Université Catholique de Louvain

Ecole Polytechnique de Louvain

Improving the minimum weight circuit constraint

Supervisor: Pierre Schaus
Readers: Sébastien Mouthuy (N-Side)
Vinasetan Ratheil Houndji

Thesis submitted for the Master's degree
in computer science and engineering
options: Computing and App. Mathematics
by Hélène Verhaeghe

Louvain-la-Neuve
Academic year 2014-2015

Abstract

Constraint programming is a well known way to solve complex problems and especially problems that can be modelled as an aggregation of multiple complex sub-problems. The branch & bound based solving methods is helped by powerful propagation procedure, helping at the reducing of the domains of the variable at each node of the search tree.

The solver OscaR implements different classes for each different types of constraint, each having a specific propagator. The development of an effective OscaR constraint for the sub-problem of the `mincircuit` (minimal Hamiltonian tour in a graph) requires effective bounding procedures.

Based on mathematical models, two main iterative methods have been analysed. Both rely on Lagrangian relaxation of the mathematical representation of the Travelling Salesman Problem. The first considers the symmetric version of the TSP applied to a symmetric equivalent transformation of the asymmetric graph. The second is based on a transposition of the initial symmetric method to asymmetric concepts directly.

The results say that from a strict theoretical point of view, both methods are equivalent and give the same result as the linear relaxation (relaxation unpractical due to a number of constraint growing in a factorial way with respect to the number of vertexes involved). But in practice, due to the instability of the iterative method caused by the step size, the asymmetric based method gives better results for a fixed given number of iterations. Concerning the time of execution of both methods, as the symmetric version is based on a greedy quick optimised algorithm, it takes less time to compute the wanted amount of iteration.

Acknowledgments

This master thesis is the end of a path of 5 years, during which I've discovered more about the world and about myself. For this, I've many people I would like to thank.

First of all, my advisor, Prof. Pierre Schaus, for his patience, support, insights and advices during this whole year that allowed me to produce this master's thesis.

Secondly, some of my teachers for the learning treasures I got from them :

- Prof. Charles Pecheur, Prof. Olivier Bonaventure and Prof. Peter Van Roy, to make me fall in love with the computer sciences.
- Prof. Vincent Blondel and Prof. Jean-Charles Delvenne, to make me discover the beauty of the discrete mathematics.
- Prof. Yves Deville, for the discovering of the wonderful field of constraint programming.

Thirdly, my friends and my family for their support and in particular:

- The Clinch Team for the days passed together in the computer room, working out on our masters thesis in good spirits.
- My friends from EPLOL, for their guidance to the young padawan I am.
- Martin Delcourt and Florimond Houssiau for their help in the relecture of my master's thesis.

Hélène Verhaeghe

Juin 2015

Louvain School of Engineering

Contents

Introduction

In our days, when efficiency drives the choices of humanity, optimisation is essential. Finding the best way to act, the best path to take, the best things to choose,... all as cheaply as possible: all these goals lead us to the field of optimisation. And a way to solve complex optimisation problems is by using constraint programming.

Solving a problem with constraint programming is done by exploring a search space.

The constraint programming solver OscaR[11], written in the JVM language Scala[13], already contains implementations for different specific high level constraints like `knapsack`, `binpacking`, `subgraph`,... Each aims to solve different specific sub-problems. Another well known problem is the Travelling Salesman Problem (or TSP in short) : a salesman, starting his day at the office, has to go to each of his clients and, at the end of the day, needs to go back to his office, and he wants to do it by driving the shortest distance. To build a specific constraint when this problem is part of a more general one (it is called, in this case, the `mincircuit` constraint), we need to find effective bounding procedures for the different variants of the problem (symmetric and asymmetric).

This master's thesis aims to create a specific constraint for `mincircuit` with different level of consistency, and answer to the following big question : Is it useful to have two separate implementations of the two versions of the TSP or can we just use a symmetric graph transform of the asymmetric TSP and solve it with a symmetric approach?

All the code is free and available on
`https://bitbucket.org/helene_verhaeghe/lingi2990-memoire-implem`

Chapter 1

Problem definition

First of all, as this master's thesis is about graphs, if you are not fluent with graph theory terms, you should begin by reading the appendix A which reminds some useful terms. If these are already known, you can proceed.

This chapter introduces the actual problem of the `mincircuit` starting by redefining a well known problem : the Travelling Salesman Problem. Formal mathematical definitions will be given.

1.1 Formal definition of the Travelling Salesman Problem

The Travelling Salesman Problem [10] is defined as :

> *A salesman must visit each of n cities exactly once and then return to his starting point. The time taken to travel from city i to city j is c_{ij}. Find the order in which he should make his tour so as to finish as quickly as possible.*

This could be more formally modelled as finding the minimal cost Hamiltonian cycle between the n vertexes of the graph (representing the different cities) and given a set of possible edges (representing the roads) with associated costs (representing the cost of the road : distance or time, depending on what has to be minimised).

The same way there are two types of graph, there are two types of TSP, both resulting in different mathematical formulations and different underlying meanings.

1.2 The symmetric TSP

The edges of a symmetric graph in the problem correspond, in the real world, to bidirectional roads with equal cost in both directions. This can be considered as utopian since the travelling cost between two towns may not be equal. Some concrete examples to illustrate it are :

- the length of the road for going from A to B could be much longer because of some work on the road with a deviation on this side and not on the other side from B to A

- the cost, by plane, for going from A to B could be more expensive than going from B to A because of the use of different companies for examples
- a road could be one-way and thus couldn't be taken in both directions

The linear mathematical formulation of the symmetric TSP is the following : Given the symmetric graph $G = (V, E)$ defined by V, the set of vertexes, numbered from 1 to $|V|$, and E, the set of edges defined as (i, j) with $i \in V$, one end, and $j \in V$, the other end of the edge (if $(i, j) \in E$, $(j, i) \notin E$ to avoid duplicates of edges), and c_{ij} $\forall (i, j) \in E$ describing the cost of taking the edge (i, j). A decision variable x_{ij} is associated to each edge $(i, j) \in E$, equal to 1 if the edge is taken and 0 if not.

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

$$\text{subject to} \sum_{j:(i,j)\in E} x_{ij} + \sum_{j:(j,i)\in E} x_{ji} = 2 \qquad \text{for } i = 1, ..., |V| \tag{1.2}$$

$$\sum_{i,j:i\in S, j\in S,(i,j)\in E} x_{ij} \leq |S| - 1 \qquad \text{for all } S \subset V, 2 \leq |S| \leq |V| - 1 \tag{1.3}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in E \tag{1.4}$$

Explanation of the model

First, let's look at the variables (1.4). Each variable is associated to an edge of the graph and represent its selection (value 1) or not (value 0).

Knowing that, we can understand the objective function (1.1) as the sum of the selected edges, which is in fact the cost of the tour. As this is a minimisation, this means we search the minimal cost tour.

The first constraint (1.2) ensures a first property of the tour : the number of edges is equal to the number of vertexes. This can be proved by the graph property saying that knowing the degrees of each node, the total number of edges is $\frac{\sum_{v\in V} degree_v}{2}$. In our case, the constraint implies $degree_v = 2$ $\forall v \in V$, which gives us $\frac{\sum_{v\in V} degree_v}{2} = \frac{\sum_{v\in V} 2}{2} = |V|$.

The second constraint (1.3) ensures that each proper subset of vertexes doesn't have more than $|S| - 1$ edges between them. This implies that no sub-cycles can be found. As a cycle of n nodes needs n edges, for a given subset size, the constraint ensures no cycle of this size exists. And as the constraint needs to be respected for each proper subset size of the graph this ensures no cycle of size less than $|V|$ exists. This leads us to only one possible arrangement of the $|V|$ edges ensured by (1.2) : a single cycle of size $|V|$.

1.3 The asymmetric TSP

When an asymmetric graph is used, the edges of the graph correspond to unidirectional roads with their own costs. If a bidirectional road must be modelled, it is done with two edges in opposite ways, each with the same associated cost.

The mathematical formulation of the asymmetric TSP is the following : given the asymmetric graph $G = (V, E)$ defined by V, the set of vertexes, numbered from 1 to $|V|$, and E, the set of edges defined as (i, j) with $i \in V$ the source and $j \in V$ the destination of the edge, and c_{ij} $\forall (i, j) \in E$ describing the cost of taking the edge (i, j). A decision variable x_{ij} is associated to each edge $(i, j) \in E$, equal to 1 if the edge is taken and 0 if not.

$$\min \sum_{i,j:(i,j)\in E} c_{ij}x_{ij} \tag{1.5}$$

$$\text{subject to} \sum_{j:(i,j)\in E} x_{ij} = 1 \qquad \text{for } i = 1, ..., |V| \tag{1.6}$$

$$\sum_{i:(i,j)\in E} x_{ij} = 1 \qquad \text{for } j = 1, ..., |V| \tag{1.7}$$

$$\sum_{i,j:i\in S, j\in S,(i,j)\in E} x_{ij} \leq |S| - 1 \qquad \text{for all } S \subset V, 2 \leq |S| \leq |V| - 1 \tag{1.8}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in E \tag{1.9}$$

Explanation of the model and differences with the symmetric model

This model contains still one decision variable associated to each edge (1.9), the same objective function (1.5) minimising the cost of the tour formed by the selected edges and the same constraint avoiding sub-tours of size strictly smaller than $|V|$.

The only change is in the degree constraint ((1.6) & (1.7)). As the tour is now directed, instead of constraining the global degree of the node to two, the in and out-degree are separately constrained to 1. This will force only one entering edge and only one outgoing edge at each node.

1.4 Usability of the models

These models seem rather simple when just laid down on paper. In fact the constraint excluding sub-tours ((1.3) & (1.8)) makes these models unpractical. This type of constraint is in fact combinatorial [10] since it has to be valid for each proper subset of the whole set of nodes.

Computing the number of constraints generated by this notation is made by computing the number of proper subsets with at least two nodes. The number of proper subsets of size k is given by the combinatorial formula

$$C^k_{|V|} = \left(\begin{array}{c} |V| \\ k \end{array} \right) = \frac{|V|!}{k!(|V| - k)!}$$

As a constraint is generated for all proper subsets, the total number of constraints generated is given by :

$$\sum_{k=2}^{|V|-1} C^k_{|V|} = \sum_{k=2}^{|V|-1} \frac{|V|!}{k!(|V| - k)!} \geq \sum_{k=2}^{|V|-1} |V|! = (|V| - 2)|V|! = (O)(|V|!)$$

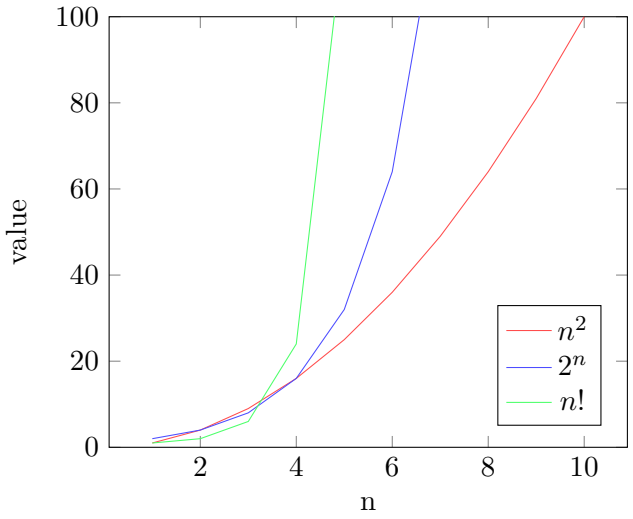


Figure 1.1: Comparison of different order of complexity

When comparing with other complexity orders (Fig.1.1), it can easily seen that the number of constraints generated become quickly unmanageable with this combinatorial formulation. This is why the solving based on the linear model is badly scalable.

1.5 Solving the linear model

However, the model is used in practice by the dedicated TSP solver Concorde [4] which solves the problem by branch & cut [10].

Branch & cut is based on a branch & bound algorithm. Instead of a fast re-optimisation at each node, the work is focused on finding the tightest dual bound by using cuts, improved formulations and also with preprocessing. This allow earliest cut of the search tree but has a largest computation cost than the branch & bound.

Chapter 2

Constraint Programming

Constraint programming [6] [12] is one of the most efficient techniques used to solve problems composed of multiple sub-problems of different types. In this chapter, the basics of constraint programming will be explained. The usefulness of finding good bounding procedures to the `mincircuit` problem will also be detailed.

2.1 Definition and usefulness of constraint programming

Constraint programming is a programming paradigm used to solve constraint satisfaction problems (CSP) and constraint optimisation problems (COP).

Definition - 2.1.1 (CSP) *A constraint satisfaction problem is composed of variables (each having their set of possible values, called the domain of the variable) and a set of constraints (proprieties involving the values of a subset of the variables). A solution of the CSP is an assignment of the variables (with value from their domain) satisfying all the constraints.*

Definition - 2.1.2 (COP) *A constraint optimisation problem is a CSP with an additional optimality constraint added (maximising or minimising a function called the objective function). The solution is now an assignment of the variables satisfying all the constraints and giving the best possible value of the objective function.*

As it can be seen, COP's are harder to solve since the goal isn't to find a satisfying solution but the best one.

The search space of a/the solution to a problem can be seen as a tree where each leaf corresponds to an assignment of the variable to some values and each node is a sub-problem of its parent.

The exploration of a search space begins with the root and will be done in a depth-first way, with a succession of two steps at each node :

1. a filtering step
2. a decision step

2.1.1 Filtering step

The filtering step removes values of the domains to make the constraints consistent with respect to the domain store.

Definition - 2.1.3 (Consistency) *A constraint is consistent with respect to the domain store if the values of the domain could be part of a solution of the constraint taken apart. Two main kinds of consistency exist :*

- *Domain-consistency : every single value of a domain can be part of a solution of the constraint.*
- *Bound-consistency : the upper and lower bound (max and min values) of the domain can be part of a solution of the constraint.*

For bigger and harder problems (such as NP-hard problem), the consistency applied is weaker and more specific to the problem.

The removal of the values is done by the fix-point algorithm (List.2.1). It will apply pruning on values until all the constraints are consistent.

Listing 2.1: Pseudo-code of the fix-point algorithm

```
1 repeat until no values can be removed from domains
2     select a constraint c not consistent with respect to the domain ↩
          store
3     apply pruning on c
4 return ok
```

When the level of consistency chosen is ensured for every constraint, the validity of the current node is checked. If there is a variable with an empty domain, this means with certainty that the current branch of the search tree contains no solution at all. There is no need to continue the exploration and the search on this node can be stopped. The search then goes back on an unexplored node by backtracking to a previous state and continues the exploration of this previous state.

The pseudo-code of the full filtering step is written down in List.2.2.

Listing 2.2: Pseudo-code of the filtering step

```
1 fix-point algorithm
2 if (exists variable with empty domain)
3     backtrack to previous state
4 else
5     continue with decision step
```

2.1.2 Decision step

The decision part consists of choosing the next branching step, the next decision of the exploration. Many branching strategies can be used :

- binary labelling : a variable and a value of its domain are chosen, the first branch will try the assignation of this value to the variable, and the second branch will try the removal of the value from its domain
- n-ary labelling : a variable is chosen and a branch will be created for each of its values of its domain, each one trying the assignation assigning the value to the variable
- domain splitting : a variable and a value of its domain are chosen, the first branch will restrain the domain of the variable to the values smaller or equal to the value chosen and the second branch will restraint the domain to the values strictly higher

Other branching strategies exist. The only thing to remember is that the decision should split the problems into some sub-problem without loosing any potential solution (the union of the solutions of the sub-problems should be the whole solutions of the problem).

To choose the variable and the value, the naive way would be by taking them in lexical order. But a better choice can be made by following heuristics (decision guidelines that aim to result in a better pruning and thus a reduced search tree) :

- first-fail heuristic on the variables : it is better to choose a variable with the smallest domain, if there are no solutions on the sub-tree, it will be sooner pruned since there are less values to try for the variable
- best chance heuristic on the values : as we want to find a value as quickly as possible, the best is to choose the values that reduces the least the domains

The combination of these two heuristics generally gives good results, but depending on the problem, other more specific heuristics can be used.

When the decisions are settled, the search will be able to continue (List.2.3).

Listing 2.3: Pseudo-code of the decision step

```
1 for (decision in decisions)
2     apply decision to domain store
3     continue with filtering
4     if (got a solution) // each variables assigned
5         return the solution
6     else
7         backtrack decision
```

2.1.3 Optimality pruning for COP

When the problem to solve is a COP, optimality pruning can be applied. The same concept is used with Branch & Bound in discrete mathematics [10]. At every node, we compute a bound (lower or upper) of the objective function. At every step we keep a best-so-far solution which is the best solution found so far (which can be initially set as the first easy solution to the problem). We compare the bound and the best-so-far solution to decide if the node can or can not be pruned. If a new solution better than the best-so-far is found, we keep it as new best-so-far.

Let's take the case of minimising a function. If we arrive at a node with lower bound higher than the best-so-far solution, we know that there is really no need to explore it. This is ensured by the fact that all solutions of this sub-tree will have a value higher than the lower bound, and thus lower than the minimum so far. This proves that solutions with lower values don't exist in this sub-tree.

The pseudo-code of the complete filtering step with the addition of optimality pruning is written down at List.2.4.

Listing 2.4: Pseudo-code of the filtering step with optimality prunning (for a minimising problem)

```
1 fix-point algorithm
2 if (exists variable with empty domain)
3     backtrack to previous state
4 else if (lower bound > best-so-far)
5     backtrack to previous state
6 else
7     continue
```

2.1.4 Example

The following example will explain the steps made by the constraint programming solver to solve the following CSP :

- Variables and their domains : $x = \{0, 1, 2, 3\}$, $y = \{0, 1, 2, 3\}$, $z = \{0, 1, 2, 3\}$
- Constraints :
  - $AllDifferent(x, y, z)$: constraints the values of the variables to be different from each other
  - $x + y = z$: constraint linking the value of the variables

The next decision will always be to take the first variable (in lexical order) unassigned (with one value in the domain) and apply a binary labelling with the smallest value still present in its domain.

To find all the solutions, the following steps will be taken by the solver :

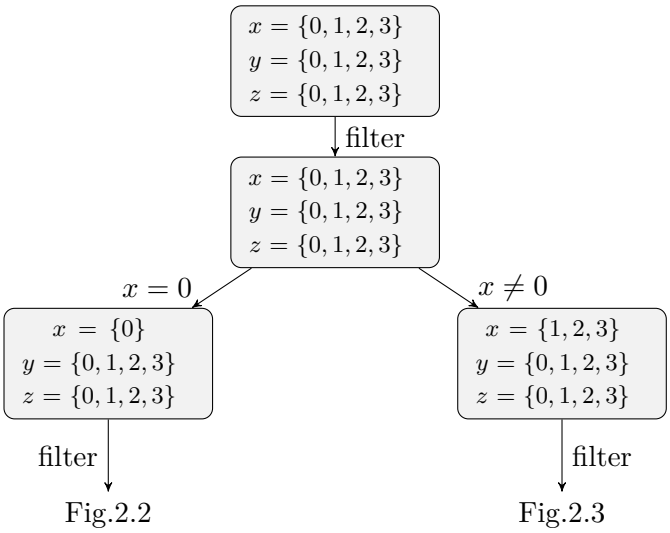


Figure 2.1: Example of the solving of a simple problem with CP (first decision)

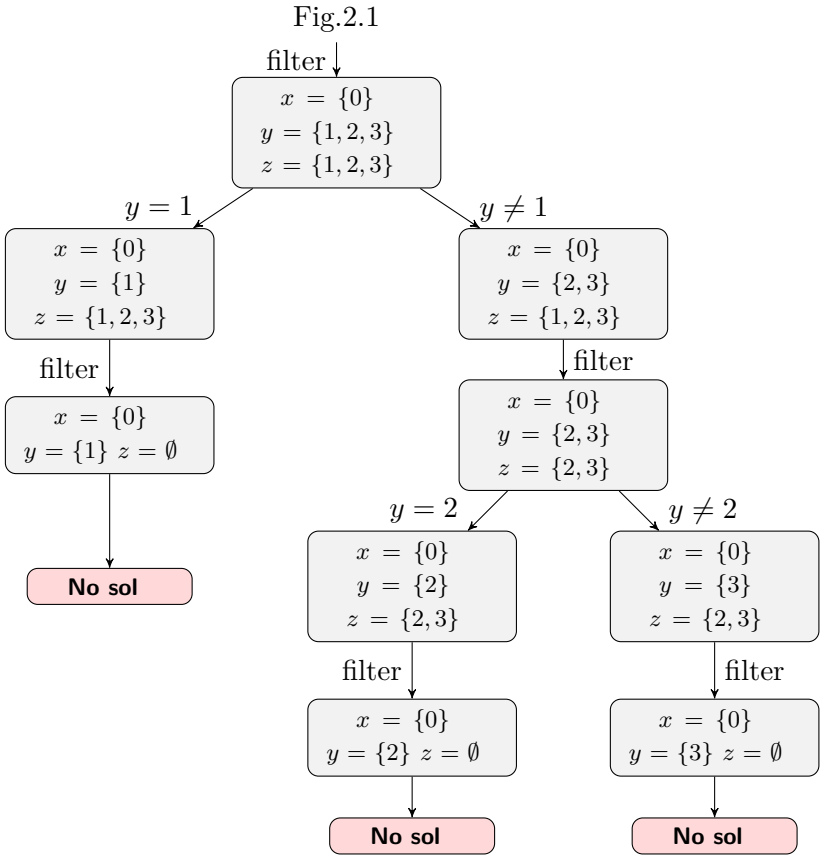


Figure 2.2: Left subtree of Fig.2.1 (decision $x == 0$)

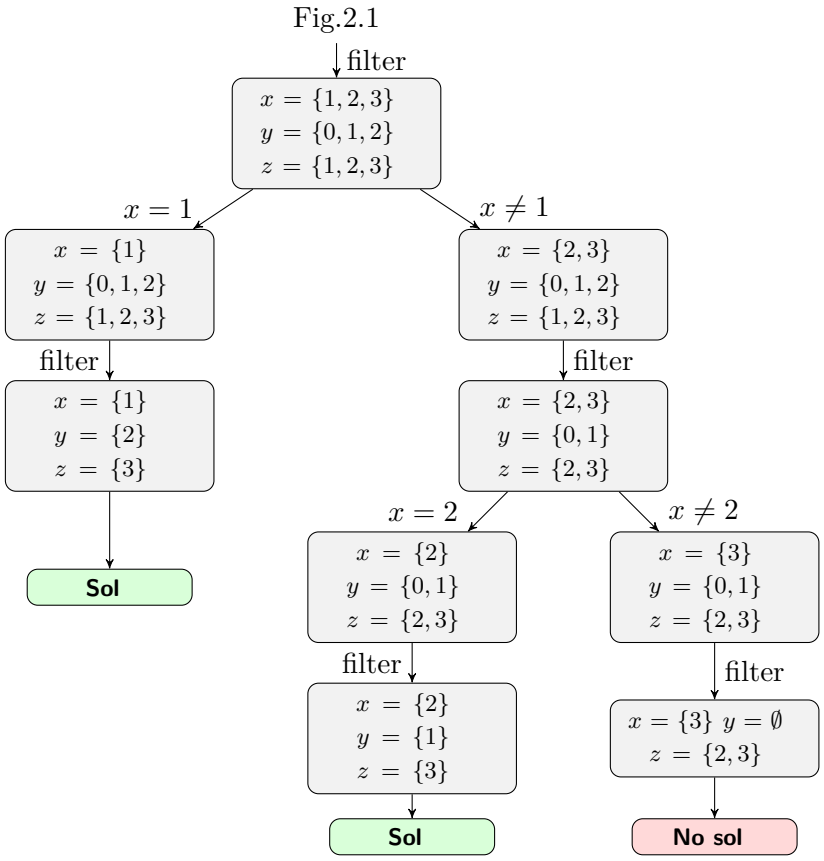


Figure 2.3: Right subtree of Fig.2.1 (decision $x! = 0$)

1. Start (Fig.2.1) the fix point algorithm with the root (the initial problem)
2. Apply a first filtering : as for both constraints, all values of the variables are part of a solution for the isolated constraint, no values are removed
3. Then the first decision is made : the first unbound variable is x and its smallest value is 0. The decision is thus to assign 0 to x
4. The search will try the first decision ($x == 0$) and apply filtering on the new sub-problem (Fig.2.2) : as the value 0 was taken by x, following the `alldifferent` constraint, this value can't be taken by the other variables, so it is removed from the domains. Concerning the `sum` constraint, as every combination of values is still viable, no other value is removed
5. As the problem is still not a solution neither a dead end, a new decision is taken, this time on y with the value 1
6. The filtering is done : 1 is removed from the domain of z because of the `alldifferent` constraint. Then, after looking at the second constraint (which is now $0 + 1 = z$ because of the assignment), we can clearly see that the values 2 and 3 should be removed from the domain of z. This lead us with an empty domain.
7. A backtracking is needed and will be done up to the last decision with unexplored branches (here the decision of assigning 1 to y) and the state before the decision will be recovered. Then the other branch will be explored ($y! = 1$)
8. It will continue the same way until the whole tree is explored (Fig.2.3). Two solutions are finally found : $x = 1, y = 2, z = 3$ and $x = 2, y = 1, z = 3$

2.2 Constraint object and propagators

The previous example shows the global process of the solving. But how exactly, do the constraints know which value to remove from the domains?

When modelling the problem, the different constraints are differentiated, each of them having their own specific filtering procedures called `propagators` and each propagator having a different level of consistency. The user can choose the level used at the definition of the constraint.

Concerning the computation time, the strongest the consistency asked, the more computation is needed, but more values are removed from the domains. The therefore user has to make the good balance. Following the problem, it can be more useful to apply a strong complexity to some constraints because they impact many aspects of the problem and therefore implies more pruning on other constraints. For others, the cost to remove all the useless values can be too high for the number of values removed at each propagation, so that it isn't worth using such a big tool as the strongest consistency. As this depends on the context, this choice is left to the programmer.

As they are specifics for each constraint type, the propagators know exactly, following the constraint, how to check if a value can still be part of a solution. Designing a new constraint then requires to create at least one propagator.

2.3 The `mincircuit` constraint

As well as there exist specific constraints for other well-known problems, allowing them to be used as sub-problem, we want to have a specific constraint for the TSP also usable as sub-problem. This constraint is called the `mincircuit` and ensures a Hamiltonian tour on the given graph.

The creation of this variable will require to model the tour and to write down bounding procedures for the cost. The modelling of the tour can be based on the already existing `circuit` constraint modelling an Hamiltonian tour : n decision variables, each associated to a vertex and containing its successor in the tour. The main propagation decision will imply the update of the bounds of the domain of the cost variable. As the goal is to minimise the cost of the tour, there is no need to define a precise upper-bound procedure. Taking the sum of the most expensive out-edge for each node can be the easiest way to do it. Concerning the lower bound, different strength of lower bounding procedure could be implemented. Ideally, the strongest bound would compute the exact value of the minimum tour in the graph, but in practice this would take too much time.

