorithms as a function of the number N of node subsets, another computational experiment was performed on complete graphs with $n \in \{100, 150\}$, $N \in \{\, .2n, .4n, .6n, .7n, .75n, .8n \,\}$, and $\gamma = .8n$. Table 2.2 shows the mean running time for both B&B implementations to find an optimal solution. For `GRASP`, the mean relative error and the mean running time to perform 100 iterations are reported.

Table 2.2: *Complete graphs,* $\gamma = .8n$.

	BBbf	BBdf	GRASP	
n, N	Time	Time	Time	ϵ
100, 20	0.6	0.8	3	0
100, 40	2.1	2.3	2.6	0
100, 60	5.4	6.3	2.7	0
100, 80	22.21	43.35	1.57	0.001
150, 20	3.1	3.1	10	0.005
150, 40	9.8	10	9.7	0.0005
150, 60	55.4	63.9	8.4	0.00004
150, 70	96.4	110.8	8.5	0.0007
150, 75	282.7	315.1	7.6	0.001

From the results, it is evident that the running time of `BBbf` and `BBdf` increases with N. This behaviour can be explained by observing that the size of the expanded graph G' depends on N, and therefore to build it a higher computational cost is required. With respect to `GRASP`, the higher the number N the lower the mean running time.

For a deeper analysis of this behaviour, Figure 2.5 plots the mean running times required by `GRASP` to find the best local optimal solution over `MaxIterations` = 100 iterations for complete graphs with $n = 200$ and $N \in \{\, .1n, .2n, .3n, .4n, .5n, .6n, .7n, .8n, .9n, n \,\}$.

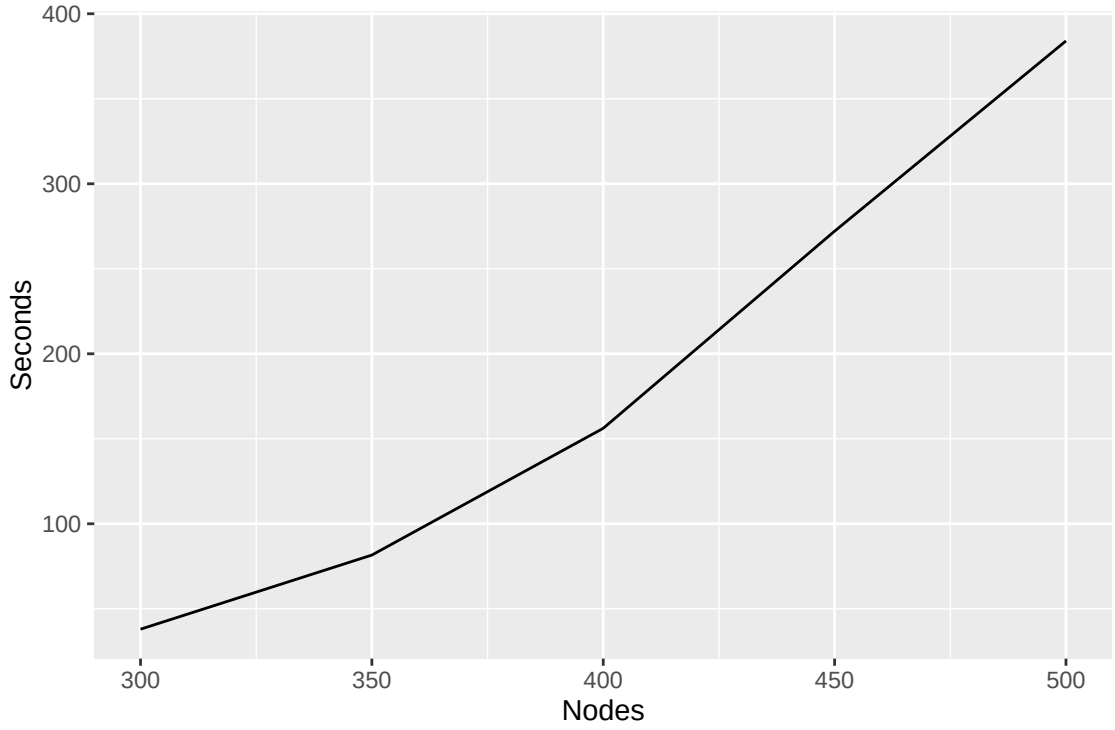


Figure 2.4: *Mean running times (over ten trials) required by* `GRASP` *to find the best solution for complete graphs with* $N = .25n$ *and* $n \in \{300, 350, 400, 450, 500\}$.

Looking at the results, for $N \sim n$ the mean running time is very low, less than about 30 seconds. This behaviour can be explained by observing that the higher N, the lower the cardinality of each T_i, $i = 1, \ldots, N$, which impacts both the construction and local search phases. In the construction phase, a smaller number of iterations is performed each time a new path must be found to be added to the partial chain. In the local search, a smaller effort must be spent to explore the lower cardinality neighborhood of the current solution.

On complete graphs with $n = 200$, $N = 10$, and

$$\gamma \in \{\, .1n, .2n, .3n, .4n, .5n, .6n, .7n, .8n, .9n, n \,\},$$

a further experiment has been performed, whose results are shown in Figure 2.6. The figure plots the mean running times required by B&B implementations to find an optimal solution and the mean running time required by `GRASP` to find the best local optimal solution.

Looking at the figure, it is evident that the running time of `GRASP` varies proportionally to γ. This result confirms the conclusion drawn above about the relationship between the cardinality of node subsets and running time to perform each local search and construction iterations. On the contrary, B&B computing time does not depend on γ. This behaviour can be explained noting that the size of the expanded graph is exactly the same.

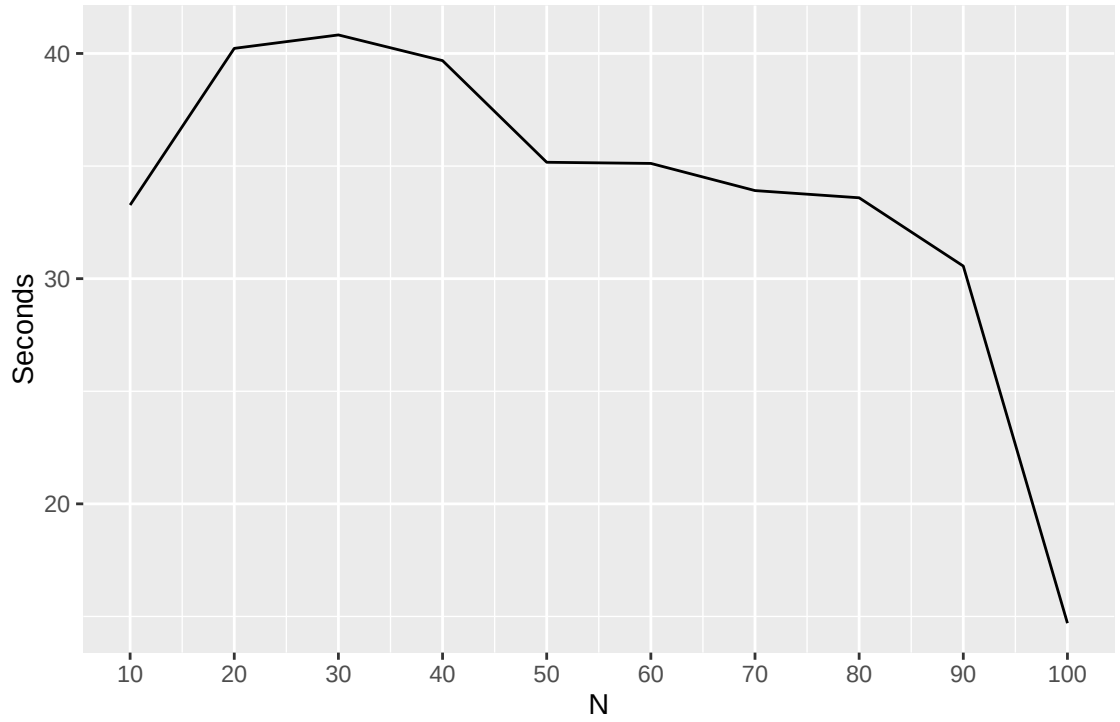


Figure 2.5: *Mean running times (over ten trials) required by* `GRASP` *to find the best solution for complete graphs with* $n = 200$ *and* $N \in \{.1n, .2n, .3n, .4n, .5n, .6n, .7n, .8n, .9n, n\}$.

Random graphs

Table 2.3 reports running times on random graphs with $n = 100$ and $m \in \{4n, 8n, 16n, 32n\}$. For these instances, `GRASP` found high quality solutions with a mean relative error ϵ less than or equal to 0.004. Running times of `GRASP` and B&B implementations are competitive, with the exception of sparse graphs with 400 arcs. This is not surprising because in case of sparse graphs B&B techniques need to perform a high number of branching operations. Contrary to the complete graphs case, the `BBdf` is outperformed by `BBbf`. This behaviour is explained by observing that in the sparse case there are less feasible solutions. Therefore, it is not worthwhile applying a depth strategy that tends to rapidly

Table 2.3: *Random graphs:* $n = 100$, $N = .24n$, *and* $\gamma = .40n$.

	`BBbf`	`BBdf`	`GRASP`	
m	Time	Time	Time	ϵ
400	24.9	85.9	0.2	0.004
800	0.7	2	0.22	0.001
1600	0.2	0.5	0.32	0
3200	0.3	0.4	0.53	0.0004

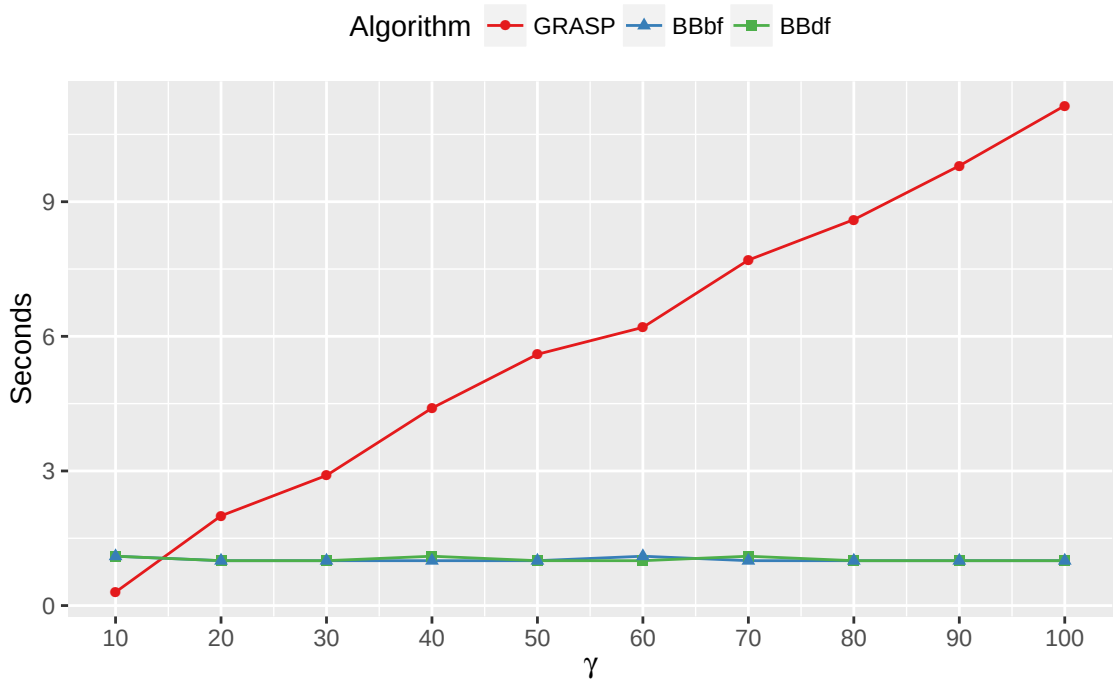


Figure 2.6: *Mean running times (over ten trials) for complete graphs with* $n = 150$, $N = 15$ *and* $\gamma \in \{.1n, .2n, .3n, .4n, .5n, .6n, .7n, .8n, .9n, n\}$.

find complete solutions that in this case can be infeasible with high probability.

Summarizing, the higher m the lower the computational effort, since the higher the density of the graph, the higher the number of paths connecting two nodes. Consequently, it is less likely that any arc constraint is violated. In other words, the denser the graph, the lower the number of the generated subproblems and coherently the depth of the corresponding branching tree.

To better analyze the performances of the two B&B implementations compared to the `GRASP`, additional experiments have been carried out on difficult instances solved within a very small computational time, that is on sparse random graphs and letting all the algorithms run for a fixed amount of time equal to 3 seconds.

For both B&B implementations, Table 2.4 reports in the second and third columns the mean running times to find an optimal solution (columns `BBbf` Time and `BBdf` Time). In the fourth, the fifth, and the sixth columns (`BBbf_time` ϵ, `BBdf_time` ϵ, and `GRASP_time` ϵ) it reports the mean relative errors of two B&B implementations and of `GRASP` after 3 seconds of running time, respectively. The symbol "—" means that the related approach is not able to find a feasible solution within the time limit of 3 seconds, for each of the considered instances.

It is evident that with the same amount of time `BBdf` has a higher probability to find a

Table 2.4: *Random graphs,* $N = .20n$ *and* $\gamma = .40n$.

Size (n, m)	BBbf Time	BBdf Time	BBbf_time ϵ	BBdf_time ϵ	GRASP_time ϵ
100, 500	0.1	0.8	0	0	0.001
120, 600	3	7.4	—	0.009	0.0005
140, 700	34.3	166.8	—	0.058	0.005
160, 800	86.2	271	—	0.068	0.003
180, 900	25.4	386.6	—	0.046	0.006

feasible solution than does BBbf. Except for the case $n = 100, m = 500$, the version of the GRASP that is executed for only 3 seconds (i.e., GRASP_time) has always a mean relative error lower than BBdf_time. This confirms the robustness of GRASP for sparse graphs.

Grid graphs

A final set of experiments involves two types of grid graphs: square grids with side length $l \in \{9, 10, 12, 14, 20, 25\}$ (i.e., $n \in \{81, 100, 144, 196, 400, 625\}$) and elongated grids with shortest side length $l_1 \in \{4, 5, 7, 10, 12\}$ and highest side length $l_2 \in \{15, 20, 25, 40, 50\}$. The number of sets T_i is $N = \in \{.15n, .16n, .17n, .18n, .19n\}$ and $\gamma = .35n$. For each grid type and for each size, we have computed the average running times employed by BBbf and GRASP over 100 different randomly generated instances. The criterion adopted to stop GRASP has been either the achievement of an optimal solution or, in the worst case, the achievement of the time limit equal to the one required by BBbf.

For small sized squared and elongated grids, the results obtained are summarized in Tables 2.5 and 2.6, where for each problem type, the instance size is reported in the first column. The second and third columns report BBbf and GRASP mean running times, respectively. The remaining columns report the number of instances optimally solved by GRASP out of 100 versus those solved by BBbf, and the mean relative error ϵ computed over all instances solved by BBbf.

The results obtained confirm that the B&B method can only be used to solve small sized instances of the problem, since only for three square grid type graphs (i.e., 09×09N15, 09×09N16, and 09×09N17) it found an optimal solution for all 100 instances in reasonable running times.

For the same three square grid type graphs, GRASP found an optimal solution for 95, 96, and 91 out the 100 instances, respectively. For 20 square grid type graphs, GRASP found high quality solutions with a mean relative error ϵ less than or equal to 0.002.

Running times of GRASP are lower by several order of magnitude compared to BBbf. This is not surprising because grid graphs are sparse and, as already shown in other experiments, in case of sparse graphs B&B techniques perform a large number of branching

Table 2.5: *Small sized square grids:* $l \in \{9, 10\}$, $N \in \{.15n, .16n, .17n, .18n, .19n\}$, *and* $\gamma = .35n$.

Instance	BBbf time	GRASP time	# GRASP optima	ϵ
$09 \times 09N15$	0.30	0.12	95 / 100	0.0004
$09 \times 09N16$	0.33	0.14	96 / 100	0.0004
$09 \times 09N17$	1.23	0.19	91 / 100	0.0008
$09 \times 09N18$	4.34	1.17	78 / 99	0.0026
$09 \times 09N19$	6.03	2.07	71 / 98	0.0022
$10 \times 10N15$	4.57	0.94	87 / 99	0.0005
$10 \times 10N16$	18.06	3.54	77 / 92	0.0012
$10 \times 10N17$	19.35	4.32	70 / 85	0.0011
$10 \times 10N18$	36.48	10.47	62 / 84	0.0020
$10 \times 10N19$	42.39	13.93	42 / 58	0.0016

Table 2.6: *Small sized elongated grids:* $l_1 \in \{5, 7\}$ *and* $l_2 \in \{15, 20\}$, $N \in \{.15n, .16n, .17n, .18n, .19n\}$, *and* $\gamma = .35n$.

Instance	BBbf time	GRASP time	# GRASP optima	ϵ
$05 \times 20N15$	16.2614	4.3863	70 / 88	0.0010
$05 \times 20N16$	28.2877	6.9315	60 / 73	0.0010
$05 \times 20N17$	24.4222	4.1111	41 / 45	0.0004
$05 \times 20N18$	68.3824	20.4412	26 / 34	0.0015
$05 \times 20N19$	62.0435	19.0435	17 / 23	0.0021
$07 \times 15N15$	8.1649	1.3402	86 / 97	0.0006
$07 \times 15N16$	111.6440	18.9333	77 / 90	0.0009
$07 \times 15N17$	21.8861	6.1392	62 / 79	0.0014
$07 \times 15N18$	32.9524	5.6031	56 / 63	0.0003
$07 \times 15N19$	39.4792	12.7292	36 / 48	0.0018

operations.

2.3 A more sophisticated exact approach

The main disadvantage of the presented B&B is that it needs to expand the graph G to a graph G' that represents the ***PAFPP*** instance. Indeed, the expanded graph is much larger than the original one, and B&B cannot solve medium instances because the expanded graph overflows memory limits.

To overcome this limitation, Ferone et al. [18] proposed an integer mathematical program that can model the ***CSPTP*** on its original graph, and allows the design of a new B&B algorithm that has better performance.

The basic observation that lies behind this new approach is that each sub-path from T_k to T_{k+1}, for each $k = 1, \ldots, N-1$, must be simple. Otherwise, it is possible to remove the cycle, obtaining a feasible solution with lower cost. Therefore, ***CSPTP*** can be viewed as a concatenation of $N-1$ simple paths, where the generic k-th path starts in a node of T_k and ends in a node of T_{k+1}.

2.3.1 Mathematical formulation

By defining for sub-path $k = 1, 2, \ldots, N-1$ and for each arc $(i,j) \in A$, the binary variables:

$$x_{ij}^k = \begin{cases} 1, & \text{if arc } (i,j) \text{ belongs to the simple path from } T_k \text{ to } T_{k+1}; \\ 0, & \text{otherwise,} \end{cases}$$

the ***CSPTP*** can be formulated as the following 0-1 integer linear program:

$$\min \sum_{(i,j)\in A} \sum_{k=1}^{N-1} c_{ij} x_{ij}^k \tag{2.2a}$$

s.t.

$$\sum_{k=1}^{N-1} x_{ij}^k \leq 1 \qquad \forall\, (i,j) \in A \tag{2.2b}$$

$$\sum_{j\in FS(s)} x_{sj}^1 = 1 \tag{2.2c}$$

$$\sum_{j\in BS(s)} x_{js}^1 = 0 \tag{2.2d}$$

$$\sum_{j\in FS(s)} x_{sj}^k - \sum_{j\in BS(s)} x_{js}^k = 0 \qquad \forall\, k = 2, \ldots, N-1 \tag{2.2e}$$

$$\sum_{j\in BS(d)} x_{jd}^{N-1} = 1 \tag{2.2f}$$

$$\sum_{j\in FS(d)} x_{dj}^k - \sum_{j\in BS(d)} x_{jd}^k = 0 \qquad \forall\, k = 1, \ldots, N-2 \tag{2.2g}$$

$$\sum_{j \in FS(i)} x_{ij}^k - \sum_{j \in BS(i)} x_{ji}^k = 0 \qquad \forall\, i \in T_1 \cup T_N \setminus \{s, d\}, \forall\, k = 1, \ldots, N-1 \tag{2.2h}$$

$$\sum_{j \in FS(i)} x_{ij}^l - \sum_{j \in BS(i)} x_{ji}^l = 0 \qquad \forall\, l = 2, \ldots, N-1,\ \forall\, k \neq l, \forall\, i \notin T_k \tag{2.2i}$$

$$\sum_{j \in FS(i)} x_{ij}^l - \sum_{j \in BS(i)} x_{ji}^l - \sum_{j \in BS(i)} x_{ji}^{l-1} = 0 \qquad \forall\, l = 2, \ldots, N-1,\ \forall\, k \neq l,\ \forall i \in T_k \tag{2.2j}$$

$$\sum_{j \in FS(i)} x_{ij}^k - \sum_{j \in BS(i)} x_{ji}^k = 0 \qquad \forall\, k = 1, \ldots, N-1, \forall\, i \notin \cup_{l=1}^{N-1} T_l \tag{2.2k}$$

$$x_{ij}^k \in \{0, 1\} \qquad \forall\ (i,j) \in A, \forall\, k = 1, \ldots, N-1. \tag{2.2l}$$

Objective function (2.2a) represents the total length of the solution path to be minimized. Constraints (2.2b) ensure that each arc $(i,j) \in A$ is included in the path tour at most once. Constraints (2.2c) and (2.2d) impose that the path tour starts at the source node s.

