An Integer Linear Programming Heuristic for the Travelling Salesman Problem

Candidato:
Paolo Marchezzolo

Relatore:
Chiar.mo Prof. Matteo Fischetti

11 Marzo 2013
A.A. 2012/2013
The Travelling Salesman Problem (TSP) is a well-known optimization problem that has many applications in a wide array of fields. It is well-known that the TSP is a NP-hard problem, thus heuristic approaches are fundamental to be able to obtain good solutions in a reasonable amount of time.

In this work, we explore a new heuristic approach to the TSP problem. We aim at improving an already existing solution by using a proximity or distance function, and iteratively looking for improvements in the “neighborhood” of the provided solution. We iteratively solve two subproblems, called master and slave; the first is an ILP relaxation of the TSP without the Subtour Elimination Constraints that tries to find the cheaper solution (compared to the existing one) which is closer to the solution of the previous iteration, while the second tries to heuristically enforce the missing constraints by finding the Shortest Spanning Tree that minimizes the Hamming Distance with the main subproblem.

The thesis, in addition to a formal presentation of the aforementioned concepts, also provides a computational analysis of the approach, tested over synthetically generated instances with various parameters settings.

The results of our testing show that our algorithm is consistently able to find the optimal value in a reasonable amount of time over our test instances. While more specialized algorithms are much faster than the implementation provided here, our approach still looks promising: it is more general, and can easily be adapted to other NP-hard problems that currently do not have good heuristics available; also, it is a new approach, so a higher degree of optimization and improvement compared to already established ones is predictable.
A.2 CPLEX 32
  A.2.1 Interactive Solver 32
  A.2.2 Callable Library 33

B SOURCE CODE 35
  B.1 Main program 35
     B.1.1 tandem.h 35
     B.1.2 tandem.c 35
  B.2 Kruskal implementation 50
     B.2.1 kruskal.h 50
     B.2.2 kruskal.c 51
  B.3 Instance generation 52
     B.3.1 script.sh 52
     B.3.2 converter.c 52
     B.3.3 merger.c 53

BIBLIOGRAPHY 55
LIST OF FIGURES

Figure 1  Example of swap of k edges for k = 2  6
Figure 2  A sample execution of our algorithm  20
Figure 3  Evolution of master and slave values on Instance 11, with CPLEX_PARAM_INTSOLLIM=1  26
Figure 4  Evolution of master and slave values on Instance 11, with the RINS approach active  26
Figure 5  Evolution of master and slave values on Instance 9, with CPLEX_PARAM_INTSOLLIM=1 and binary search active  27

LIST OF TABLES

Table 1  Test Instances  22
Table 2  Test Results, Instance 1 (300 nodes, 2243 edges)  22
Table 3  Test Results, Instance 2 (300 nodes, 4485 edges)  23
Table 4  Test Results, Instance 3 (300 nodes, 6728 edges)  23
Table 5  Test Results, Instance 4 (400 nodes, 3990 edges)  23
Table 6  Test Results, Instance 5 (400 nodes, 7980 edges)  24
Table 7  Test Results, Instance 6 (400 nodes, 11970 edges)  24
Table 8  Test Results, Instance 7 (500 nodes, 6238 edges)  24
Table 9  Test Results, Instance 8 (500 nodes, 12475 edges)  25
Table 10  Test Results, Instance 9 (500 nodes, 18713 edges)  25
Table 11  Test Results, Instance 10 (600 nodes, 8985 edges)  25
Table 12  Test Results, Instance 11 (600 nodes, 17970 edges)  26
INTRODUCTION

OVERVIEW

The Travelling Salesman Problem (TSP) is one of the most relevant problems in combinatorial optimization, and has been widely studied in the fields of operational research and computer science. Informally it is the task of finding the shortest route that, given a list of “cities” and their pairwise distances, visits all of them once and returns to the origin “city”. It has an extremely wide number of practical applications, ranging from planning, logistics and chip manufacturing to apparently unrelated ones, for example DNA sequencing.

Unfortunately, it is well-known that the TSP problem belongs to the class of NP-hard problems. This means that the time required to solve an arbitrary instance of the problem can, at worst, increase exponentially with the size of said instance, making an exact algorithm unpractical for many applications.

For this reasons, heuristic algorithms are extremely important to obtain “acceptable” solutions in a reasonable amount of time. A very large amount of works investigates the task of finding an almost-optimal solution in short computing time. Nowadays, we can find a good solution to instances with thousands, or even millions of “cities”, using modern heuristic methods.