Chapter 3

TSP lower bounds

This chapter describes some algorithms known to find a lower bound on the minimum circuit over a graph. Most of them concern symmetric/undirected graphs.

3.1 Lower bound for both types of graphs

3.1.1 Relaxation by removing a constraint

An easy way to have a relaxation is to drop completely a constraint. This is what is done when using the *assignment problem* as a relaxation of the TSP.

The assignment problem is defined [10] [8] as assigning one job to one worker, each worker having only one job and each job being done by only one worker. A cost is associated to each (worker,job) pair. The goal is to minimise the total cost.

The mathematical model of the assignment problem is

$$\begin{aligned} \min & \sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}x_{ij} & \\ \text{subject to} & \sum_{j=1}^{n} x_{ij} = 1 & \text{for } i = 1, ..., n \\ & \sum_{i=1}^{n} x_{ij} = 1 & \text{for } j = 1, ..., n \\ & x_{ij} \in \{0, 1\} & \forall (i,j) \in E \end{aligned}$$

with x_{ij} the variable of decision to assign job j to worker i with the associated cost c_{ij}.

This model is quite similar to the model of the asymmetric TSP. If we consider a worker as a vertex, a job as the following vertex and the cost from i to j as the cost of the edge linking i and j. A solution of the assignment problem will correspond to the pairs (vertex, successor) corresponding to the selected edges.

This corresponds in fact to the initial model of the TSP without the sub-tour elimination constraint. The solution of this relaxation will correspond either to a single tour either to multiple tours.

3.1.2 Linear programming relaxation

A first intuitive bound for mathematicians [2] is to take the optimisation model and apply a linear relaxation to the variables, that is, to release the integer constraint to allow real values.

The linear relaxation for the symmetric case is :

$$\begin{aligned}
\min \quad & \sum_{i,j:(i,j)\in E} c_{ij}x_{ij} & \\
\text{subject to} \quad & \sum_{j:(i,j)\in E} x_{ij} + \sum_{j:(j,i)\in E} x_{ij} = 2 & \text{for } i = 1, ..., |V| \\
& \sum_{i,j:i\in S, j\in S,(i,j)\in E} x_{ij} \leq |S| - 1 & \text{for } S \subset V, 2 \leq |S| \leq |V| - 1 \\
& 0 \leq x_{ij} \leq 1 & \forall (i,j) \in E
\end{aligned}$$

The linear relaxation for the asymmetric case is :

$$\begin{aligned}
\min \quad & \sum_{i,j:(i,j)\in E} c_{ij}x_{ij} & \\
\text{subject to} \quad & \sum_{j:(i,j)\in E} x_{ij} = 1 & \text{for } i = 1, ..., |V| \\
& \sum_{i:(i,j)\in E} x_{ij} = 1 & \text{for } j = 1, ..., |V| \\
& \sum_{i,j:i\in S, j\in S,(i,j)\in E} x_{ij} \leq |S| - 1 & \text{for } S \subset V, 2 \leq |S| \leq |V| - 1 \\
& 0 \leq x_{ij} \leq 1 & \forall (i,j) \in E
\end{aligned}$$

As these relaxations are linear optimisation problems, they are solvable by the simplex algorithm or the interior point method. But as they still contain the combinatorial constraint, they remain impractical for graphs with more than a few nodes, which concerns most realistic cases.

3.1.3 Lagrangian Relaxation

Another mathematical relaxation is the Lagrangian relaxation. If you don't recall the technique, see Appendix B. Unlike a simple relaxation where some constraints are simply removed or replaced by weakest ones, the objective function is also modified to include an influence from the removed constraints. This aims to make the solution of the relaxation more likely to be close to the solution of the initial problem.

A possible Lagrangian relaxation of the symmetric model could be

$$\begin{aligned}
\min \quad & \sum_{i,j:(i,j)\in E} c_{ij}x_{ij} + \sum_{i\in V} \lambda_i \left(2 - \sum_{j:(i,j)\in E} x_{ij} - \sum_{j:(j,i)\in E} x_{ji} \right) \\
\text{subject to} \quad & \sum_{i,j:(i,j)\in E} x_{ij} = |V| \\
& \sum_{i,j:i\in S, j\in S,(i,j)\in E} x_{ij} \leq |S| - 1 && \text{for } S \subset V, 2 \leq |S| \leq |V| - 1 \\
& x_{ij} \in \{0,1\} && \forall (i,j) \in E
\end{aligned}$$

with $\lambda \in \mathbb{R}^{|V|}$. Here the degree constraints are replaced by the weaker constraints on the total number of edges and the objective now includes a new term.

Other Lagrangian relaxations could be designed by choosing an other set of constraints to relax. Unfortunately, the combinatorial constraints, even if they are set in the objective, stay too complicated to be used like this in a classical solver with graphs with more than a few vertexes.

The Lagrangian relaxation of the asymmetric case is done the same way, and is also unpractical with classical solvers for graph slightly big.

3.2 Lower bound for symmetric graphs only

3.2.1 Minimum spanning tree

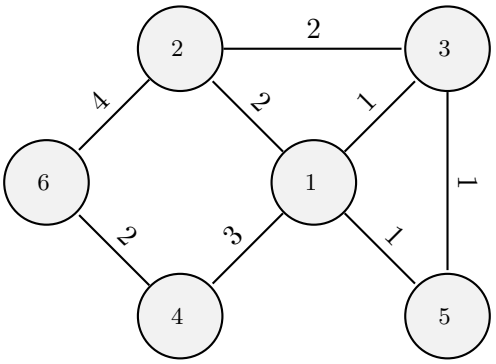
One of the naivest way [14] to relax a tour in an undirected graph is to take the minimum spanning tree (easily calculated with a Prim or a Kruskal approach [5]). Unfortunately for us, this relaxation is far to weak to be used in practice. Only $|V| - 1$ edges will be taken into account and the degree constraint is also relaxed. So even if it is easy to compute, it isn't used in practice.

3.2.2 1Trees

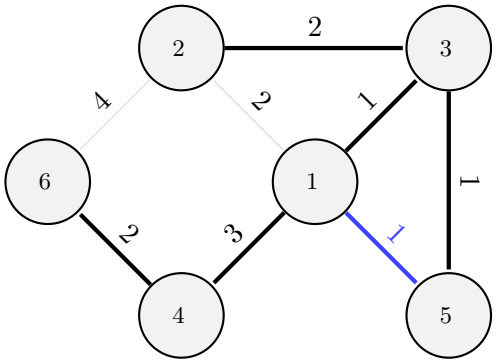
minimal 1Tree

A 1Tree is defined [14] as a connected sub-graph of an undirected graph with $|V|$ edges. To find the minimal 1Tree of a graph, just simply compute the minimum spanning tree of the graph and then add a last edge by taking the remaining untaken edge with the smallest cost.

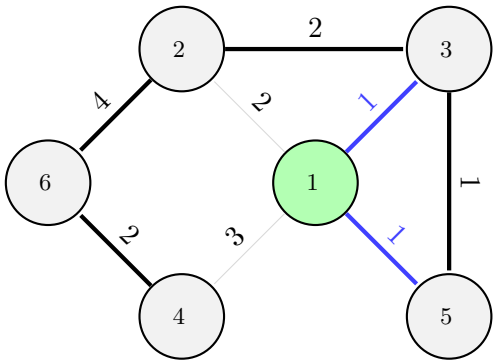
This relaxation corresponds to a tour for which both constraints have been weakened. Implicitly, the degree constraint forces the total number of edges to be equal to $|V|$ with each node having a degree of 2. Here, the weakening is made by keeping the global constraint on the number of edges but forgetting about the degrees of the nodes. The other constraint implied no cycle in every strict subset of nodes and, by combination with the degree constraint, implied the connectivity between all the nodes. The weakening is made by keeping only the connectivity.



(a) Symmetric graph with costs



(b) Minimal spanning 1Tree of value 10 on the graph Fig.3.1a



(c) Rooted 1Tree with root 1 and value 11 on the graph Fig.3.1a

Figure 3.1: Example of 1Tree

An example of 1Tree can be seen on Fig.3.1b. The black edges are the edges selected from the tree step, the blue edge is the minimum edge added and the grey ones are the discarded ones.

This minimum 1Tree is a first approximation but a better one can be found.

Rooted 1Tree

The rooted 1Tree [9] is a 1Tree constructed by first finding the tree in the sub-graph where the root and all edges linked to it are removed, and then adding the two edges linked to the root with the minimal cost. This corresponds to the same relaxation as the minimal spanning 1Tree except we did keep the degree constraint on the root. As this is less relaxed, this will give a better lower bound.

An example can be seen at Fig.3.1c where the green vertex is the root. The rooted 1Tree is better than the simple minimum spanning 1Tree.

3.2.3 Held-Karp iterative bounding procedure

The Held-Karp iterative method [9] [14] [15] [2] takes the advantages of the easily computable lower bound given by the rooted 1Tree and combines it to the principle of the Lagrangian relaxation. This results in a method having the same strong bounding result as the linear relaxation but without the combinatorial number of constraints contained in it.

Mathematical basis

The mathematical formulation of the Lagrangian relaxation associated is :

$$\begin{aligned}
&\min \sum_{e:e\in E} c_e x_e + \sum_{i\in V} \pi_i \left(2 - \sum_{e:e\in\delta(i)} x_e\right) \\
&\text{subject to} \sum_{i,j:i\in S, j\in S,(i,j)\in E} x_{ij} \leq |S| - 1 && \text{for } S \subseteq V\backslash\{1\}, 2 \leq |S| \\
&\sum_{e:e\in\delta(1)} x_e = 2 \\
&\sum_{e:e\in E} x_e = |V| \\
&x_e \in \{0,1\} && \forall e \in E
\end{aligned}$$

with $\pi_i \in \mathbb{R} \forall i \in V\backslash\{1\}$ and $\delta(i) = \{(i,j)|(i,j) \in E\} \cup \{(j,i)|(j,i) \in E\}$.

The remaining constraint corresponds to the rooted 1Tree of root 1. This leads to the following iterative method ending with, as hoped, the best Lagrangian multipliers maximising the lower bound.

As said in [10] (at p.173), the remaining constraints of this Lagrangian relaxation form a set for which the linear relaxation is it's convex hull. This implies that the value obtainable with the optimal Lagrangian multipliers is the same as the result of the linear relaxation of the TSP.

Intuitive meaning of the multipliers

Before going further, a more intuitive meaning can be given to the value of the Lagrangian multipliers π_i. Each one is associated with a node (except the root) and represents, in some sense, the attractiveness of the vertex. Let us analyse the part of the objective concerned by π_i, $\pi_i \left(2 - \sum_{e:e\in\delta(i)} x_e\right)$,

- if $\pi_i = 0$, the term won't influence the objective
- if $\pi_i > 0$, the term will influence the objective in a good way if the number of taken edges linked to i is more than 2, this will make i an "attractive" node, tending to be linked as much as possible
- if $\pi_i < 0$, the term will influence the objective in a good way if the number of taken edges linked to i is less than 2, this will make i a "repulsive" node, tending to be linked as little as possible

The method will in fact update the Lagrangian multipliers following this intuitive meaning.

Description of the method

The method associates to each vertex a variable containing the value of the associated multiplier to the vertex. Their initial value will be 0. A different step size β will be associated to each iteration (its value will be given later). Each iteration takes place in three steps :

1. Find the rooted 1Tree with respect to the updated weight
$$\bar{c}_{ij} = c_{ij} - \pi_i - \pi_j$$
2. Compute the lower bound
$$cost = cost(1Tree) + 2\sum_{i\in V} \pi_i$$
3. If the 1Tree is not a tour, update the multipliers following the formula
$$\pi_i += (2 - degree(i)) * \beta$$
else stop the algorithm by returning the tour

To understand how this method comes from the model, let's reformulate the objective :

$$\begin{aligned}
\sum_{e:e\in E} c_e x_e + \sum_{i\in V} \pi_i \left(2 - \sum_{e:e\in\delta(i)} x_e\right) &= \sum_{e:e\in E} c_e x_e + 2\sum_{i\in V} \pi_i - \sum_{i\in V} \pi_i \sum_{e:e\in\delta(i)} x_e \\
&= \sum_{e:e\in E} c_e x_e + 2\sum_{i\in V} \pi_i - \sum_{i,j:(i,j)\in E} (\pi_i x_{ij} + \pi_j x_{ij}) \\
&= \sum_{i,j:(i,j)\in E} (c_{ij} - \pi_i - \pi_j)\, x_{ij} + 2\sum_{i\in V} \pi_i \\
&= \sum_{i,j:(i,j)\in E} \bar{c}_{ij} x_{ij} + 2\sum_{i\in V} \pi_i
\end{aligned}$$

The reformulated objective is composed of the cost of the 1Tree considering the updated costs and the constant term depending on the multipliers uniquely.

Optimal value of β

As this is a Lagrangian relaxation, the classical updates of the step length can be used [10] [15] :

- First choice, a step size of β_k with $\sum_k \beta_k \to \infty$ with $\beta_k \to 0$ as $k \to \infty$. This method guarantees the convergence but in practice this convergence may be too slow to be practicaly used.
- Second choice, a step size of $\beta_k = \beta_0 \rho^k$ with some parameter $\rho < 1$. This method guarantees convergence if β_0 and ρ are sufficiently large to avoid having the geometrical series $\beta_0 \rho^k$ tending too rapidly to 0.
- Third choice, $\beta_k = \epsilon_k \frac{UB - LB}{\sum_{v \in V} (2 - \sum_{e \in \delta(i)} x_e)^2}$ with $0 < \epsilon_k < 2$. This may not converge if the upper bound of the solution (UB) isn't the optimal.

As described in [3], one update that has given good performance in practice is the third one. And in particular, an adaptation of it, as described in [18]. The iterative method starts with $\epsilon_k = 2$ for $2n$ iterations, with n a description of the size of the problem. ϵ_k and the number of iterations are then successively halved until the number of iteration reach a threshold z. After that, ϵ_k is halved every z iteration until the end of the algorithm.

End of the algorithm

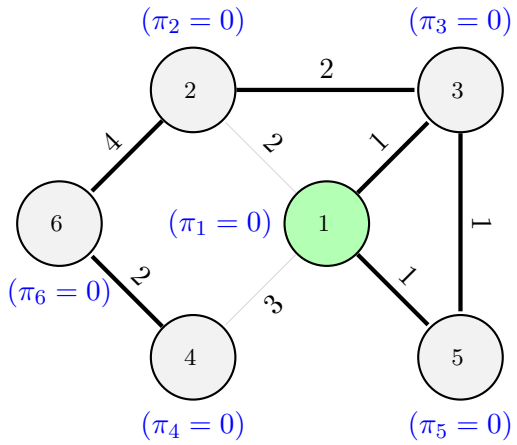
As explained in [2] and [10], the value of the Lagrangian relaxation with the optimal multipliers (the one maximising the optimal value of the relaxation), in the case of the TSP (symmetric or asymmetric), has the same result as the linear relaxation.

But unfortunately for us, there is no guarantee that the iterative algorithm stops. But, if it stops, the optimum has been found. And there is also no guarantee that the cost of the k^{th} iteration will be better that the previous one.

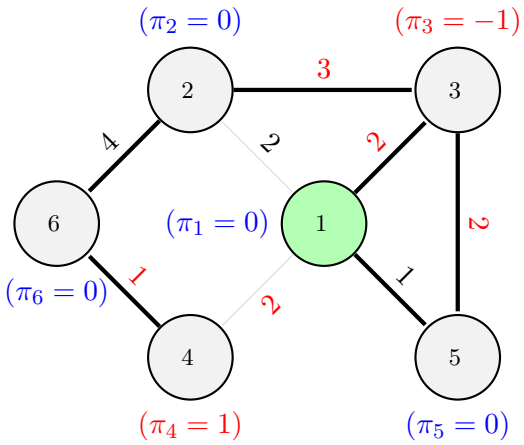
A quick example

Let's take the graph Fig.3.1a and apply to it the algorithm with the following values for β along the iteration : $1, 0.5, 0.25, ...$ The evolution of the graph is at Fig.3.2.

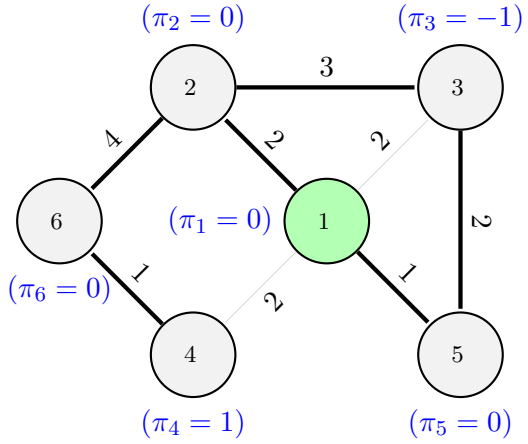
1. First iteration ($\beta = 1$) :
   - The first thing done is the computation of the first 1Tree rooted in 1. It gives us the graph Fig.3.2a
   - Then the cost of this graph is computed :
     $cost = 11 + 2 \times (0 + .. + 0) = 11$
   - As this 1Tree isn't yet a tour, the update of the multipliers is computed from the degrees of the vertexes in the 1Tree. The costs of the edges are then updated. This gives us the updated graph Fig.3.2b.



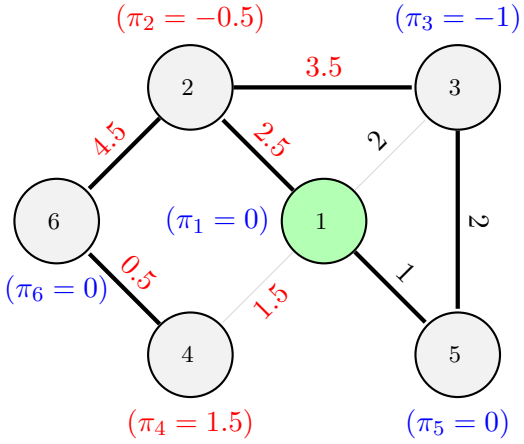
(a) 1Tree of the first iteration



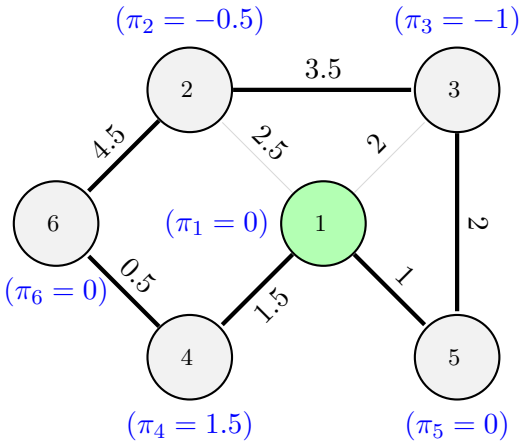
(b) Update of the costs with $\beta = 1$



(c) 1Tree of the second iteration



(d) Update of the cost with $\beta = 0.5$



(e) 1Tree of the third iteration

Figure 3.2: Execution of the Held-Karp algorithm on the graph Fig.3.1a

2. Second iteration ($\beta = 0.5$) :
    - A new 1Tree (Fig.3.2c) is computed with respect to the new costs of the edges.
    - The cost of this new tree is computed :
      $cost = 13 + 2 \times (1 + -1 + 0 + .. + 0) = 13$
    - As this is again not a tour, the multipliers and the costs are updated again (Fig.3.2d).
3. Third iteration ($\beta = 0.25$) :
    - The third tree is computed (Fig.3.2d).
    - Its cost is $cost = 13 + 2 \times (1.5 + -0.5 + -1.5 + 0 + .. + 0) = 13$
    - As this 1Tree is also an Hamiltonian tour, we can stop the algorithm because the lower bound has reached its optimal value.

Chapter 4

Adaptation of the Held-Karp iterative algorithm to asymmetric graphs

If we want to solve the Hamiltonian tour in an asymmetric graph the same way as in a symmetric one, we need to adjust the Held Karp bounding algorithm. Two ways are possible. First, keep the same algorithm but use a symmetric transformation of the asymmetric graph as input. The second way is to adapt the algorithm by finding equivalences (find a equivalent tree, a parameter,...).

4.1 Transformation of the asymmetric graph to a symmetric graph

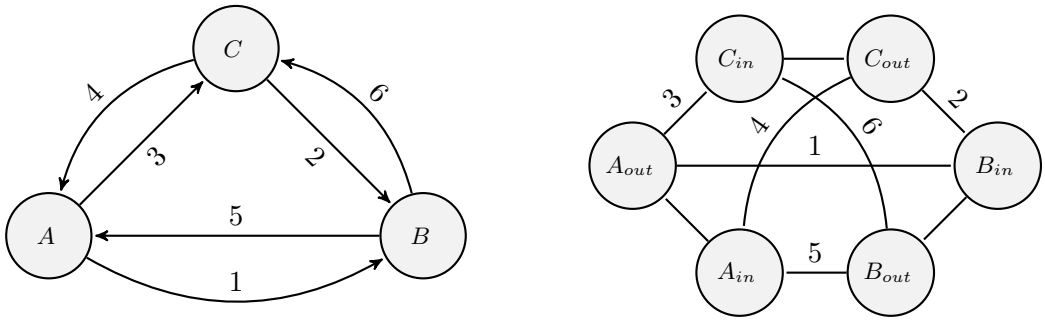
The first way is by transformation of the graph, leading to none or few changes in the Held-Karp algorithm. This is the less invasive method.

This has one practical advantage for the maintenance of a constraint programming solver since only one main method is needed to be maintained.

A way to transform an asymmetric graph into a symmetric graph [16] is by doubling the number of nodes.

Each node of the asymmetric graph corresponds to two nodes in the symmetric graph. The first one act as entering node (*in*-node) of the initial node and the second as outgoing node (*out*-node). Each directed edge of the asymmetric graph corresponds to an edge in the symmetric graph going from the *out*-node of the source to the *in*-node of the destination.

A new edge is created between the *in*-node and the corresponding *out*-node. Multiple choices of cost for this linking edge are possible, with their advantages and disadvantage. They will be discussed latter.



(a) Asymmetric graph

(b) Symmetric version of the graph Fig.4.1a

Figure 4.1: Asymmetric graph and its corresponding symmetric transformation

An example of this transformation can be seen on Fig.4.1a and Fig.4.1b. The first one represents the initial asymmetric graph and the second one represents the symmetric result of the transformation. Each corresponding node has the same name but with the *in* and *out* variant in the symmetric version.

Let's now talk about the possibilities for the cost of the linking edges.

To ensure a total correspondence between the graph and its symmetric transformation, we must ensure a bijection between the symmetric and asymmetric tour.

First, let's see that every asymmetric tour still correspond to tour after the transformation. This is trivially implied by the transformation.

Secondly, a symmetric tour can only correspond to an asymmetric tour if the linking edges are involved in the tour. As the reverse transformation will remove the linking edges, the number of edges selected in the asymmetric graph will correspond to the number of edges selected in the symmetric minus the linking edges selected. If not all linking edges are involved, the remaining selected edge in the asymmetric will be higher than the number of nodes and thus won't correspond to a tour.

The cost should then ensure the selection of the linking edge at each time.

To test the different costs, the linear relaxations on small graphs (up to 15 nodes) were computed on the asymmetric and the symmetric graph and compared to ensure the graphs gives equivalent results, at least for the linear relaxation.

As in the paper : Negative costs

In the paper [16], they talked about using a cost of $-\infty$ for the linking edges. This is theoretically the best way to do it since every time the minimum 1Tree computed will take these edges with the minimum cost (ensured by the greedy nature of the minimum 1Tree algorithm).

But practically, the value of $-\infty$ doesn't exist and is approximated to the minimum value representable by the type used to represent the costs. This leads to overflow computation in some case (computation of the cost of the tree) or needs an adaptation of the computation of the cost.

To avoid overflows, we can set the cost of the linking edge to a negative number B not so small as the smallest representable number. This works but can still possibly lead to an overflow if the number of linking edges is too high. Furthermore, we need to add a bias of $-Bn$ (with n the number of nodes) to the total cost to get the real cost of the 1Tree.

An other problem can be that the cost of the linking edge can be modified, and following the updates, may become higher than other ones. This leads to some iterations where the 1Tree doesn't include all the linking edges. So, potentially, tours not corresponding to tours in the asymmetric version can be found. This hardly ever happens but the probability isn't null.

This first choice of costs gives equivalent values for the linear relaxations.

Cost of 0

We can set the value of the edge to 0. With this, the probability of an overflow is strongly reduced. The creation of the 1Tree would include the linking edges. But as the algorithm will update the cost of the edges, after a while, there is a non-negligible probability that the value of the linking edge would be such that the 1Tree doesn't take them anymore. This was seen when testing and worse bounds were found because the tour found didn't correspond to one in the asymmetric graph.

Even when testing with the linear relaxation, the values for asymmetric and symmetric are not the same each time. The symmetric linear relaxation always gives a lower or equal value as the asymmetric linear relaxation. This method can't therefore be used.

Cost of 0 and forcing of the edges

We can still set the cost of the linking edge to 0 but if we force the linking edges to be part of the 1Tree, this lead to a better approximation since a 1Tree that will also be a symmetric tour will correspond to an asymmetric tour.

Also, overflow problems won't append because of the cost of the linking edges will tend to stay small.

When adding the forcing on the linking edges to the symmetric linear relaxation, its value is always the same as the asymmetric linear relaxation which confirms the bijection between the tours for this adaptation.

4.2 Transformation of the method

4.2.1 Equivalence for the 1Tree

The first modification will concern the notion of the 1Tree undefined in directed graphs. As the tree itself doesn't have a match, let's first define a potential notion for it.

Arborescence and Edmond algorithm

An arborescence [8] rooted in a vertex a is a connected sub-graph with $|V| - 1$ edge and with a directed path from the root to every other vertex of the graph. A polynomial algorithm to found it was described by Jack Edmonds [1].

The algorithm can be find in List.4.1.