Since the source node s can be visited again in sub-paths $i = 2, \ldots, N-1$, conditions (2.2e) represent the flow balance constraints at source node, whereas constraints (2.2f) and (2.2g) are the specular flow balance constraints at the destination node.

Constraints (2.2h) impose that the nodes belonging to $T_1 \cup T_N$ cannot be neither the destination nor the source node of any intermediate simple path. Constraints (2.2i) and (2.2j) guarantee that any node belonging to T_k, for some $k = 2, \ldots, N-1$, can be the destination node of the sub-path $k-1$ and the source node of the sub-path k, but it cannot be neither destination nor source node for other sub-paths.

Finally, conditions (2.2k) represent the flow balance constraints at each node that does not belong to any T_k.

2.3.2 An advanced exact approach

Relying on the mathematical formulation (2.2), a new Branch & Bound algorithm (referred in sequel as B&B^{new}) was proposed. This new algorithm is more efficient than the algorithm presented in Section 2.2.1.

A path tour is a concatenation of simple paths between T_i and T_{i+1}, $\forall\, i = 1, \ldots, N-1$. A generic node P_t of the branching tree corresponds to a subproblem of the original ***CSPTP*** problem. By relaxing per elimination constraints (2.2b), the relaxed subproblem P_t^r is a shortest path tour problem that can be polynomially solved. Once optimally solved P_t^r, if its optimal solution results infeasible for P_t, there must exist a pair of indices i and j such that the P_t^r solution path crosses at least twice an arc (v, w), one time in the sub-path from T_i to T_{i+1}, the other time in the sub-path from T_j to T_{j+1}. In this case, B&B^{new} generates two new branching nodes: it is imposed that the solution must not include the arc (v, w) in the sub-path from T_i to T_{i+1} and in the sub-path from T_j to T_{j+1}, respectively.

To efficiently address the solution of each subproblems, each branching tree node uses

the following data structures:

- an $N-1$-dimensional array *paths* that stores in position i, $i = 1, \ldots, N-1$, all paths from each node in T_i to all nodes in T_{i+1};
- an $N-1$-dimensional array *constraints* that stores in position i, $i = 1, \ldots, N-1$, all the arcs that cannot be used in the solution for the sub-path from T_i to T_{i+1};
- an index i that identifies the sub-path that must be recomputed.

Algorithm 2.7: *Function that generates a new branching tree node.*

```
1 Function GenerateNode(paths, constraints, i, (v, w))
2     Node.paths ← paths ;
3     Node.constraints ← constraints ;
4     Node.constraints[i] ← Node.constraints[i] ∪ {(v, w)} ;
5     Node.paths[i] ← ∅ ;
6     Node.index ← i ;
7     return Node ;
```

The pseudo-code of B&B^{new} is reported in Algorithm 2.8. To generate a new branching node, it invokes the function described in Algorithm 2.7. B&B^{new} starts by computing a shortest path between all pairs of vertices (line 2). Then, it looks for the best path tour that can be constructed with the computed shortest paths. If the resulting tour is feasible, then it is also optimal for the ***CSPTP*** and it is returned by the algorithm (lines 3 to 5). Otherwise, all shortest paths are stored for later use (lines 6 to 9). At line 10, the algorithm finds an arc (v, w) that is repeated in the solution, since it belongs both to the path from T_i to T_{i+1} and to the path from T_j to T_{j+1}. Hence, two branching nodes are generated. In particular, the node related to the index i

- saves the constraint, imposing that the arc (v, w) cannot belong to the path from T_i to T_{i+1};
- discards the paths from T_i to T_{i+1};
- memorizes the index i.

Similar information are stored at the node corresponding to index j (lines 11 to 14).

The algorithm iterates until the queue Q of current active nodes becomes empty (line 16). At each iteration, the algorithm selects a node and removes from the graph the constrained arcs (lines 17 to 19), computes a shortest path between all nodes in T_i and T_{i+1}, and then applies a dynamic programming algorithm (described in the following) to find the best path tour (lines 20 to 23). Afterwards, the graph is restored (line 24). The

Algorithm 2.8: *New Branch & Bound algorithm.*

```
1  Function B&Bnew(G = (V, A, C), s, d, {T_i}_{i=1,...,N})
2      ShortestPaths ← FloydWarshall(G) ;
3      x ← DP(V, A, s, {T_i}_{i=1,...,N}, ShortestPaths) ;
4      if x is feasible then
5          return (x, z(x)) ;
6      for i ← 1 to N − 1 do
7          for v ∈ T_i do
8              for w ∈ T_{i+1} do
9                  Paths[i] ← Paths[i] ∪ {ShortestPaths[v][w]} ;
10     ((v, w), i, j) ← Find(x) ;
11     Node_1 ← GenerateNode(Paths, [∅]_{i=1}^{N−1}, i, (v, w)) ;
12     Node_2 ← GenerateNode(Paths, [∅]_{i=1}^{N−1}, j, (v, w)) ;
13     k ← 2 ;
14     Q ← {Node_1, Node_2} ;
15     x* ← Nil, z(x*) ← +∞ ;
16     while Q is not empty do
17         Node ← Pop(Q) ;
18         i ← Node.index;
19         A ← A \ Node.costraints[i] ;
20         for v ∈ T_i do
21             for w ∈ T_{i+1} do
22                 Node.paths[i] ← Node.paths[i] ∪ {Dijkstra(G, v, w)};
23         x ← DP(Node.paths) ;
24         A ← A ∪ Node.costraints[i] ;
25         if x is feasible then
26             if z(x) < z(x*) then
27                 x* ← x; z(x*) ← z(x);
28         else if z(x) < z(x*) then
29             k ← k + 2;
30             ((v, w), i, j) ← Find(x);
31             Node_{k−1} ← GenerateNodes(Node.paths, Node.constraints, i, (v, w));
32             Node_k ← GenerateNodes(Node.paths, Node.constraints, j, (v, w));
33             Q ← Q ∪ {Node_k, Node_{k−1}};
34     return (x*, z(x*)) ;
```

last part of the algorithm is concerned with examining the solution. The following two cases can occur: 1) the current solution is feasible; 2) the current solution is infeasible. In the first case (line 25), if the current solution is better than the incumbent solution, the latter is updated. In the second case, if the bounding criterion is not satisfied, two branching nodes are generated (lines 28 to 33). As soon as the queue $\mathcal{Q}$ becomes empty, the incumbent solution is returned as final result (line 34).

An interesting property of this approach is that it does not need to recompute from scratch the entire path tour at each node. In fact, when the analyzed node contains the index i, only the sub-path from T_i to T_{i+1} is redetermined, by computing the shortest paths between T_i and T_{i+1}. Generally speaking, this leads to a reduction of the running time needed to solve the subproblem associated to each node.

Dynamic programming algorithm

The main operations executed by the dynamic programming algorithm implemented to solve the ***SPTP*** problem are given in Algorithm 2.9.

Algorithm 2.9: *Dynamic programming algorithm for the solution of* ***SPTP***.

```
1  Function DP(V, A, s, {T_i}_{i=1,...,N}, L[,])
2      for l ← 1 to 2 do
3          for w ∈ T_l do
4              C[w] ← L[s, w];
5              if L[s, w] < +∞ then
6                  Pred[w] ← s;
7              else
8                  Pred[w] ← Nil
9      for l ← 3 to N do
10         for w ∈ T_l do
11             C[w] ← +∞;
12             Pred[w] ← Nil;
13             for z ∈ T_{l−1} do
14                 if C[z] + L[z, w] < C[w] then
15                     C[w] ← C[z] + L[z, w];
16                     Pred[w] ← z;
17     return (C, Pred)
```

Let L be the matrix of the shortest path costs between all pairs of nodes. The first lines 2 to 8 compute the best path between each node $v \in T_2$ and s, then in lines 9 to 16, for each $l \in 3, \ldots, N$, the best path tour is computed from the nodes in T_{l-1} to the current node $v \in T_l$. At the end, in $C[d]$ there will be the cost of the optimal solution, and the array *Pred* contains the sequence of nodes in T_i, for $i = 1, \ldots, N$.

2.3.3 Experimental results

Test environment

The B&B^{new} has been implemented in C++ and compiled into bytecode with g++ (Ubuntu 5.2.1-22ubuntu2) 5.2.1 Flag: -std=c++14. Running times reported are UNIX real wall-clock times in seconds, excluding the time to read the instance. The random number generator used was Matsumoto and Nishimura's *Mersenne Twister* [46].

All experiments were run on S.Co.P.E.[1], a cluster of nodes, connected by 10 Gigabit Infiniband technology, each of them with two processors Intel Xeon E5-4610v2@2.30 Ghz. Each execution was limited to a single processor.

Test Problems

Three classes of pseudo-randomly generated instances are considered to assess the behaviour of the considered solution approaches: complete, random sparse, and grid graphs. All test problems are built by using the generator proposed by Festa and Pallottino [28].

Networks with different density values and number of nodes have been taken into account. For all networks, the arc costs are chosen according to a uniform distribution in the range from $[10, 100]$.

Complete Graphs Nine complete graphs are considered. Each of these test problems, referred to as $C1, \ldots, C9$, in the sequel, includes all the possible $n(n-1)$ arcs. The number of nodes are $n \in \{100, \ldots, 500\}$ with a step of 50.

Grid Graphs Six grid graphs have been considered (i.e., $G1, \ldots, G6$), whose characteristics are given in Table 2.7. The nodes are arranged in a planar grid. Each pair of adjacent nodes are connected in both directions.

Random Sparse Graphs A set of 10 random graphs of different size (named $R1, \ldots, R10$, in the sequel) have been generated. The number of nodes n has been set equal to $250, 500$ and the number of arcs m has been chosen as $\{0.1, 0.2, 0.3, 0.4, 0.5\} \cdot n(n-1)$.

Table 2.7: *Characteristics of the Grid Networks.*

Problem	Dimension	Nodes	Arcs
G1	5×10	50	170
G2	10×20	200	740
G3	15×30	450	1710
G4	5×5	25	80
G5	10×10	100	360
G6	15×15	125	840

[1] S.Co.P.E. is a computing infrastructure at the University of Napoli FEDERICO II.

Algorithm 2.10: *Generation algorithm.*

```
1  Function GenerateGraph(type, n, m, a, γ, seed)
2      SetSeed(seed);
3      Generate G = (V, A, C) with the generator by [28] using parameters (type, n, m);
4      N ← ⌊a · n⌋; γ ← ⌊γ · n⌋;
5      Z ← ∅;
6      s ← Rand(V \ Z); Z ← Z ∪ {s};
7      d ← Rand(V \ Z); Z ← Z ∪ {d};
8      T_1 ← {s}; T_N ← {d};
9      for i ← 2 to N − 1 do
10         v ← Rand(V \ Z);
11         T_i ← {v}; Z ← Z ∪ {v};
12     for i ← N + 1 to γ do
13         v ← Rand(V \ Z);
14         i ← Rand(1, N);
15         T_i ← T_i ∪ {v}; Z ← Z ∪ {v};
16     return (G, s, d, {T_i}_{i=1,...,N});
```

For each of the aforementioned network, a set of ***CSPTP*** instances has been defined by choosing the number N of subsets T_i as a percentage a of the number of nodes n. In particular, a has been set equal to 0.10 and 0.25. In addition, the size of the subsets T_i has been defined, by selecting randomly a percentage γ of the number of nodes and inserting them into the subsets T_i, according to the procedure outlined in Algorithm 2.10.

Firstly, the weighted graph is generated with the generator by Festa and Pallottino [28] (line 3). Then the source and destination nodes randomly picked in non constrained nodes and inserted in T_1 and T_N, respectively (lines 6–8). To ensure that all subsets T_i are non-empty, a random node is assigned to each subset T_i, $i = 2, \ldots, N-1$ (lines 9–10). Finally, until the instance is complete, a random node is added in a random set T_i. All RAND calls use a uniform distribution between the candidates.

In the computational experiments, γ has been chosen equal to 0.35 and 0.70. For each problem type, ten different instances have been generated. The average running time (in seconds) has been computed and stored.

Algorithms

The following algorithms and models are compared:

- M1: the mathematical model of ***PAFPP*** on the expanded graph presented in Section 2.2.1;
- B&B: the Branch & Bound algorithm presented in Section 2.2.1;
- GRASP: the GRASP metaheuristic (Section 2.2.2);

- M2: the mathematical model presented in Section 2.3.1;
- B&B^{new}: the Branch & Bound algorithm presented Section 2.3.2.

Empirical evaluation of the mathematical models

In this section, the mathematical models M1 and M2 are compared, in terms of efficiency. In particular, the number of times in which each model is solved (i.e, either the optimal or a feasible solution is found), on the 10 different instances, are chosen as statistics to evaluate the performance of the considered models. In the computational experiments, a time limit of 30 minutes on the solver execution time was imposed.

The performance profiles for both models M1 and M2 are highlighted in Figures 2.7 and 2.8, where the computational results collected on random sparse and complete graphs are reported, respectively. Two plots are reported for each set: one related to the optimal solution and the other to the feasible solution. On the grid graphs, the results of both the models are equal, except for the case G3 $a = 0.1$, $\gamma = 0.7$. In this case, M1 is optimally solved on 10 instances, conversely with M2 the optimum is reached in 8 cases.

The collected computational results clearly underline that the mathematical formulation M2 outperforms M1 on random and complete graphs. From Figures 2.7 and 2.8, it is evident that M2 is solved (i.e, either the optimal or a feasible solution is obtained) a greater number of times respect to M1, within the imposed time limit.

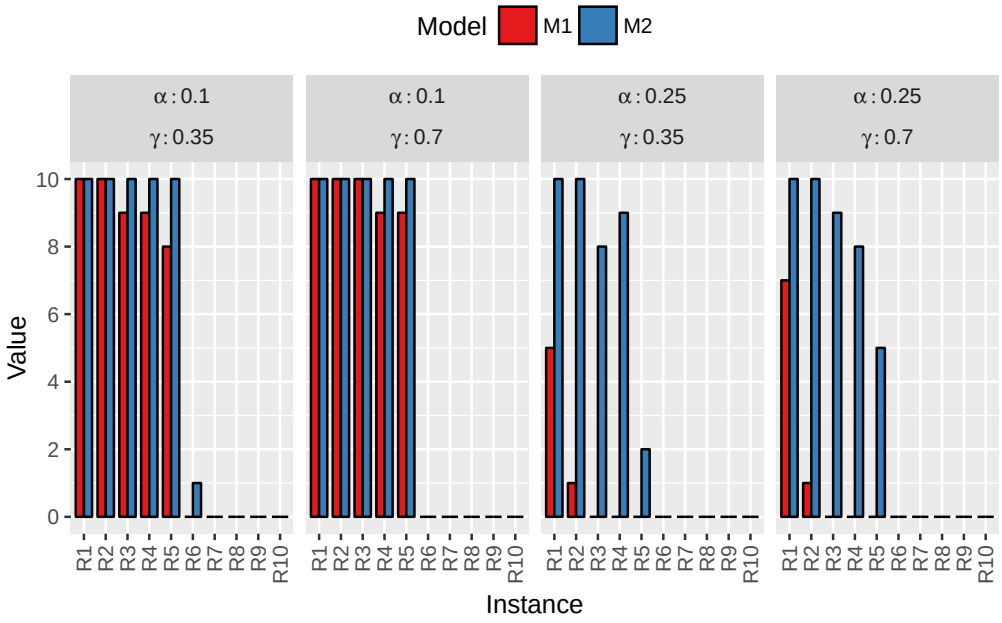
To better analyze the behaviour of the two models, it is useful to consider also the computational times, given in Figures 2.9 to 2.11. It is quite evident that M2 completely outperforms M1 on Random and Complete networks, and it is competitive on most of the Grid instances. Moreover, M2 does not need the construction of the expanded graph, and it requires less memory.

Empirical evaluation of the Branch & Bound approaches

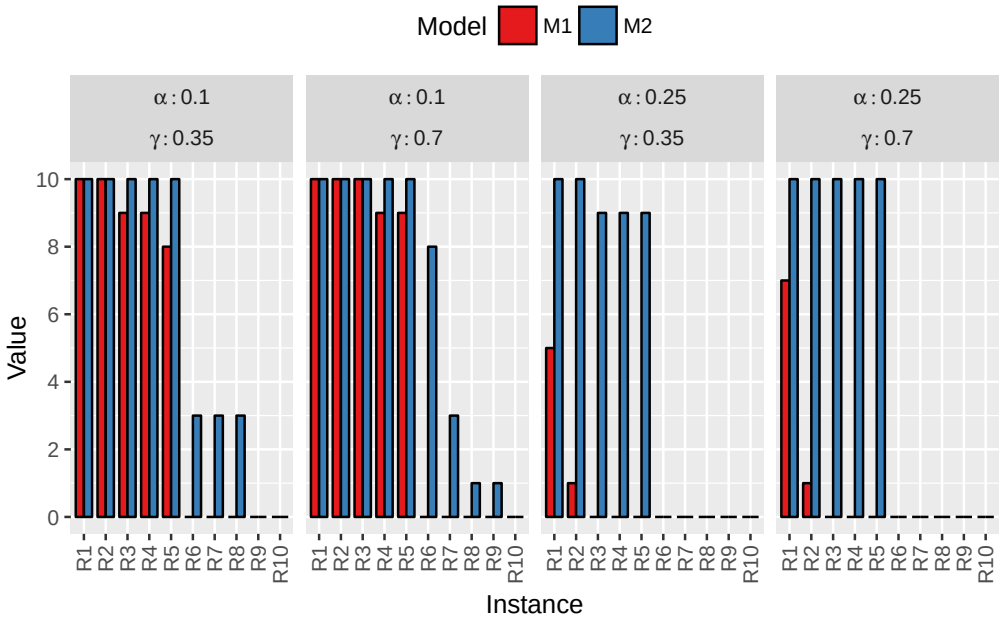
In this section, a comparison of B&B^{new} with B&B is presented. The computational experiments have been carried out, by considering, for both the approaches, the best found first strategy to build the branching tree. This choice has been motivated by the fact that, as shown in Section 2.2.3, this strategy outperforms the depth first one.

In Figure 2.12, the performance profiles of the two approaches are shown. It is evident that B&B^{new} outperforms consistently B&B. For all graph's typologies, B&B^{new} is able to find the optimal solution on a greater number of instances than those solved by B&B. Particularly, for complete graphs (Figure 2.12a), B&B^{new} always find an optimal solution.