Our work approaches the subject in a slightly different manner. As already said, finding a good solution to even large instances is often not too difficult; however, it could be interesting to try to improve it further. To rephrase the concept, we will investigate if it is possible to find effective algorithms that obtain a better solution by taking a “good” one as input. Obviously, we still aim at obtaining a heuristic method, since the NP-completeness of the problem still prevents us to reach an optimal solution in a reasonable amount of time on some instances.

The main idea behind our algorithm is that, given a good solution to a TSP instance, it is likely that other good solutions (possibly better ones) are not too far from the given one, using a suitable metric. Thus an effective method that iteratively tries to lower the solution cost while minimizing the distance from the incumbent could be effective in finding an improvement. This kind of heuristic would take advantage of the information contained in the given solution, providing a practical tool to use in combination with other heuristic methods.
CONTENT STRUCTURE

THE FIRST CHAPTER offers a brief introduction to the Travelling Salesman Problem, its Integer Linear Programming formulation, relaxations of the model, and some exact and heuristic algorithms.

THE SECOND CHAPTER introduces the particular approach to the Travelling Salesman Problem which is addressed in our work.

THE THIRD CHAPTER presents in detail the implementation of the main topics of Chapter 2.

THE FOURTH CHAPTER reports the main experimental results found while testing the code.

THE FIFTH CHAPTER analyzes the outcome of the testing phase and outlines possible future improvements of the ideas developed in our work.
THE TRAVELLING SALESMAN PROBLEM

This chapter provides a brief introduction to the Travelling Salesman Problem, along with definitions and concepts that will be used in following chapters. We formally define the TSP, give its ILP model and some relaxations of it, and also illustrate some heuristic algorithms that can either create or improve a tour for a given TSP instance.

1.1 PROBLEM DEFINITION

An informal definition of the Travelling Salesman Problem was already sketched in the introduction. A more precise one follows.

**Definition 1.** Given a weighted graph \( G = (V, E) \) the Travelling Salesman Problem requires to find a cycle such that:

1. every vertex \( v \in V \) is visited exactly once;
2. the weight (or cost) of the cycle is minimum.

If the graph is undirected, the problem is called Symmetric Travelling Salesman Problem (STSP). If the graph is directed, the problem is called Asymmetric Travelling Salesman Problem (ATSP). The focus of this work is on the STSP variant of the problem. In the following, we will refer to the undirected version when generically talking of the TSP problem unless otherwise stated.

1.2 INTEGER LINEAR PROGRAMMING MODEL

The Travelling Salesman Problem is modeled in an elegant way as an Integer Linear Programming problem. Since such approach will be widely used in the rest of this work, a brief introduction to Linear Programming and Integer Linear Programming is given.

1.2.1 Linear Programming

Linear Programming (LP) is a framework used to optimize a linear objective function subject to linear equality or inequality constraints. A LP problem is usually stated as follows:

\[
\text{max } c^T x \\
Ax \geq b \\
x \geq 0
\]

It is known that Linear Programming belongs to the P class, thus there exists an algorithm that solves every instance of a LP problem in polynomial
time. Also, there exist practically efficient algorithms to solve these kinds of problems, like the simplex algorithm (which, despite not having a polynomial worst-case complexity, is often preferred thanks to many desirable properties, for example the relative ease of adding new constraints to an already solved problem).

Unfortunately, the LP is not powerful enough to effectively model the TSP\(^1\). Representing a problem like the TSP requires some variables of the LP to be integer, which is possible in an Integer Linear Programming problem.

### 1.2.2 Integer Linear Programming

An Integer Linear Programming problem looks exactly like a LP problem, except that it allows to express integrality constraints over some variables of the model.

\[
\begin{align*}
\text{max } & c^T x \\
\text{subject to } &Ax \geq b \\
&x \geq 0 \quad \text{integer.}
\end{align*}
\]

Adding integrality constraints to the model allows one to represent a much broader class of problems than it was previously possible with Linear Programming. In many cases, it is useful to force some variables to assume a value of either 0 or 1. Those variables are called binary variables, and are often used when the model needs to make a “choice”; for example, as it will be explained soon, in a TSP model, an edge is chosen in a solution only if the corresponding variable has a value of 1, and discarded otherwise.

However, an ILP model is, in its general case, NP-hard, which implies the lack of efficient, exact algorithms to solve a problems stated in that form. For this reason, heuristics for general ILP models are often the only way to approach problems modeled as ILP for practical applications.