Listing 4.1: Pseudo-code of Edmonds's algorithm to find the minimum-cost arborescence

```
for each node v!=root
    mincost_v = minimal cost of the entering edges of v
    for each entering edge e of v
        cost_e = cost_e - mincost_v
    select one 0-cost entering edge for v
if selected edge form arborescence
    return arborescence
else
    find selected edges forming a cycle C
    contract C in a new super node leading to new graph G*=(V*,E*)
    find arborescence in G*
    extend the arborescence in G* to one in G by adding all but one ↩
       edge involved in C
    return arborescence
```

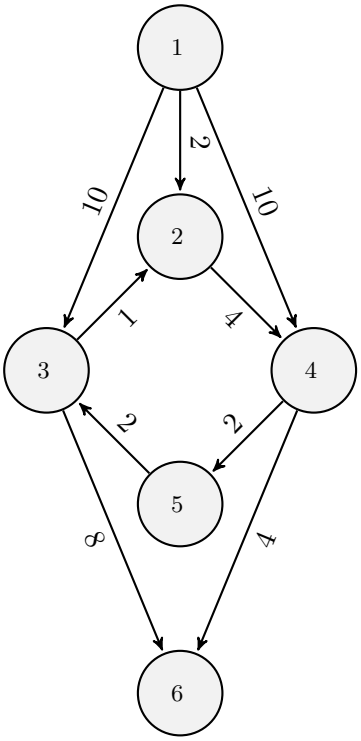
Quick example

The algorithm starts with the graph Fig.4.2a and with the node 1 as root.

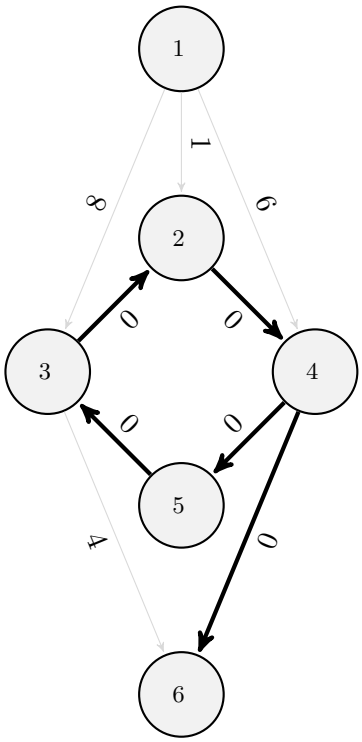
1. For each node, the minimal cost entering edge is found and the costs of each edges are modified, giving the cost at Fig.4.2b.
2. One 0-cost entering edge is chosen for each node giving a potential arborescence (Fig.4.2b).
3. As this is not an arborescence (not connected), there is a cycle (with node $2, 3, 4$ and 5). A contraction is made of these nodes resulting at the graph Fig.4.2c.
4. The recursive call is made : reduction of the cost and selection of the 0-cost edges. As this leads to an arborescence (Fig.4.2c), the recursive call returns.
5. The contraction is extended again and all edges but one of the cycles is chosen (Fig.4.2d).
6. As the selected edges form an arborescence (Fig.4.3), the algorithm ends.

Anti-arborescence

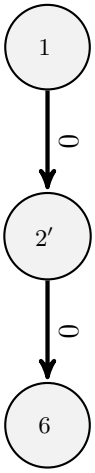
An anti-arborescence [8] anti-rooted in a vertex a is a connected sub-graph with $|V| - 1$ edges and with a directed path from every other node of the graph to the root.



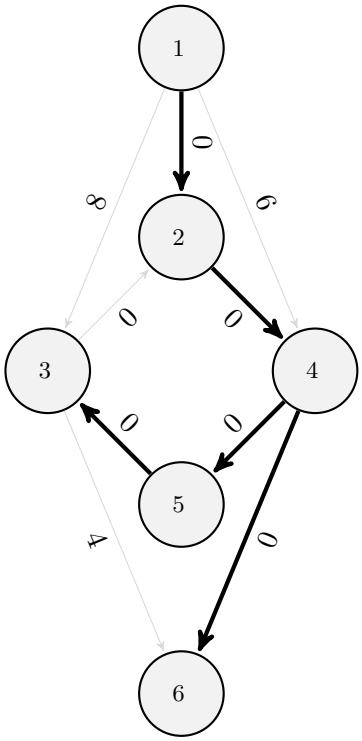
(a) Initial graph



(b) Reduction of costs and selection of the 0-cost edges



(c) Contraction of C and recursive call



(d) Expansion of the contracted node and selection of edges

Figure 4.2: Example of the process of Edmonds' algorithm

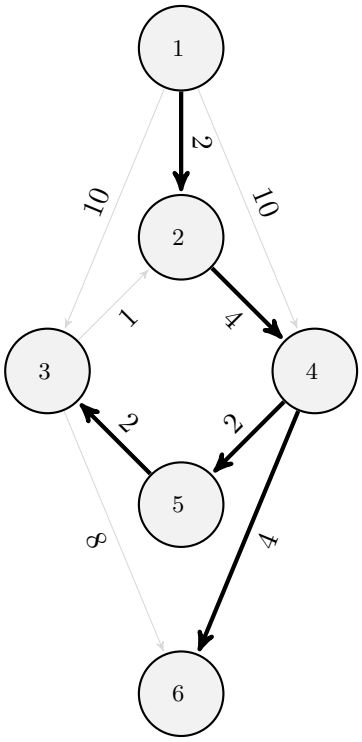


Figure 4.3: Resulting minimal cost arborescence of tree Fig.4.2a

Two ways exist to compute the anti-arborescence. The first is to take the inverted graph (graph where all the edges have been inverted : the initial destination became the source and the initial source became the destination) and compute the arborescence rooted at the anti-root. The inverted graph of the arborescence found correspond to the anti-arborescence in the initial graph. The second way is to adapt the Edmond algorithm (List.4.2).

Listing 4.2: Pseudo-code of the adaptation of Edmonds's algorithm to find the minimum-cost anti-arborescence

```
for each node v!=anti-root
    mincost_v = minimal cost of the outgoing edges of v
    for each outgoing edge e of v
        cost_e = cost_e - mincost_v
    select one 0-cost outgoing edge for v
if selected edge form anti-arborescence
    return anti-arborescence
else
    find selected edges forming a cycle C
    contract C in a new super node leading to new graph G*=(V*,E*)
    find anti-arborescence in G*
    extend the anti-arborescence in G* to one in G by adding all but ↩
       one edge involved in C
    return anti-arborescence
```

1Arborescence and 1Anti-Arborescence

Now that an equivalence for the tree was found, we can transpose the idea of the 1Tree in 1Arborescence and 1Anti-Arborescence. We defined a 1Arborescence as an arborescence with the addition of the minimal cost entering edge of the root and a 1Anti-arborescence as an anti-arborescence with the addition of the minimal cost outgoing edge of the anti-root.

4.2.2 Modified Held-Karp

Now that an equivalent formulation to the 1Tree is defined for asymmetric graph, the rest of the algorithm can be adapted following the same reasoning underlying the Held-Karp method.

From Arborescence

From this base, we can develop an algorithm based on the Held-Karp [8] for symmetric graph. Let's first define a Lagrangian Relaxation of the asymmetric problem with its remaining constraint corresponding to a 1Arborescence. We obtained the following problem :

$$\begin{aligned}
\min \quad & \sum_{i,j:(i,j)\in E} c_{ij}x_{ij} + \sum_{i\in V} \pi_i \left(1 - \sum_{j:(i,j)\in E} x_{ij}\right) & \\
\text{subject to} \quad & \sum_{i:(i,j)\in E} x_{ij} = 1 & \text{for } j = 1, ..., |V| \\
& \sum_{i,j:i\in S, j\in S, (i,j)\in E} x_{ij} \leq |S| - 1 & \text{for } S \subset V\backslash\{root\}, 2 \leq |S| \\
& x_{ij} \in \{0, 1\} & \forall (i,j) \in E
\end{aligned}$$

From this we can define the new algorithm :

1. Find the 1Arborescence with respect to the updated weight

$$\bar{c}_{ij} = c_{ij} - \pi_i$$

2. Compute the lower bound

$$cost = cost(1Arborescence) + \sum_{i\in V} \pi_i$$

3. If the 1Arborescence is not a tour, update the multipliers following the formula

$$\pi_i += (1 - degree_{out}(i)) * \beta$$

   else return the tour found

From Anti-arborescence

The same can be done for the 1Anti-arborescence. Following the Lagrangian relaxation :

$$\begin{aligned}
\min \quad & \sum_{i,j:(i,j)\in E} c_{ij}x_{ij} + \sum_{i\in V} \pi_i \left(1 - \sum_{j:(j,i)\in E} x_{ji}\right) & \\
\text{subject to} \quad & \sum_{j:(i,j)\in E} x_{ij} = 1 & \text{for } i = 1, ..., |V| \\
& \sum_{i,j:i\in S, j\in S,(i,j)\in E} x_{ij} \leq |S| - 1 & \text{for } S \subset V\backslash\{root\}, 2 \leq |S| \\
& x_{ij} \in \{0, 1\} & \forall (i,j) \in E
\end{aligned}$$

The new algorithm can be defined as :

1. Find the 1Anti-arborescence with respect to the updated weight

$$\bar{c}_{ij} = c_{ij} - \pi_j$$

2. Compute the lower bound

$$cost = cost(1Anti - Arborescence) + \sum_{i\in V} \pi_i$$

3. If the 1Anti-arborescence is not a tour, update the multipliers following the formula

$$\pi_i += (1 - degree_{in}(i)) * \beta$$

   else return the tour found

4.2.3 Step parameter

The step parameter must also be adapted but always following the possibilities explained in section 3.2.3 to guarantee a best convergence. The effect of the choices is explained in details in the chapter 6 where the comparison test between the methods are done.

4.2.4 How good can this bound be?

Let's set some notations :

- x_a* and x_s* : optimal solutions of the asymmetric TSP (Section 1.3) and the symmetric TSP (Section 1.2) used to solve the transformation of the input graph that doubles the nodes and forces of the linking edges
- r_a* and r_s* : optimal solution of the linear relaxation of the asymmetric TSP (Section 3.1.2) and of the symmetric TSP (3.1.2)
- w_a* : optimal solution of the Lagrangian relaxation used in the modified Held-Karp algorithm defined in this chapter

- w_s* : optimal solution of the Lagrangian relaxation used in the initial Held-Karp algorithm defined in Section 3.2.3

Let's also say that our input graph is the asymmetric graph defined by $G = (V, E)$ and its transformation is named $G' = (V', E')$.

This section will prove that $w_s* = w_a*$.

First equality : $x_a* = x_s*$

As already mention in Section 4.2.1, this is proven [16] by the fact that each tour in G' including all the linking edges corresponds to a tour in the initial graph G with the same cost. As all the linking edges are taken, a tour will only have one other edge entering the in-node and one other exiting the out-node (because of the degree constraint). There will thus be only one corresponding entering edge and one corresponding exiting edge in the asymmetric graph. By merging the pairs of in and out-nodes, we obtain directly the tour in the asymmetric graph with the same cost as the linking edge were 0-cost edges.

And every tour in G correspond to a tour including the linking edges in G' with the same cost. This is done by applying the transformation to the tour sub-graph. As the only action made was to split each node in two and link the two sub-nodes with a 0-cost edge, this still stays a tour of the same cost.

Second equality : $r_a* = r_s*$

The same way the exact value of the minimal cost tour is equal in the symmetric and asymmetric cases, we can prove that each linear relaxation of the asymmetric graph corresponds to a linear relaxation of a symmetric tour with the forcing of the edge. This can done by simply applying the transformation to the relaxation and keep the same associated variables. The cost is the same as the only edges added are linking edges with a 0-cost.

On the other hand, each symmetric relaxation with forcing of the linking edge correspond to a linear relaxation of the asymmetric tour with the same cost. We just have to merge the corresponding in and out-node in the symmetric version to get a valid linear relaxation of the asymmetric tour. As both models of relaxation are equivalent and keep the same cost, the minimum of one will have as corresponding transformation the minimum of the other and the optimum will be the same.

Third equality : $r_s* = w_s*$

As already said in Section 3.2.3, this is already explained in [10] by the fact that the possible solutions are proven to be on the convex hull of the domain.

Fourth equality : $r_a* = w_a*$

This demonstration is based on mathematical basis. Not all of them will be explained in detail. All the mathematical theorems and concepts come from [10]. The page number will be indicated every time.

Only the case where the Lagrangian Relaxation is based on the 1Arborescence will be demonstrate. The case where the 1Anti-arborescence is used is demonstrated the same way.

First, let's formulate the constraints of the Lagrangian relaxation in the form $Ax \leq b$:

$$\sum_{i:(i,j)\in E} x_{ij} \leq 1 \qquad \text{for } j = 1, ..., |V|$$

$$\sum_{i:(i,j)\in E} -x_{ij} \leq -1 \qquad \text{for } j = 1, ..., |V|$$

$$\sum_{i,j:i\in S, j\in S,(i,j)\in E} x_{ij} \leq |S| - 1 \qquad \text{for } S \subset V\backslash\{root\}, 2 \leq |S|$$

Following the corollary p.172, if the domain is $X = \{x \in \mathbb{Z}^n_+ : Ax \leq b\}$ and its convex hull is $conv(X) = \{x \in \mathbb{R}^n_+ : Ax \leq b\}$, then the best bound obtained by the Lagrangian relaxation is equal to the optimal of the linear relaxation of the initial problem. We will prove that this is the case for our set of constraints.

As said in the seventh approach (p.147) of problem 2 (p.145) of determining if the polyhedron $P = \{x \in \mathbb{R}^n_+ : Ax \leq b\}$ is the convex hull of a subset $X \subset \mathbb{Z}^n_+$, if $b \in \mathbb{Z}^n$ and the inequalities $Ax \leq b$ form a TDI (*Totally Dual Integral*) system, then $P = conv(X)$.

As our vector b is composed of integers, the first condition is fulfilled. Let's now prove that the set of constraints forms a TDI system.

First let's prove it for the first two simple general inequalities. The matrix defined by these two general constraints is TU (*Totally Unimodular*). We can prove it with the help of the proposition 3.2 (p.39) which gives us a sufficient condition to have a TU matrix.

- First, the matrix needs to have $a_{ij} \in \{+1, -1, 0\}$. This is ensured by the coefficient before the variables that all are $+1$ in the first general constraint, -1 in the second, and 0 if the edge isn't involved in a constraint.
- Secondly, each column of the matrix must contains at most two nonzero coefficients. We can prove it by seeing that each edge can be involved only twice in the constraints, once for each. An edge x_{ab} can only be involved into the constraint where $j = b$.
- Thirdly, there must exist a partition (M_1, M_2) of the set M of rows such that each column j containing two nonzero coefficients satisfies $\sum_{i\in M_1} a_{ij} - \sum_{i\in M_2} a_{ij} = 0$. The partition valid in our case is $M_1 = M$ and $M_2 = \emptyset$. Since a variable only exists in two constraints, one with a $+1$ coefficient and one with a -1 coefficient, for every column (each corresponding to a variable), the sum of the coefficients is 0. Thus we have $\sum_{i\in M_1} a_{ij} = \sum_{i\in M} a_{ij} = 1 - 1 = 0$ and $\sum_{i\in M_2} a_{ij} = \sum_{i\in\emptyset} a_{ij} = 0$ leading to $\sum_{i\in M_1} a_{ij} - \sum_{i\in M_2} a_{ij} = 0$.

The two firsts general inequalities form a TU matrix. As said in p.49, if A is TU, $Ax \leq b$ is TDI. We have thus the first inequalities forming a TDI system.

Secondly, let's prove that the combinatorial constraint forms also a TDI system. As said in p.49, in the proof of the Theorem 3.11, this shows that the linear system $\{\sum_{j\in S} x_j \leq f(S)\ for\ S \subseteq N, x_j \geq 0\ for j \in N\}$ is TDI (where $f(S)$ is a submodular nondecreasing function defined by $f : \mathcal{P}(N) \to \mathbb{R}$ where $\mathcal{P}(N)$ denotes the set of subsets of N). Our combinatorial constraint can be viewed as a linear system if we redefine it as

$$\sum_{e\in K} x_e \leq f(K) \text{ with } K = \{e | e \in E,\ e = (i,j),\ i,j \in S\},\ \text{for all } S \subset V\backslash\{root\}, 2 \leq |S|$$

where $f(K) = (\textit{size of the cover of } K) - 1$. $f(K)$ is nondecreasing (p.46) since adding any edge to K can only add a new vertex to the set of vertexes covered by the edges in K. $f(K)$ is submodular if $f(A) + f(B) \geq f(A \cap B) + f(A \cup B)$ (p.46). The property 3.10 (p.46) also said that $f(S)$ is submodular and nondecreasing if $f(A) \leq f(B) + \sum_{j\in A\backslash B}[f(B \cup \{j\}) - f(B)]$ for all $A, B \subseteq N$. This is true for our function. We can prove it by subdividing A into 3 subsets of edges : the first who also are contained in B, the second who are not in B but cover vertexes already covered by B and the other not in B and with at least one vertex not covered by B. For the first one, there is no term in the sum since they are contained in B. For the second subset, the term $[f(B \cup \{j\}) - f(B)] = 0$ and their covering vertexes are counted either in $f(A)$ or in $f(B)$. For the third, $[f(B \cup \{j\}) - f(B)] = 1$ *or* 2 (if one or two vertexes are added to the cover). The sum can thus be bounded by $\sum_{j\in A\backslash B}[f(B \cup \{j\}) - f(B)] \geq f(A) - f(A \cap B)$ since it will at least count all the vertexes covered by A but not by B. We have thus

$$\begin{aligned} f(B) + \sum_{j\in A\backslash B} [f\,(B \cup \{j\}) - f\,(B)] &\geq f(B) + f(A) - f(A \cap B) \\ &\geq f(B) + f(A) - f(B) \qquad \text{because of nondecreasing} \\ &= f(A) \end{aligned}$$

As we have now proved that $f(N)$ is submodular and nondecreasing, we know that the linear system defined by the combinatorial constraint is a TDI system.

Knowing two TDI systems, we can prove the intersection of them to be also TDI. So let's prove $Ax \leq b$ (with $A = \begin{pmatrix} A' \\ A" \end{pmatrix}$ and $b = \begin{pmatrix} b' \\ b" \end{pmatrix}$) is TDI knowing that $A'x \leq b'$ and $A'x \leq b'$ are TDI as well. If they are TDI, for all $c' \in \mathbb{Z}^n$ for which $\max\{c'x : A'x \leq b'\}$ has a finite optimal value, the dual $\min\{y'b' : y'A' = c', y' \geq 0\}$ has an optimal solution with y integral and for all $c" \in \mathbb{Z}^n$ for which $\max\{c"x : A"x \leq b"\}$ has a finite optimal value, the dual $\min\{y"b" : y"A" = c", y" \geq 0\}$ has an optimal solution with y integral. We will thus verify if for all $c \in \mathbb{Z}^n$ for which $\max\{cx : Ax \leq b\}$ has a finite optimal value, the dual $\min\{yb : yA = c, y \geq 0\}$ has an optimal solution with y integral : we can first write

$$\begin{aligned} \min\{yb : yA = c, y \geq 0\} &= \min\{\begin{pmatrix} y' & y" \end{pmatrix} \begin{pmatrix} b' \\ b" \end{pmatrix} : \begin{pmatrix} y' & y" \end{pmatrix} \begin{pmatrix} A' \\ A" \end{pmatrix} = c, y', y" \geq 0\} \\ &= \min\{y'b' + y"b" : y'A' + y"A" = c, y', y" \geq 0\} \\ &= \min\{y'b' + y"b" : y'A' = c', y"A" = c", c' + c" = c, y', y" \geq 0\} \\ &= \min\{y'b' + : y'A' = c', y' \geq 0\} + \min\{y"b" : y"A" = c", y" \geq 0\} \end{aligned}$$

By hypothesis, those two minimisation's have optimal values with respectively y' and $y"$ integral. This makes the whole solution $y = \begin{pmatrix} y' & y" \end{pmatrix}$ also integral for $c = c' + c"$ which can be whatever value $\in \mathbb{Z}^n$ since c' and $c"$ are whatever values $\in \mathbb{Z}^n$. This made the whole system TDI and concludes our demonstration to prove that $X = conv(X)$ and thus that $w_a* = r_a*$.

Final equality : $w_a* = w_s*$

As we have proven that $r_a* = w_a*$, $r_a* = r_s*$ and $r_s* = w_s*$, trivialy,

$$w_a* = r_a* = r_s* = w_s*$$

This implies the best lower bound we can get from our modified Held-Karp method to be exactly the same as the best lower bound we can get from the initial Held-Karp algorithm applied to the transformed graph with the forcing of the linking edges

Chapter 5

Analysis of the weak bound

Let's first focus on the simple weak bounds we have : the 1Tree, the 1Arborescence and the 1Anti-Arborescence.

Let's first define some notations for this chapter : firstly, for the algorithms :

- **1Arbo** : the 1Arborescence relaxation
- **1Anti** : the 1Anti-arborescence relaxation
- **1Tree** : the 1Tree relaxation
- **1TreeI** : the 1Tree relaxation only on in-node
- **1TreeO** : the 1Tree relaxation only on out-node

Secondly for the benchmark of graphs used for the tests :

- **RandA** : Constituted of randomly generated graphs with a number of vertexes between 3 and 17, a uniform random cost for the edges between 0 and 100. Most of them are not full graphs.
- **RandB** : Constituted of randomly generated graphs with a number of vertexes between 20 and 50, a uniform random cost for the edges between 0 and 1000. Most of them are not full graphs.
- **RandC** : Constituted of randomly generated full graphs with a number of vertexes between 20 and 50. These graphs have from one fifth up to one third of their vertexes with all of either their in or their out edges with a cost between 0 and 10000. All the other edges having a cost between 0 and 1000. This is specially made to have also nonuniform graphs in the benchmark.
- **TSPlib** : The asymmetric graph solved of the TSPLIB [17]. The descriptions of them are in Appendix C

	better	strictly better
mean	1.00	0.96
max	1.00	0.96
min	1.00	0.96
var	0.25	0.22
min 1Arbo vs max 1Tree	0.96	

Table 5.1: 1Arbo vs 1Tree : percentages for TSPLIB benchmark

5.1 The 1Tree as a relaxation of the 1Arbo and the 1Anti

5.1.1 1TreeI vs 1Arbo

Given an asymmetric graph G and its symmetric transformation G', it can be easily seen that each 1Arbo rooted in a node r corresponds to a 1Tree rooted in r_in. This corresponding 1Tree is obtained by applying the same graph transformation as to transform the whole asymmetric graph into a symmetric one.

If we take the minimal 1Arbo rooted in r, its corresponding 1Tree (rooted in r_in) has the same cost but it can have a higher cost than the minimal 1Tree rooted in r_in. We thus have

$$\min 1Arbo(r) \geq \min 1TreeI(r) \quad \forall r \in V$$

If we now take the best minimal 1Arbo we can found, following the previous inequality, we know for sure that it will be higher than the best minimal 1TreeI for the graph.

$$\max_{v \in V}(\min 1Arbo(v)) \geq \max_{v \in V}(\min 1TreeI(v))$$

5.1.2 1TreeO vs 1Anti

The same reasoning can be done concerning the 1Anti and the 1TreeO, leading to the following inequalities :

$$\min 1Anti(r) \geq \min 1TreeO(r) \quad \forall r \in V$$

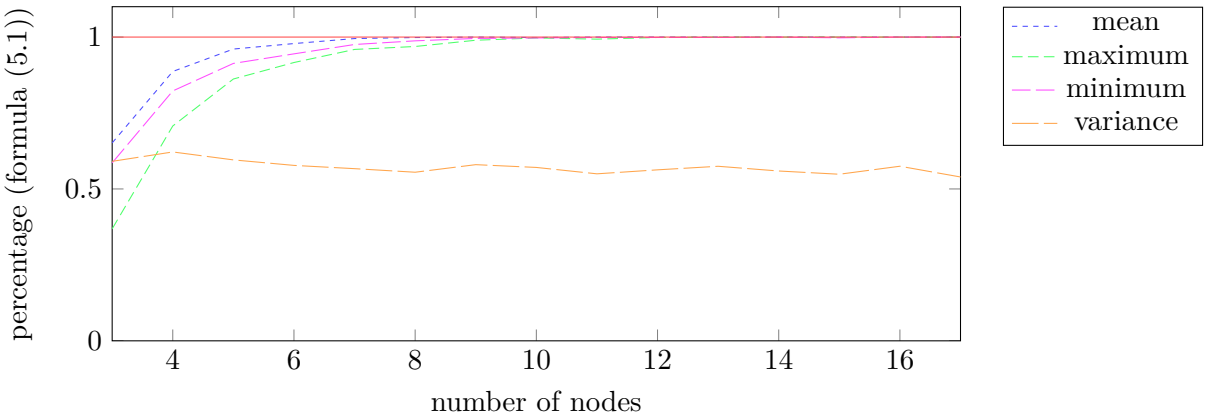
$$\max_{v \in V}(\min 1Anti(v)) \geq \max_{v \in V}(\min 1TreeO(v))$$

5.1.3 1Arbo vs 1Tree

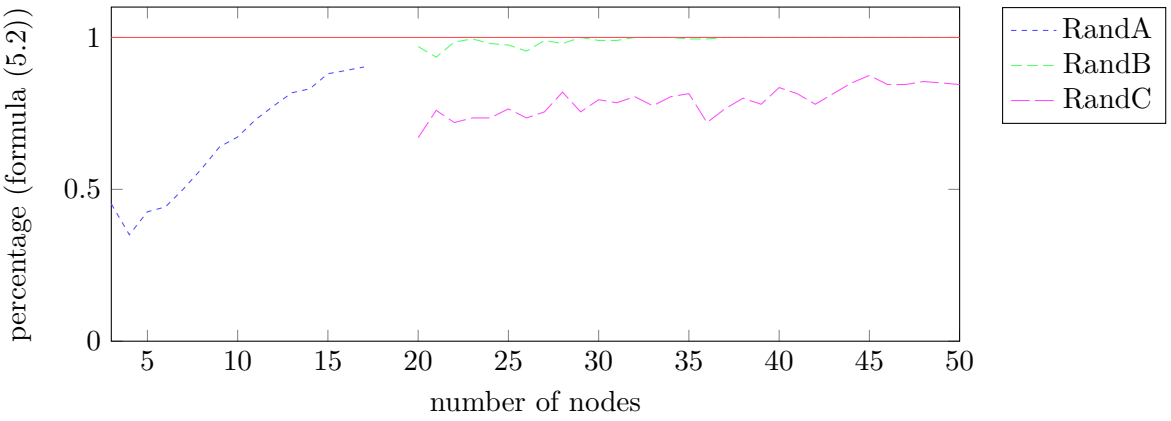
Let's now compare the 1Arbo with the 1Tree in general.