It was an expected behaviour for two main reasons. Firstly, B&B^{new} shows a better time complexity than B&B, since at each node, B&B^{new} have to recalculate only a small part of the path tour. Secondly, B&B^{new} does not need to expand the graph, thus a significant difference in memory usage is observed. Indeed, B&B is not able to solve a

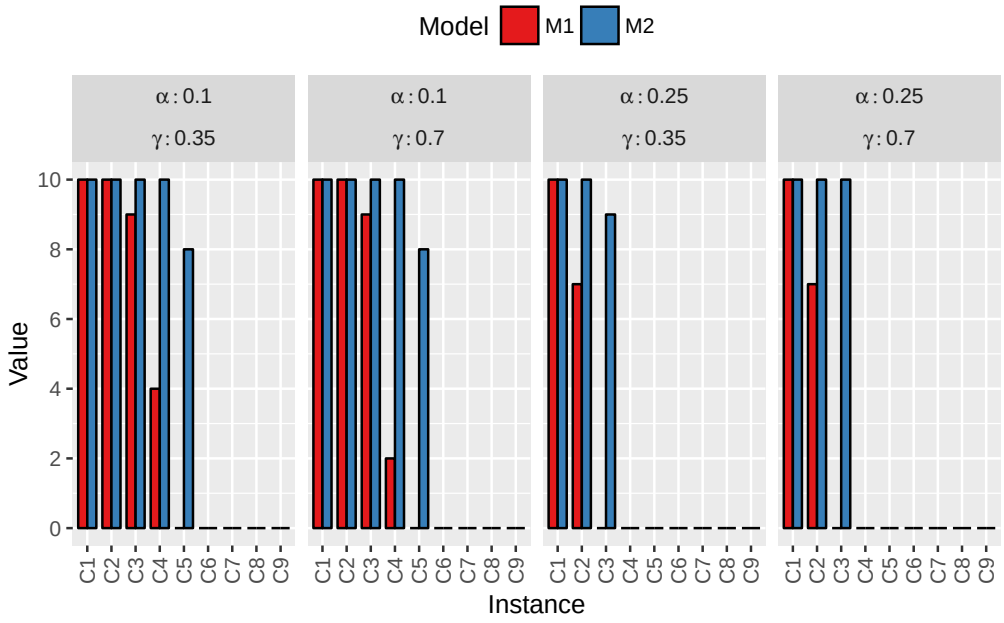


(a) *Optimal solutions*

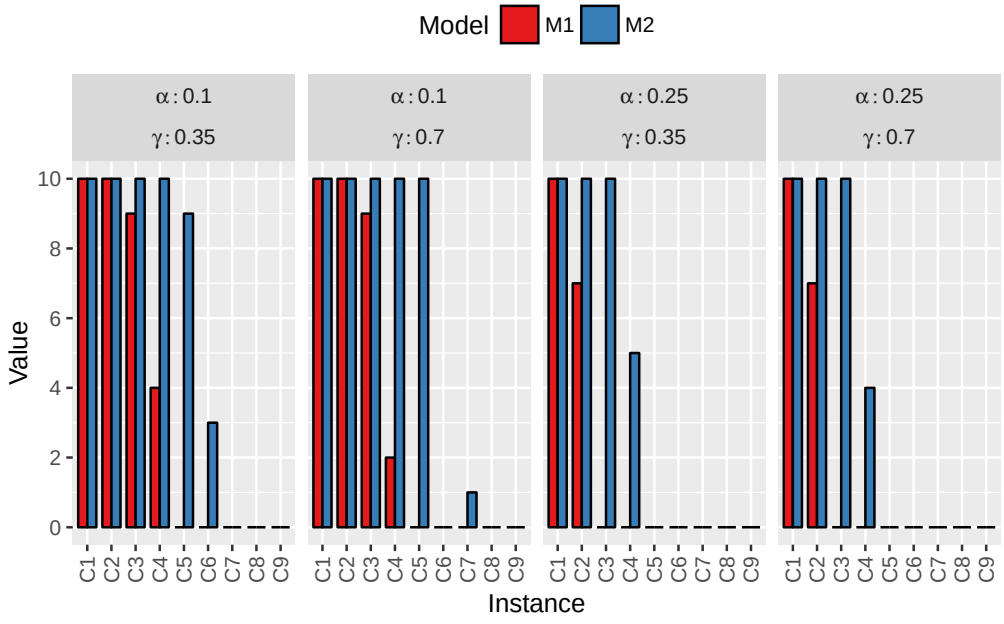


(b) *Feasible solutions*

Figure 2.7: *Performance profiles for optimal solutions of M1 and M2 on sparse random graphs.*



(a) *Optimal solutions*



(b) *Feasible solutions*

Figure 2.8: *Performance profiles of M1 and M2 on complete graphs.*

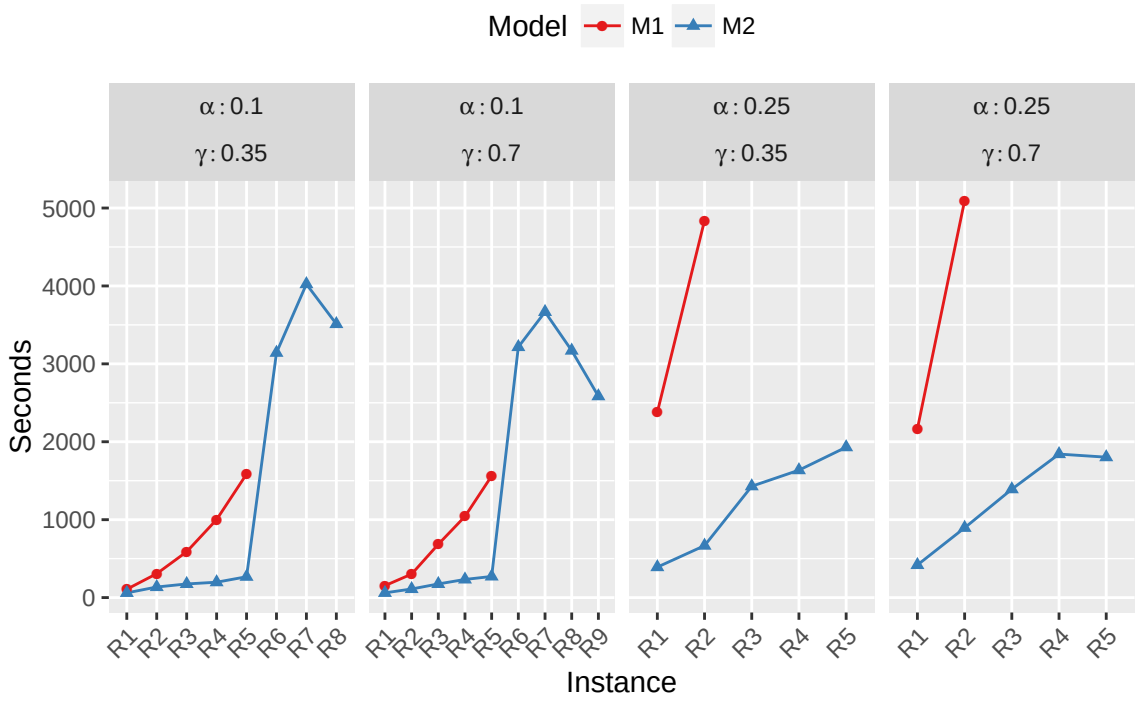


Figure 2.9: *Computational times to solve M1 and M2 formulations on Random graphs.*

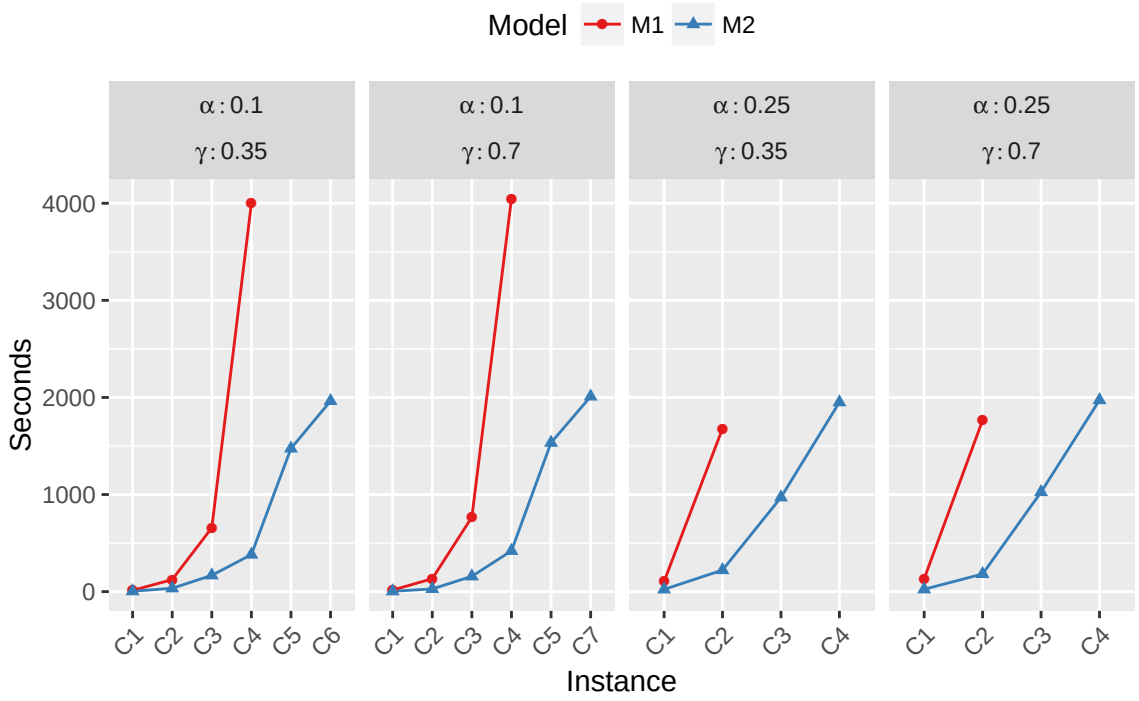


Figure 2.10: *Computational times to solve M1 and M2 formulations on Complete graphs.*

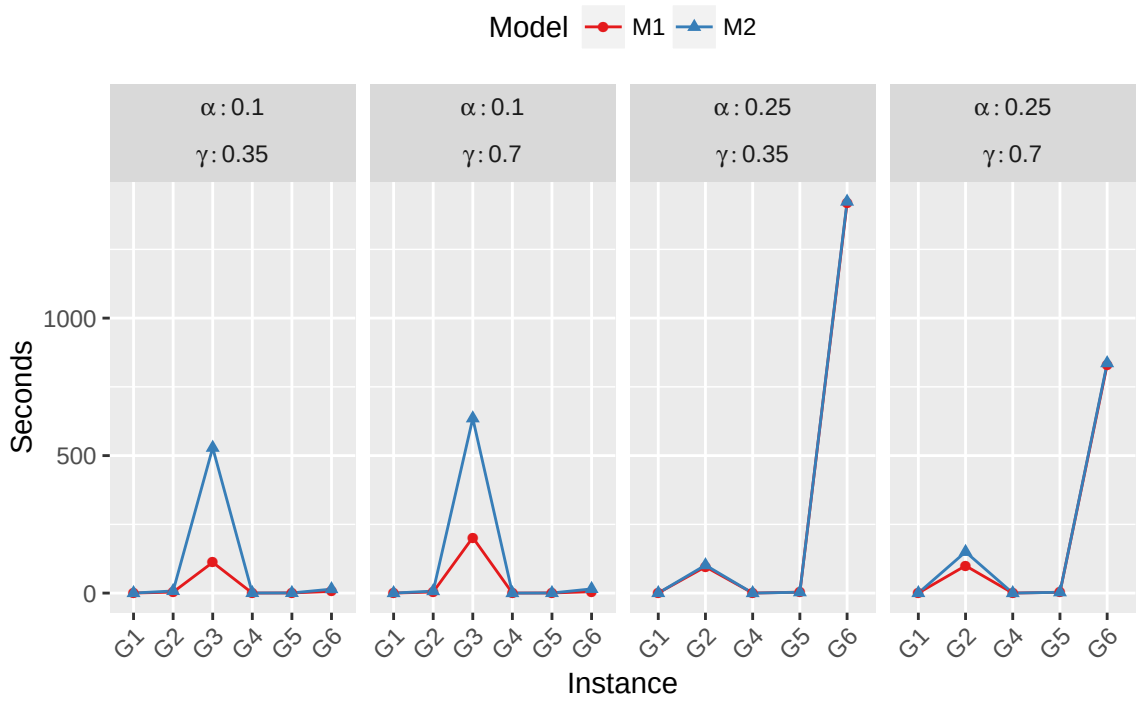


Figure 2.11: *Computational times to solve M1 and M2 formulations on Grid graphs.*

large number of instances, because it requires a huge amount of memory and fails to allocate it.

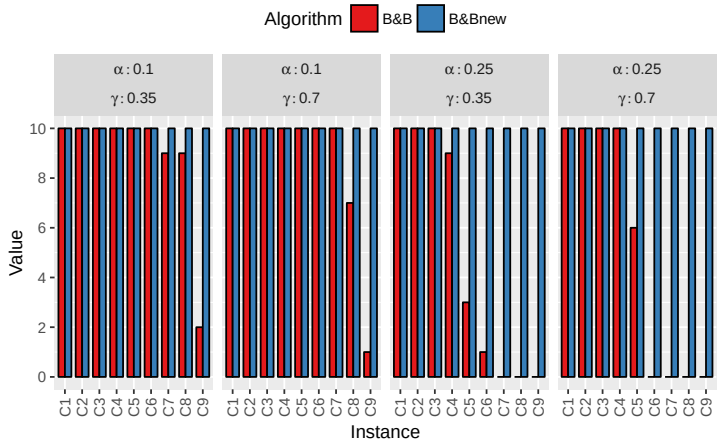
Comparison with the GRASP

In this section, the best performing approaches (i.e., B&B^{new} and M2) are compared with the GRASP metaheuristic.

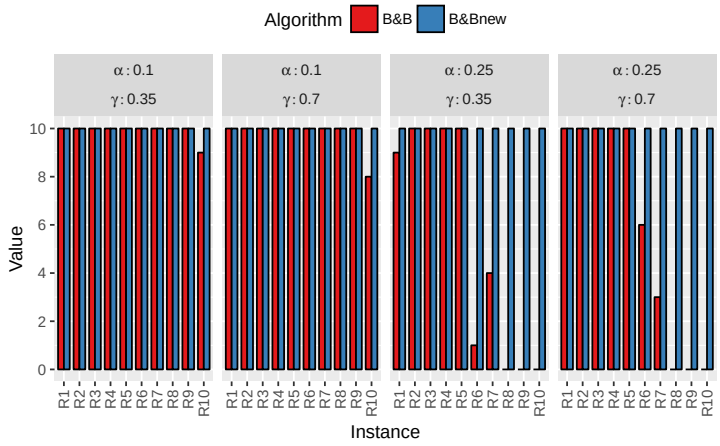
The results for complete graphs are shown in Figures 2.13 and 2.14. It is evident that B&B^{new} is very accurate and efficient on complete graphs, since it is able to find always the optimum. GRASP shows quite good performance and is able to solve to optimality several instances (and it finds a feasible solution on all the test problems). On the other hand, M2 behaves quite poorly, especially on large-size instances. It is evident that B&B^{new} outperforms CPLEX, because it follows a solution strategy well-tailored to the problem under study, that relies on the solution of only ***SPP*** sub-problems.

The computational results collected on the random graphs (see Figures 2.15 and 2.16) underline that B&B^{new} behaves the best and for the big instances, GRASP is not able to find optimal solutions.

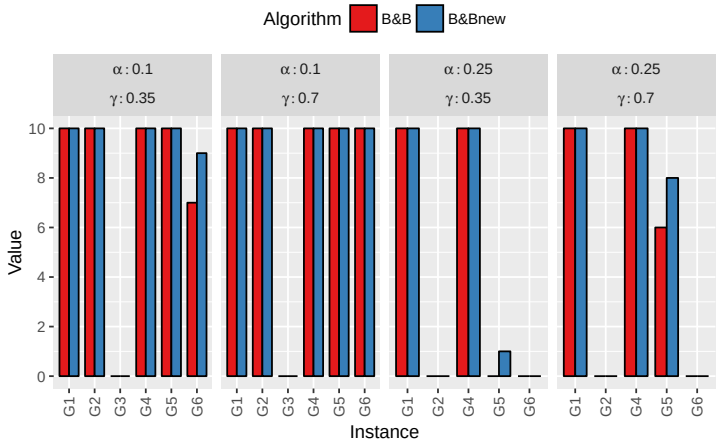
A completely different behaviour is observed for the grid graphs. More specifically, the results reported in Figure 2.17 show that B&B^{new} and M2 do not find feasible solutions on several instances. On the contrary, GRASP always finds a feasible solution. This



(a) *Complete graphs*



(b) *Random graphs*



(c) *Grid graphs*

Figure 2.12: *Performance profiles of B&B and B&B^{new} algorithms for optimal solutions.*

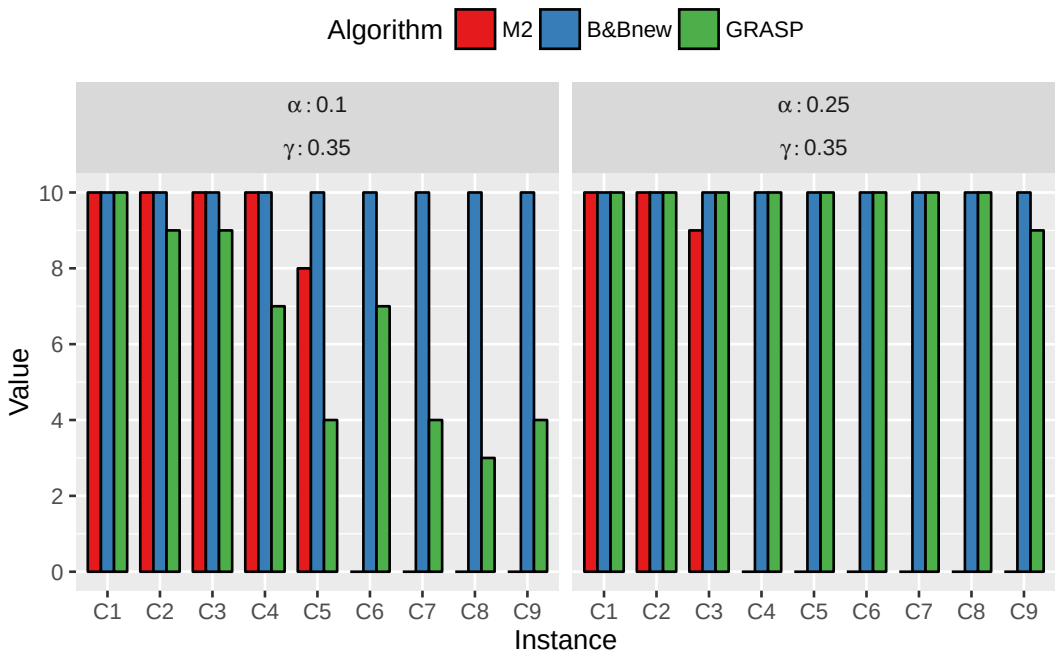


Figure 2.13: *Performance profile on complete graphs with* $\gamma = 0.35$ *for optimal solutions.*

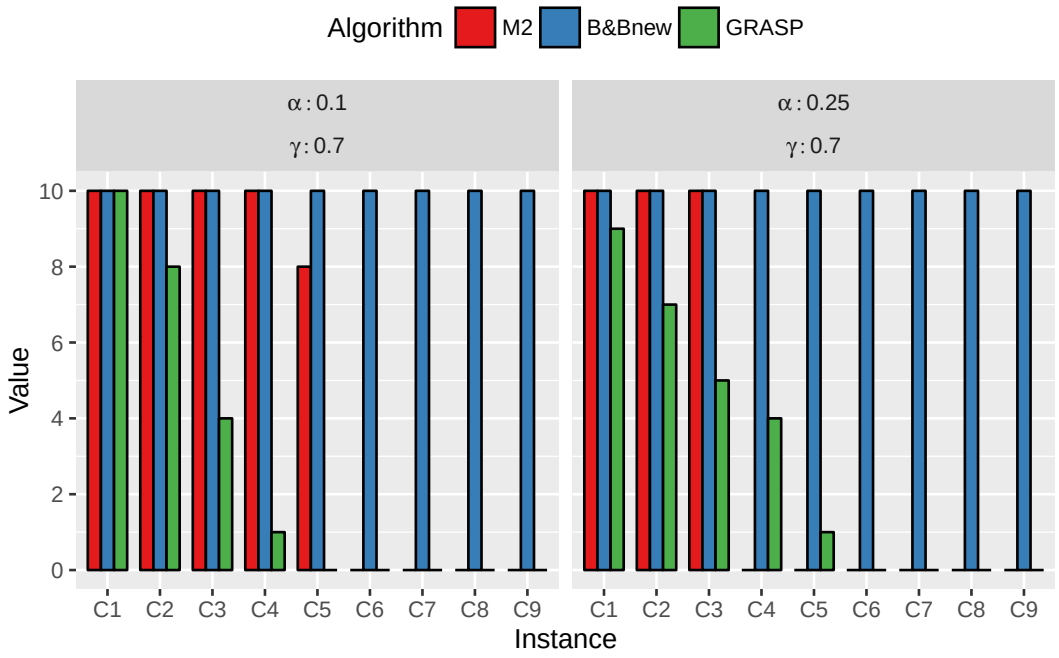


Figure 2.14: *Performance profile on complete graphs with* $\gamma = 0.70$ *for optimal solutions.*

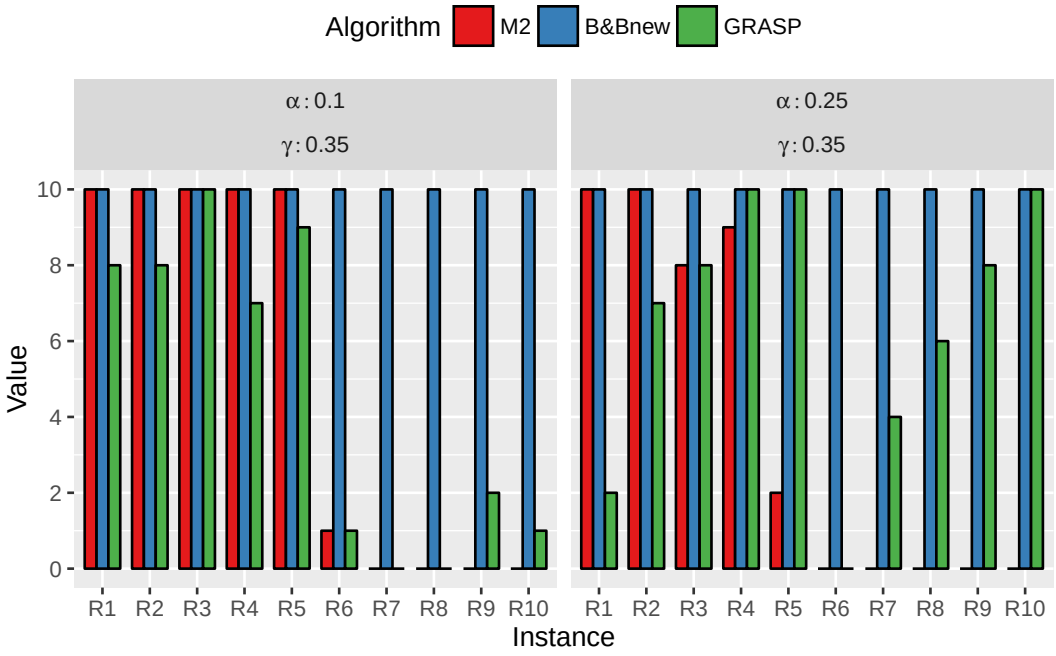


Figure 2.15: *Performance profile on random sparse graphs with* $\gamma = 0.35$ *for optimal solutions.*

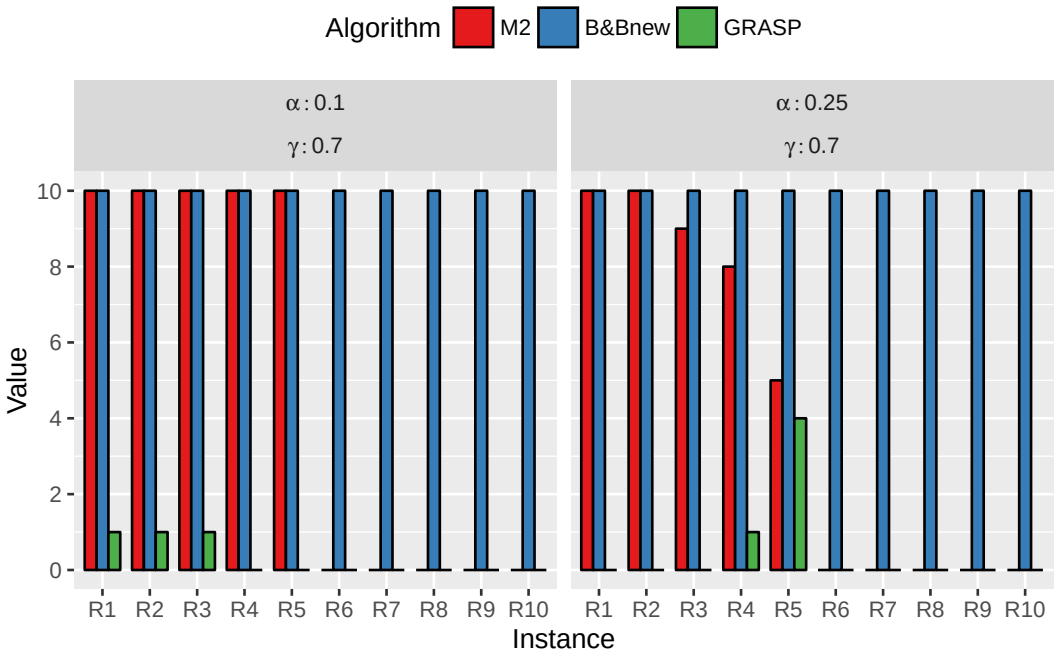


Figure 2.16: *Performance profile on random sparse graphs with* $\gamma = 0.70$ *for optimal solutions.*

behaviour was expected, given the computational complexity of ***CSPTP***.