### 1.2.3 An ILP model for the Travelling Salesman Problem

As already mentioned, the most natural way to represent mathematically the TSP is using an ILP model. Using the notation intruduced in Definition 1 and assuming that edge \(e \in E\) has a weight of \(w_e\), we obtain the following:

\[
\begin{align*}
\text{min } & \sum_{e \in E} w_e x_e \\
\sum_{e \in \delta(i)} x_e &= 2 & i & \in V \\
\sum_{e \in \delta(S)} x_e &\geq 2 & S & \subset V, 2 \leq |S| \leq |V| - 2 \\
x_e &\in \{0, 1\} & e & \in E
\end{align*}
\]

In the model, each variable \(x_e\) is binary, as stated by the constraints (4), and each edge \(e\) is selected if and only if the corresponding variable \(x_e\) is 1 in the solution of the ILP. The set of constraints (2)\(^2\) ensures that each node

\(^1\) If it could, it would imply that the conjecture \(P = NP\) actually holds, while the opposite is widely thought.

\(^2\) Notation \(\delta(i)\) represents all the edges incident at \(i\), and by extension \(\delta(S)\) with \(S \in V\) is the set of all edges with one vertex in \(S\) and the other one outside \(S\).
has exactly two incident edges, and the constraints (3), called Subtour Elimination Constraints (SECs), are necessary to avoid subtours in the solution.

Constraints (3) can also be formulated in an alternative way: instead of requiring each subset of nodes $S \in V$ to have at least two edges in their $\delta(S)$, it is also possible to force all the selected edges with both ends inside $S$ to be less or equal to $|S| - 1$:

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \quad S \subset V, 3 \leq |S| \leq |V| - 2$$

(5)

It is possible to show that the two forms (3) and (5) are equivalent to enforce the absence of subtours in the final solution. Both of them, however, include a number of constraints that increases exponentially with the size of the problem. So, not only solving the ILP model associated with a TSP problem is a NP-hard problem itself, as described in Subsection 1.2.2, but the size of the problem itself is also exponential. This makes impossible to utilize this model as-is for practical applications.

1.3 RELAXATIONS OF THE ILP MODEL

Since, as already said in Subsection 1.2.3, trying to directly solve the ILP formulation for the Travelling Salesman Problem is not a feasible approach, it is possible to relax some constraints of the formulation to obtain a solvable model. Doing so may destroy the feasibility of the solution found, but allows us to obtain a lower bound of it in a reasonable amount of time.

Formally, a relaxation of a minimization problem (as the TSP is) is defined as follows:

**Definition 2.** Assume that our problem, $P$, is a minimization problem:

$$z = \min f(x), \ x \in F(P).$$

Then a new problem $R$ is defined as follows:

$$z_R = \min \Phi(x), \ x \in F(R).$$

The problem $R$ is called a relaxation of the problem $P$ if the following conditions hold:

(a) $F(P) \subseteq F(R)$

(b) $\Phi(x) \leq f(x) \ \forall x \in F(P)$

Two relaxations are mainly useful for this work, so a brief explanation of them is provided.

1.3.1 Relaxation by SECs elimination

The first, and maybe most obvious idea to “simplify” the model is to just remove the SECs. After doing so, the problem becomes both theoretically and practically easy.

This relaxation for the TSP model reads:

The set of all the edges with this property is written as $E(S)$.
\[
\min \sum_{e \in E} w_e x_e \\
\sum_{e \in \partial(i)} x_e = 2 \quad i \in V \\
x_e \in \{0, 1\} \quad e \in E
\]

(6) (7) (8)

It is also possible to obtain an exact algorithm to solve the TSP from this relaxation by simply checking if each node is reachable from a fixed node. If the answer is positive, we have a solution of the original TSP; if not, the SECs corresponding to the subtours found in the current solution are added to the model, that is solved again. Iterating this process, however, can at worst generate every SEC present in the original TSP formulation, severely diminishing the approach’s usefulness.

1.3.2 1-tree relaxation

Another way to relax the TSP is to notice that, if the two edges incident on some node (say, node 1) are removed from the solution, the remaining edges form a spanning tree on the subgraph induced by \( V \setminus \{1\} \). Thus, any solution of the TSP problem has the following structure:

(a) every node \( v \neq 1 \) has degree 2;
(b) the node 1 has two incident edges;
(c) removing the node 1 from the graph (and the two associated edges) we obtain a tree on the subgraph induced by \( V \setminus \{1\} \).