As all the minimal 1Tree's are composed of all minimal 1TreeI's and all minimal 1TreeO's, and as we have already proven that $\max_{v \in V}(\min 1Arbo(v)) \geq \max_{v \in V}(\min 1TreeI(v))$, we have no guarantee that using only the 1Arbo's to compute the lower bound will be higher than computing all 1Tree's : the maximum of the minimal 1Tree's could be better if the following two conditions are satisfied :

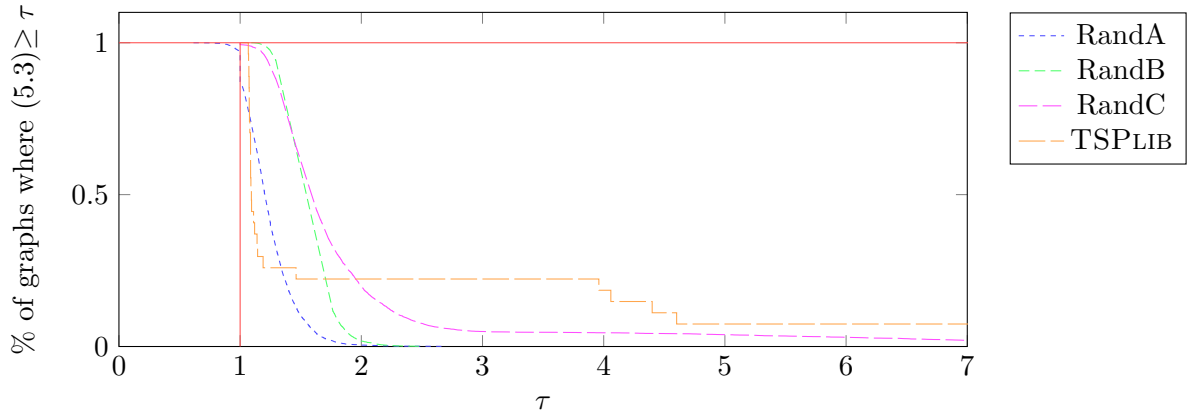
$$\max_{v \in V}(\min 1TreeO(v)) \geq \max_{v \in V}(\min 1TreeI(v))$$



(a) Percentage of graphs of RandA where the statistical data for the minimal 1Arbo's is strictly better than the one for the minimal 1Tree's



(b) Percentage of graphs where the minimum of the minimal 1Arbo's is better than the maximum of the minimal 1Tree's



(c) Performance profile of the ratio between the maximum of the minimal 1Arbo's and the maximum of the minimal 1Tree's

Figure 5.1: 1Arbo versus 1Tree

and

$$\max_{v \in V}(\min 1Arbo(v)) = \max_{v \in V}(\min 1TreeI(v))$$

We therefore want to know if this case happens and for which graphs.

Comparison by statistical datas

The first graph (Fig.5.1a) depicts some statistical analysis on the graphs of RandA. For each graphs, the values of the minimum 1Arbo rooted in all the vertexes ($\min 1Arbo(v)\ \forall v \in V$) and the values of the minimum 1Tree rooted in all the vertexes ($\min 1Tree(v)\ \forall v \in V',\ V' = \bigcup_{v \in V}\{v_{in}, v_{out}\}$) are computed. For each graph and for both techniques, the following data are computed :

- the mean
- the maximum
- the minimum
- the variance

For each graph, the corresponding data (mean of the 1Arbo with mean of the 1Tree, max with max,...) are compared. For each statistical data, the following percentage is computed :

$$\frac{|\{G \mid G \in G_x,\ data_{1Arbo}(G) > data_{1Tree}(G)\}|}{|G_x|} \tag{5.1}$$

with G_x the subset of the benchmark containing all the graphs with $|V| = x$.

This give us the first graph (Fig.5.1a). It can be seen on it that when the graphs are big enough (more than 10 vertexes), the mean, the maximum and the minimum of the minimal 1Arbo's computed for one graph become higher than all the minimal 1Tree's ones and not only the minimal 1TreeI's. This indicates that basing a weak bound on the minimal 1Arbo's should give better results than with the minimal 1Tree's.

The comparison of the variance doesn't follow this dominance of the 1Arbo over the 1Tree but as it stays close to 50%, it doesn't say anything.

The same results where obtained on the RandB (App.D), RandC (App.D) and TSPLib benchmarks (Tab.5.1) where for almost all of them the statistical data is strictly higher for the 1Arbo.

Is each minimal 1Arbo is better than each minimal 1Tree?

The second graph (Fig.5.1b) gives us something even more interesting. Here, the percentage computed is the following :

$$\frac{|\{G \mid G \in G_x,\ min_{1Arbo}(G) > max_{1Tree}(G)\}|}{|G_x|} \tag{5.2}$$

	better	strictly better
mean	1.00	0.96
max	1.00	0.96
min	1.00	0.93
var	0.48	0.44
min 1Anti vs max 1Tree	0.96	

Table 5.2: 1Anti vs 1Tree : percentages for TSPLIB benchmark

As we can see on the graph regrouping the statistics from the RandA and RandB (up to 50 nodes) benchmarks, the more nodes there is, the more we have most of the graphs satisfying $min_{1Arbo}(G) > max_{1Tree}(G)$. If this is satisfied, this implies all the minimal 1Arbo's are better than all minimal 1Tree's, meaning taking a single minimal 1Arbo will always give a better result than every minimal 1Tree.

How good is the 1Arbo with respect to the 1Tree?

Knowing that the 1Arbo became better than the 1Tree's at the end, the last graph (Fig.5.1c) answer the following question : how much can we gain by taking the 1Arbo instead of the 1Tree? To answer it, the ratio

$$\frac{\max_{v_i nV}(\min 1Arbo(v))}{\max_{v_i nV}(\min 1Tree(v))} \tag{5.3}$$

is computed for each graph of the three benchmark. The performance profile at Fig.5.1c regroups these data. It can be seen on it that the ratios tend to be higher for benchmarks with bigger graphs. This confirms what has been said before : the more vertexes the graph has, the better the 1Arbo is with respect to the 1Tree.

5.1.4 1Anti vs 1Tree

The same reasoning leading to a similar conclusion can be done between the 1Anti and the 1Tree.

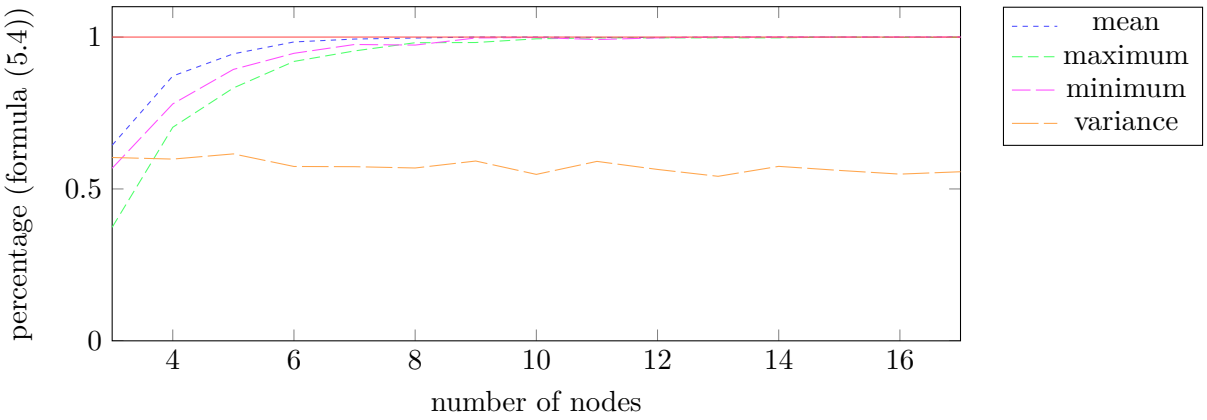
The three formulas for the percentage and the performance profiles are now :

$$\frac{|\{G \mid G \in G_x,\ data_{1Anti}(G) > data_{1Tree}(G)\}|}{|G_x|} \tag{5.4}$$

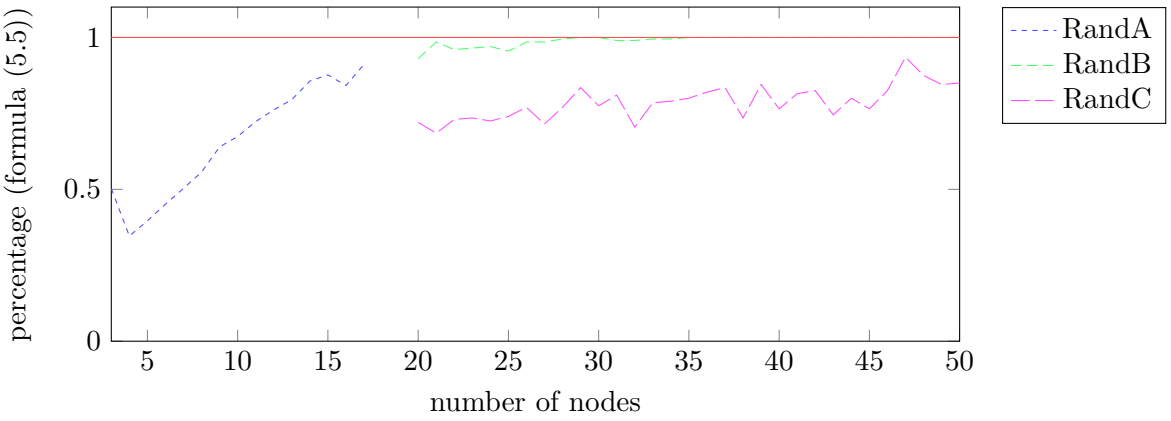
$$\frac{|\{G \mid G \in G_x,\ min_{1Anti}(G) > max_{1Tree}(G)\}|}{|G_x|} \tag{5.5}$$

$$\frac{\max_{v_i nV}(\min 1Anti(v))}{\max_{v_i nV}(\min 1Tree(v))} \tag{5.6}$$

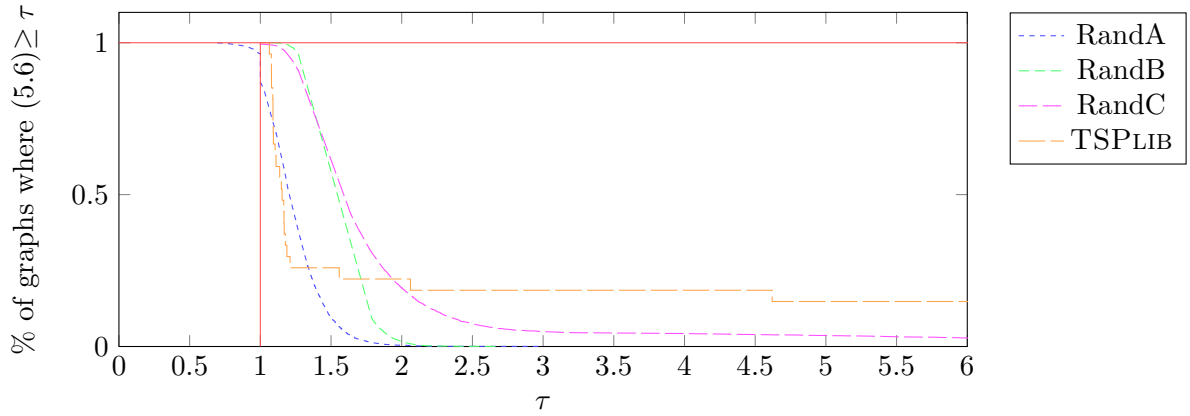
The results are shown on Fig.5.2 for the global results, Tab.5.2 for statistical data on TSPLIB and App.D for the RandB and RandC.



(a) Percentage of graphs of RandA where the statistical data for the minimal 1Anti's is strictly better than the one for the minimal 1Tree's



(b) Percentage of graphs where the minimum of the minimal 1Anti's is better than the maximum of the minimal 1Tree's



(c) Performance profile of the ratio between the maximum of the minimal 1Anti's and the maximum of the minimal 1Tree's

Figure 5.2: 1Anti versus 1Tree

	better	strictly better
mean	0.15	0.11
max	0.15	0.11
min	0.19	0.15
var	0.15	0.11
min 1Arbo vs max 1Anti	0.07	
min 1Anti vs max 1Arbo	0.41	

Table 5.3: 1Arbo vs 1Anti : percentages for TSPLIB benchmark

5.2 Comparison between 1Arbo and 1Anti

Now that the benefit of the 1Arbo and 1Anti have been shown compared to the 1Tree, let's compare them.

First, the same analysis can be performed with the following formulas :

$$\frac{|\{G \mid G \in G_x,\ data_{1Arbo}(G) > data_{1Anti}(G)\}|}{|G_x|} \tag{5.7}$$

$$\frac{|\{G \mid G \in G_x,\ min_{1Arbo}(G) > max_{1Anti}(G)\}|}{|G_x|} \tag{5.8}$$

$$\frac{|\{G \mid G \in G_x,\ min_{1Anti}(G) > max_{1Arbo}(G)\}|}{|G_x|} \tag{5.9}$$

$$\frac{\max_{v_i nV}(\min 1Arbo(v))}{\max_{v_i nV}(\min 1Anti(v))} \tag{5.10}$$

The results are at Fig.5.3 and Fig.5.4 for the global results, Tab.5.3 for statistical data on TSPLIB , App.D for RandB and RandC.

We can see on the graphs that for RandA, RandB and RandC, half of the graph tends to have a better bound with the 1Arbo and the other half with the 1Anti. When analysing the data's on the random benchmark, no clear difference can be deduced.

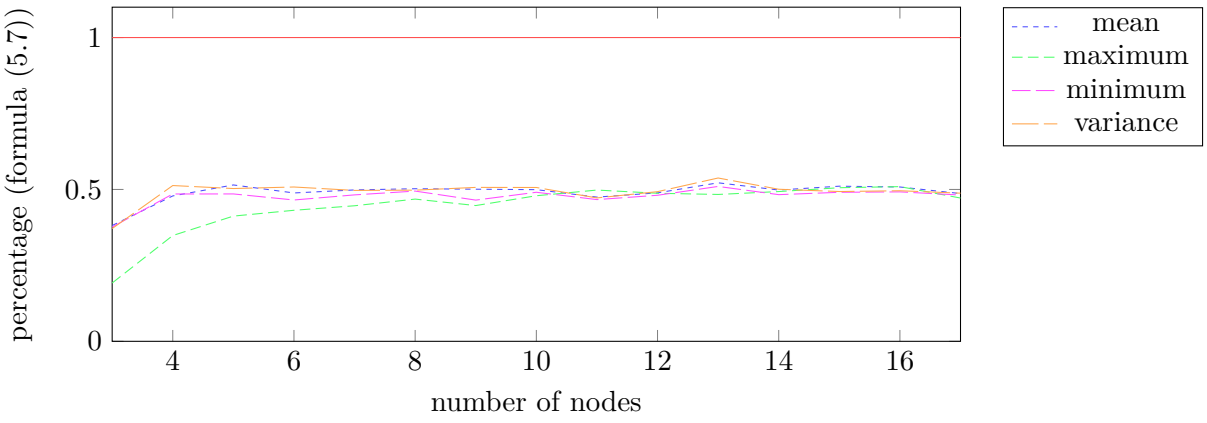
Concerning the TSPLIB , we can see on the table and the performance profile graph that almost all the graphs have a better result with the 1Anti (only 15% of the graphs have the maximum of the minimal 1Arbo's better than the maximum of the minimal 1Anti's). which differs from the result obtained via the random benchmarks.

The reason is that this benchmark is biased compared to the random benchmark.

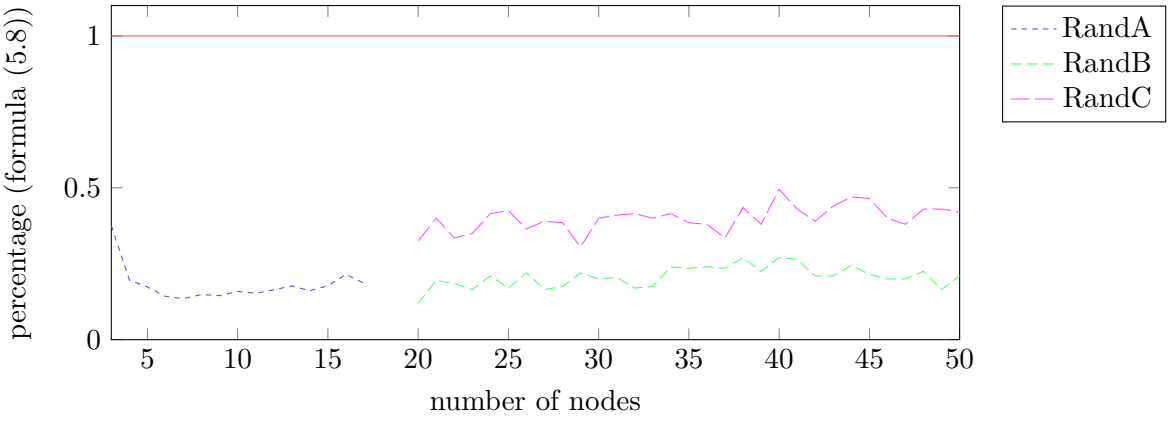
5.2.1 Influence of the variance at the nodes

For almost all of the graphs of the TSPLIB benchmark it can be seen that the following ratio is higher than 1 :

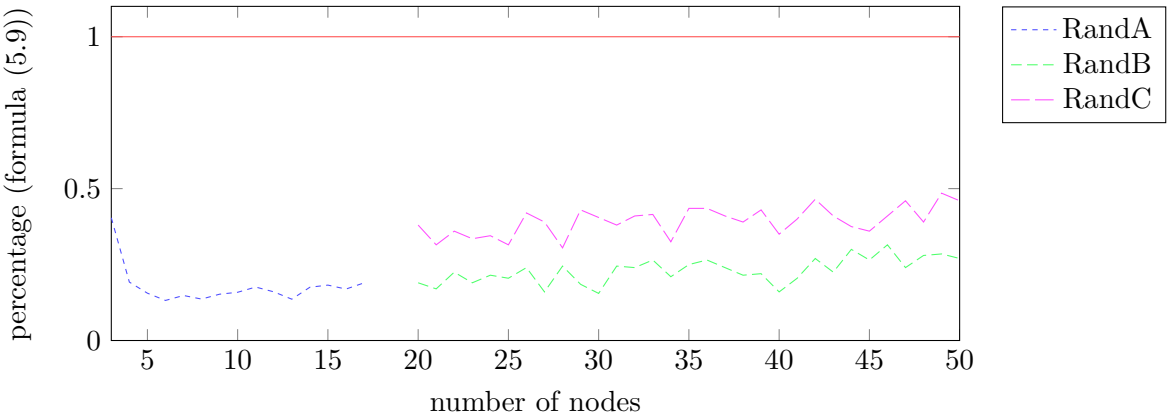
$$variance\_ratio = \frac{r_i}{r_o} \tag{5.11}$$



(a) Percentage of graphs of RandA where the statistical data for the minimal 1Arbo's is strictly better than the one for the minimal 1Anti's



(b) Percentage of graphs where the minimum of the minimal 1Arbo's is better than the maximum of the minimal 1Anti's



(c) Percentage of graphs where the minimum of the minimal 1Anti's is better than the maximum of the minimal 1Arbo's

Figure 5.3: 1Arbo versus 1Anti

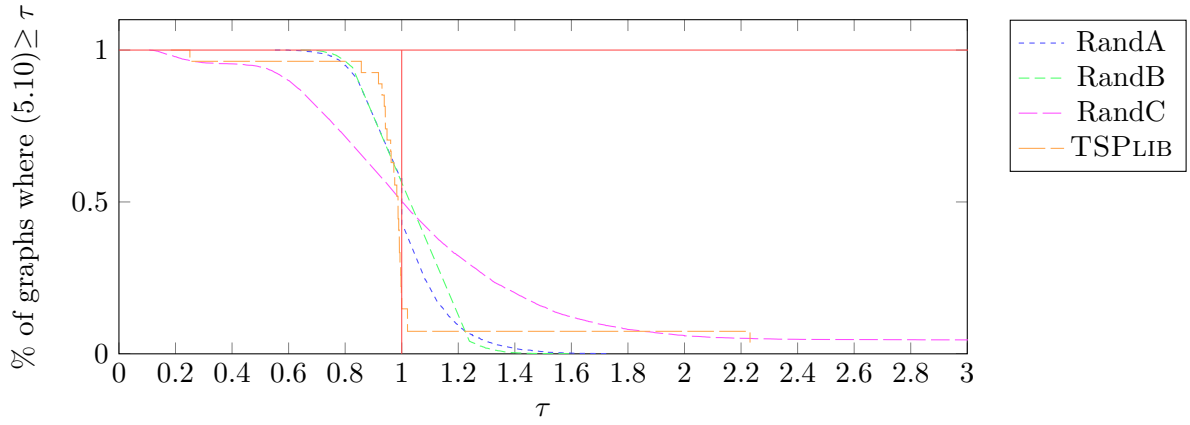


Figure 5.4: Performance profile of the ratio between the maximum of the minimal 1Arbo's and the maximum of the minimal 1Anti's

with

$$r_i = \frac{\sum_{v \in V} var_i(v)}{|V|} \qquad r_o = \frac{\sum_{v \in V} var_o(v)}{|V|}$$

$$var_i(v) = \frac{\sum_{e \in in-edge(v)} (c_e - mean_i(v))^2}{|in - edge(v)|} \qquad var_o(v) = \frac{\sum_{e \in out-edge(v)} (c_e - mean_o(v))^2}{|out - edge(v)|}$$

$$mean_i(v) = \frac{\sum_{e \in in-edge(v)} c_e}{|in - edge(e)|} \qquad mean_o(v) = \frac{\sum_{e \in out-edge(v)} c_e}{|out - edge(e)|}$$

Let's now verify if there exists a real correlation between this ratio and the advantage of taking either the 1Arbo or the 1Anti. To do it, a first plot is made of the ratio $\frac{mean\ 1Anti}{mean\ 1Arbo}$ function of the *variance_ratio* (Fig.5.5a).

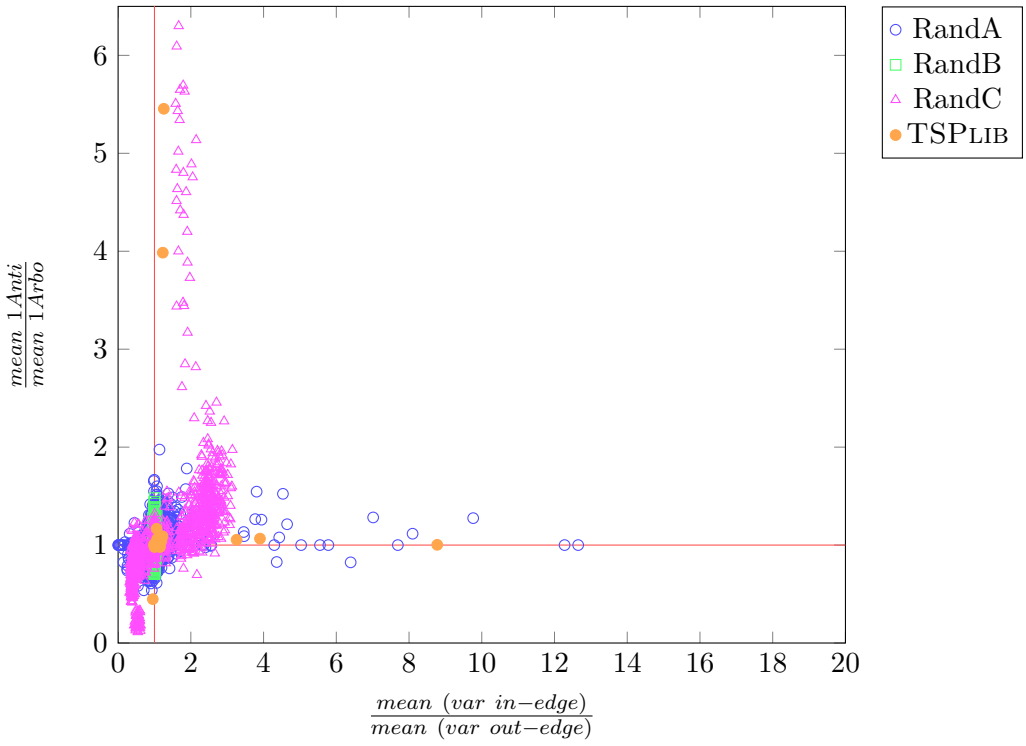
On this graph, it's easy to see that when the *variance_ratio* tends to be higher than 1, the ratio between the means tends also to be higher than 1 meaning a 1Anti is preferred. On the other hand, when the *variance_ratio* is lower than 1, the ratio of the means tends also to be lower than 1, meaning a 1Arbo is preferred.

When comparing with the ratio between the maximum of the two techniques (Fig.5.5b), the conclusions are the same.

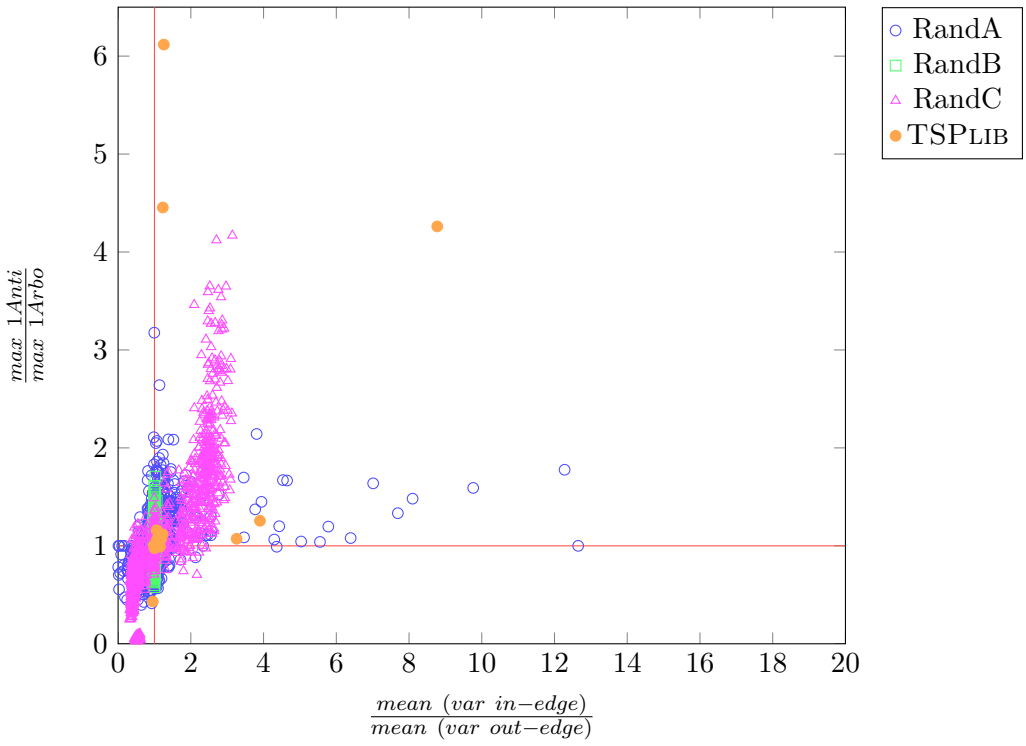
5.2.2 Influence of the degrees at the nodes

On Fig.5.5a and Fig.5.5b, a slight difference can be seen between the RandA (composed not only of full graphs), RandB and RandC benchmarks. RandA tends to have more graphs where the ratio won't predict the best choice. This is due to the fact that the degrees of the nodes influence also a little bit the choice.