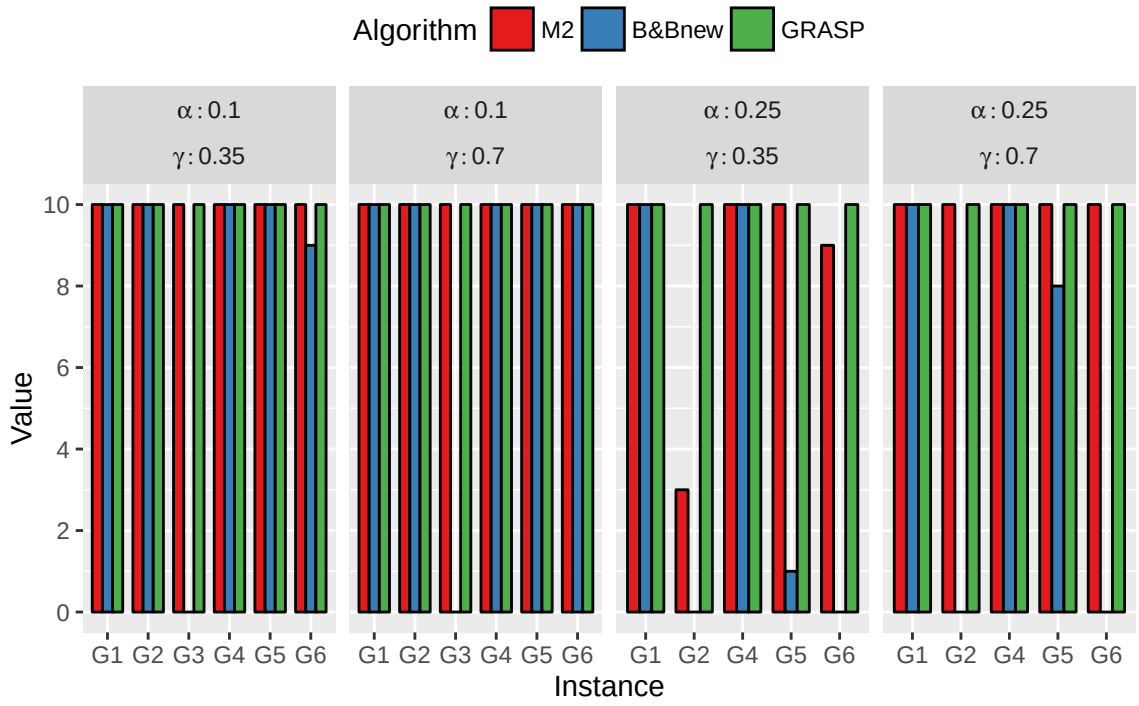


Figure 2.17: *Performance profile on grid graphs for feasible solutions.*

A detailed comparison of computational times of the three algorithms is given in Figures 2.18 to 2.20. The average computational times over ten instances for all types of test problems is reported. If the algorithm was not able to solve any instances, the value is missed.

The computational times confirm the results of the performance profiles: for complete and random graphs the B&B^{new} algorithm is the best one, followed by GRASP. For grid graphs, the GRASP shows in general the smallest time of execution.

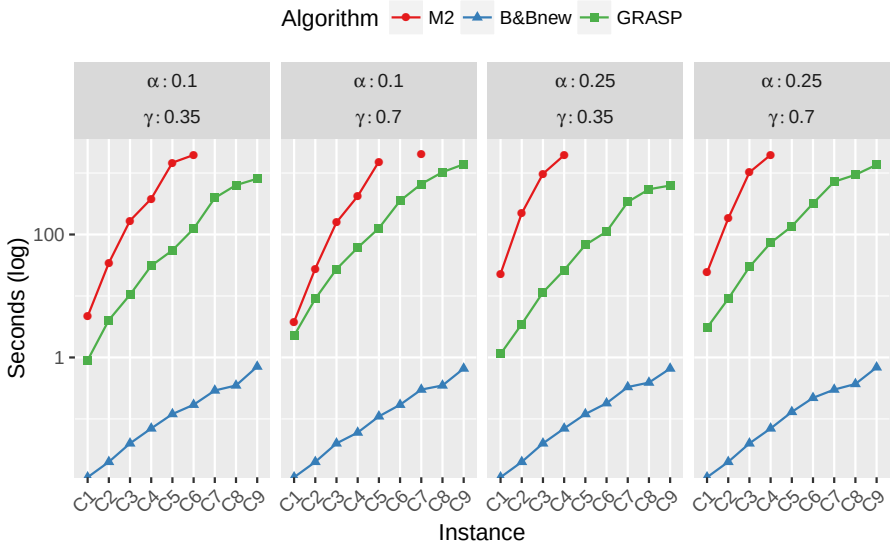


Figure 2.18: *Computational times over complete graphs.*

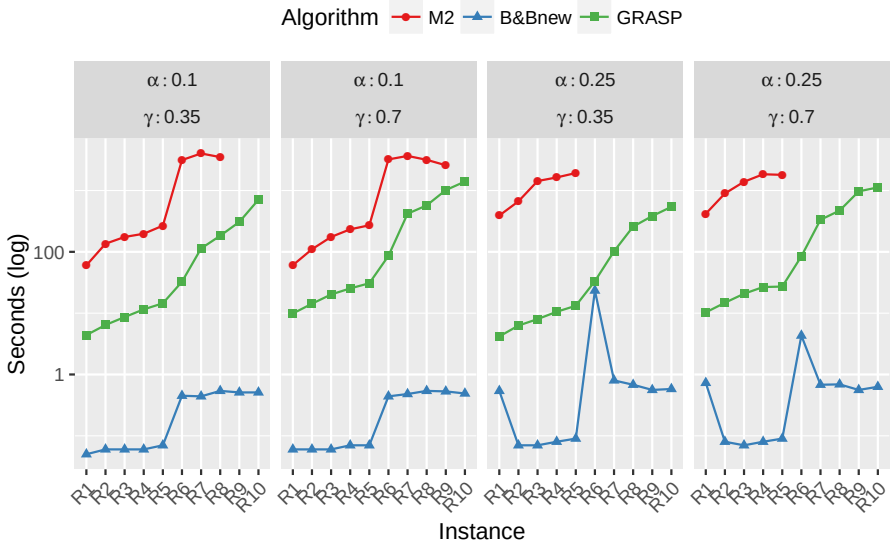


Figure 2.19: *Computational times over random graphs.*

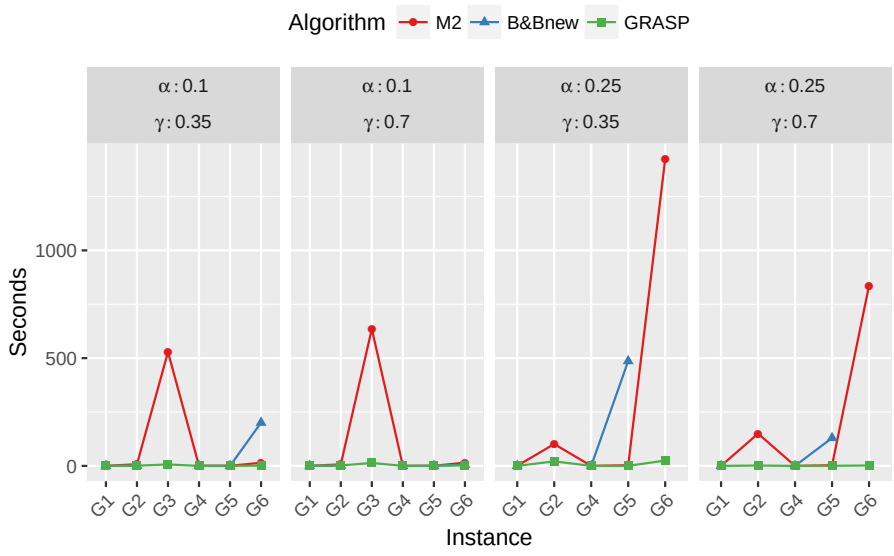


Figure 2.20: *Computational times over grid graphs.*

2.4 Shortest Path Tour Problem with Time Windows

2.4.1 Introduction

The *Resource-Constrained Shortest Path Problem* (***RCSPP***) [51] is a variant of classical ***SPP***, where besides the usual cost c_{ij} associated with each arc (i,j), there is an additional r-dimensional resource vector R. Formally, let $G = (V, A)$ a directed graph, a cost c_{ij} and $|R|$ scalars w_{ij}^h, $h = 1, \ldots, |R|$, which represent the consumption along arc (i,j) of resource h, are associated with each arc $(i,j) \in A$.

Let $s, d \in V$ two distinct nodes, ***RCSPP*** aims to find a minimum cost path from s to d, such that all feasibility constraints on the resources $h = 1, \ldots, |R|$ are satisfied. Indeed, a resource window $[a_i^h, b_i^h]$, with $a_i^h < b_i^h, h = 1 \ldots, |R|$, is associated with each node $i \in V$. A path Π is feasible if and only if $a_i^h \leq w^h(\Pi) \leq b_i^h$, where $w^h(\Pi) = \sum_{(i,j)\in\Pi} w_{ij}^h$, for $h = 1, \ldots, |R|$.

A widely studied type of ***RCSPP*** is the *Shortest Path Problem with Time Windows* (***SPPTW***). In this case, the only resource associated with the arcs is time. A cost c_{ij} and a *transit time* t_{ij} are associated with each arc $(i,j) \in A$, whereas a *time window* $[e_i, l_i]$ is associated to each node $i \in V$. A path Π_{sd} from s to d is feasible if and only if $e_i \leq \tau(\Pi_{si}) \leq l_i$, for each sub-path Π_{si}, where

$$\tau(\Pi_{si}) = \max \left\{ e_i, \sum_{(j,k)\in\Pi_{si}} \right\}.$$

Indeed, every node i has to be served during the time window $[e_i, l_i]$, but it is possible to wait in a node until the time window will be open.

According to the ***SPPTW*** principles, it is easy to extend the ***SPTP*** to include time window constraints, resulting in the *Shortest Path Tour Problem with Time Windows* (***SPTPTW***).

Let $G = (V, A, C)$ be a directed and weighted graph. A non-negative transit time t_{ij} is associated with each arc $(i,j) \in A$. Moreover, a service time s_i and a time window $[e_i, l_i]$ are associated with each node i. Let τ_i be the arrival time to node i. The ***SPTPTW*** can be defined as follows.

Definition 2.6. *The Shortest Path Tour Problem* ***with Time Windows*** *(**SPTPTW**) consists in finding a shortest path from a source node* s *to a destination node* d, $s, d \in V$, $s \neq d$, *by ensuring that at least one node of each node disjoint subsets* $T_1, \ldots, T_N$, *is crossed according to the sequence wherewith the subsets are ordered. Any intermediate nodes between visits to the subsets* T_h, $h = 1, \ldots, N$ *are allowed.* ***Furthermore, the solution path must satisfy the equations*** $\tau_i \leq l_i$, ***for all*** $i \in V$.

Note that, if $\mathcal{T} = \cup_{h=1}^{N} T_h$, it is assumed $s_i = 0, e_i = 0, l_i = +\infty$, for all nodes $i \notin \mathcal{T}$.

2.4.2 Dynamic programming algorithm

Since ***RCSPP*** are **NP**-hard problems [34], and ***SPTPTW*** is a generalization of a specific ***RCSPP*** problem, it follows that ***SPTPTW*** belongs to the class of **NP**-hard problems.

Despite of the hardness of the problem, an exact method has been proposed. It relies on the well known technique of *Dynamic Programming* (DP) [10]. The algorithm's core is the *label* concept. To each path P_{si} from node s to node i, is associated a label y_i. Each label is characterized by the cost of the associated path c_i, the last index h of the last subset T_h visited, that is r_i, and the arrival time τ_i.

Definition 2.7 (Feasibility)**.** *A label y_i is feasible if the condition $\tau_i \leq l_i$ holds.*

Definition 2.8 (Dominance)**.** *Given two labels y_i and $\bar{y}_i$ associated with node $i \in V$, y_i dominates $\bar{y}_i$ if the following conditions hold:*

$$c_i \leq \bar{c}_i,$$
$$r_i \geq \bar{r}_i,$$
$$\tau_i \leq \bar{\tau}_i,$$

and at least one inequality is strict.

Given the two definitions above, it is clear that a feasible and non dominated label y_d must be associated to an optimal shortest path tour with time windows from source node s to destination node d.

Algorithm 2.11 looks for a such label. In lines 2 to 4, the first label y_s^0 is created associated to the path that contains only s, the list L of the labels to process, and the set D of all non dominate labels are initialized.[2]

The main loop (line 5) iterates until L is empty. At each iteration a label is removed from L, and if it is a solution better than the incumbent, the latter is updated (lines 6 to 9). Otherwise, the label is extended examining the forward start of the last node i of the path. The arrival time into the node j is calculated in line 13. If the node $j \in T_{r_i+1}$ and the window time is not open yet, a label is created in which the current node is not served (lines 14 to 16). Furthermore, a new label denoting that the node j has been served is created (lines 17 to 21).

In the case that $j \in T_{r_i+1}$ and $time < e_j$, two labels are generated to avoid the *waiting problem*. Generally, in shortest path problem with time windows, if a node is reached before the opening of the window, it is possible to wait until the window is open and restart. In our case, the only nodes with time windows are the node in $\{T_i\}_{i=1,\dots,N}$, but waiting in a node can exclude some feasible solution.

[2] Each label is associated to the corresponding path, but the paths are not reported in pseudocode.

Algorithm 2.11: *Dynamic programming algorithm to solve* ***SPTPTW***.

```
1  Function Labeling(G, s, d, {T_i}_{i=1,...,N})
2      y_s^0 ← (0, 0, 1);
3      L ← {y_s^0}; D[s] ← {y_s^0}; D[j] ← ∅ ∀j ∈ V, j ≠ s;
4      best ← Nil;
5      while L ≠ ∅ do
6          y_i ← Pop(L); L ← L \ {y_i};
7          if i = d and r_i = N then
8              if c_i < c_best then
9                  best ← y_i;
10         else
11             foreach j ∈ FS(i) do
12                 r̄ ← r_i;
13                 time ← τ_i + s_i + t_ij;
14                 if j ∈ T_{r_i+1} and time < e_j then
15                     y_j ← (c_i + c_ij, time, r̄);
16                     AddLabel(L, D, y_j);
17                 time ← max{time, e_j};
18                 if j ∈ T_{r_i+1} then
19                     r̄ ← r_i + 1;
20                 y_j ← (c_i + c_ij, time, r̄);
21                 AddLabel(L, D, y_j);
22     return best
23 Function AddLabel(L, D, y_j)
24     if y_j is not dominated by any label y'_j belonging to D[j] then
25         Remove from D[j] and from L all label y'_j that are dominated by y_j;
26         L ← L ∪ {y_j}; D[j] ← D[j] ∪ {y_j};
```

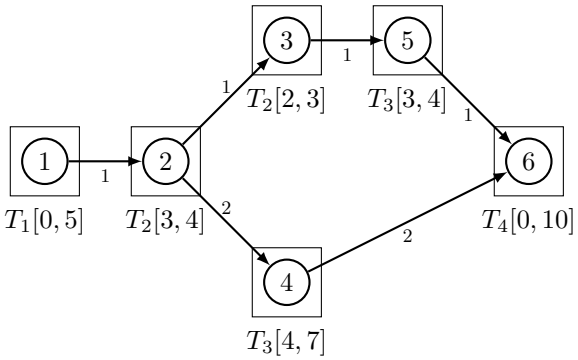


Figure 2.21: *A small example of waiting problem.*

An example instance to illustrate the problem is depicted in Figure 2.21. All service and transit times are equal to 1. When the node $2 \in T_2$ is reached the time is 2, waiting until time 3 makes it infeasible to pass through node 5 (reached at time 5), so the best found solution is $(1, 2, 4, 6)$, that is feasible and has a cost of 5. But the best solution is $(1, 2, 3, 5, 6)$ with a cost equal to 4 and feasible because T_2 is served in node 3.

Therefore, when a "problematic" node is reached, two different labels are generated: in the first one, it does not wait for the opening time, the node is not served and r_j is not updated; in the second, the node is served waiting the opening of the time windows, and r_j is incremented.

2.4.3 Bounds

To improve the performance of the algorithm, several bounds can be used to reduce the number of feasible and non dominated labels generated by the algorithm.

Obtain a feasible upper bound

There is a simple method to obtain a feasible solution that is an upper bound to the optimal solution. It is possible to solve the ***SPTP*** using transit times t_{ij} instead of normal weight c_{ij}. If the minimal Shortest Path Tour calculated in this manner is not feasible with respect to the time windows, there is no feasible solution, since the path between the sets T_i are all minimal respect to transit times. In Algorithm 2.12 the pseudo-code to obtain a such upper bound is shown.

Cost bounding

Let P_{sv} be a partial solution path from source node s to node v. If the cost $L(P_{sv})$ is greater than the cost of the current incumbent, then P_{sv} can be discarded. Moreover, it is possible to calculate the better cost that can be obtained starting from P_{sv} to arrive to destination d. For each $v \in \cup_{i=1}^{N}\{T_i\}$, let $SPTcost[v]$ be the cost of the best shortest path

Algorithm 2.12: *Algorithm to obtain an upper bound for* ***SPTPTW***.

```
1 Function UpperBound(G, s, d, {T_i}_{i=1,...,N})
2 |  L[,] ← FloydWarshall(G) ;    // Using transit times as cost function
3 |  UB ←DP(V, A, s, d, {T_i}_{i=1,...,n}) ;                    // Algorithm 2.9
4 |  if UB is feasible respect to time windows then
5 |  |  return UB
6 |  else
7 |  |  return Nil
```

tour from v to d, then $L(P_{sv}) + SPTcost[v]$ is a lower bound to each solution starting with P_{sv}.

For nodes that are not in $\cup_{i=1}^{N}\{T_i\}$, the lower bound defined above is not applicable. Let be $C[v,w]$ the cost of the shortest path from v to w, and let r_v the index of the last set T_i visited in the path, then the lower bound for any node $v \notin \cup_{i=1}^{N}\{T_i\}$ can be defined as $\min_{j \in T_{r_v+1}} \{ C[v,j] + SPTcost[j] \}$.

Given these lower bounds, a label y_i can be discarded if the lower bound associated is greater than the cost of the upper bound solution calculated with Algorithm 2.12. Clearly, if a feasible solution better the UB is found during the computation, then UB is updated with the new solution.

Time bounding

A second bounding technique can be used on times, rather than costs. Let be $T[v,w]$ the total time of shortest path respect transit times from v to w, and r_v the index of the last set T_i visited in the path. If the time consumed by a partial solution P_{sv} plus the time to reach the nodes in $v \in T_{r_v+1}$ exceed the time windows of all nodes v, than the current partial solution cannot generate a feasible complete solution. Therefore, a label y_i is not feasible if not exists $j \in T_{r_i+1}$ such that $\tau_i + T[i,j] \leq l_j$.

2.4.4 Experimental results

Several experiments have been performed to test the performance of the `Labeling` algorithm. The comparison aims at showing that the extra computational times, needed to calculate lower and upper bounds, is less than the time saved by pruning the bounded labels.