The graph in Fig.5.6 is the same as Fig.5.5b but with only the RandA benchmark. The graphs were sorted following their distribution of number of edges by node. The first three sets correspond to unbalanced graphs (at least one node has a degree in different from its degree out). The



(a) Correlation with the ratio of the means



(b) Correlation with the ratio of the maximums

Figure 5.5: Correlation graphs

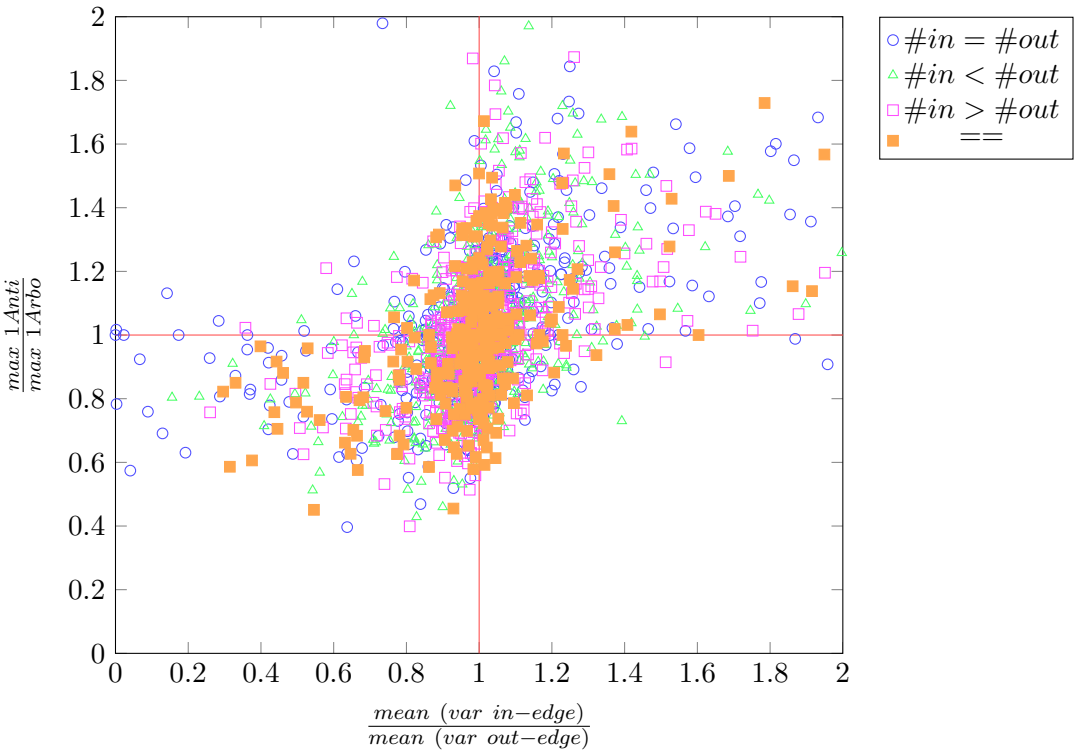


Figure 5.6: Influence of the degree over the graph of RandA

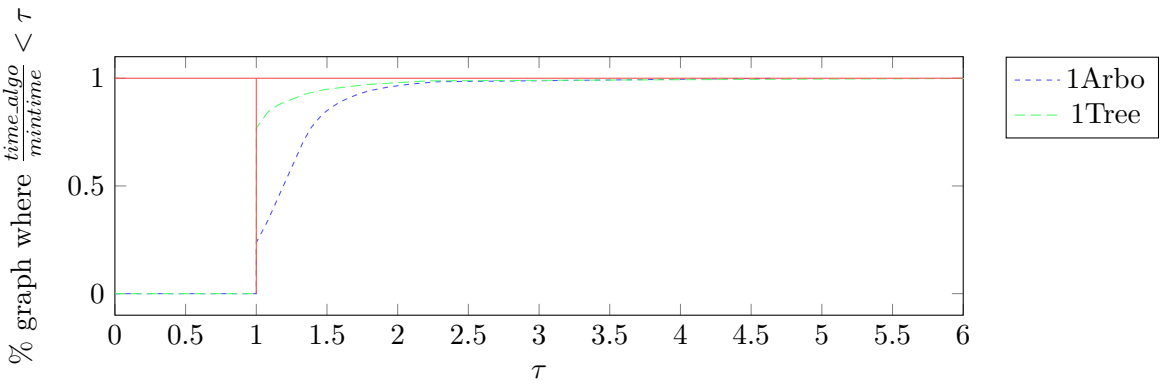
count of vertexes with a strictly higher in-degree than out ($\#in$) and strictly higher out-degree than in were made ($\#out$). The sets depict when $\#in > \#out$, $\#in = \#out$ and $\#in < \#out$. It can be seen that when the graph is balanced (4^{th} set), the prediction is almost always correct as thought. The set with the most out-predicted graphs is the second but the difference with the first and the third is too small to conclude something else than seen between RandA and RandB at the previous paragraph : When there are unbalanced vertexes (in-degree not equal to out-degree), the prediction is a little be more likely to lead to wrong choice of method.

5.3 Time comparison

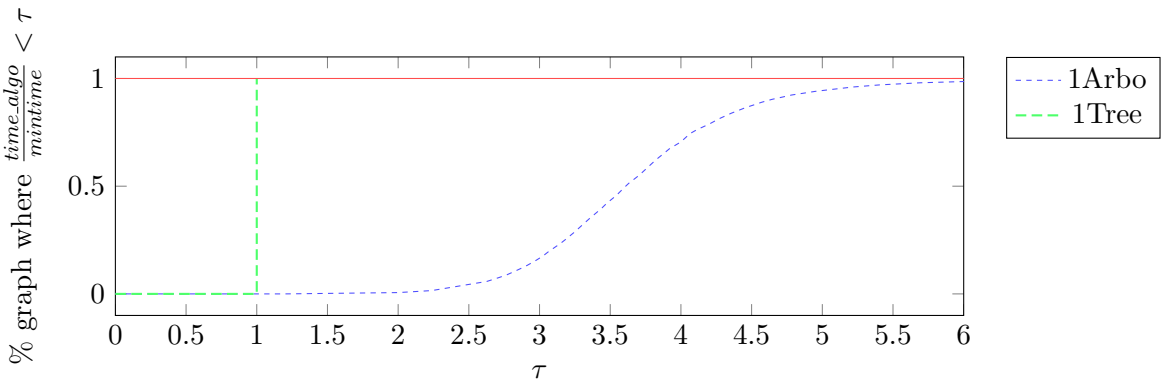
First, as the 1Arbo and the 1Anti are basically the same algorithms, we only compare the 1Arbo with the 1Tree. The comparison will be done on the RandB, RandC and TSPLib benchmarks between the time taken to do a 1Arbo and the time taken to do the 1Tree over the transformed graph. So even if the solving of a 1Tree over n nodes is quicker than the 1Arbo on n nodes, the relevant comparison will compare 1Tree's on $2n$ nodes against 1Arbo's on n nodes.

As to solve a 1Arbo or 1Anti, first a graph data structure is created and then the algorithm is executed, the measurements of the creation and the algorithm where done separately. By graph, the time to create the structure and the time to compute all the 1Arbo/1Tree were measured. Then the mean of the time to compute one 1Arbo/1Tree was retrieved from the graphs. We then have the measurements $time_{create}$ and $\bar{t}_{compute}$.

Two performance profiles on the RandB benchmark where made out of this. The first (Fig.5.7a)



(a) Performance profile of the global time (creation + solving) for RandB



(b) Performance profile of the execution time only for RandB

Figure 5.7: Performance profile graphs on RandB

considers the solving of the 1tree/1arbo with the creation of the data structure. The time used is then $t_{method} = time_{create} + \bar{t}_{compute}$. For each graph and each method the ratio $\frac{t_{method}}{\min(t_{1Arbo}, t_{1Tree})}$ was computed and the performance profile was draw. The second (Fig.5.7b) performance profile considers the same ratio but with $t_{method} = \bar{t}_{compute}$.

We can see that globally the 1Tree is more efficient than the 1Arbo and for the computation only, it is always the more efficient (almost at least 3 times more efficient for each graph). This difference can be explained by two factors :

- the fact that the implementation of the 1Tree is optimised by the use of Disjoint Sets while for the 1Arbo the algorithm is probably not as optimised as possible
- the fact that the 1Arbo algorithm is a more complex recursive algorithm with a number of recursions depending on the graph structure while the 1Arbo is basically a greedy algorithm with about the same number of operations for every graph of a given size

The same results are obtained with RandC and TSPLib (App.D).

5.4 Conclusion

The bounds computed with the 1Arbo or the 1Anti tend to be always better than the one computed with the 1Tree and the gap between their values grows with the size of the input graph. But this gain is achieved at the expense of more time to compute the bound.

Concerning the difference between the 1Arbo and the 1Anti, for some graphs one gives better result than the other. The variance ratio defined by (5.11) can act as a predictor to predict the possibly better value. The strength of the prediction depends on the internal structure of the graph. The prediction will be more accurate if the in and out degrees are the same for each node of the graph

Chapter 6

Analyse the iterative method and impact of the step size

This chapter contains a comparison analysis of the adaptation of the iterative method against the initial iterative algorithm with a modified graph. A first thing to remind in head is that, as said in [18],

> *We would particularly point out the choice of step size as an area which is imperfectly understood.*

A consequence of this is that no step size is proven mathematically to be better and a good step size of a graph may be not so good for another. This was verified during the analysis.

6.1 Different choices of step size allow mathematically

As already said in Section 3.2.3, the solving of the Lagrangian dual problem (finding the multiplier leading to the tightest bound for the optimal) can be iteratively done with three possible classes of step sizes that have a convergence guarantee :

- β_k with $\sum_k \beta_k \to \infty$ and $\beta_k \to 0$ as $k \to \infty$ (simple step size)
- $\beta_k = \beta_0 \rho^k$ with some parameter $\rho < 1$ (geometric step size)
- $\beta_k = \epsilon_k \frac{UB - LB}{\sum_{v \in V} (2 - \sum_{e \in \delta(i)} x_e)^2}$ with $0 < \epsilon_k < 2$ (complex step size)

The first was known to give the right bound but slowly. The second could converge if ρ and β_0 were big enough to avoid having the geometrical series going to 0 to fast. The last may possibly not converge to the best lower bound when the upper bound is not the value of the optimal solution of the initial problem.

To test the difference between these, let's fix the different step size used for each family:

- simple : $\beta_k = \frac{\lambda}{k}$ where λ is a fixed parameter
- geometric : $\beta_k = \beta_0 \rho^k$ where the parameters are β_0 and ρ

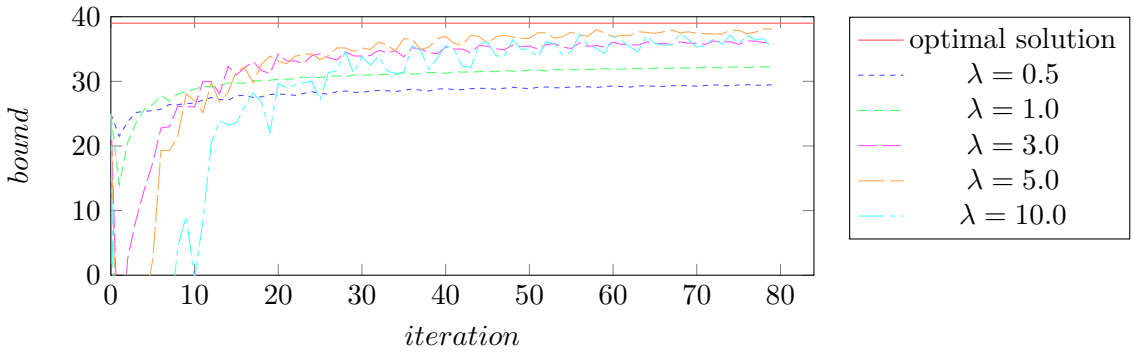


Figure 6.1: Convergence to wrong lower bounds for the simple step size

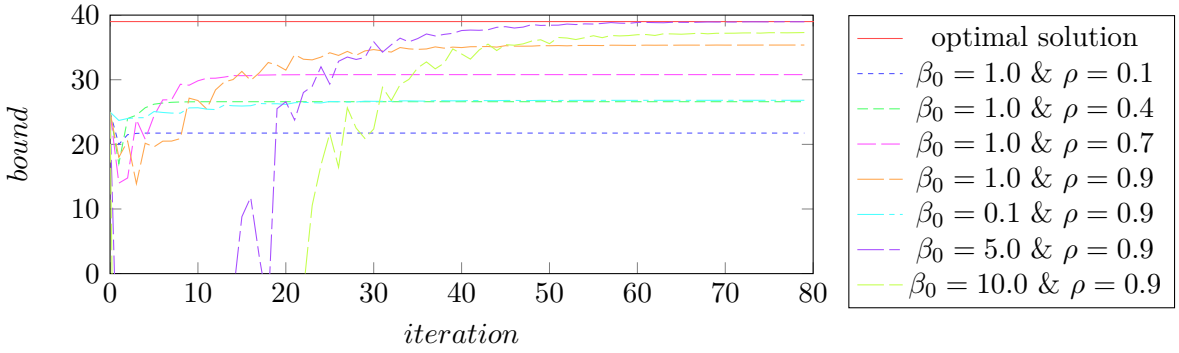


Figure 6.2: Convergence to wrong lower bounds for the geometric step size

- complex : $\beta_k = \epsilon_k \frac{UB - LB}{\sum_{v \in V} (2 - \sum_{e \in \delta(i)} x_e)^2}$ where the parameter is UB and where the $epsilon_k$ is update following the update method described in [18]. $\epsilon_k = 2$ for $2n$ iterations, where n is a measure of the problem size. Then successively halved both the value of ϵ_k and the number of iterations until the number of iterations reaches a threshold z. From this point, halved ϵ_k every z iteration until end. Here n has been fixed to half the number of nodes of the graph and z to $\left\lceil \frac{|V|}{100} \right\rceil$.

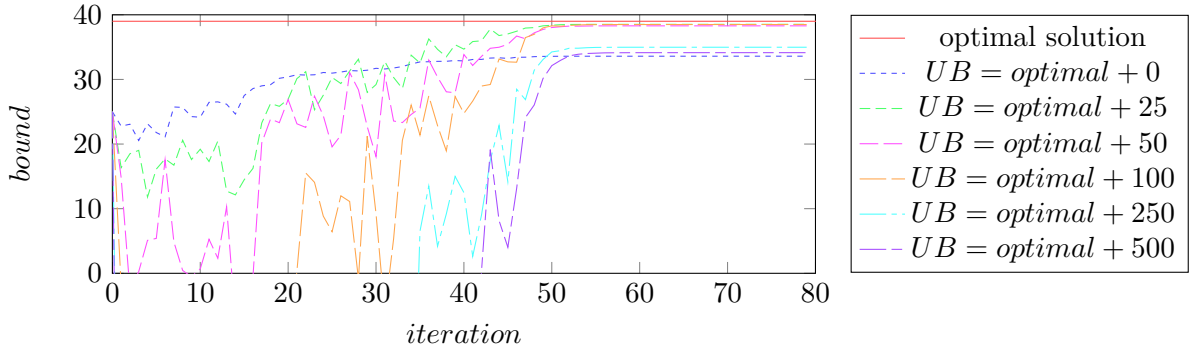


Figure 6.3: Convergence to wrong lower bounds for the complex step size

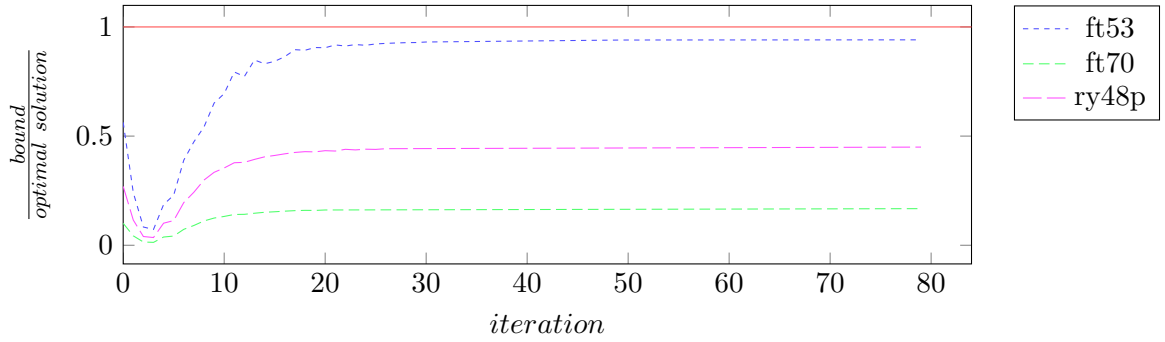


Figure 6.4: Same step size for similar graphs

6.2 Illustration of the drawbacks of the different methods

At graph Fig.6.1, Fig.6.2 & Fig.6.3, you can see the computation of the lower bound of the graph `br17` of TSPLib by the adapted Held Karp method with the different step size, each with different values of the parameters.

The first graph uses the simplest bound. After 80 iterations, the value of the bound continues to grow, but too slowly to let the algorithm continue until the end of convergence. Following the parameter, the point where the convergence became very slow can arise at different values. The parameter leading to this best value is neither the smallest, neither the highest.

The second uses a geometric step with different parameters. We can see that the bound can converge to different values following the parameters. We can also see that the parameters giving the best convergence are not the highest ($\beta_0 = 5$ & $\rho = 0.9$ give a better result than $\beta_0 = 10$ & $\rho = 0.9$).

The last uses a complex step with the UB parameter varying from the value of the optimal tour to higher values. It can be seen that the convergence happens not only at the optimal lower bound. And we can also see that setting the optimal value as UB doesn't give the best bound.

At Fig.6.4, the same step size (geometric with $\beta_0 = 100$ and $\rho = 0.9$) was used on similar graphs of the TSPLib benchmark (ft53, ft70 and ry48p). These graphs have a similar range of costs (between 0 and about 2500), about the same number of vertexes (respectively 53, 70 and 48) and about the same mean of cost by edges (respectively about 500, 1000 and 1100). The ratio $\frac{bound}{optimal\ value}$ shows us how far from the optimal we are. We can see that a same bound can end up to be very good for one graph, while being very bad for another. The parameters leading to a good bound depend highly on the graph seems to be unpredictable.

With these examples, the weakness of each step size family has been shown. An the impact of good parameters has also been illustrated.

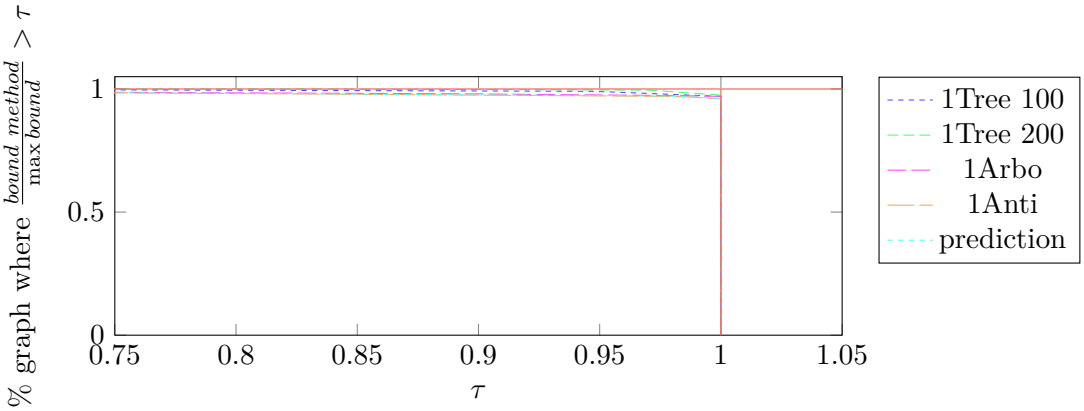


Figure 6.5: Bound performance profile of the simple step for RandB

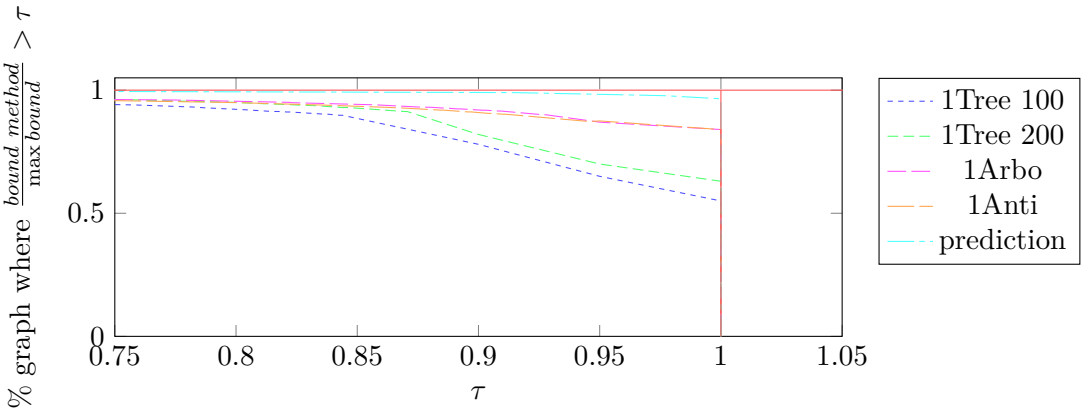


Figure 6.6: Bound performance profile of the simple step for RandC

6.3 Comparison of the different methods with a same step size

For each step sizes, some fixed parameters will be used and the bound for each method (Held-Karp symmetric with the transformation with 2 nodes and the linking edge forced, the adaptation of the Held-Karp based on the 1Arbo, the adaptation of the Held-Karp based on the 1Anti and the adaptation of the Held-Karp with either the 1Arbo or the 1Anti following the predictor) will be computed and the performance profile will be computed. As our goal is to have a best bound as quickly as possible, the number of iterations will be set to 100 for the asymmetric method and 200 for the symmetric one since the computation of a 1Tree has been shone to be quicker. The test where made on the RandA, RandB, RandC and TSPLib benchmarks. The remaining graphs are in Appendix E.

6.3.1 Simple step size

The simple step size was set with $\lambda = 100$.

On the graphs Fig.6.5, 6.6 & 6.7, we can see the performance profile concerning the bound found. We can see that for each the one behaving the better is the asymmetric with prediction to chose between 1Arbo and 1Anti. For the RandC benchmark (Fig.6.6), the difference between

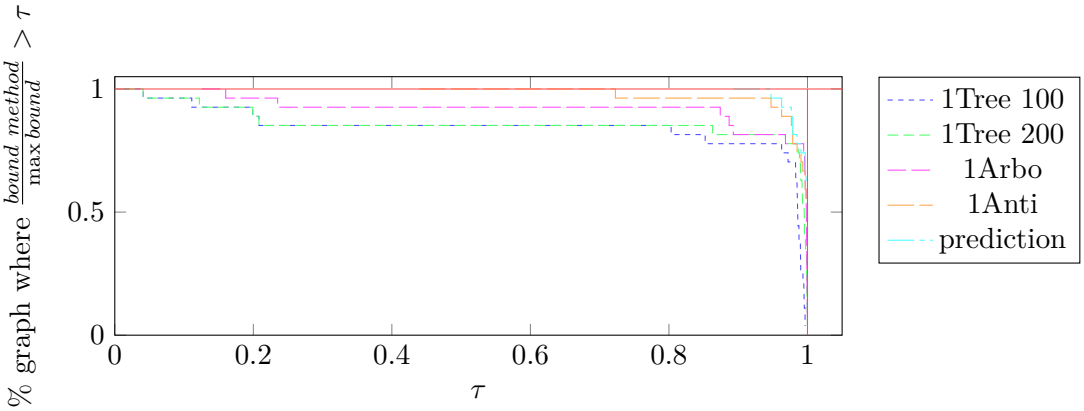


Figure 6.7: Bound performance profile of the simple step for TSPLIB

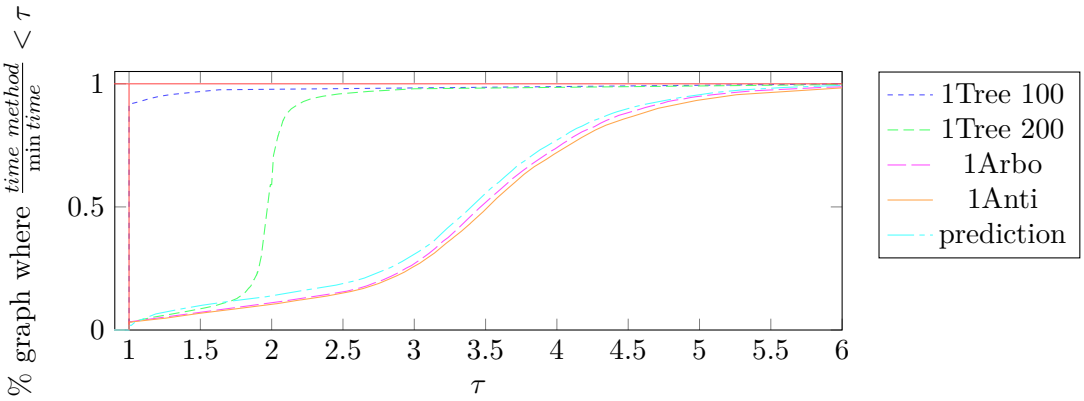


Figure 6.8: Time performance profile of the simple step for RandC

the simple 1Arbo or 1Anti with the one with prediction is much more pronounced. The 1Tree, either with 100 iterations or 200, gives the worst results.

Concerning the time taken (Fig.6.8), as expected, the 1Tree with 100 iterations is the best for most of the graphs, following by the 1Tree with 200 iterations. The 1Arbo and 1Anti are approximately the same. The choice with predictor has a slightly better profile, this can arise if the chosen method is the quicker for the iteration. The results are similar for the other benchmarks.

6.3.2 Geometric step size

The geometric step size was set with $\beta_0 = 100$ and $\rho = 0.9$.

In the case of the geometric step size (Fig.6.9 & 6.10), the best is the asymmetric with prediction following by the two other asymmetric still similar. The symmetric ones are still the worsts.

On the first graph (Fig.6.9), we can also notice the effect of the convergence to some value as 1Tree with 100 iterations and 1Tree with 200 has exactly the same profiles.