Instance generation

All test problems have been pseudo-randomly generated by using a generator inspired by Powell and Chen [50]. The generation procedure performs the following steps:

1. The nodes are randomly dispersed in a square matrix $[0, 500]\times[0, 500]$ by a uniform distribution, and the source node is located at $(250, 250)$;
2. The transit time, t_{ij} for each arc, is assigned to their Euclidean distance plus a perturbation uniformly distributed in $[5, 25]$;
3. The cost of each arc, c_{ij} is equal to $t_ij - \min_{(i,j)\in A}\left\{ t_{ij} \right\}$, to ensure that all costs are greater or equal to zero;
4. The nodes are assigned to the sets T_i, $i = 1, \ldots, N$;
5. To ensure the feasibility of each instance, the center of the time windows of a node $v \in \cup_{i=1}^{N}\{T_i\}$ is equal to the of the best shortest path tour respect to times from s to v;
6. The width of the time windows of each constrained node is a random number in $U[\frac{2}{3}AVG, \frac{3}{4}AVG]$, where AVG is the input average. The time window is derived from the center and the width.

Three types of networks have been utilized in the experiments: complete, grid, and random graphs. For complete graphs, the number of nodes n ranges from 100 to 500 with a step of 50, a is equal to 0.25, resulting in $N = \lfloor 0.25n \rfloor$, and $\gamma = 70$. As last parameter, the average width of the time windows has been set to 100. Random graphs share the same a, γ and AVG of the complete graphs, but the number of nodes is $n \in \{ 250, 500, 750, 100 \}$ and the density of the graph $\frac{m}{n}$ equals to $\{ 10, 15, 20 \}$.

Lastly, grid graphs have the following shapes $\{05 \times 20, 07 \times 15, 09 \times 09, 10 \times 10, 10 \times 40, 14 \times 30, 15 \times 15, 20 \times 20, 20 \times 40, 30 \times 60, 40 \times 40\}$, and a spreads from 0.15 to 0.19. The remain parameters are the same of the other network types.

Results

A first set of experiments involves complete graphs. Figure 2.22 shows the performance profile, that reports the number of instances that can be resolved in six hours by the algorithms. The `Labeling` algorithm uses both bounds on time and cost, `nocostbound` algorithm does not use the cost bounding, and no bounding techniques are used in `nobound`. In Figure 2.23, the computational times of the solved instances are reported.

A second set of experiments involves random instances, and the results are shown in Figure 2.24. In Figure 2.25, the computational times confirm that the use of the bounding strategies results in time savings.

Finally, the performance profile and computational times for grid networks are reported in Figures 2.26 and 2.27, respectively. The results confirm previous behaviour.

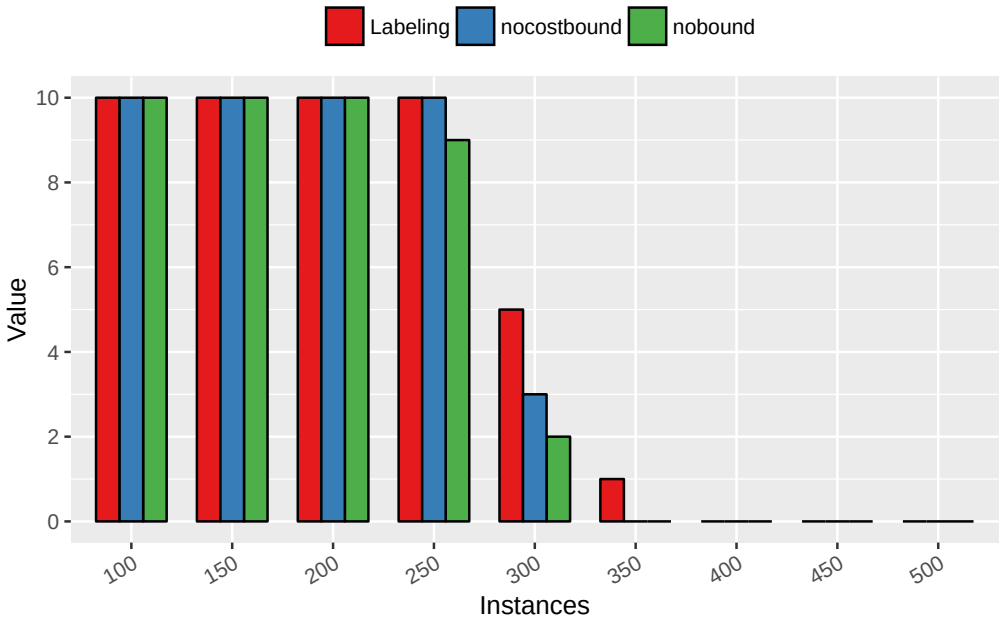


Figure 2.22: *`Labeling` performance profiles for complete graphs.*

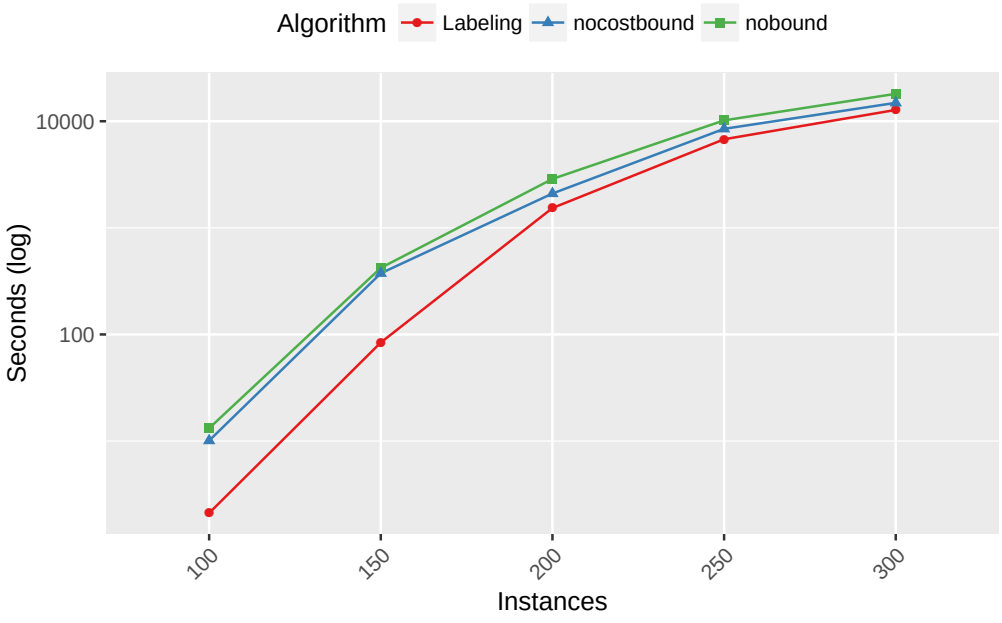


Figure 2.23: *`Labeling` computational times for complete graphs.*

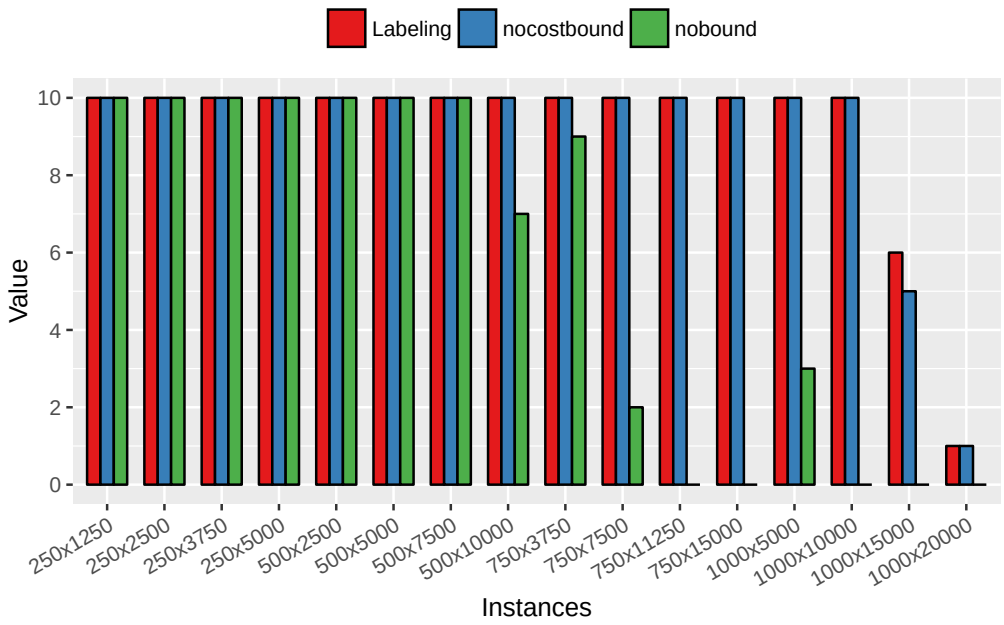


Figure 2.24: *`Labeling` performance profiles for random graphs.*

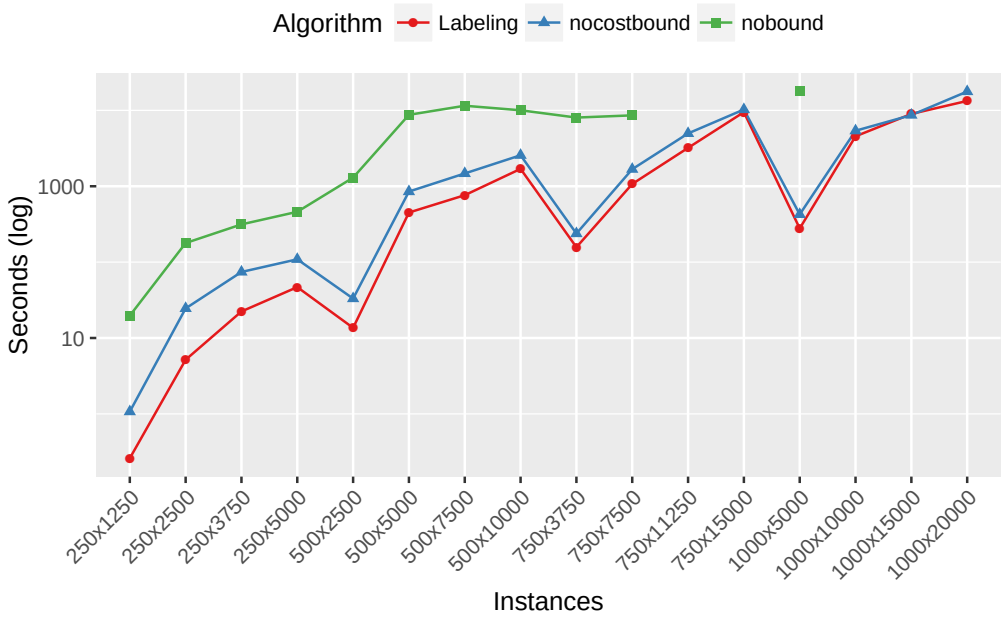


Figure 2.25: *`Labeling` computational times for random graphs.*

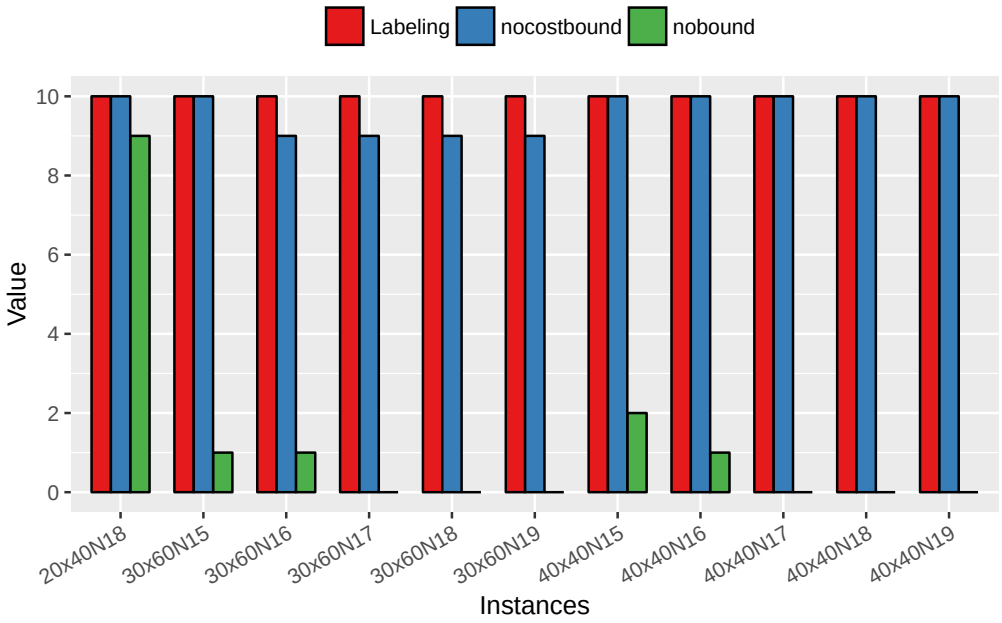


Figure 2.26: *`Labeling` performance profiles for grid graphs.*

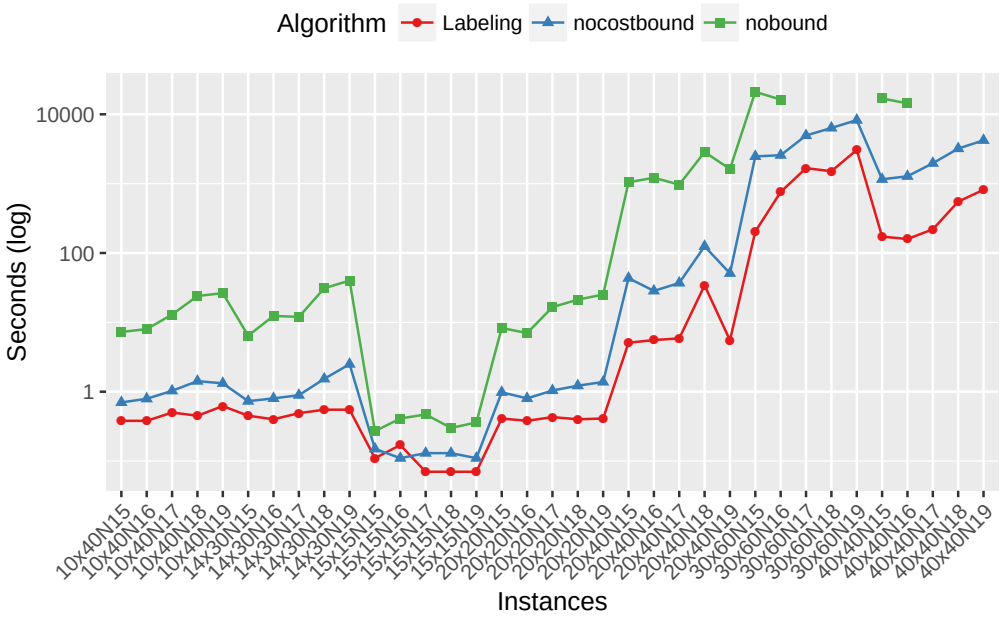


Figure 2.27: *`Labeling` computational times for grid graphs.*

Chapter 3

GRASP algorithms for the FFMSP, p-Center, and MCS

3.1 Far From Most String Problem

3.1.1 Introduction

The *Far From Most String Problem* (***FFMSP***) is one of the so called *string selection and comparison problems*, that belong to the more general class of problems known as *sequences consensus*, where a finite set of sequences is given and one is interested in finding their *consensus*, i.e., a new sequence that agrees as much as possible with all the given sequences. In other words, the objective is to determine a sequence called consensus, because it represents, in some sense, all the given sequences. For the ***FFMSP***, the objective is to find a sequence that is far from as many sequences as possible in a given set of sequences all having the same length. The problem has applications in several fields [23], including molecular biology where one is interested in creating diagnostic probes for bacterial infections or in discovering potential drug targets. Moreover, as a result of the linear coding of DNA and proteins, one is also interested in computing distance/proximity among biological sequences.

To formally state the problem, the following notation is needed:

- An *alphabet* $\Sigma = \{c_1, c_2, \ldots, c_k\}$ is a finite set of elements, called *characters*;
- $s^i = (s^i_1, s^i_2, \ldots, s^i_m)$ is a sequence of length m ($|s^i| = m$) on Σ ($s^i_j \in \Sigma$, $j = 1, 2, \ldots, m$);
- Given two sequences s^i and s^l on Σ such that $|s^i| = |s^l|$, $d_H(s^i, s^l)$ denotes their Hamming distance and is given by

$$d_H(s^i, s^l) = \sum_{j=1}^{|s^i|} \Phi(s^i_j, s^l_j), \tag{3.1}$$

where s^i_j and s^l_j are the characters in position j in s^i and s^l, respectively, and Φ :

$\Sigma \times \Sigma \to \{0, 1\}$ is the predicate function such that

$$\Phi(a, b) = \begin{cases} 0, & \text{if } a = b; \\ 1, & \text{otherwise.} \end{cases}$$

The FFMSP consists in determining a sequence far from as many sequences as possible in the input set Ω. This can be formalized by stating that, given a threshold t, a string s must be found maximizing the variable x such that

$$d_H(s, s^i) \geq t, \text{ for } s^i \in P \subseteq \Omega \text{ and } |P| = x. \tag{3.2}$$

Lanctot et al. [44] demonstrated that for sequences over an alphabet Σ with $|\Sigma| \geq 3$, approximating the ***FFMSP*** within a polynomial factor is **NP**-hard.

3.1.2 GRASP with Path Relinking

Path relinking

Path-relinking is a heuristic proposed by Glover [35] as an intensification strategy exploring trajectories connecting elite solutions obtained by tabu search or scatter search.

Starting from one or more elite solutions, paths in the solution space leading towards other guiding elite solutions are generated and explored in the search for better solutions. This is accomplished by selecting moves that introduce attributes contained in the guiding solutions. At each iteration, all moves that incorporate attributes of the guiding solution are analyzed and the move that best improves (or least deteriorates) the initial solution is chosen.

Algorithm 3.1 illustrates the pseudo-code of path-relinking for the ***FFMSP***. It is applied to a pair of sequences $(s', \hat{s})$. Their common elements are kept constant, and the space of solutions spanned by this pair of solutions is searched with the objective of finding a better solution. This search is done by exploring a path in the solution space linking solution s' to solution $\hat{s}$. s' is called the *initial solution* and $\hat{s}$ the *guiding solution*.

The procedure then computes (line 4) the symmetric difference $\Delta(s', \hat{s})$ between the two solutions as the set of components for which the two solutions differ:

$$\Delta(s', \hat{s}) = \{\, i = 1, \ldots, m \;\big|\; s'_i \neq \hat{s}_i \,\}.$$

Note that, $|\Delta(s', \hat{s})| = d_H(s', \hat{s})$ and $\Delta(s', \hat{s})$ represents the set of moves needed to reach $\hat{s}$ from s', where a move applied to the initial solution s' consists in selecting a position $i \in \Delta(s', \hat{s})$ and replacing s'_i with $\hat{s}_i$.

Path-relinking generates a path of solutions $s'_1, s'_2, \ldots, s'_{|\Delta(s', \hat{s})|}$ linking s' and $\hat{s}$. The best solution s^* in this path is returned by the algorithm (line 11).

Algorithm 3.1: *Pseudo-code of path-relinking for the* ***FFMSP****.*

```
1  Function PathRelinking(t, m, f_t(·), s', ŝ, Seed)
2      f* ← max{f_t(s), f_t(ŝ)};
3      s* ← arg max{f_t(s), f_t(ŝ)};
4      Δ(s', ŝ) ← { i = 1, ..., m | s'_i ≠ ŝ_i };
5      while Δ(s', ŝ) ≠ ∅ do
6          i* ← arg max { f_t(s' ⊕ i) | i ∈ Δ(s', ŝ) };
7          Δ(s' ⊕ i*, ŝ) ← Δ(s', ŝ) \ {i*};
8          s' ← s' ⊕ i*;
9          if f_t(s') > f* then
10             f* ← f_t(s'); s* ← s';
11     return s*;
```

The path of solutions is computed in the loop in lines 5 through 10. This is achieved by advancing one solution at a time in a greedy manner. At each iteration, the procedure examines all moves $i \in \Delta(s', \hat{s})$ from the current solution s' and selects the one which results in the highest cost solution (line 6), i.e., the one which maximizes $f_t(s' \oplus i)$, where $s' \oplus i$ is the solution resulting from applying move i to solution s'. The best move i^* is made, producing solution $s' \oplus i^*$ (line 8). The set of available moves is updated (line 7). If necessary, the best solution s^* is updated (lines 9–10). Clearly, the algorithm stops when $\Delta(s', \hat{s}) = \emptyset$.

Hybridization of GRASP with PR

Since GRASP iterations are independent of one another, it does not make use of solutions produced throughout the search. One way to add memory to GRASP is its hybridization with path-relinking. The first proposal of such a hybrid method was published by Laguna and Martí [43].

For the ***FFMSP***, the path-relinking intensification procedure described previously, has been integrated in the GRASP by Ferone et al. [24]. Path-relinking is applied at each GRASP iteration to pairs $(s, \hat{s})$ of solutions, where s is the locally optimal solution obtained by the GRASP local search and $\hat{s}$ is randomly chosen from a pool with at most `MaxElite` high quality solutions found along the search.

3.1.3 Results

Several strategies for implementing path-relinking have been adopted. Given two solutions s' and $\hat{s}$, the following strategies have been implemented:

Forward PR The path emanates from s' which is the worst solution between s' and $\hat{s}$ (`grasp-h-f`);

Backward PR The path emanates from s' which is the best solution between s' and $\hat{s}$ (`grasp-h-b`);

Mixed PR Two paths are generated, one emanating from s' and the other emanating from $\hat{s}$: the process stops as soon as an intermediate common solution is met (`grasp-h-m`).

The algorithms were implemented in *C*, compiled by gcc 4.1.3, and run on an Intel i7 Quad core with a 2.67 GHz clock and 6 Gigabytes of RAM memory. The following two sets of problem instances were used.

$\mathcal{A}$. This is the set of benchmark instances introduced in Ferone et al. [22], consisting of 600 random instances of different size. More specifically, the number n of input sequences in Ω is in $\{100, 200, 300, 400\}$, and the length m of each of the input sequences is in $\{300, 600, 800\}$. In all cases, the alphabet size is four, i.e., $|\Sigma| = 4$. For each combination of n and m, the set $\mathcal{A}$ consists of 100 random instances. Finally, the algorithms were applied on all instances for different settings of the threshold parameter t varying from $.75m$ to $.85m$.

$\mathcal{B}$. This set of problem instances was used in Mousavi et al. [49] and was kindly provided by the authors. In this set, some instances were randomly generated, while the remaining are real-world instances from biology (Hyaloperonospora parasitica V 6.0 sequence data). In both the randomly generated and the real-world instances, the alphabet size is four, i.e., $|\Sigma| = 4$, the number n of input sequences in Ω is in $\{100, 200, 300\}$, and the length m of each of the input sequences ranges from 100 to 1200. Finally, the threshold parameter t varies from $.75m$ to $.95m$.

For each problem size in set $\mathcal{A}$, all the variants were run on 100 random instances and average solution value and average running times were computed. The set $\mathcal{B}$ contains 10 randomly generated instances and three real-world instances, for each problem size. The results obtained are summarized in Tables 3.1 to 3.3, where for each problem type, in the first column the instance size (n, m, and t) is reported. The remaining columns report the average objective function values (z) obtained by each algorithm and the standard deviation.

Looking at the average objective function values and standard deviation , it can be concluded that `grasp-h-b` finds better quality solution within a fixed running time. In order to validate the good behaviour of the backward strategy, in Figures 3.1 to 3.2, it is plotted the empirical distributions of the random variable *time-to-target-solution-value* (TTT-plots), involving algorithms `grasp`, and `grasp-h-b`. The plots show that, given any fixed amount of computing time, `grasp-h-b` has a higher probability of finding a good quality target solution.

Table 3.1: *Comparison between the different hybrid GRASP with path-relinking strategies on instances in set $\mathcal{A}$ after* 30 *seconds of running time.*

	grasp		grasp-h-f		grasp-h-b		grasp-h-m	
(n, m, t)	z	σ^2	z	σ^2	z	σ^2	z	σ^2
100, 300, 225	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00
100, 300, 240	73.84	0.87	74.60	0.97	**76.26**	0.99	75.89	1.02
100, 300, 255	**29.54**	0.57	29.39	0.61	29.42	0.59	29.42	0.64
100, 600, 450	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00
100, 600, 480	76.18	0.97	76.46	0.95	**77.53**	1.02	77.44	1.19
100, 600, 510	27.46	0.82	27.46	0.82	27.46	0.82	**27.47**	0.83
100, 800, 600	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00
100, 800, 640	81.50	1.26	81.63	1.20	82.06	1.13	**82.17**	1.11
100, 800, 680	**26.54**	0.90	26.50	0.93	26.51	0.92	26.51	0.93
200, 300, 225	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00
200, 300, 240	87.84	1.55	90.20	2.24	**94.71**	2.30	94.33	2.29
200, 300, 255	30.30	0.92	30.29	0.93	**30.37**	0.92	30.35	0.94
200, 600, 450	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00
200, 600, 480	79.98	1.73	79.73	1.77	**80.94**	1.89	80.80	1.84
200, 600, 510	**26.35**	1.28	26.33	1.30	26.33	1.31	26.33	1.31
200, 800, 600	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00
200, 800, 640	85.67	2.21	85.31	2.19	85.50	2.12	**85.71**	2.29
200, 800, 680	24.41	1.24	**24.42**	1.24	**24.42**	1.24	24.41	1.26
300, 300, 225	296.65	0.96	299.17	0.86	**299.22**	0.73	299.14	0.79
300, 300, 240	102.98	2.27	107.09	2.95	**112.83**	3.07	111.91	2.97
300, 300, 255	31.82	1.14	31.80	1.16	**31.83**	1.13	**31.83**	1.13
300, 600, 450	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00
300, 600, 480	82.69	2.51	82.46	2.71	83.02	2.63	**83.12**	2.60
300, 600, 510	24.94	1.55	**24.95**	1.55	24.94	1.55	24.94	1.55
300, 800, 600	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00
300, 800, 640	**90.26**	2.34	89.94	2.40	90.01	2.44	90.02	2.39
300, 800, 680	23.51	1.20	**23.53**	1.20	23.48	1.25	23.48	1.25
400, 300, 225	377.45	1.65	386.18	2.64	**388.62**	2.03	387.92	2.39
400, 300, 240	107.89	2.63	112.82	3.05	**119.32**	3.22	119.29	3.36
400, 300, 255	32.76	1.27	32.77	1.26	**32.78**	1.27	**32.78**	1.27
400, 600, 450	**400.00**	0.00	**400.00**	0.00	**400.00**	0.00	**400.00**	0.00
400, 600, 480	85.47	2.94	85.40	3.01	**85.99**	2.74	85.96	2.81
400, 600, 510	**24.56**	1.27	**24.56**	1.27	**24.56**	1.27	**24.56**	1.27
400, 800, 600	**400.00**	0.00	**400.00**	0.00	**400.00**	0.00	**400.00**	0.00
400, 800, 640	92.81	3.36	92.75	3.39	92.84	3.35	**92.86**	3.28
400, 800, 680	22.81	1.02	**22.82**	1.01	22.81	1.02	22.81	1.02

Table 3.2: *Comparison between the different hybrid GRASP with path-relinking strategies on random instances in set $\mathcal{B}$ after* 30 *seconds of running time.*

	grasp		grasp-h-f		grasp-h-b		grasp-h-m	
(n, m, t)	z	σ^2	z	σ^2	z	σ^2	z	σ^2
100, 100, 75	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00
100, 100, 85	32.60	1.02	32.90	1.22	**33.80**	1.17	**33.80**	1.17
100, 100, 95	**7.10**	0.30	**7.10**	0.30	**7.10**	0.30	**7.10**	0.30
100, 200, 150	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00
100, 200, 170	28.80	1.08	**28.90**	1.04	**28.90**	1.04	**28.90**	1.04
100, 200, 190	**5.50**	0.50	**5.50**	0.50	**5.50**	0.50	**5.50**	0.50
100, 400, 300	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00
100, 400, 340	27.90	0.54	27.90	0.54	**28.00**	0.45	**28.00**	0.45
100, 400, 380	**4.20**	0.40	**4.20**	0.40	**4.20**	0.40	**4.20**	0.40
200, 200, 150	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00
200, 200, 170	30.90	0.94	30.80	0.98	**31.00**	0.89	**31.00**	0.89
200, 200, 190	**5.00**	0.00	**5.00**	0.00	**5.00**	0.00	**5.00**	0.00
200, 400, 300	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00
200, 400, 340	**30.20**	1.40	**30.20**	1.40	**30.20**	1.40	**30.20**	1.40
200, 400, 380	**3.60**	0.49	**3.60**	0.49	**3.60**	0.49	**3.60**	0.49
200, 800, 600	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00
200, 800, 680	**23.90**	0.94	**23.90**	0.94	**23.90**	0.94	**23.90**	0.94
200, 800, 760	**3.10**	0.30	**3.10**	0.30	**3.10**	0.30	**3.10**	0.30
300, 300, 225	296.90	2.02	299.00	1.10	**299.20**	0.87	298.80	1.40
300, 300, 255	**31.10**	0.94	31.00	1.00	31.00	1.00	31.00	1.00
300, 300, 285	**3.90**	0.30	3.80	0.40	3.80	0.40	3.80	0.40
300, 600, 450	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00
300, 600, 510	**25.50**	0.92	**25.50**	0.92	**25.50**	0.92	**25.50**	0.92
300, 600, 570	**2.40**	0.66	2.30	0.64	**2.40**	0.66	**2.40**	0.66
300, 1200, 900	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00
300, 1200, 1020	**21.40**	0.66	**21.40**	0.66	**21.40**	0.66	**21.40**	0.66
300, 1200, 1140	**1.00**	0.77	**1.00**	0.77	**1.00**	0.77	**1.00**	0.77

Table 3.3: *Comparison between the different hybrid GRASP with path-relinking strategies on real instances in set $\mathcal{B}$ after* 30 *seconds of running time.*

	grasp		grasp-h-f		grasp-h-b		grasp-h-m	
(n, m, t)	z	σ^2	z	σ^2	z	σ^2	z	σ^2
100, 100, 75	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00
100, 100, 85	63.00	0.82	63.33	0.47	**64.00**	0.00	63.00	0.82
100, 100, 95	**10.67**	1.25	**10.67**	1.25	**10.67**	1.25	**10.67**	1.25
100, 200, 150	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00
100, 200, 170	55.33	1.25	56.00	1.41	55.67	1.25	**56.33**	1.70
100, 200, 190	**8.00**	0.82	**8.00**	0.82	**8.00**	0.82	**8.00**	0.82
100, 400, 300	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00	**100.00**	0.00
100, 400, 340	**57.33**	2.05	**57.33**	2.05	**57.33**	2.05	**57.33**	2.05
100, 400, 380	**7.33**	0.47	**7.33**	0.47	**7.33**	0.47	**7.33**	0.47
200, 200, 150	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00
200, 200, 170	88.67	6.55	90.67	4.92	**92.67**	6.65	92.00	7.35
200, 200, 190	**10.00**	0.82	**10.00**	0.82	**10.00**	0.82	**10.00**	0.82
200, 400, 300	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00
200, 400, 340	72.67	5.25	73.00	4.97	**74.67**	4.64	73.67	4.92
200, 400, 380	7.00	0.00	**7.33**	0.47	6.67	0.47	7.00	0.00
200, 800, 600	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00	**200.00**	0.00
200, 800, 680	**77.33**	6.34	**77.33**	6.34	**77.33**	6.34	**77.33**	6.34
200, 800, 760	**3.67**	1.70	**3.67**	1.70	**3.67**	1.70	**3.67**	1.70
300, 300, 225	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00
300, 300, 255	99.33	10.66	104.00	11.58	105.67	9.98	**106.67**	9.98
300, 300, 285	6.00	0.82	**6.33**	0.47	**6.33**	0.47	6.00	0.82
300, 600, 450	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00
300, 600, 510	94.67	2.87	94.33	2.49	95.33	1.25	**95.67**	1.70
300, 600, 570	**2.67**	0.47	**2.67**	0.47	**2.67**	0.47	**2.67**	0.47
300, 1200, 900	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00	**300.00**	0.00
300, 1200, 1020	96.33	3.86	96.33	3.86	**96.67**	4.03	95.33	5.91
300, 1200, 1140	**1.67**	0.47	**1.67**	0.47	**1.67**	0.47	**1.67**	0.47

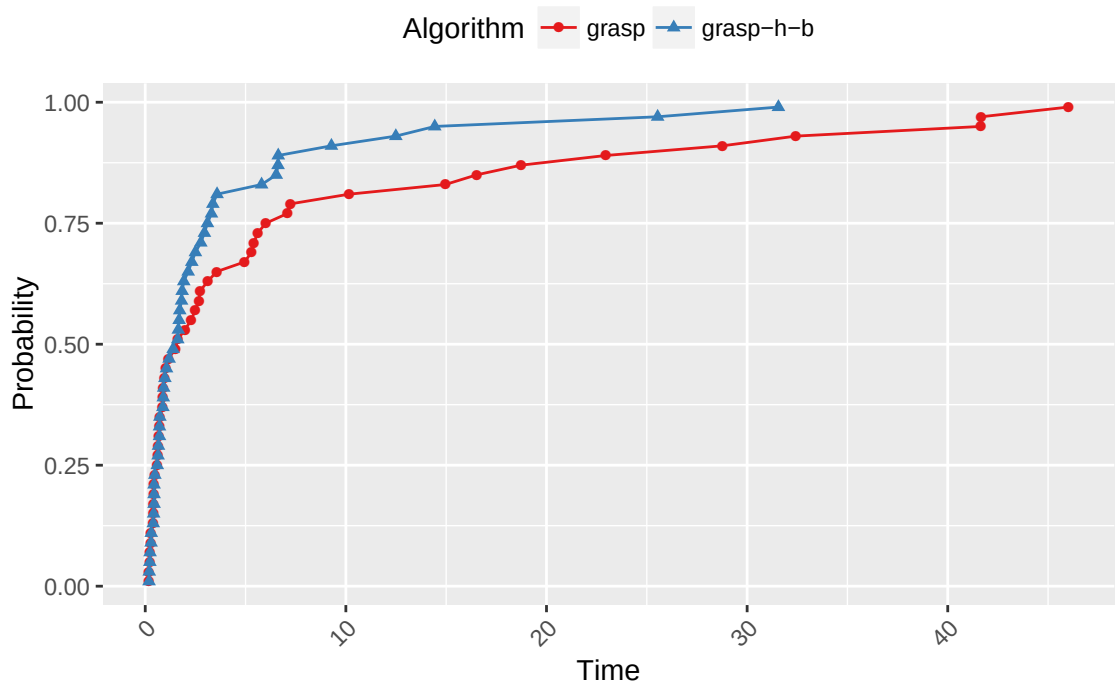


Figure 3.1: *Time to target distributions (in seconds) comparing* `grasp` *and* `grasp-h-b` *on random instance with* $n = 100$, $m = 300$, $t = 240$, *and target value* $\hat{z} = 0.73 \times n$.

3.2 p-Center

3.2.1 Introduction

The p-center problem is one of the best-known discrete location problems first introduced by Hakimi [36]. It consists of locating p facilities and assigning clients to them in order to minimize the maximum distance between a client and the facility to which the client is assigned (i.e., the closest facility). It is needless to say that this problem arises in many different real-world contexts, whenever one designs a system for public facilities, such as schools or emergency services.

Formally, given a complete undirected edge-weighted bipartite graph $G = (V \cup U, E, c)$, where

- $V = \{1, 2, \ldots, n\}$ is a set of n potential locations for facilities;
- $U = \{1, 2, \ldots, m\}$ is a set of m clients or demand points;
- $E = \{(i, j) |\ i \in V, j \in U\}$ is a set of $n \times m$ edges;
- $c : E \mapsto \mathbb{R}^+ \cup \{0\}$ is a function that assigns a nonnegative distance c_{ij} to each edge $(i, j) \in E$.

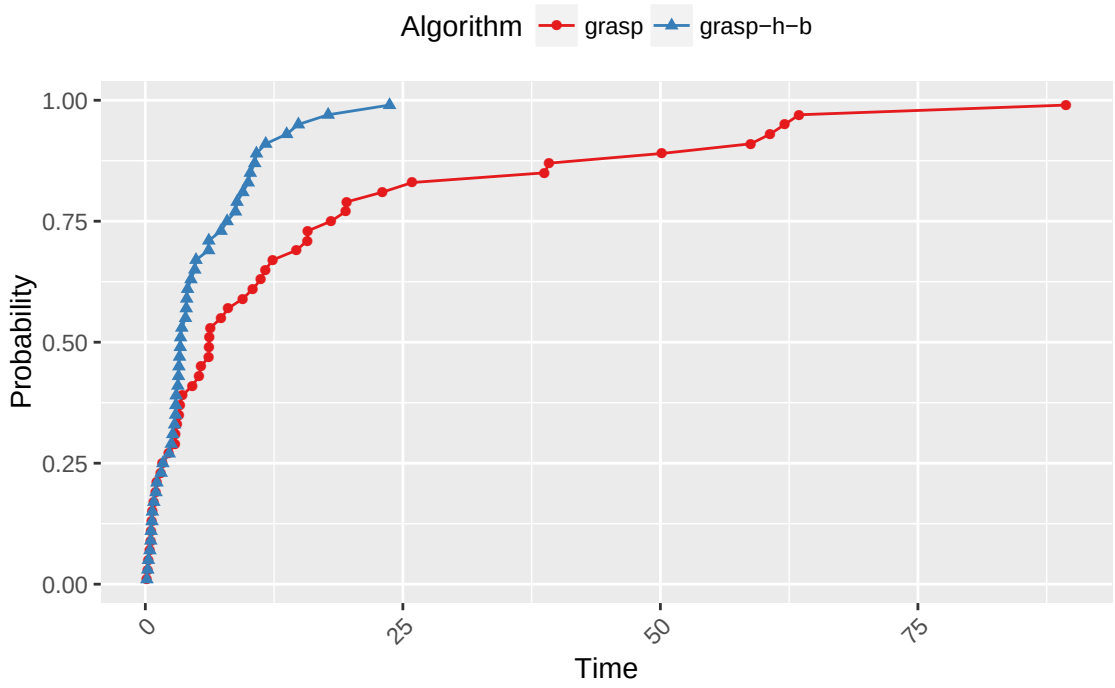


Figure 3.2: *Time to target distributions (in seconds) comparing* `grasp` *and* `grasp-h-b` *on random instance with* $n = 200$, $m = 300$, $t = 240$, *and target value* $\hat{z} = 0.445 \times n$.

The p-center problem is to find a subset $P \subseteq V$ of size p such that its weight, defined as

$$\mathbb{C}(P) = \max_{i \in U} \min_{j \in P} c_{ij} \tag{3.3}$$

is minimized. The minimum value is called the *radius*. Although it is not a restrictive hypothesis, in this work we consider the special case where $V = U$ is the vertex set of a complete graph $G = (V, E)$, each distance c_{ij} represents the length of a shortest path between vertices i and j ($c_{ii} = 0$), and hence the triangle inequality is satisfied. Kariv and Hakimi [41] proved that the problem is **NP**-hard, even in the case where the input instance has a simple structure (e.g., a planar graph of maximum vertex degree 3).

3.2.2 A new local search for the p-center

Ferone et al. [20] proposed a new local search integrated in a GRASP framework, in order to solve the p-center problem. The classical local search for p-center has been proposed by Hansen and Mladenović [37], here addressed as `Interchange`. It consists in swapping a facility $f \in P$ with a facility $\overline{f} \notin P$ which results in a decrease of the current cost function. Especially in the case of instances with many vertices, usually a single swap does not strictly improve the current solution, because there are several facilities whose

distance is equal to the radius of the solution. In other words, the objective function is characterized by large plateaus and the `Interchange` local search cannot escape from such regions. To face this type of difficulties, we propose a refined way for comparing between valid solutions by introducing the concept of *critical vertex*. Given a solution $P \subseteq V$, let $\delta_P : V \mapsto \mathbb{R}^+ \cup \{0\}$ be a function that assigns to each vertex $i \in V$ the distance between i and its closest facility according to solution P. Clearly, the cost of a solution P can be equivalently written as $\mathbb{C}(P) = \max\{\, \delta_P(i) \colon i \in V \,\}$.

Definition 3.1 (Critical vertex)**.** *Let $P \subseteq V$ be a solution whose cost is $\mathbb{C}(P)$. For each vertex $i \in V$, i is said to be a critical vertex for P, if and only if $\delta_P(i) = \mathbb{C}(P)$.*

In the following, we denote by $\max_{\delta_P} = |\{\, i \in V \colon \delta_P(i) = \mathbb{C}(P) \,\}|$ the number of vertices whose distance from their closest facility results in the objective function value corresponding to solution P. The comparison operator $<_{cv}$ is defined, and $P <_{cv} P'$ if and only if $\max_{\delta_P} < \max_{\delta'_P}$.

The main idea of the *plateau surfer* local search is to use the concept of critical vertex to escape from plateaus, moving to solutions that have either a better cost than the current solution or equal cost but fewer critical vertices.

Starting from an initial solution P, the algorithm tries to improve the solution replacing a vertex $j \notin P$ with a facility $i \in P$. Clearly, this swap is stored as an improving move if the new solution $\bar{P} = P \setminus \{i\} \cup \{j\}$ is strictly better than P according to the cost function $\mathbb{C}$. If $\mathbb{C}(\bar{P})$ is better than the current cost $\mathbb{C}(P)$, then $\bar{P}$ is compared also with the incumbent solution and if it is the best solution found so far, the incumbent is update and the swap that led to this improvement stored (lines 10 to 12 of Algorithm 3.2).

Otherwise, the algorithm checks if it is possible to reduce the number of critical vertices. If the new solution $\bar{P}$ is such that $\bar{P} <_{cv} P$, then the algorithm checks if $\bar{P}$ is the best solution found so far (line 13), the value that counts the number of critical vertices in a solution is updated (line 14), and the current swap stored as an improving move (line 15).

3.2.3 Experimental results

A GRASP framework with both the local search algorithms proposed by Mladenović et al. [47] and the `PlateauSurfer` has been tested. The algorithms were implemented in C++, compiled with `gcc 5.2.1` under Ubuntu with `-std=c++14` flag. The stopping criterion is $maxTime = 0.1 \cdot n + 0.5 \cdot p$. All the tests were run on a cluster of nodes, connected by 10 Gigabit Infiniband technology, each of them with two processors Intel Xeon E5-4610v2@2.30GHz.