Concerning the time (Fig.6.11), the 100 iterations' 1Tree is still the better followed by the other

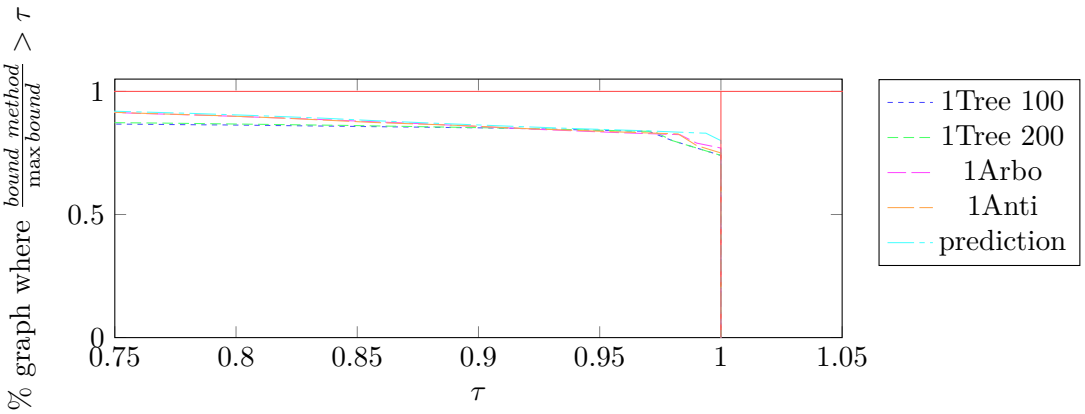


Figure 6.9: Bound performance profile of the geometric step for RandB

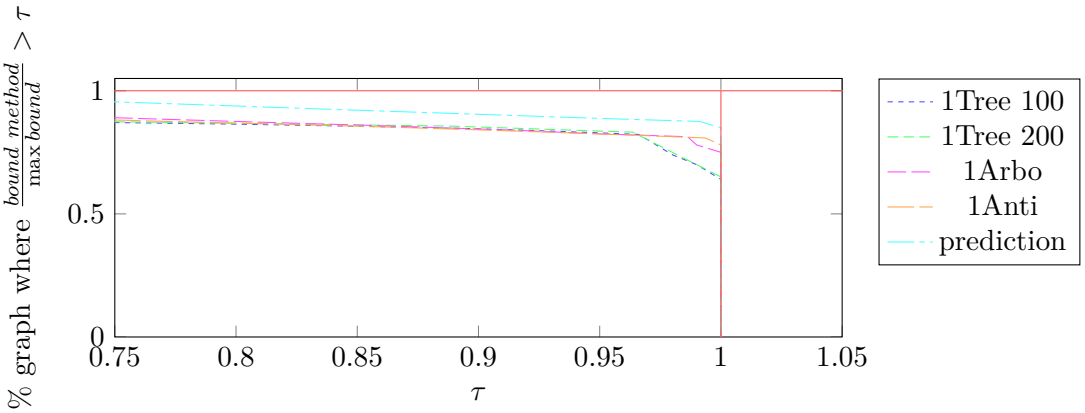


Figure 6.10: Bound performance profile of the geometric step for RandC

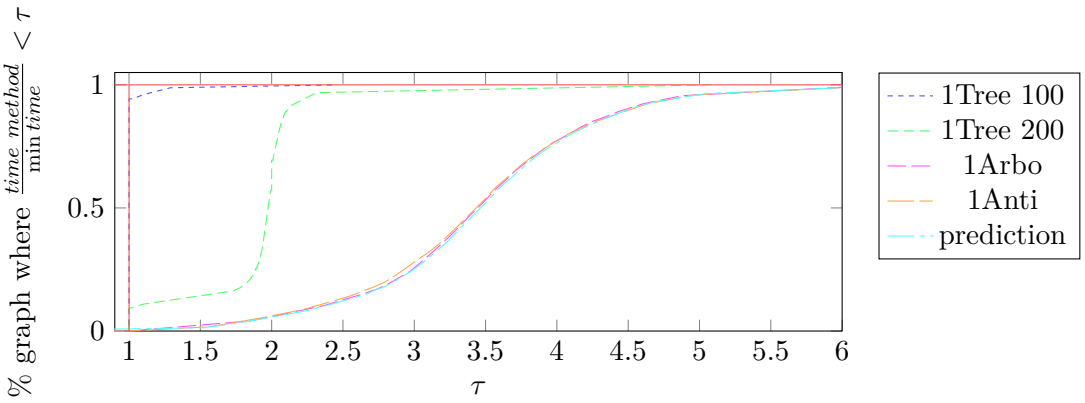


Figure 6.11: Time performance profile of the geometric step for RandB

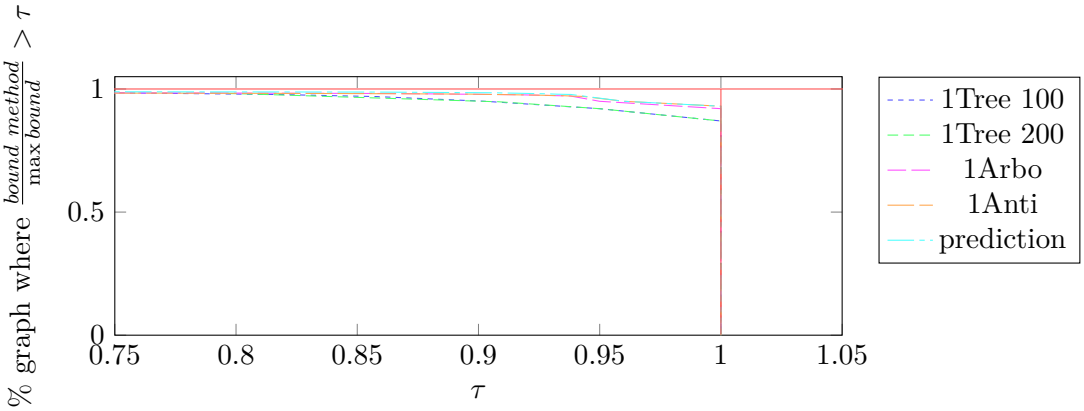


Figure 6.12: Bound performance profile of the complex step for RandA

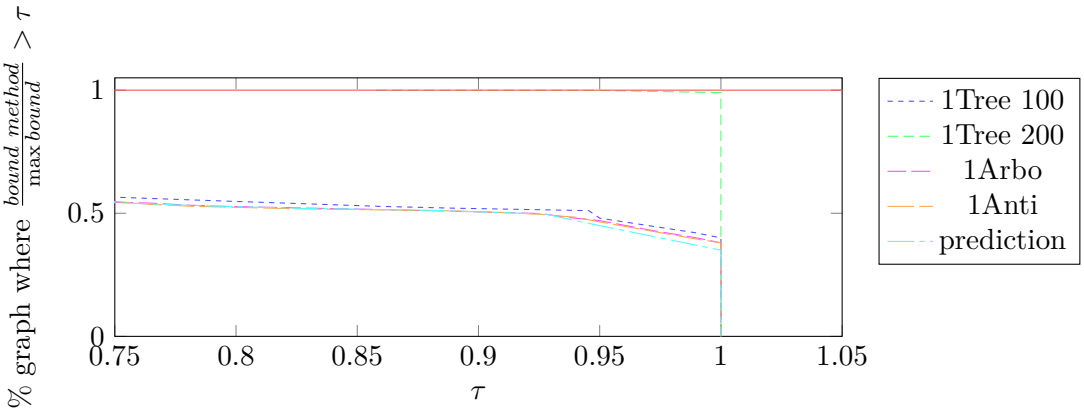


Figure 6.13: Bound performance profile of the complex step for RandB

1Tree and then all the asymmetric based method with profiles similar to each other.

The other results on the other benchmarks show the same trends.

6.3.3 Complex step size

The complex step size was tested with the update of ϵ and was set with parameter $ub = \min\{\sum_v(\max in - edge(v)), \sum_v(\max out - edge(v))\}$.

For the bound (Fig.6.12 & 6.13), here the conclusion differs from the previous ones. In this case, the best seems to be the 1Tree with 200 iterations. This is probably due to the fact that this step size with our arbitrary parameters make a little bit more time to be good enough. More iterations should be used to have again the asymmetric bounds better than the symmetric ones.

Concerning the time (Fig.6.14), the conclusions are still the same : the 1Tree is the fastest.

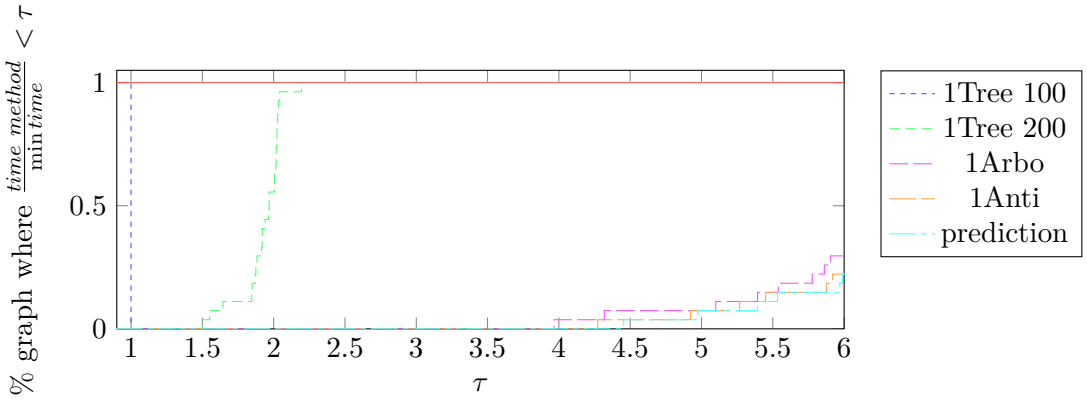


Figure 6.14: Time performance profile of the complex step for TSPlib

6.4 Conclusion

The first thing learnt is that the right parameters for a given step size family hardly depend on the graph.

Secondly, for a given step size and given parameters, if the number of iterations is enough, the asymmetric based methods give generally the best lower bound. But the cost to achieve such a result is much higher for the same number of iterations. To become more practical the asymmetric based methods should be optimised.

The last thing saw is the effect of the predictor (Section 5.2.1) with which a generally better bound is achieved.

Chapter 7

`mincircuit` as an OscaR constraint

This chapter details the implementation of the `mincircuit` constraint as a constraint of the solver OscaR.

7.1 The `Constraint` interface

The `Constraint` interface [11] has only one abstract method :

```
def setup(l: CPPropagStrength): CPOutcome
```

This method has to contain the registrations to the domain modification and also a first consistency check.

Another interesting method is the `propagate` method, called when a registered event for propagation occurs.

```
def propagate(): CPOutcome = CPOutcome.Suspend
```

These are the two methods from the interface implemented/overrided to make our own `mincircuit` constraint [7].

7.2 `mincircuit` as an aggregation of other constraints

A big principle in programming is the reuse of already existing code. This ensures more maintainability and also reduces the bugs.

This is why this `mincircuit` constraint is built from a first `circuit` constraint. To be able to do it, the successor model must be followed and an array `succ` ($\forall i; 0 \leq i \leq succ.size - 1; succ(i) = successor\ of\ node\ i$) containing the successor nodes will be used. This constraint will take care that at the end we have a single circuit, an Hamiltonian circuit.

Now concerning the cost of the tour, a variable will be used to contain it. To compute the cost, we will need the list of the possible edges, their costs and a mean to now if they are selected or not. To do this, a set containing a tuple by edge will be kept. This tuple contains the source,

the destination and the cost of the edge and also a boolean variable representing the selection or not of the edge. A sum constraint linking the variable containing the total cost of the tour and the sum for all edges of the cost multiplied by their boolean variable will be set.

The last thing needed will be to link the successor model with the edge selection model. To do it, for all pairs of vertexes, a constraint is added. If their exists an edge going from the first vertex to the second, the constraint created will link the value of the boolean variable to the boolean equality between the value of the variable succ(first) and the second ($x_{ij} == (succ(i) == j)$). If their exists no edge between the two vertexes, the value of the second vertex is retrieved from the variable containing the successor of the first by an inequality constraint ($succ(i)! = j$).

7.3 `setup` of the constraint

First, the setup will first add the constraint of the model to the solver. Then, if the solver has not failed, the registration to events is made. As the cost must change only when one edge is selected or discarded, the `setup` will register a call to the `propagation` method each time the domain of one of the variables in the `succ` array changes.

7.4 `propagation` of the constraint

Up to this point, the constraint is built the same way it will either use a weaker bounding procedure or not. This is why this was set in an `MinCircuit` interface implemented by three classes : `MinCircuitWeak` `MinCircuitMedium` and `MinCircuitStrong`. The abstract method will be the one computing the lower bound.

The propagate method of the `mincircuit` will first update possibly the upper bound of the cost. If the solver hasn't failed, the lower bound will be computed by calling the abstract method. Then the lower bound will be updated.

The different lower bound computation are :

- weak propagation : the maximum of all the 1Arbo or 1Anti, the method is choose following the predictor.
- medium propagation : the bound is computed with the symmetric iterative Held-Karp method with 200 iterations and the geometric step size (with $\beta_0 = 100$ and $\rho = 0.9$).
- strong propagation : the bound is computed with the asymmetric iterative Held-Karp method using the predictor and with the same parameters as the medium propagation (200 iterations and geometric step size)

7.5 Adding a `mincircuit` constraint to a model

To add a `mincircuit` constraint, four things need to be given in argument to a companion object used to return the corresponding constraint :

- an array for the successors
- a variable to carry out the cost of the tour
- the set of tuples (source, destination, cost) describing the edges
- a boolean, set as true if the stronger propagation is wanted

7.6 Tests

To do the test, a simple cp search with a binary first fail on the successor gives :

- With the strongest bound, the best value found was not the optimal but very close (40 instead of 39) and found after a few seconds.
- With the medium bound, no solution where found.
- With the weak bound, no solution where found.

For the symmetric method, because of the speed of the computation, the number of nodes visited was very high (+-144000 nodes explored) with respect to the number of nodes visited by the asymmetric one (+-4500 nodes explored). But as the lower bound isn't effective enough, almost no branches are cut. For the weakest bound, the amount of node visited is very high (+- 271500 nodes) but almost no pruning is done.

The same tests were done with `ft53` leading to the same conclusions : for a same parameter, the Medium and the Weak bound go over many nodes but too few pruning arise.

Conclusion

To conclude this master's thesis, let's remind our initial question : Is a differentiation between the symmetric and the asymmetric `mincircuit` useful?

Specific constraint The simplest bound

Concerning the simplest bound, from a theoretical point of view there is clearly an advantage in using the 1Arborescence/1Anti-Arborescence instead of the 1Tree. But from a performance point of view, without an optimised version of the computation of the 1Arborescence, its actual cost may be a disadvantage.

If using the 1Arborescence/1Anti-Arborescence, the prediction based on the variances of the in and out-edges of the nodes is useful can most likely lead to the best bound without the high cost of computing all the 1Arborescence and 1Anti-Arborescence.

Specific constraint The iterative bound

From a theoretical point of view, both techniques should converge to the same bound. But in practice this isn't all the time the case.

First, following the step size and the parameters chosen, a convergence to another value than the optimal lower bound can be seen. Then, we can see that, given a same step size with the same parameters, the asymmetric methods results in a better bound for the same number of iterations. And there is also no guarantee (except for the slow simple step size family) that the symmetric method would reach the same bound value since it can converge to a lower value due to the drawback of the step size. From this point of view, the 1Arborescence/1Anti-Arborescence should be privileged. Also, the use of the predictor has also shown better results in general.

Concerning the time, again, the actual 1Arborescence takes much more time.

Specific constraint As for the constraint the sooner a value is removed from the domain the better it is, a 1Arborescence/ 1Anti-arborescence bound may be useful at the beginning, even if it takes more time to reduce at most the search tree close to the root.

References

[1] Algorithm Design, Jon Kleinberg and Eva Tardos, Tsinghua University Press, 2005, ISBN: 0-321-29535-8, p.177-183, *Chapter 4, section 9 : Minimum-cost Arborescence: A multi-phase greedy algorithm*

[2] Analysis of the Held-Karp Heurisic for the Traveling Salesman Problem, David Paul Williamson, 1990,

`http://people.orie.cornell.edu/dpw/papers/masters.pdf`

[3] Combinatorial Optimization, William J. Cook, William H. Cunningham, William R. Pulleyblank and Alexander Schrijver, 1997, Wiley-Interscience, ISBN: 978-0-471-55894-1

- p.241-271 : The travelling salesman problem
- p.252-261 : Lower bounds on TSP (Held Karp, Linear,...)

[4] Concorde TSP solver,

`http://www.math.uwaterloo.ca/tsp/concorde.html`

[5] Course LINMA1691 Discrete mathematics - Graph theory and algorithms, UCL, Vincent Blondel, followed in 2012-2013,

`http://www.uclouvain.be/en-cours-2014-LINMA1691.html`

[6] Course LINGI2365 Constraint Programming, UCL, Yves Deville, followed in 2013-2014, `http://www.uclouvain.be/en-cours-2013-LINGI2365.html`

[7] CP for the impatient,

`https://www.info.ucl.ac.be/~pschaus/cp4impatient/`

[8] Exact algorithms for the asymmetric travelling salesman problem, Paolo Toth, DEIS, University of Bologna

[9] *Hybrid method (CP+OR)*, Willem-Jan van Hoeve, ACP Summer School on Practical Constraint Programming, 2014, slides 32-43

`http://school.a4cp.org/summer2014/`

[10] Integer Programming, Laurence A. Wolsey, 1998, Wiley-Interscience, ISBN: 978-0-471-28366-9

- p.5-6 : Definition of the assignment problem

- p.7-8 : Definition of the asymmetric TSP
- p.8-9 : Combinatorial explosion of constraints
- p.157-160 : Branch & cut technique
- p.167-172 : Definition of Lagrangian Relaxation
- p.172-173 : Strength of the Lagrangian relaxation with respect to linear relaxation
- p.173-177 : Iterative solving of the Lagrangian Dual explained with the TSP

[11] OscaR: Scala in OR, OscaR Team, 2012

`http://oscarlib.bitbucket.org/`

[12] Principles of Constraint Programming, Krzysztof R. Apt, 2003, ISBN: 978-0-521-12549-9

[13] Scala programming language,

`http://www.scala-lang.org/`

[14] The traveling-salesman problem and minimum spanning trees, Michael Held and Richard M. Karp, 1969

`http://www.jstor.org/stable/169411?seq=1#page_scan_tab_contents`

[15] The traveling-salesman problem and minimum spanning trees: part 2, Michael Held and Richard M. Karp, 1970

`http://download-v2.springer.com/static/pdf/733/art%253A10.1007%252FBF01584070.pdf?token2=exp=1431099905~acl=%2Fstatic%2Fpdf%2F733%2Fart%25253A10.1007%25252FBF01584070.pdf*~hmac=3bffb8e3777fc83ee4e65d35d60664ddb510e23ec1ad3c4f6a2a34cbba35a25a`

[16] Transforming asymmetric into symmetric traveling salesman problems, Roy Jonker and Ton Volgenant

`http://www.sciencedirect.com/science/article/pii/0167637783900482`

[17] TSPlib,

`http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

[18] Validation of a subgradient optimization, Michael Held, Philip Wolfe and Harlan P. Crowder , 1973

`http://download-v2.springer.com/static/pdf/127/art%253A10.1007%252FBF01580223.pdf?token2=exp=1432630316~acl=%2Fstatic%2Fpdf%2F127%2Fart%25253A10.1007%25252FBF01580223.pdf*~hmac=6240bbd300077a814c71fb4976c117389d6aaafb667e921b64451149b3cbeaf4`

Appendix

Appendix A

Reminders on Graph Theory

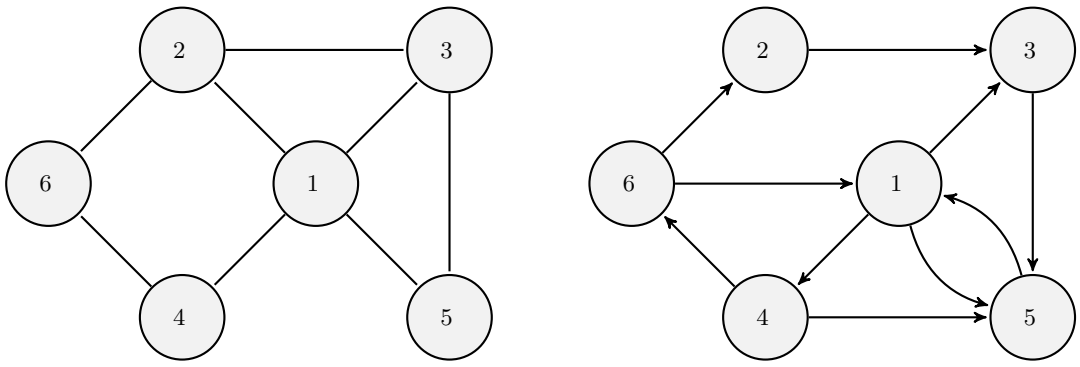
This appendix can be viewed as a reminder of basic knowledge on graph theory [5]. This basis is required to understand this master's thesis. If you are not fluent with the terms used or if you don't recall some of them, it is advised to read it.

A.1 Graphs in general

Definition - A.1.1 (Graph) *A graph (Fig.A.1a) $G = (V, E)$ is composed of a set V of **vertexes** (also called nodes) and a set E of **edges** (also called arcs) linking two vertexes of the graph together. These edges can be taken in both directions. They are also called undirected or symmetric graph.*

Definition - A.1.2 (Directed graph) *A directed (or asymmetric) graph (Fig.A.1b) is a graph $G = (V, E)$ composed of a set V of vertexes and a set E of directed edges going from one vertex (the source) to another (the destination) of the graph. The edges can only be taken from the source to the destination and not in the opposite direction.*

Definition - A.1.3 (Sub-graph) *A sub-graph (Fig.A.2) of a graph defined by $G = (V, E)$ is a graph defined by $G_s = (V_s, E_s)$ where $V_s \subseteq V$, $E_s \subseteq E$*



(a) Undirected graph (b) Directed graph

Figure A.1: Example of graphs

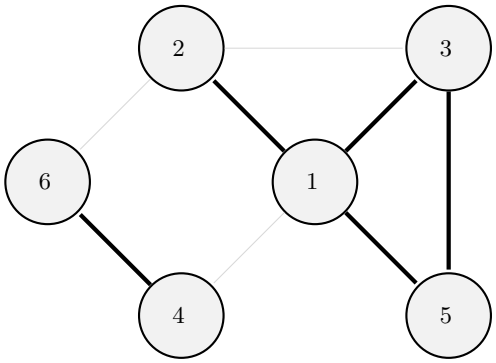
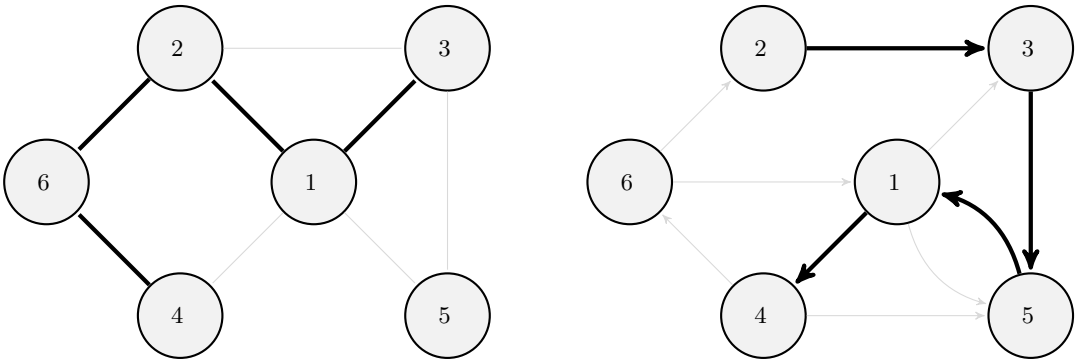


Figure A.2: Sub-graph of graph Fig.A.1a



(a) Path in graph Fig.A.1a

(b) Directed path in graph Fig.A.1b

Figure A.3: Example of paths

A.2 Path, cycle and connectivity

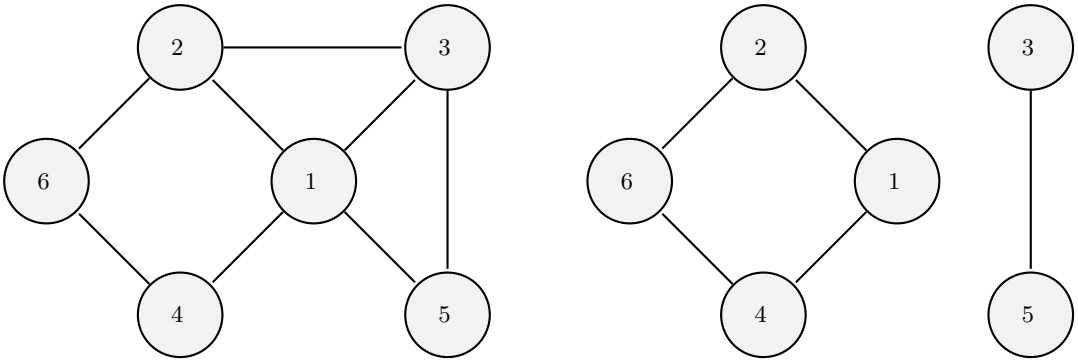
Definition - A.2.1 (Path) *A path (Fig.A.3) between s and t ($s, t \in V$) in a graph $G = (V, E)$ is a succession of vertexes $v_0 - v_1 - v_2 - ... - v_n$ for which $v_0 = s$, $v_n = t$, all vertexes of the path are different and there is an edge $(v_i, v_{i+1}) \in E$ for each $0 \leq i \leq n - 1$.*

Definition - A.2.2 (Connected graph) *An undirected graph (Fig.A.4) is connected if there exists a path between each pair of vertexes of the graph.*

A directed graph (Fig.A.5) is weakly connected if the undirected graph obtained by replacing all directed edge by undirected one is connected. A directed graph is strongly connected if there exist a path between each couple of vertexes.