Table 3.4 summarizes the results on a set of `ORLIB` instances, originally introduced by Beasley [3]. It consists of 40 graphs with number of vertices ranging from 100 to 900, each with a suggested value of p ranging from 5 to 200. Each vertex is both a user and a potential facility, and distances are given by shortest path lengths. Each algorithm was

Algorithm 3.2: *Pseudocode of the plateau surfer local search algorithm based on the critical vertex concept.*

```
1  Function PlateauSurfer(G = ⟨V, A, ℂ⟩, P, p)
2      repeat
3          modified ← false;
4          forall i ∈ P do
5              best_flip ← best_cv_flip ← Nil;
6              bestNewSolValue ← ℂ(P);
7              best_cv ← max_δ(P̄);
8              forall j ∈ V \ P do
9                  P̄ ← P \ {i} ∪ {j};
10                 if ℂ(P̄) < bestNewSolValue then
11                     bestNewSolValue ← ℂ(P̄);
12                     best_flip ← j;
13                 else if best_flip = Nil and max_δ(P̄) < best_cv then
14                     best_cv ← max_δ(P̄);
15                     best_cv_flip ← j;
16             if best_flip ≠ Nil then
17                 P ← P \ {i} ∪ {best_flip};
18                 modified ← true;
19             else if best_cv_flip ≠ Nil then
20                 P ← P \ {i} ∪ {best_cv_flip};
21                 modified ← true;
22     until modified = false;
23     return P;
```

run with 10 different seeds, and minimum (min), average (E) and variance (σ^2) values are listed in the table. The second to last column lists the %-Gap between average solutions. To deeper investigate the statistical significance of the results obtained by the two local searches, a t-test has been performed. The last column of the table lists the p-values where the %-Gap is significant, all the values are less than $\alpha = 0.01$. This outcome confirms that `PlateauSurfer` is better performing than `Interchange` local search.

3.3 Minimum Cost SAT Problem

3.3.1 Introduction

Propositional Satisfiability (***SAT***) and its derivations are well known problems in logic and optimization, and belong to the special class of **NP**-complete problems [34]. Beside playing a special role in the theory of complexity, they often arise in applications, where they are used to model complex problems whose solution is of particular interest.

One such case surfaces in logic supervised learning. Here, there is a dataset of samples, each represented by a finite number of logic variables, and a particular extension of the classic ***SAT*** problem - the *Minimum Cost Satisfiability Problem* (***MCS***) - can be used to iteratively identify the different clauses of a compact formula in *Disjunctive Normal Form* (DNF) that possesses the desirable property of assuming the value **True** on one specific subset of the dataset and the value **False** on the rest.

The use of ***MCS*** for learning propositional formula from data is described in Felici and Truemper [14] and Truemper [53]. There are several reasons that motivate the validity of such an approach to supervised learning, and that it proved to be very effective in several applications, particularly on those derived from biological and medical data analysis [1, 5].

One of the main drawbacks of the approach described in Felici and Truemper [14] lies in the difficulty of solving ***MCS*** exactly or with an appropriate quality level. Such drawback is becoming more and more evident as, in the era of Big Data, the size of the datasets that one is to analyze steadily increases. Feature selection techniques may be used to reduce the space in which the samples are represented.

The need for efficient ***MCS*** solvers is strong, and in particular for solvers that may take advantage of the specific structure of those ***MCS*** that represent supervised learning problems.

Indeed, in Felici et al. [13], a GRASP-based metaheuristic designed to solve ***MCS*** problems that arise in supervised learning is proposed. The method has been tested on several instances derived from artificial supervised problems in logic form, and successfully compared with three established solvers in the literature (**Z3** from Microsoft Research [48], **bsolo** [45], and **MiniSat+** [12]).

Table 3.4: *Results on ORLIB instances.*

	Interchange			PlateauSurfer				
Instance	min	E	σ^2	min	E	σ^2	%-Gap	p-value
pmed01	127	127	0	127	127	0	0.00	
pmed02	98	98	0	98	98	0	0.00	
pmed03	93	93.14	0.12	93	93.54	0.25	0.43	
pmed04	74	76.21	1.33	74	74.02	0.04	-2.87	1.20E-16
pmed05	48	48.46	0.43	48	48	0	-0.95	
pmed06	84	84	0	84	84	0	0.00	
pmed07	64	64.15	0.27	64	64	0	-0.23	
pmed08	57	59.39	1.36	55	55.54	0.73	-6.48	3.37E-18
pmed09	42	46.87	2.83	37	37.01	0.01	-21.04	2.80E-18
pmed10	29	31.21	0.81	20	20.01	0.01	-35.89	9.38E-19
pmed11	59	59	0	59	59	0	0.00	
pmed12	51	51.89	0.1	51	51.41	0.24	-0.93	
pmed13	42	44.47	0.73	36	36.94	0.06	-16.93	1.04E-18
pmed14	35	38.59	3.24	26	26.85	0.13	-30.42	2.11E-18
pmed15	28	30.23	0.7	18	18	0	-40.46	1.09E-18
pmed16	47	47	0	47	47	0	0.00	
pmed17	39	40.71	0.23	39	39	0	-4.20	8.69E-20
pmed18	36	37.95	0.29	29	29.41	0.24	-22.50	6.37E-19
pmed19	27	29.32	0.42	19	19.13	0.11	-34.75	6.25E-19
pmed20	25	27.05	0.99	14	14	0	-48.24	1.46E-18
pmed21	40	40	0	40	40	0	0.00	
pmed22	39	40.06	0.24	38	38.94	0.06	-2.80	1.30E-18
pmed23	30	32.02	0.44	23	23.21	0.17	-27.51	7.16E-19
pmed24	24	25.38	0.34	16	16	0	-36.96	4.37E-19
pmed25	22	22.62	0.24	11	11.89	0.1	-47.44	2.77E-19
pmed26	38	38	0	38	38	0	0.00	
pmed27	33	33.96	0.06	32	32	0	-5.77	2.15E-22
pmed28	26	26.78	0.17	19	19	0	-29.05	2.20E-20
pmed29	23	23.43	0.31	13	13.68	0.22	-41.61	8.00E-19
pmed30	20	21.18	0.47	10	10	0	-52.79	6.50E-19
pmed31	30	30	0	30	30	0	0.00	
pmed32	30	30.37	0.23	29	29.62	0.24	-2.47	
pmed33	23	23.76	0.2	16	16.28	0.2	-31.48	4.31E-19
pmed34	21	22.42	0.66	11	11.56	0.25	-48.44	1.59E-18
pmed35	30	30.01	0.01	30	30	0	-0.03	
pmed36	28	29.37	0.31	27	27.65	0.23	-5.86	4.52E-18
pmed37	23	24.07	0.37	16	16	0	-33.53	2.74E-19
pmed38	29	29	0	29	29	0	0.00	
pmed39	24	25.08	0.11	23	23.98	0.02	-4.39	4.68E-21
pmed40	20	21.81	0.43	14	14	0	-35.81	5.14E-19
						Average	-16.78	

3.3.2 Mathematical formulation of the problem

The Minimum Cost SAT Problem (***MCS***) - also known as Binate Covering Problem - is a special case of the well known Boolean Satisfiability Problem. Given a set of n Boolean variables $X = \{x_1, \ldots, x_n\}$, a non-negative cost function $c\colon X \mapsto \mathbb{R}^+$ such that $c(x_i) = c_i \geq 0$, $i = 1, \ldots, n$, and a Boolean formula $\varphi(X)$ expressed in CNF, the ***MCS*** problem consists in finding a truth assignment for the variables in X such that the total cost is minimized while $\varphi(X)$ is satisfied. Accordingly, the mathematical formulation of the problem is given as follows:

$$\text{(MinCostSAT)} \quad z = \min \sum_{i=1}^{n} c_i x_i$$

subject to:

$$\varphi(X) = 1,$$
$$x_i \in \{0, 1\}, \qquad \forall i = 1, \ldots, n.$$

It is easy to see that a general SAT problem can be reduced to a ***MCS*** problem whose costs c_i are all equal to 0. Furthermore, the decision version of the ***MCS*** problem is **NP**-complete [32].

3.3.3 A GRASP for Minimum Cost SAT

In order to allow a better and easier implementation of the GRASP algorithm, the ***MCS*** has been treated as particular covering problem with incompatibility constraints. Indeed, each literal $(x, \neg x)$ is considered as a separate element, and a clause is covered if at least one literal in the clause is contained in the solution. The algorithm tries to add literals to the solution in order to cover all the clauses and, once the literal x is added to the solution, then the literal $\neg x$ cannot be inserted (and vice versa). Therefore, if the literal x is in solution, the variable x is assigned to true and all clauses covered by x are satisfied. Similarly, if the literal $\neg x$ is in solution, the variable x is assigned to false.

The construction phase adds a literal at a time, until all clauses are covered or no more literals can be assigned. At each iteration of the construction, if a clause can be covered only by a single literal x –due to the choices made in previous iterations– then x is selected to cover the clause. Otherwise, if there are not clauses covered by only a single literal, the addition of literals to the solution takes place according to a penalty function $penalty(\cdot)$, which greedily sorts all the candidates literals, as described below.

Let $cr(x)$ be the number of clauses yet to be covered that contain x. We then compute:

$$penalty(x) = \frac{c(x) + cr(\neg x)}{cr(x)}. \tag{3.4}$$

This penalty function evaluates both the benefits and disadvantages that can result

from the choice of a literal rather than another. The benefits are proportional to the number of uncovered clauses that the chosen literal could cover, while the disadvantages are related to both the cost of the literal and the number of uncovered clauses that could be covered by $\neg x$. The smaller the penalty function $penalty(x)$, the more favorable is the literal x. According to the GRASP scheme, the selection of the literal to add is not purely greedy, but a Restricted Candidate List (RCL) is created with the most promising elements, and an element is randomly selected among them.

Algorithm 3.3: *Pseudo-code of GRASP construction phase.*

```
1  Function ConstructSolution(C, X, β)
       /* C is the set of uncovered clauses                          */
       /* X is the set of candidate literals                         */
2      s ← ∅ ;
3      while C ≠ ∅ do
4          if c ∈ C can be covered only by x ∈ X then
5              s ← s ∪ {x};
6              X ← X \ {x, ¬x};
7              C ← C \ {c̄ | x ∈ c̄};
8          else
9              compute penalty(x) ∀ x ∈ X;
10             th ← min_{x∈X}{penalty(x)} + β(max_{x∈X}{penalty(x)} − min_{x∈X}{penalty(x)}) ;
11             RCL ← { x ∈ X : penalty(x) ≤ th } ;
12             x̂ ← rand(RCL) ;
13             s ← s ∪ {x̂};
14             X ← X \ {x̂, ¬x̂};
15             C ← C \ {c̄ | x̂ ∈ c̄};
16     return s
```

Let $|C| = m$ be the number of clauses. Since $|X| = 2n$, in the worst case scenario the **while** loop (line 3) in the `construct-solution` function pseudo-coded in Algorithm 3.3 runs m times and in each run the most expensive operation consists in the construction of the RCL. Therefore, the computational complexity is $O(m \cdot n)$.

In the local search phase, the algorithm uses a 1-exchange (flip) neighborhood function, where two solutions are neighbors if and only if they differ in at most one component each other. Therefore, if there exists a better solution $\bar{s}$ that differs only for one literal from the current solution s, the current solution s is set to $\bar{s}$ and the procedure restarts. If such a solution does not exists, the procedure ends and returns the current solution s. The local search procedure would also re-establish feasibility if the current solution is not covering all clauses of $\varphi(X)$. A best improvement strategy has been used.

3.3.4 Experimental results

GRASP has been implemented in C++ and compiled with `gcc 5.4.0` with the flag `-std=c++14`. All tests were run on a cluster of nodes, connected by 10 Gigabit Infiniband technology, each of them with two processors Intel Xeon E5-4610v2@2.30GHz.

The algorithm has been compared with different solvers proposed in literature. In particular, it has been used: **Z3** solver freely available from Microsoft Research [48], **bsolo** solver kindly provided by the authors Manquinho and Marques-Silva [45], and the **MiniSat+** [12] available at web page `http://minisat.se/`. The aim of computational experiment is the evaluation of the quality of the solutions obtained by GRASP within a certain time limit. More specifically, the stopping criterion for GRASP and **bsolo** is a time limit of 3 hours, for **Z3** and **MiniSat+** is the reaching of an optimal solution.

For testing, the datasets used to test feature selection methods in Bertolazzi et al. [4] have been used. Such testbed is composed of 4 types of problems (A,B,C,D), for each of which 10 random repetitions have been generated. Problems of type A and B are of moderate size (100 positive examples, 100 negative examples, 100 logic features), but differ in the form of the formula used to classify the samples into the positive and negative classes (the formula being more complex for B than for A). Problems of type C and D are much larger (200 positive examples, 200 negative examples, 2500 logic features), and D has a more complex generating logic formula than C.

Table 3.5 reports both the value of the solutions and the time needed to achieve them (in the case of GRASP, it is average over ten runs).[1] For problems of moderate size (A and B), the results show that GRASP finds an optimal solution whenever one of the exact solvers converges. Moreover, GRASP is very fast in finding the optimal solution, although here it runs the full allotted time before stopping the search. For larger instances (C and D), GRASP always provides a solution within the bounds, while two of the other tested solvers fail in doing so and the only one that is successful (**bsolo**) always obtains values of inferior quality.

3.4 Biased Randomized SimGRASP

Many real-life combinatorial optimization problems (COPs) are shaped by large problem sizes and inherent complexity through various problem constraints. Typically, this results in **NP**-hard problem settings which cannot be solved to optimality in reasonable computing times. In this context, metaheuristics have proven to be very efficient in finding near-optimal solutions to a wide range of problem settings. Nevertheless, as a drawback, generally metaheuristics are able to solve only deterministic problem settings. Real-life problems often present uncertainty, resulting in stochastic versions of the classical deter-

[1] For missing values, the algorithm was not able to find the optimal solution in 24 hours.

Table 3.5: *Comparison between GRASP and other **MCS** solvers.*

	GRASP		Z3		bsolo		MiniSat+	
Instance	Time (s)	Value	Time (s)	Value	Time (s)	Value	Time (s)	Value
A1	6.56	78.0	10767.75	78.0	0.09	78.0	0.19	78.0
A2	1.71	71.0	611.29	71.0	109.59	71.0	75.46	71.0
A3	0.64	65.0	49.75	65.0	598.71	65.0	10.22	65.0
A4	0.18	58.0	4.00	58.0	205.77	58.0	137.82	58.0
A5	0.29	66.0	69.31	66.0	331.51	66.0	9.03	66.0
A6	21.97	77.0	5500.17	77.0	328.93	77.0	32.82	77.0
A7	0.21	63.0	30.57	63.0	134.20	63.0	19.34	63.0
A8	0.25	62.0	6.57	62.0	307.69	62.0	16.84	62.0
A9	12.79	72.0	1088.83	72.0	3118.32	72.0	288.76	72.0
A10	0.33	66.0	42.23	66.0	62.03	66.0	37.75	66.0
B1	6.17	78.0	8600.60	78.0	304.36	78.0	121.25	78.0
B2	493.56	80.0	18789.20	80.0	4107.41	80.0	48.21	80.0
B3	205.37	77.0	7037.00	77.0	515.25	77.0	132.74	77.0
B4	38.26	77.0	7762.03	77.0	376.00	77.0	119.49	77.0
B5	19.89	79.0	15785.35	79.0	3025.26	79.0	214.52	79.0
B6	28.45	76.0	4087.14	76.0	394.45	76.0	162.31	76.0
B7	129.76	78.0	10114.84	78.0	490.30	78.0	266.25	78.0
B8	44.42	76.0	5186.45	76.0	5821.19	76.0	1319.21	76.0
B9	152.77	80.0	14802.00	80.0	5216.95	82.0	36.28	80.0
B10	7.55	73.0	1632.87	73.0	760.28	79.0	370.30	73.0
C1	366.24	132.0	86400	–	8616.25	178.0*	86400	–
C2	543.11	131.0	86400	–	323.90	150.0*	86400	–
C3	5883.6	174.1	86400	–	6166.06	177.0*	86400	–
C4	4507.63	176.3	86400	–	6209.69	178.0*	86400	–
C5	5707.51	171.2	86400	–	314.18	179.0*	86400	–
C6	6269.91	172.1	86400	–	1547.90	177.0*	86400	–
C7	6193.15	165.9	86400	–	794.90	177.0*	86400	–
C8	596.58	137.0	86400	–	306.27	169.0*	86400	–
C9	466.3	136.0	86400	–	433.32	179.0*	86400	–
C10	938.54	136.0	86400	–	3703.94	180.0*	86400	–
D1	3801.61	145.3	86400	–	307.25	175.0*	86400	–
D2	2040.64	139.0	86400	–	7704.92	177.0*	86400	–
D3	1742.78	143.0	86400	–	309.10	145.0*	86400	–
D4	1741.95	135.0	86400	–	6457.79	177.0*	86400	–
D5	1506.22	134.0	86400	–	6283.27	178.0*	86400	–
D6	1960.87	144.5	86400	–	309.11	173.0*	86400	–
D7	1544.42	143.0	86400	–	4378.73	179.0*	86400	–
D8	1756.15	144.0	86400	–	1214.97	179.0*	86400	–
D9	2779.38	137.0	86400	–	303.11	146.0*	86400	–
D10	5896.86	149.0	86400	–	319.45	170.0*	86400	–
Y	16.05	0.0	0.73	0.0	9411.06	974*	1.96	0

*sub-optimal solution

– no optimal solution found in 24 hours

ministic COPs.

We next describe *SimGRASP*. By integrating simulation at different stages of GRASP, this constitutes an accessible way to tackle problem settings under stochastic input uncertainty. SimGRASP can be characterized as so called *simheuristic* [39]. Furthermore, another possible extension of traditional GRASP is the *Biased Randomized GRASP*, that tries to simplify and improve the construction phase of the classical GRASP.

3.4.1 BR-GRASP and SimGRASP methodologies

Introducing biased randomization during the construction phase

The first enhancement to the traditional GRASP introduces biased randomization in the construction phase of the algorithm. This technique is proposed in Ferone et al. [25], in contrast to the classical framework, the BR-GRASP methodology does not rely on a RCL to build a feasible solution. The main idea is similar to Bresina [8], but it has never combined in a GRASP framework. As shown in Algorithm 3.4, the first steps of the construction phase and the creation of a CL (lines 2-4) are similar to the ones applied in the traditional GRASP. However, the element selection process varies in a couple of points. On the one hand, a non-symmetric, non-uniform distribution function is applied to define the probabilities of including element $e \in CL$ in the current solution. Different skewed (non-symmetric) probability distribution can be used at this point, e.g., geometric or triangular distributions. Thus, the probabilities are biased towards the most promising solution elements. On the other hand, the BR-GRASP framework does not restrict the candidate list of elements. This means that all feasible and non-selected elements are potentially eligible. Figure 3.3 highlights the difference between the traditional- and the biased randomization construction phase.

Algorithm 3.4: *Construction phase with Biased Randomization.*

```
1  Function BiasedConstruction(D, β)
2      s ← ∅;
3      initialize candidate set: CL ← E;
4      order the Candidate List (CL) elements according to c(·) ;
5      while solution s is not complete do
6          Randomly select pos ∈ {1, ..., |CL|} according to distribution D(β);
7          s ← s ∪ {CL[pos]};
8          CL ← CL \ {CL[pos]} ;
9          Reorder CL;
10     return s;
```

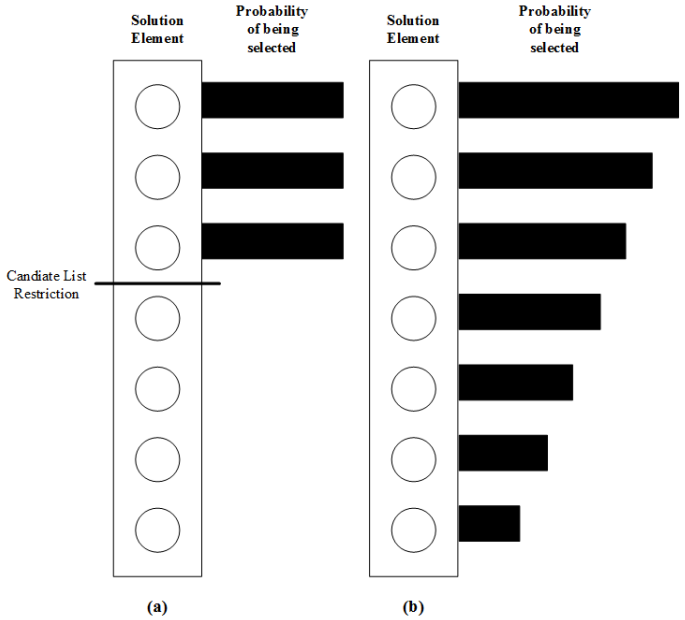


Figure 3.3: *(a) Traditional GRASP element selection process; (b) BR-GRASP element selection process.*

3.4.2 Including simulation in GRASP

Like other metaheuristics, GRASP alone (with or without biased randomization) is not able to consider any input uncertainty. The SimGRASP framework, presented in Ferone et al. [17], and outlined in the following overcomes this drawback by including simulation at different stages of the algorithm. In general, this simheuristic methodology does not depend on the GRASP construction phase. Thus, both GRASP and BR-GRASP could be applied. However, as discussed by Juan et al. [39], the quality of stochastic solutions are directly related to the deterministic output.

The general SimGRASP framework is outlined in Algorithm 3.5. Given a stochastic COP, the problem settings is transformed into its deterministic counterpart (line 1). Considering a set X of stochastic variables, each variable $x \in X$ is transformed into a deterministic value x^* by considering the expected values $E[x] = x^*$. Using the deterministic values, an initial solution is constructed using the BR-GRASP framework (lines 2-3). Then, the deterministic solution is evaluated in an uncertainty environment by applying simulation (line 4). During $nIter_{short}$ simulation runs, all stochastic variables are hereby simulated from any suitable empirical or theoretical probability distribution, using the expected values x^* as distribution mean. Moreover, the variability of the stochastic variable is defined by parameter k. Note that, high uncertainty levels can be modeled by increasing this value, while $k = 0$ is equal to the deterministic case. The simulation procedure leads to important decision making information. Not only can the stochastic costs be evaluated, but valuable solution information and statistics of different stochastic COP solutions can

Algorithm 3.5: *General SimGRASP framework.*

```
1  Function SimGRASP(Stochastic COP)
2      Transform stochastic COP into deterministic counterpart;
3      s_0 ← BiasedConstruction(·);
4      s* ← LocalSearch(s_0);
5      (s*, sf(s*), statistics) ← Simulation (s*, short);
6      while stopping criterion not reached do
7          s** ← GenerateSolution;
8          s** ← LocalSearch(s**)
10         (s**, sf(s**), statistics) ← Simulation (s**, short)
12         if sf(s**) < sf(s*) then
14             EliteSolutions ← add(s**)
16             s* ← s**
18     foreach solution s ∈ EliteSolutions do
20         (s, sf(s), statistics) ← Simulation (s, long)
22     return Set of stochastic solutions;
```

be obtained. Finally, the initial solution s^* is set as current best solution.

3.4.3 Experiments

To show the functionality and competitiveness of both GRASP adaptions, BR-GRASP and SimGRASP have been applied to a well-known problem setting related to production, transportation, and logistics: the capacitated vehicle routing problem (***VRP***). In addition to solving the deterministic case with BR-GRASP, the problem is also considered in its stochastic version, the ***VRP*** with stochastic demands (***VRPSD***). This problem is defined by its large size and **NP**-hardness, making the use of metaheuristics and their extensions suitable to solve it.

The stochastic variables x_i are generated according to a log-normal distribution function, which considers the deterministic values x_i^* as distribution mean and a variance factor k. Thus, the log-normal probability function can be formulated through a location parameter, μ_i, and a scale parameter σ_i, with $E[D_i] = x_i$ and $Var[D_i] = k \cdot x_i$, such that:

$$\mu_i = \ln(E[D_i]) - \frac{1}{2} \cdot \ln\left(1 + \frac{Var[D_i]}{E[D_i]^2}\right)$$

$$\sigma_i = \left|\sqrt{\ln\left(1 + \frac{Var[D_i]}{E[D_i]^2}\right)}\right|$$

All experiments were implemented as Java application and run on Eclipse on an Ubuntu operating system with a personal computer with an Intel i7 Quad core, 2.67 GHz clock, and 6 GB RAM. Geometric distribution with β parameter has been used in

`BiasedConstruction`.

The ***VRP*** can be formulated on a graph $G = (V, E)$, whereby customer set $V = 0, 1, \ldots, n$ describes n clients with demand $d_i, i \in V \setminus \{0\}$, which have to be served by a set of vehicles located at the central depot $0 \in V$. Travel distances between any two nodes are described by a set E of weighted edges. Typically, the objective function consists of the minimization of overall travel distances or costs [52].

In order to compare BR-GRASP with the traditional GRASP metaheuristic implementation, threshold parameter a was set to 0.3, while the geometric distribution parameter β was set to 0.5 respectively. As test instance, the sets A and B of the benchmark set for the capacitated VRP proposed by Augerat et al. [2] are used. The algorithms running time for the deterministic case was set to 30 seconds for both algorithm implementations. All results are compared to the optimal results for the chosen instances elaborated in the work of Fukasawa et al. [33]. Furthermore, the ***VRPSD*** was used to evaluate the performance of the SimGRASP methodology. As performance indicator, the results obtained with the simheuristic based on a multi-start heuristic procedure discussed by Juan et al. [40] are used. As done in their paper, the deterministic demand levels d_i given by the benchmarks are turned into their stochastic counterpart by applying different variance levels k, whereby a variance level of $k = 0.25$ is used to obtain the results reported in this work. Computational running times are set to 10 seconds for the stochastic case.

As summarized in Table 3.6, the traditional implementation of GRASP is on average 4.59% worse than the optimal solution. This gap can be significantly reduced to 1.88% by introducing a biased randomized edge selection process. When comparing the combination of a multi-start heuristic with simulation to our SimGRASP algorithm, it can be seen that SimGRASP outperforms previous simulation-optimization approaches by -9.81%.

Table 3.6: *Performance of BR-GRASP and SimGRASP for the VRP.*

Instance	BKS (1)	GRASP (2)	BR-GRASP (3)	%-Gap (1)-(2)	%-Gap (1)-(3)	SimMultiStart (4)	SimGRASP (5)	%-Gap (4)-(5)
A-n32-k5	787.08	807.09	787.2	2.54	0.02	993.20	890.95	-10.30
A-n33-k5	662.11	687.91	662.11	3.90	0.00	815.40	750.63	-7.94
A-n33-k6	742.69	768.84	742.69	3.52	0.00	912.60	837.63	-8.21
A-n37-k5	664.8	707.81	685.26	6.47	3.08	795.00	734.44	-7.62
A-n38-k5	716.5	768.13	747.14	7.21	4.28	885.10	824.37	-6.86
A-n39-k6	822.8	863.08	835.25	4.90	1.51	1010.60	926.11	-8.36
A-n45-k6	938.1	1006.45	957.06	7.29	2.02	1184.30	1091.36	-7.85
A-n45-k7	1139.3	1199.98	1155.22	5.33	1.40	1502.00	1336.08	-11.05
A-n55-k9	1067.4	1099.84	1088.45	3.04	1.97	1408.40	1258.72	-10.63
A-n60-k9	1344.4	1421.88	1363.58	5.76	1.43	1795.70	1579.79	-12.02
A-n61-k9	1022.5	1102.23	1042.96	7.80	2.00	1330.60	1224.44	-7.98
A-n63-k9	1607	1687.96	1649.33	5.04	2.63	2203.70	1897.29	-13.90
A-n65-k9	1166.5	1239.42	1197.49	6.25	2.66	1555.30	1437.84	-7.55
A-n80-k10	1754	1860.94	1798.01	6.10	2.51	2328.40	2177.10	-6.50
B-n31-k5	676.09	681.16	676.09	0.75	0.00	855.70	757.60	-11.46
B-n35-k5	956.29	978.33	961.77	2.30	0.57	1255.50	1098.00	-12.54
B-n39-k5	549	566.71	553.27	3.23	0.78	695.90	621.34	-10.71
B-n41-k6	826.4	878.3	844.7	6.28	2.21	1103.20	1005.62	-8.84
B-n45-k5	747.5	757.16	754.23	1.29	0.90	904.60	828.04	-8.46
B-n50-k7	741	748.8	744.23	1.05	0.44	945.80	859.48	-9.13
B-n52-k7	745.8	764.9	755.85	2.56	1.35	944.40	848.71	-10.13
B-n56-k7	704	733.74	719.03	4.22	2.13	920.00	845.37	-8.11
B-n57-k9	1596	1653.42	1602.92	3.60	0.43	2199.70	1885.69	-14.27
B-n64-k9	859.3	921.56	903.43	7.25	5.14	1179.60	1064.53	-9.75
B-n67-k10	1024.4	1099.95	1086.01	7.38	6.01	1404.50	1247.67	-11.17
B-n68-k9	1263	1317.77	1305.32	4.34	3.35	1754.70	1515.88	-13.61
Average				***4.59***	***1.88***			***-9.81***

Chapter 4

Conclusions

In this thesis the Shortest Path Tour Problem and its variants have been studied. The ***SPTP*** is a generalized *Shortest Path Problem*, where given some disjoint node subset T_i, $i = 1, \ldots, N$, the shortest path must visits at least one node for each subset in the given order.

Here, two main variants of the problem have been considered. The former is the *Constrained Shortest Path Tour Problem*, in which a feasible solution must not include any repeated arc. The latter is the *Shortest Path Tour Problem with Time Windows*, where to each constrained node is assigned a time window in which the node can be served.

On one hand the general Shortest Path Tour Problem is polynomially solvable. On the other hand, the two variants are **NP**-hard problems.

For the ***CSPTP***, two different mathematical formulations and two exact approaches have been proposed. Moreover, due to the complexity of the problem, a GRASP meta-heuristic has been applied to solve the problem.

An extensive computational study has been carried out on a variety of network instances with the goal of assessing the behaviour of the proposed solution procedures. The computational results obtained have shown that, given the intrinsic complexity of the ***CSPTP***, the second Branch & Bound (B&B^{new}) method shows good results on medium size instances. Furthermore, when the network is dense B&B^{new} is able to tackle big instances, too. Nevertheless, the GRASP meta-heuristic can be very useful on general big instances, where the exact techniques fails to solve the problem, due to its theoretical complexity.

The research on the ***SPTPTW*** is still at an initial stage, but an exact dynamic programming algorithm and several bounding strategies to improve its performance have been proposed. The algorithm is able to cope medium size instances, but it is needed to propose some sort of (meta-)heuristic, that can solve large size instances in reasonable times.

A lot of work has still to be done. The problem has been proposed recently, and the scientific literature is poor. Therefore, several research lines are open. An interesting

generalization of the ***SPTPTW***, is the *Shortest Path Tour Problem with* ***Tour*** *Windows*. In this case, the nodes could belong to different subset T_i depending in the time they are visited. Tour windows $[e_v^i, l_v^i]$ are associated with each node and indicates the interval of time in which the node v belongs to the subset T_i.

Furthermore, an interesting and promising research line could be the application of the aforementioned problems to more realistic-case scenarios, where the determinism is resigns in favor of stochasticity and uncertainly.

In this thesis different side projects have been also presented. A common solving approach, GRASP meta-heuristic, has been proposed for all the problems addressed here. Although the GRASP has been largely studied and several hybridization have been proposed, to the best of my knowledge it is the first time that simulation has been integrated in this well known framework. All existing GRASP variants and extensions can be integrated in this new strategy, resulting in new tools to cope with many real-life problems.

Bibliography

[1] I. Arisi, M. D'Onofrio, R. Brandi, A. Felsani, S. Capsoni, G. Drovandi, G. Felici, E. Weitschek, P. Bertolazzi, and A. Cattaneo. "Gene expression biomarkers in the brain of a mouse model for Alzheimer's disease: mining of microarray data by logic classification and feature selection". *Journal of Alzheimer's Disease* 24.4 (2011), pp. 721–738 (cit. on p. 78).

[2] P. Augerat, J. M. Belenguer, E. Benavent, A. Corberàn, D. Naddef, and R. Giovanni. *Computational results with a branch and cut code for the capacitated vehicle routing problem*. Tech. rep. IASI Research Report n. 495. Rome, Italy: Institute for Systems Analysis and Computer Science, 1998. URL: http://www.iasi.cnr.it/reports%5C_html/R495 (cit. on p. 87).

[3] J. E. Beasley. "A note on solving large *p*-median problems". *European Journal of Operational Research* 21 (1985), pp. 270–273 (cit. on p. 76).

[4] P. Bertolazzi, G. Felici, P. Festa, G. Fiscon, and E. Weitschek. "Integer programming models for feature selection: New extensions and a randomized solution algorithm". *European Journal of Operational Research* 250.2 (2016), pp. 389–399 (cit. on p. 82).

[5] P. Bertolazzi, G. Felici, and E. Weitschek. "Learning to classify species with barcodes". *BMC bioinformatics* 10.14 (2009), p. 1 (cit. on p. 78).

[6] D. P. Bertsekas. "An Auction Algorithm for Shortest Paths". *SIAM Journal on Optimization* 1.4 (1991), pp. 425–447 (cit. on p. 16).

[7] D. P. Bertsekas. *Dynamic Programming and Optimal Control*. 3rd. Vol. I. Athena Scientific, 2005 (cit. on p. 17).

[8] J. L. Bresina. "Heuristic-biased Stochastic Sampling". *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*. AAAI'96. AAAI Press, 1996, pp. 271–278 (cit. on p. 84).

[9] R. Cerulli, P. Festa, and G. Raiconi. "Shortest Path Auction Algorithm Without Contractions Using Virtual Source Concept". *Computational Optimization and Applications* 26.2 (2003), pp. 191–208. ISSN: 1573-2894 (cit. on p. 16).

[10] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001 (cit. on p. 59).

[11] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs”. *Numer. Math.* 1 (Dec. 1959), pp. 269–271 (cit. on p. 16).

[12] N. Eén and N. Sörensson. “Translating pseudo-boolean constraints into SAT”. *Journal on Satisfiability, Boolean Modeling and Computation* 2 (2006), pp. 1–26 (cit. on pp. 78, 82).

[13] G. Felici, D. Ferone, P. Festa, A. Napoletano, and T. Pastore. “A GRASP for the Minimum Cost SAT Problem”. *Proceedings of 11th Learning and Intelligent OptimizatioN Conference*. Vol. Lecture Notes in Computer Science. 2017. Forthcoming (cit. on p. 78).

[14] G. Felici and K. Truemper. “A minsat approach for learning in logic domains”. *INFORMS Journal on computing* 14.1 (2002), pp. 20–36 (cit. on p. 78).

[15] T. A. Feo and M. G. C. Resende. “A probabilistic heuristic for a computationally difficult set covering problem”. *Operations Research Letters* 8.2 (1989), pp. 67–71 (cit. on p. 28).

[16] T. A. Feo and M. G. C. Resende. “Greedy Randomized Adaptive Search Procedures”. *Journal of Global Optimization* 6 (1995), pp. 109–133 (cit. on p. 28).

[17] D. Ferone, P. Festa, A. Gruler, and A. A. Juan. “Combining simulation with a GRASP metaheuristic for solving the permutation flow-shop problem with stochastic processing times”. *2016 Winter Simulation Conference (WSC)*. Dec. 2016, pp. 2205–2215 (cit. on p. 85).

[18] D. Ferone, P. Festa, and F. Guerriero. “An Efficient Exact Approach for the Constrained Shortest Path Tour Problem”. *Computers & Operations Research* (2017). Submitted (cit. on p. 40).

[19] D. Ferone, P. Festa, F. Guerriero, and D. Laganà. “The constrained shortest path tour problem”. *Computers & Operations Research* 74 (2016), pp. 64–77 (cit. on p. 19).

[20] D. Ferone, P. Festa, and A. Napoletano. “A new local search for the p-center problem based on the critical vertex concept”. *Proceedings of 11th Learning and Intelligent OptimizatioN Conference*. Vol. Lecture Notes in Computer Science. 2017. Forthcoming (cit. on p. 75).

[21] D. Ferone, P. Festa, A. Napoletano, and T. Pastore. “Reoptimizing shortest paths: From state of the art to new recent perspectives”. *2016 18th International Conference on Transparent Optical Networks (ICTON)*. 2016, pp. 1–5 (cit. on p. 16).

[22] D. Ferone, P. Festa, and M. G. C. Resende. “Hybrid metaheuristics for the far from most string problem”. *Proceedings of 8th International Workshop on Hybrid Metaheuristics*. Vol. 7919 of Lecture Notes in Computer Science. 2013, pp. 174–188 (cit. on p. 70).

[23] D. Ferone, P. Festa, and M. G. C. Resende. "On the Far from Most String Problem, One of the Hardest String Selection Problems". *Dynamics of Information Systems: Computational and Mathematical Challenges*. Ed. by C. Vogiatzis, J. L. Walteros, and P. M. Pardalos. Cham: Springer International Publishing, 2014, pp. 129–148 (cit. on p. 67).

[24] D. Ferone, P. Festa, and M. G. C. Resende. "Hybridizations of GRASP with path relinking for the far from most string problem". *International Transactions in Operational Research* 23.3 (2016), pp. 481–506. ISSN: 1475-3995 (cit. on p. 69).

[25] D. Ferone, A. Gruler, P. Festa, and A. A. Juan. "Enhancing and Extending the Classical GRASP Framework with Biased Randomization and Simulation". *Journal of the Operational Research Society* (2017). Submitted (cit. on p. 84).

[26] P. Festa. "Complexity analysis and optimization of the shortest path tour problem". *Optimization Letters* 6.1 (2012), pp. 163–175 (cit. on pp. 17, 18, 33).

[27] P. Festa, F. Guerriero, D. Laganà, and R. Musmanno. "Solving the shortest path tour problem". *European Journal of Operational Research* 230.3 (2013), pp. 464–474 (cit. on p. 18).

[28] P. Festa and S. Pallottino. *A pseudo-random networks generator*. Tech. rep. Department of Mathematics and Applications "R. Caccioppoli", University of Napoli FEDERICO II, Italy, 2003 (cit. on pp. 33, 45, 46).

[29] P. Festa and M. G. C. Resende. "GRASP: An annotated bibliography". *Essays and surveys in metaheuristics*. Springer, 2002, pp. 325–367 (cit. on p. 28).

[30] P. Festa and M. G. C. Resende. "An annotated bibliography of GRASP - Part I: Algorithms". *International Transactions in Operational Research* 16.1 (2009), pp. 1–24 (cit. on p. 28).

[31] P. Festa and M. G. C. Resende. "An annotated bibliography of GRASP - Part II: Applications". *International Transactions in Operational Research* 16.2 (2009), pp. 131–172 (cit. on p. 28).

[32] Z. Fu and S. Malik. "Solving the minimum-cost satisfiability problem using sat based branch-and-bound search". *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*. 2006, pp. 852–859 (cit. on p. 80).

[33] R. Fukasawa, H. Longo, J. Lysgaard, M. P. de Aragão, M. Reis, E. Uchoa, and R. F. Werneck. "Robust Branch-and-Cut-and-Price for the Capacitated Vehicle Routing Problem". *Mathematical Programming* 106.3 (2006), pp. 491–511 (cit. on p. 87).

[34] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979 (cit. on pp. 59, 78).

[35] F. Glover. "Tabu search and adaptive memory programing – Advances, applications and challenges". *Interfaces in Computer Science and Operations Research*. Ed. by R.S. Barr, R.V. Helgason, and J.L. Kennington. Kluwer, 1996, pp. 1–75 (cit. on p. 68).

[36] S. L. Hakimi. "Optimum Locations of Switching Centers and the Absolute Centers and Medians of a Graph". *Operations Research* 12.3 (1964), pp. 450–459 (cit. on p. 74).

[37] P. Hansen and N. Mladenović. "Variable neighborhood search for the p-median". *Location Science* 5.4 (1997), pp. 207–226 (cit. on p. 75).

[38] P. E. Hart, N. J. Nilsson, and B. Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968), pp. 100–107 (cit. on p. 16).

[39] A. A. Juan, J. Faulin, S. E. Grasman, M. Rabe, and G. Figueira. "A review of simheuristics: extending metaheuristics to deal with stochastic combinatorial optimization problems". *Operations Research Perspectives* 2 (2015), pp. 62–72 (cit. on pp. 84, 85).

[40] A. A. Juan, J. Faulin, J. Jorba, J. Caceres, and J. M. Marquès. "Using parallel & distributed computing for real-time solving of vehicle routing problems with stochastic demands". *Annals of Operations Research* 207.1 (2013), pp. 43–65 (cit. on p. 87).

[41] O. Kariv and S. L. Hakimi. "An Algorithmic Approach to Network Location Problems. Part I: The p-Centers". *SIAM Journal on Applied Mathematics* 37.3 (1979), pp. 513–538 (cit. on p. 75).

[42] R. M. Karp. "Reducibility among Combinatorial Problems". *Complexity of computer computations*. Springer US, 1972, pp. 85–103 (cit. on p. 19).

[43] M. Laguna and R. Martí. "GRASP and path relinking for 2-layer straight line crossing minimization". *INFORMS J. on Computing* 11 (1999), pp. 44–52 (cit. on p. 69).

[44] J. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. "Distinguishing string selection problems". *Information and Computation* 185.1 (2003), pp. 41–55 (cit. on p. 68).

[45] V. M. Manquinho and J. P. Marques-Silva. "Search pruning techniques in SAT-based branch-and-bound algorithms for the binate covering problem". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.5 (May 2002), pp. 505–516 (cit. on pp. 78, 82).

[46] M. Matsumoto, T. Nishimura, and Mersenne Twister. "A 623-dimensionally equidistributed uniform pseudorandom number generator". *ACM Transactions on Modeling and Computer Simulation* 8.1 (1998), pp. 3–30 (cit. on p. 45).

[47] N. Mladenović, M. Labbé, and P. Hansen. "Solving the p-Center Problem with Tabu Search and Variable Neighborhood Search". *Networks* 42.April (2003), pp. 48–64 (cit. on p. 76).

[48] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* 2008, pp. 337–340 (cit. on pp. 78, 82).

[49] S. R. Mousavi, M. Babaie, and M. Montazerian. "An improved heuristic for the far from most strings problem". *Journal of Heuristics* 18 (2012), pp. 239–262 (cit. on p. 70).

[50] W. B. Powell and Z. Chen. "A generalized threshold algorithm for the shortest path problem with time windows". *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 40 (1998), pp. 303–318 (cit. on p. 62).

[51] L. D. P. Pugliese and F. Guerriero. "A survey of resource constrained shortest path problems: Exact solution approaches". *Networks* 62.3 (2013), pp. 183–200 (cit. on p. 58).

[52] P. Toth and D. Vigo. *Vehicle Routing - Problems, Methods and Applications.* Ed. by Paolo Toth and Daniele Vigo. 2nd. Philadelphia: SIAM Monographs on Discrete Mathematics and Applications, 2014. ISBN: 978-1-61197-358-7 (cit. on p. 87).

[53] K. Truemper. *Design of logic-based intelligent systems.* John Wiley & Sons, 2004 (cit. on p. 78).