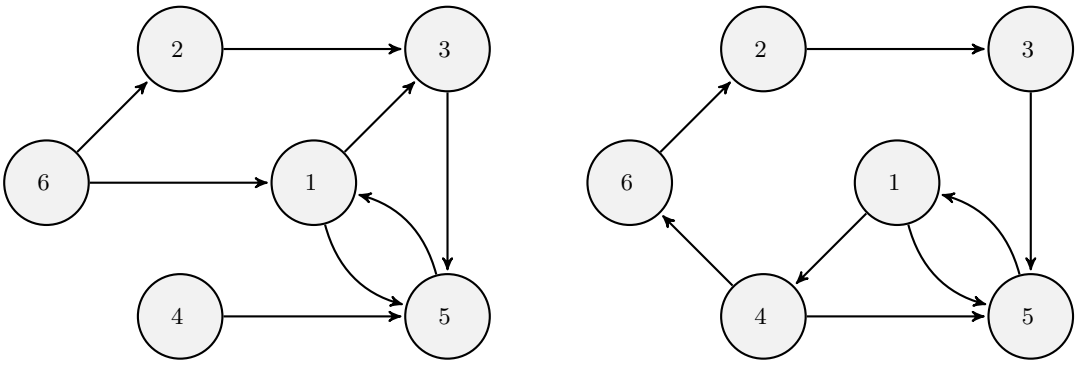
Definition - A.2.3 (Cycle) *A cycle (Fig.A.6) is a succession of vertexes $v_0 - v_1 - ... - v_{n-1}$ where the same vertex doesn't appear twice in the sequence and there is an edge $(v_i, v_{(i+1) \ mod \ n}) \in E$ for each $0 \leq i \leq n - 1$.*

Definition - A.2.4 (Hamiltonian cycle) *An Hamiltonian cycle (Fig.A.7) is a cycle involving each vertex of the graph.*

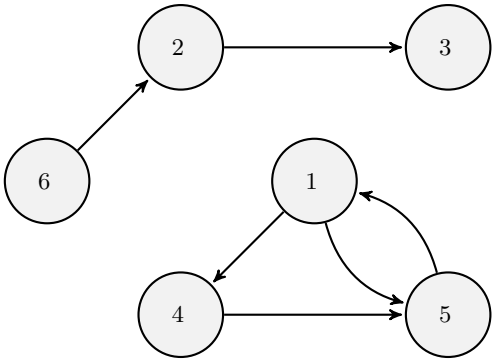


(a) connected graph (b) unconnected graph

Figure A.4: Example of connected graphs

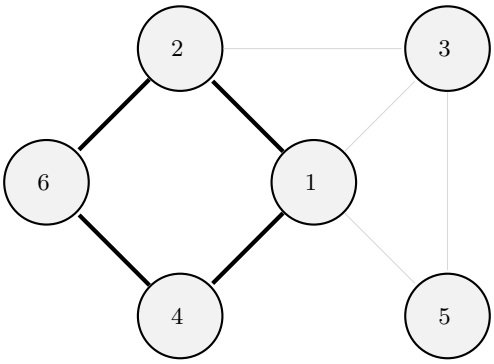


(a) weakly connected directed graph (b) strongly connected directed graph

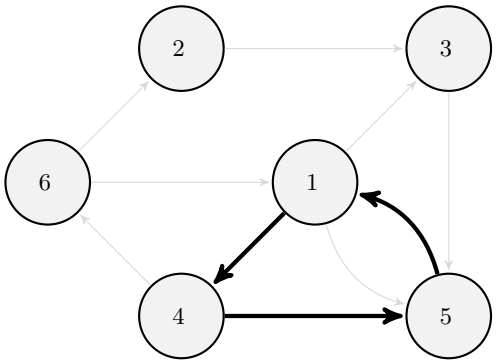


(c) unconnected directed graph

Figure A.5: Example of connected directed graphs

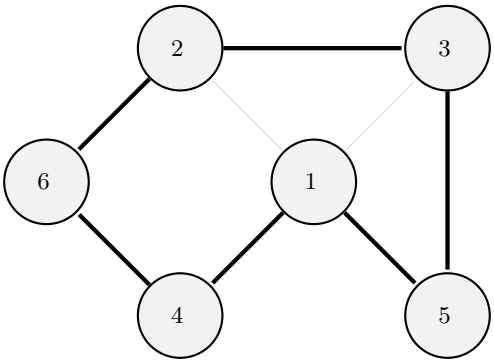


(a) Cycle in graph Fig.A.1a

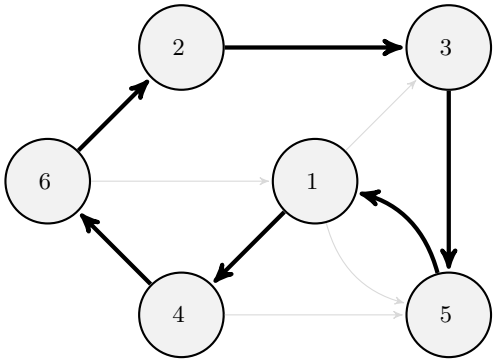


(b) Cycle in directed graph Fig.A.1b

Figure A.6: Example of a cycle



(a) Hamiltonian cycle in graph Fig.A.1a



(b) Hamiltonian cycle in asymmetric graph Fig.A.1b

Figure A.7: Example of a Hamiltonian cycle

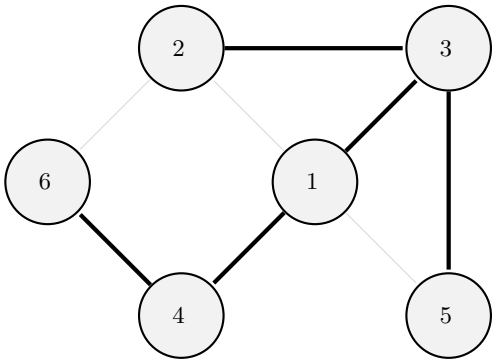


Figure A.8: Tree sub-graph of graph Fig.A.1a

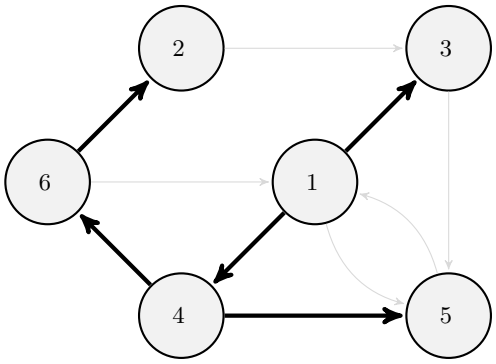


Figure A.9: Arborescence sub-graph (rooted in 1) of graph Fig.A.1b

A.3 Trees and arborescences

Definition - A.3.1 (Tree) *A tree (Fig.A.8) in an undirected graph is a subgraph $G_t = (V, E_t)$ of the graph $G = (V, E)$ where $E_t \subseteq E$ and where the following conditions are respected :*

- $|E_t| = |V| - 1$
- *the subgraph is connected*

The definition stays the same if the second property is replaced by : the subgraph doesn't contain any cycles

Definition - A.3.2 (Arborescence) *An arborescence (Fig.A.9) rooted in $r \in V$ in a directed graph is a sub-graph $G_t = (V, E_t)$ where the following conditions are respected :*

- $|E_t| = |V| - 1$
- *there exists a path from r to each other vertex of the graph*

An arborescence has exactly one edge entering each node (except the root).

Definition - A.3.3 (Anti-arborescence) *An anti-arborescence (Fig.A.10) anti-rooted in $r \in V$ in a directed graph is a sub-graph $G_t = (V, E_t)$ where the following conditions are respected :*

- $|E_t| = |V| - 1$
- *there exists a path from each other vertex of the graph to the anti-root r*

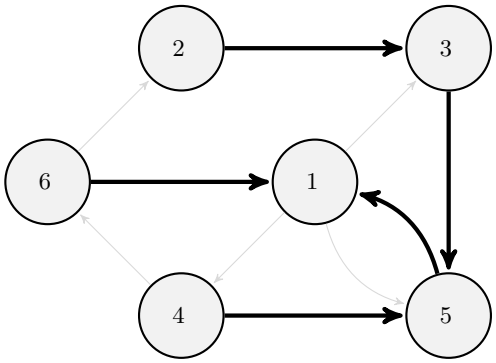


Figure A.10: Anti-arborescence sub-graph (anti-rooted in 1) of graph Fig.A.1b

An anti-arborescence has exactly one edge going out of each node (except the anti-root).

A.4 Costs on graphs

Definition - A.4.1 (Costs on a graph) *A cost can be associated to each edge of $G = (V, E)$. The meaning of this cost can be the time, the amount of resource needed,... to go through this edge. The cost of the graph is then the sum of all costs of its edges.*

Definition - A.4.2 (Minimum spanning tree) *The minimum spanning tree of a graph is a tree for which its cost is the minimum among all the possible trees.*

Definition - A.4.3 (Minimum arborescence) *The minimum arborescence of a directed graph is an arborescence for which its cost is the minimum among all the possible arborescences.*

Definition - A.4.4 (Minimum anti-arborescence) *The minimum anti-arborescence of a directed graph is an anti-arborescence for which its cost is the minimum possible among all the possible anti-arborescences.*

Appendix B

Lagrangian Relaxation

This appendix explains the mechanisms and the mathematical concepts underlying the Lagrangian Relaxation.

Relaxation

Lagrangian relaxation consists of relaxing a complex optimisation problem, making it easier to solve. However, this implies for the optimal solution to be relaxed and no more optimal for the initial problem.

How useful can this be, if the solution found isn't the solution to the original problem? The new optimum can be used as a bound for the optimum of the initial problem (lower bound for minimising problems and upper bound for maximising ones).

Model of the Lagrangian Relaxation

We consider the following optimisation problem :

$$\begin{aligned} z = \min f(x) \\ Ax \leq b \\ x \in X \subseteq \mathbb{R}^n \end{aligned} \tag{P}$$

with $f : \mathbb{R}^n \to \mathbb{R}$, $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^{m \times 1}$.

Let's assume the constraint $Ax \leq b$ is tricky and the polyhedron X is easy to solve.

A naive relaxation would be to just drop the difficult constraint and solve the resulting problem. But as this totally ignores some constraints, this relaxation can be still weak compared to the initial problem.

The principle of the Lagrangian relaxation is to drop the difficult constraint (like a simple relaxation) but also to modify the objective by adding a term composed of the constraint dropped.

The goal of this term is to penalise the solutions of the relaxation violating the constraint and rewarding the ones satisfying it.

The Lagrangian relaxation of the problem (P) is :

$$\begin{aligned} z(\lambda) = &\min f(x) - \lambda(b - Ax) \\ &x \in X \subseteq \mathbb{R}^n \end{aligned} \tag{P(λ)}$$

with $\lambda \in \mathbb{R}_+^{1\times m}$.

The model slightly changes when the constraint is different. When the constraint removed is of the form $Ax \geq b$, the multipliers need to be negative ($\lambda \in \mathbb{R}_-^{1\times m}$). And when the constraint removed is of the form $Ax = b$, there are no constraint on the multipliers ($\lambda \in \mathbb{R}^{1\times m}$).

Is the Lagrangian relaxation always a lower bound?

A simple proof that $z(\lambda)^*$ (the cost of the optimal solution x_r^* of the relaxation) is a lower bound to z^* (the cost of the optimal solution x^* of the initial problem) can be done this way :

Knowing that x_r^* is the optimal solution of the relaxing problem, it can be said that $\forall x \in X$: $f(x_r^*) - \lambda(b - Ax_r^*) \leq f(x) - \lambda(b - Ax)$. As the polyhedron of the initial problem is fully contained in the one of the relaxed one ($\{x : x \in X \text{ and } Ax \leq b\} \subseteq \{x : x \in X\}$), this inequality is also right for all the solutions of the initial problem, including the optimal solution x^*. But as x^* is a solution of the initial problem, we can state that $-\lambda(b - Ax_r^*) \leq 0$. This concludes the proof, as :

$$\begin{aligned} z(\lambda)^* &= f(x_r^*) - \lambda(b - Ax_r^*) \\ &\leq f(x^*) - \lambda(b - Ax^*) && \text{by optimality of } x_r^* \\ &= z^* - \lambda(b - Ax^*) && \text{as } x^* \text{ is solution of the initial problem} \\ &\leq z^* \end{aligned}$$

Choice of the multipliers

Each multiplier respecting the multiplier constraint can be used, but some are better than others. The Lagrangian dual is the problem $z(\lambda^*) = \max_\lambda z(\lambda)$ which corresponds to finding the best multipliers maximising the lower bound. This can be done in an iterative way.

Appendix C

TSPlib

TSPlib [17] is a library regrouping multiple TSP instances from different sources. These instances are from different type (symmetric, asymmetric,...). It also inventory the best known solution so far for each instance. As the focus is made on the asymmetric TSP, only this type of instance from this library was used. This allows to test the implementations over bigger and more realistic instances.

The table Tab.C.1 describes the different instances and their characteristics (number of vertexes,...).

Name	Optimum	Number of vertice	Minimum cost	Maximum cost	Mean of the costs
br17	39	17	0	74	14.529411764705882
ft53	6905	53	21	1834	492.91835994194486
ft70	38673	70	331	2588	1030.3155279503105
ftv33	1286	34	7	332	128.451871657754
ftv35	1473	36	7	332	135.20714285714286
ftv38	1530	39	7	332	132.76788124156545
ftv44	1613	45	7	332	136.94343434343435
ftv47	1776	48	7	348	142.37677304964538
ftv55	1608	56	6	324	131.8168831168831
ftv64	1839	65	5	348	135.25913461538462
ftv70	1950	71	5	321	118.96519114688128
ftv90	1579	91	5	321	123.37838827838827
ftv100	1788	101	5	321	133.7749504950495
ftv110	1958	111	5	331	139.8997542997543
ftv120	2166	121	5	331	144.33243801652893
ftv130	2307	131	5	353	149.2601291837933
ftv140	2420	141	5	353	150.42264437689968
ftv150	2611	151	5	368	153.38366445916114
ftv160	2683	161	4	368	154.0144409937888
ftv170	2755	171	4	368	153.62751977984175
kro124p	36230	100	81	4545	1910.0977777777778
p43	5620	43	0	5160	593.6976744186046
rbg323	1326	323	0	33	19.190594773378457
rbg358	1163	358	0	33	19.329726303929395
rbg403	2465	403	0	33	19.080836512227943
rbg443	2720	443	0	33	18.64608847532762
ry48p	14422	48	54	2782	1138.7672872340424

Table C.1: Data concerning the TSPlib instances

Appendix D

Complementary results for Chap. 5

This Appendix contains some of the results of the statistical analysis over the RandB benchmark of the Chap.5. They all lead to the same conclusion as the analysis in the Chapter 5.

1Arbo vs 1Tree : statistical analysis

Fig.D.1 concerns the comparison of the statistical data's for the 1Arbo and the 1Tree over the RandB benchmark.

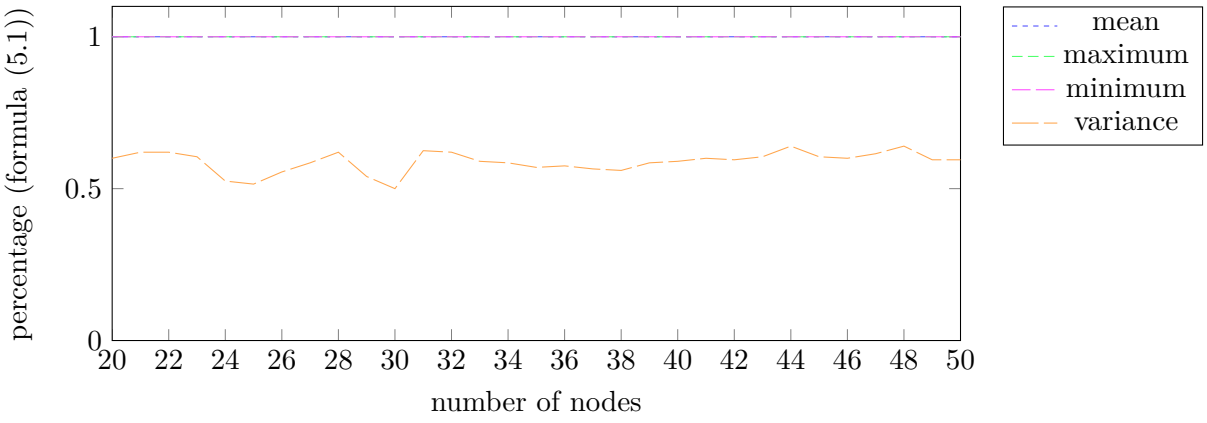
1Anti vs 1Tree : statistical analysis

Fig.D.2 concerns the comparison of the statistical data's for the 1Anti and the 1Tree over the RandB benchmark.

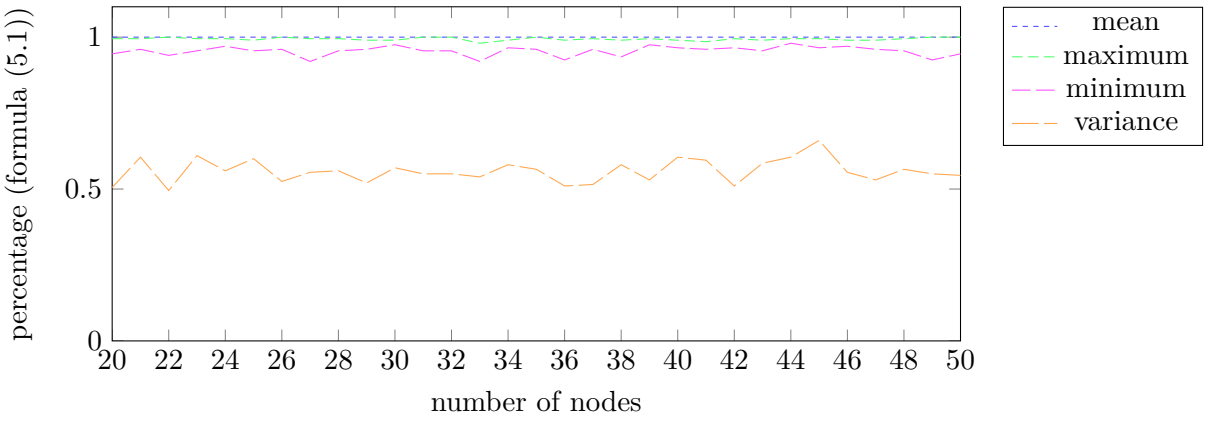
1Arbo vs 1Anti : statistical analysis

Fig.D.3 concerns the comparison of the statistical data's for the 1Arbo and the 1Anti over the RandB benchmark.

Time comparison

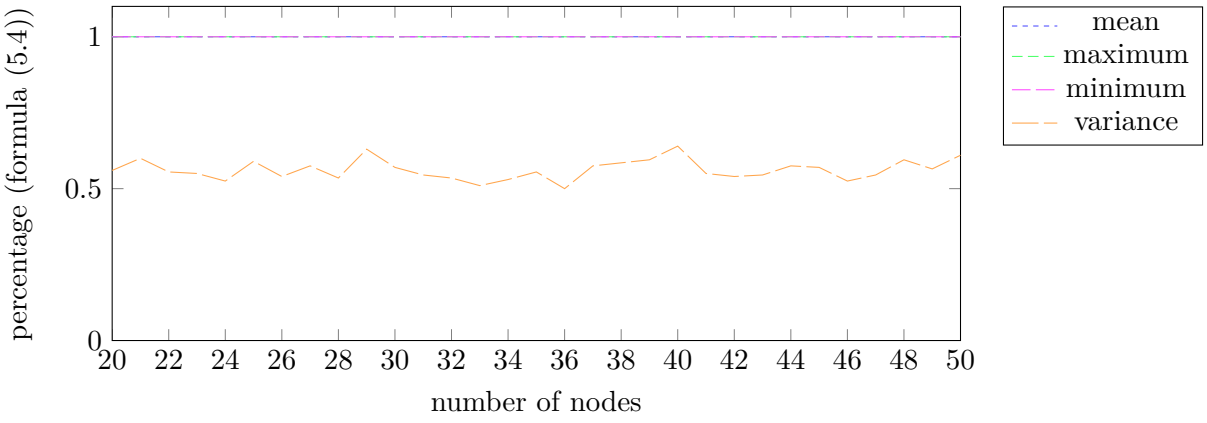


(a) Percentage of graphs of RandB where the statistical data for the minimal 1Arbo's is strictly better than the one for the minimal 1Tree's

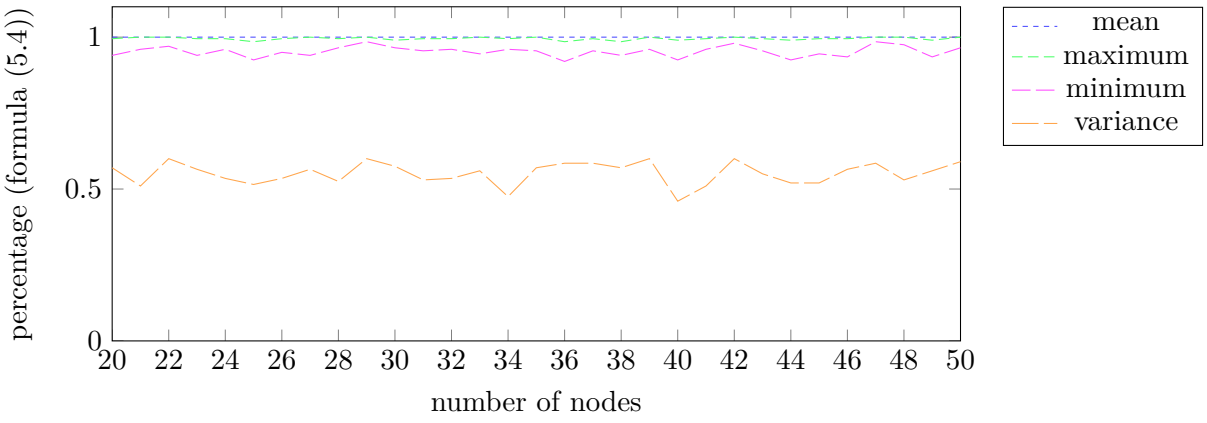


(b) Percentage of graphs of RandC where the statistical data for the minimal 1Arbo's is strictly better than the one for the minimal 1Tree's

Figure D.1: 1Arbo versus 1Tree for RandB

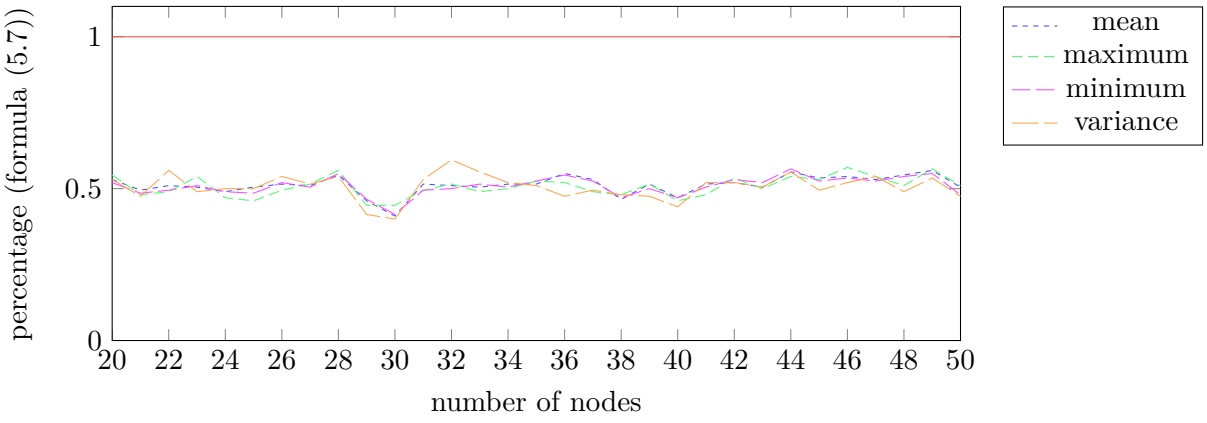


(a) Percentage of graphs of RandB where the statistical data for the minimal 1Anti's is strictly better than the one for the minimal 1Tree's

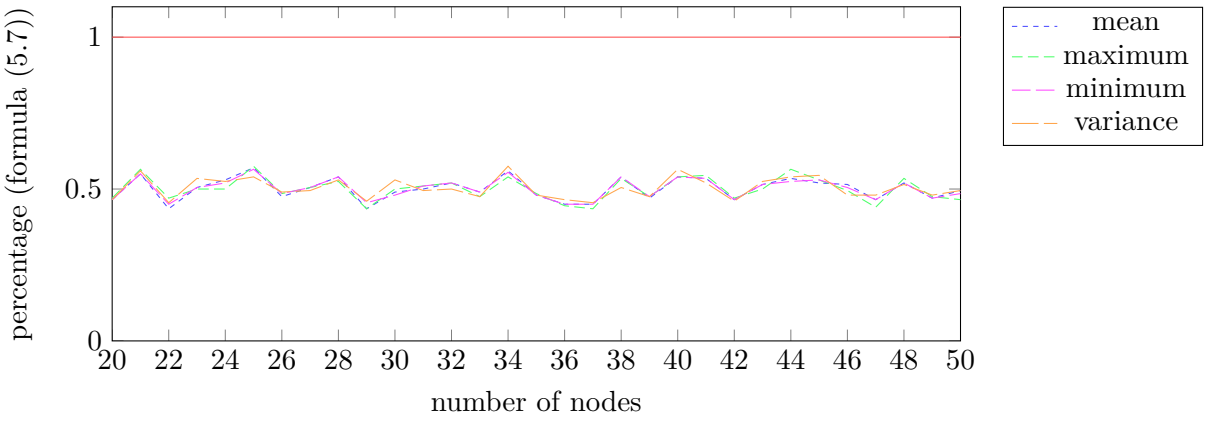


(b) Percentage of graphs of RandC where the statistical data for the minimal 1Anti's is strictly better than the one for the minimal 1Tree's

Figure D.2: 1Anti versus 1Tree for RandB

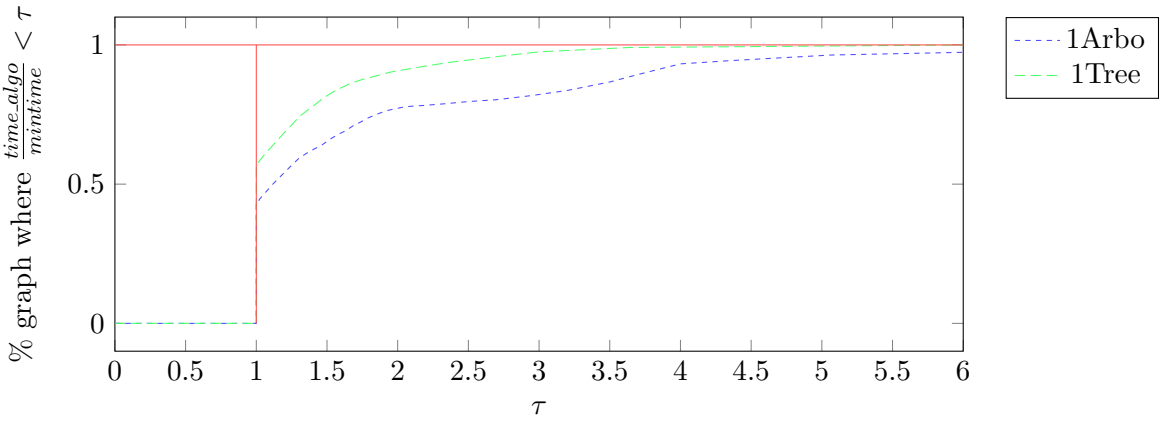


(a) Percentage of graphs of RandB where the statistical data for the minimal 1Arbo's is strictly better than the one for the minimal 1Anti's

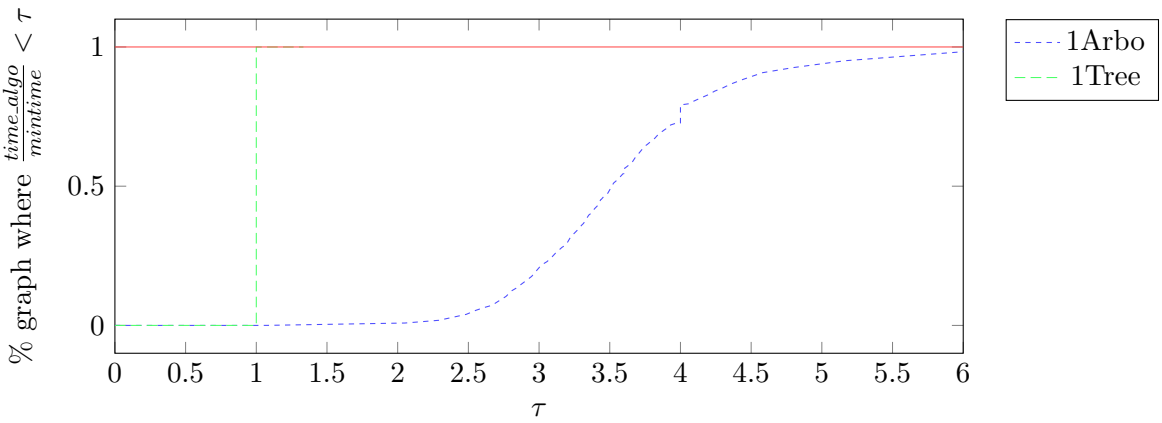


(b) Percentage of graphs of RandC where the statistical data for the minimal 1Arbo's is strictly better than the one for the minimal 1Anti's

Figure D.3: 1Arbo versus 1Anti for RandB

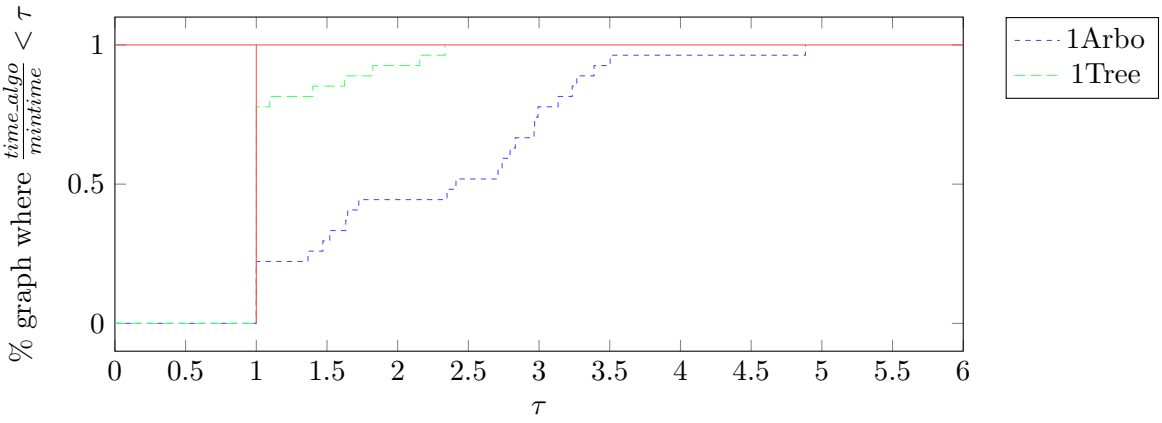


(a) Performance profile of the global time (creation + solving) for RandC

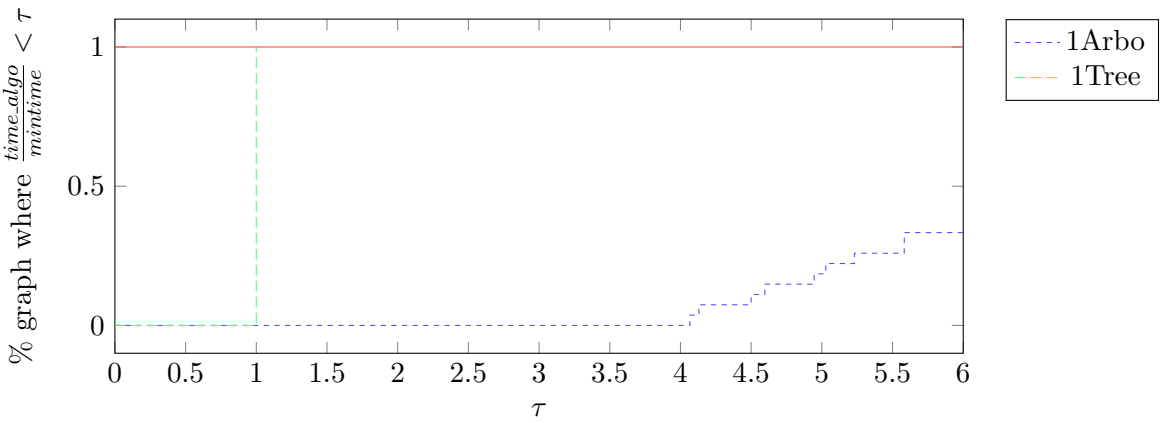


(b) Performance profile of the execution time only for RandC

Figure D.4: Performance profile graphs on RandB



(a) Performance profile of the global time (creation + solving) for TSPLib



(b) Performance profile of the execution time only for TSPLib

Figure D.5: Performance profile graphs on RandB

Appendix E

Complementary results for Chap. 6

This Appendix contains some of the results of the comparison between the different methods with a same step size (Section 6.3). Their results follow the analysis given during the chapter.

Simple step size

The Fig.E.1 gives the optimal performance profile for RandA and Fig.E.2, E.3 and E.4 give the time performance profile for RandA, RandB and TSPLIB

Geometric step size

The Fig.E.1 & 6.7 give the optimal performance profile for RandA and TSPLIB and Fig.E.7, E.8 & E.9 represent the time performance profile for the RandA, RandC and TSPLIB benchmarks.

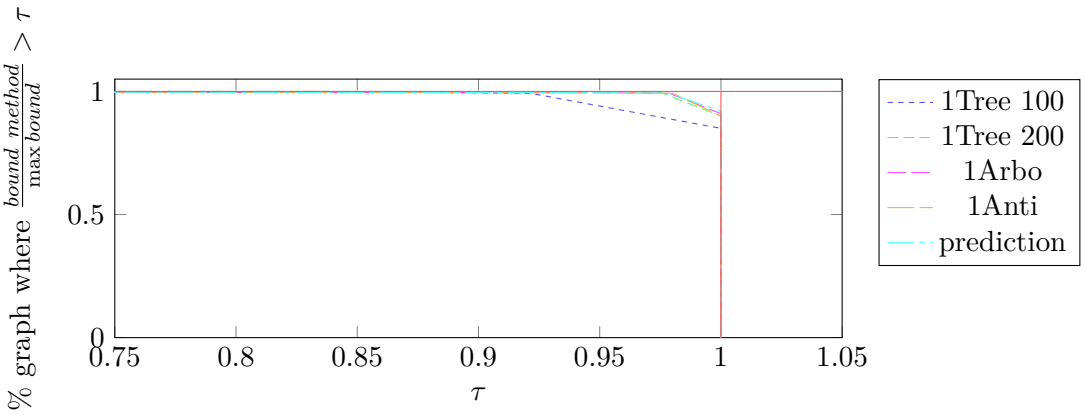


Figure E.1: Bound performance profile of the simple step for RandA

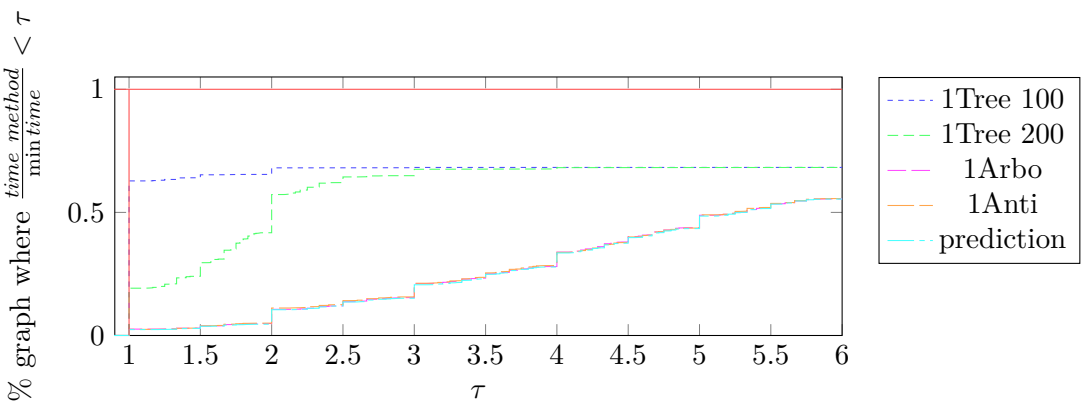


Figure E.2: Time performance profile of the simple step for RandA

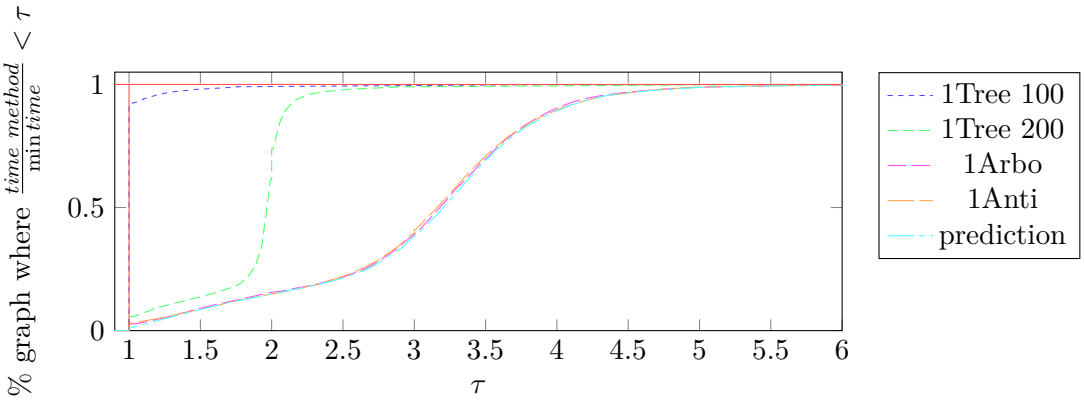


Figure E.3: Time performance profile of the simple step for RandB

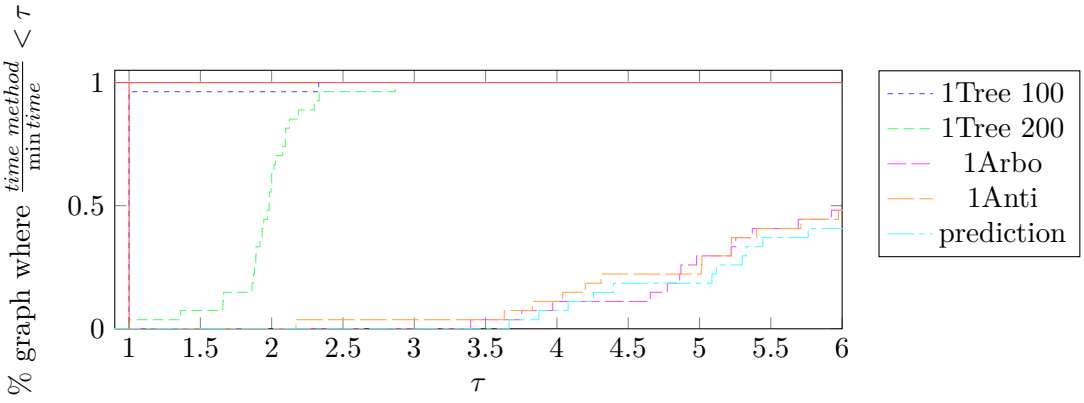


Figure E.4: Time performance profile of the simple step for TSPLib

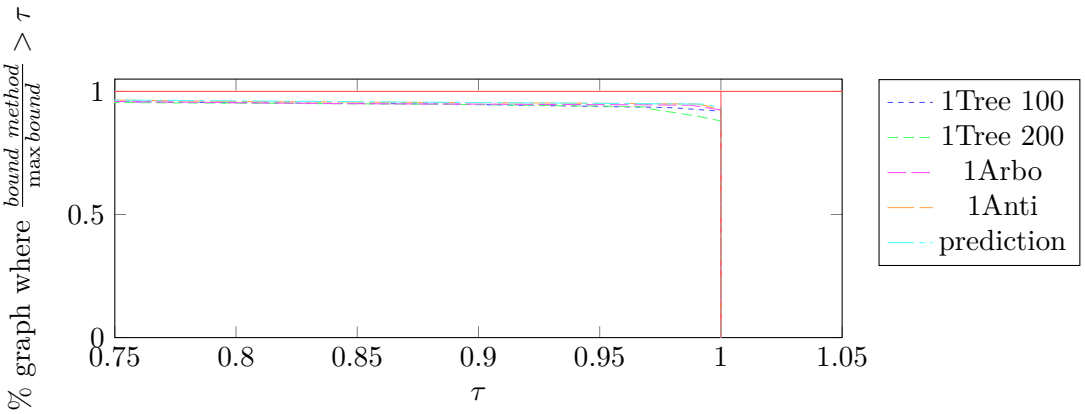


Figure E.5: Bound performance profile of the geometric step for RandA

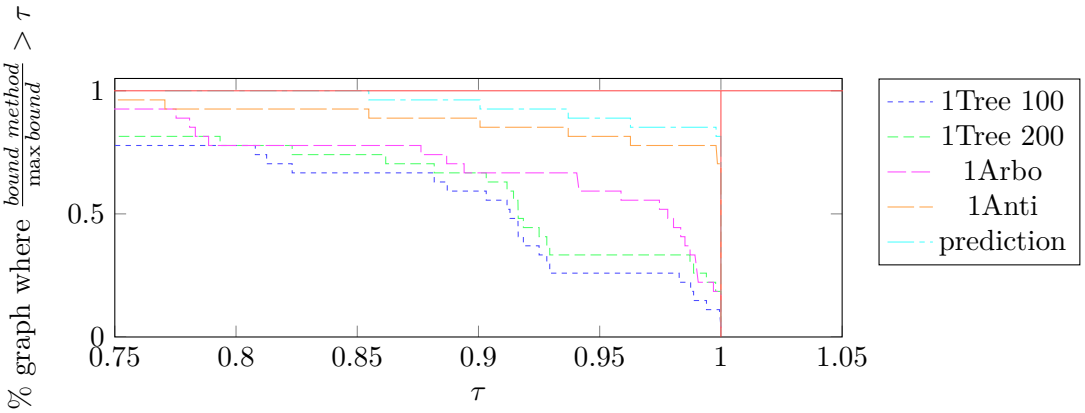


Figure E.6: Bound performance profile of the geometric step for TSPLib

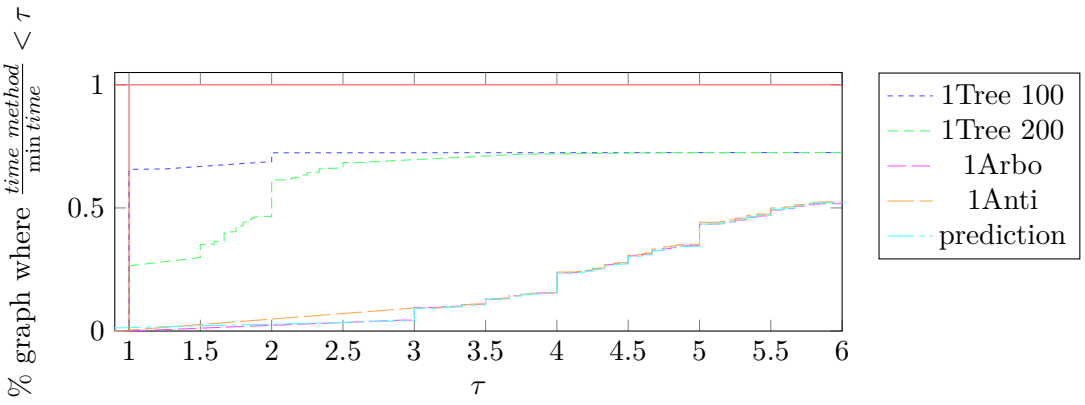


Figure E.7: Time performance profile of the geometric step for RandA

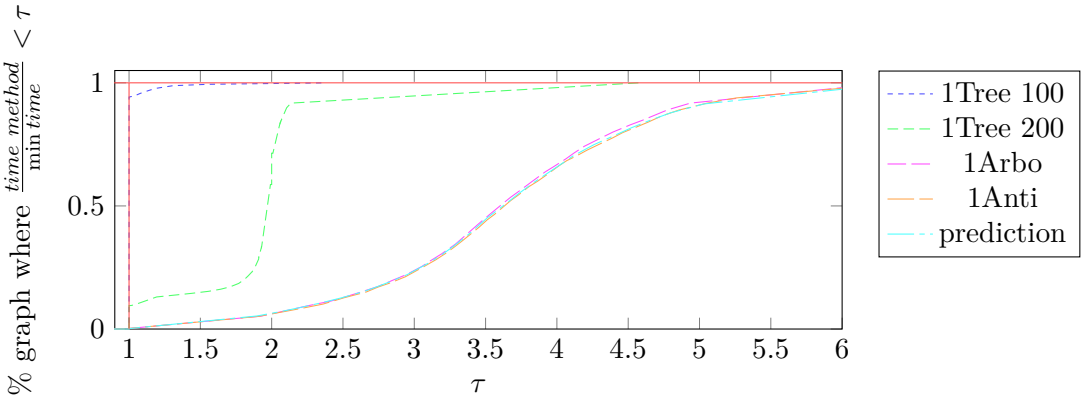


Figure E.8: Time performance profile of the geometric step for RandC

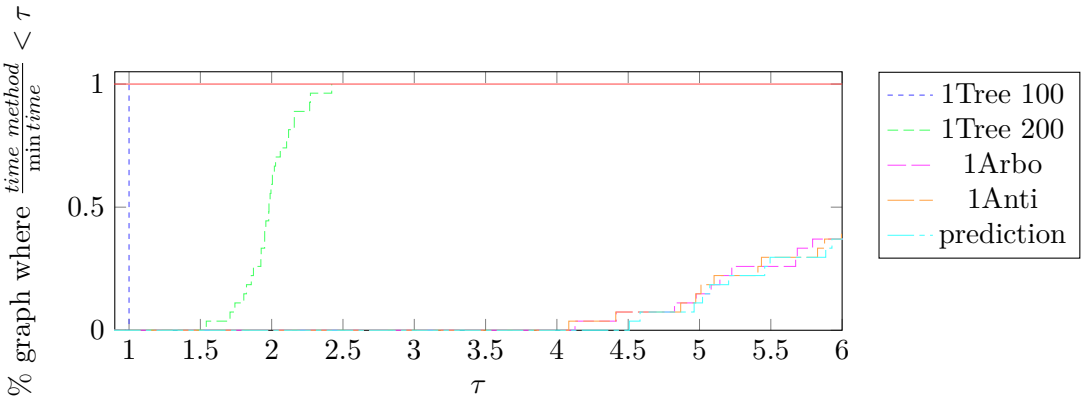


Figure E.9: Time performance profile of the geometric step for TSPLib

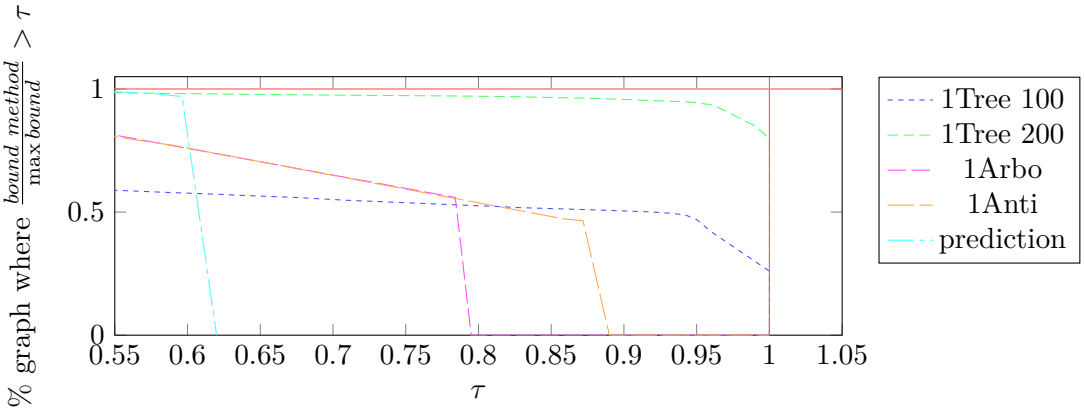


Figure E.10: Bound performance profile of the complex step for RandC

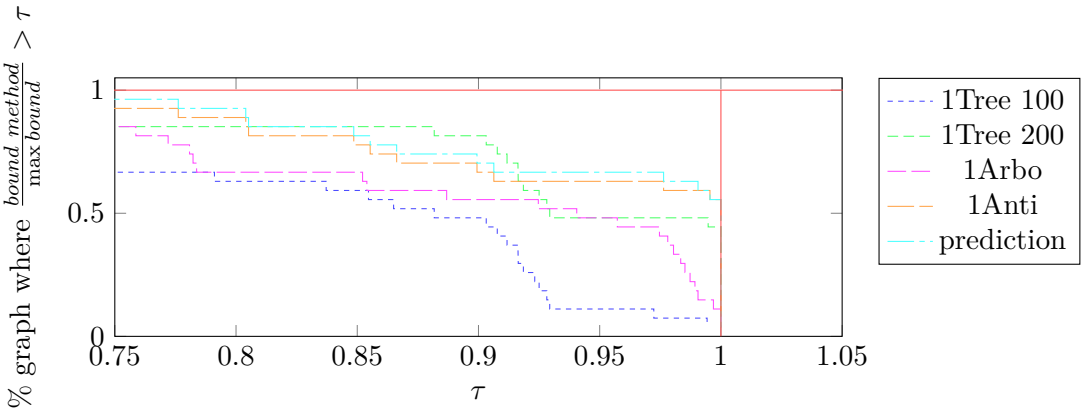


Figure E.11: Bound performance profile of the complex step for TSPlib

Complex step size

The Fig.E.10 & E.11 give the optimal performance profile for RandC and TSPLIB and Fig.E.12, E.13 & E.14 represent the time performance profile for the RandA, RandB and RandC benchmarks.

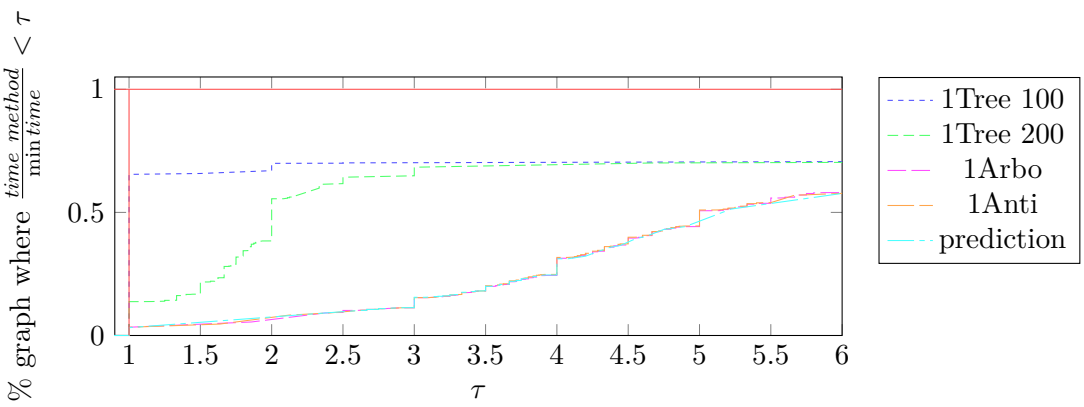


Figure E.12: Time performance profile of the complex step for RandA

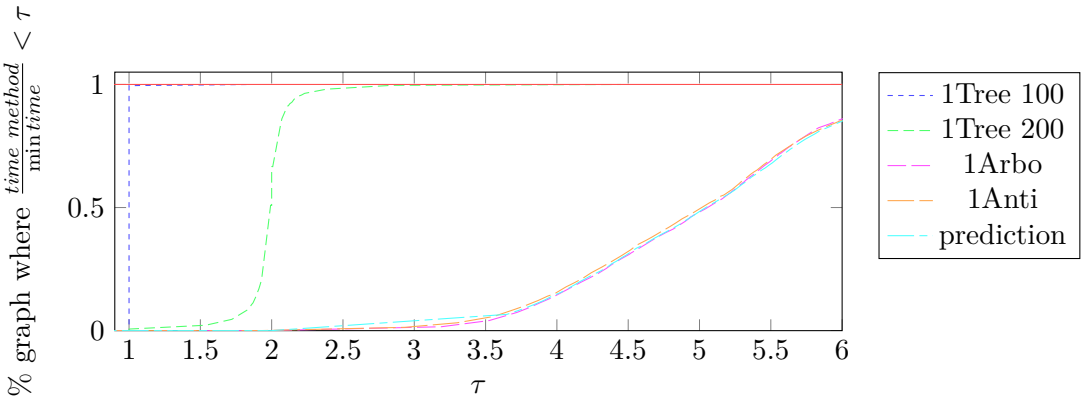


Figure E.13: Time performance profile of the complex step for RandB

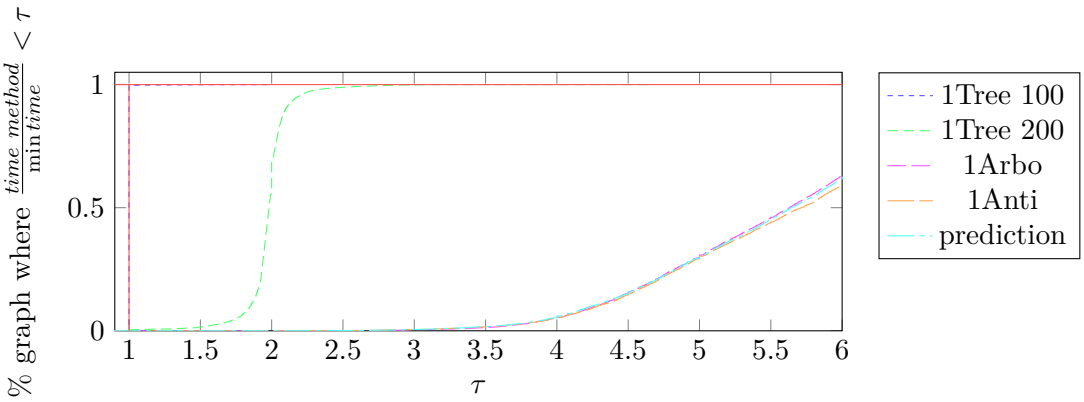


Figure E.14: Time performance profile of the complex step for RandC