Removing (a), we obtain a relaxed problem whose solution is:

1. find the Shortest Spanning Tree (SST) on the subgraph obtained removing the node 1;
2. add to the solution the cheapest two edges incident at node 1.

A solution constructed as previously described is called a 1-tree. An ILP model to solve the SST problem involved can be obtained easily by the ILP model illustrated in Subsection 1.2.3.

\[
\min \sum_{e \in E} w_e x_e \\
\sum_{e \in \partial(1)} x_e = 2 \\
\sum_{e \in E} x_e = n \\
\sum_{e \in \partial(S)} x_e \geq 1 \quad S \subset V, 1 \leq |S| \leq |V| - 2 \\
x_e \in \{0, 1\} \quad e \in E
\]

(9) (10) (11) (12) (13)

The constraint (11) forces the solution to have exactly \( n \) selected edges, while constraint (10) ensures that two edges incident on node 1 are selected. Since
the connection is guaranteed from the set of constraint (12), this ILP formulation is indeed a model of the 1-tree problem. However it is preferrable to not solve the 1-tree problem with an ILP solver, since the Shortest Spanning Tree (SST) problem is solvable in polynomial time with simple greedy algorithms, like Kruskal’s one. The 1-tree relaxation can be easily obtained after having solved the SST over the graph induced by \( V \setminus \{1\} \) by just adding the two cheapest edges incident on node 1.

1.4 SOME HEURISTIC APPROACHES

This section gives a brief outline of some simple heuristic approaches to the Travelling Salesman Problem.

1.4.1 Tour Construction Procedures

These heuristics work iteratively on the TSP instance they are processing. To build the heuristic solution, they define three rules, that control the choice of the starting node (or the starting subtour, depending on the method), the selection of the next node to be added to the solution, and the pair of nodes what will be the immediate predecessor and successor of the new node. Different choices for these rules generate different heuristics. We list three examples, that work better on average.

1. The **Nearest Neighbour Algorithm** starts from a random node on the graph, then it selects the node that has the lowest distance from the last picked node, and it adds it to the solution. The algorithm has a time complexity which is \( O(n^2) \), where \( n \) is the number of nodes of the graph.

2. The **Cheapest Insertion Algorithm** chooses a random node at first, and its initial subtour is composed of the most “expensive” edge incident from the first node. Then the algorithm iteratively chooses the node that minimizes the insertion cost of any unselected node between any pair of already selected nodes. The insertion cost of adding node \( k \) between node \( i \) and node \( j \) is simply calculated as the difference between the cost of the two edges \( (i, k) \) and \( (k, j) \) and the cost of the edge previously part of the subtour \( (i, j) \). The algorithm has a time complexity which is \( O(n^2 \log n) \) for with graph of \( n \) nodes.

3. The **Multi-Path Algorithm** is somewhat similar to Kruskal’s algorithm to determine the Sortest Spanning Tree on a graph. It first sorts the edge set in nondecreasing order, then it starts examining the edges in the sorted list. If an edge does not form a subtour with the already selected ones and one of its vertices does have less than two incident edges, then it is added to the solution. Considering a graph of \( n \) nodes, the algorithm terminates after selecting \( n \) edges, and its time complexity is \( O(n^2 \log n) \).

These heuristic procedures will not be used in this work, however they can be extremely valuable. They allow, in fact, to obtain a good solution in a very short amount of time; that solution can be then fed to our algorithm, which will try to improve it.

---

4 See Section 2.4 for additional details.
1.4.2 Tour Improvement Procedures

These procedures start from an already existing solution and try to improve it. The most natural idea to locally search for improvements when given a TSP solution is to swap out some edges and substitute them with some others. More formally, if we consider a generic solution \( s \) described with its successor vector \( \sigma(i), i \in V \), we can remove \( k \) edges from \( s \) and add another \( k \) edges not previously in \( s \). If \( k = 2 \), a possible example is shown in Figure 1; the edges \((i, \sigma(i))\) and \((j, \sigma(j))\) are removed and \((i, j)\) and \((\sigma(i), \sigma(j))\) are added. The variation in the solution cost can be easily evaluated:

\[
\Delta C = -c_{i, \sigma(i)} - c_{j, \sigma(j)} + c_{i, j} + c_{\sigma(i), \sigma(j)}
\]

The number of edges that have to be considered this way is \( O(n^k) \).

This strategy, apparently very simple, has been implemented with great success by Lin & Kernighan, and is useful in many applications.

Figure 1: Example of swap of \( k \) edges for \( k = 2 \).
This chapter presents the main ideas behind our approach: we define our heuristic method and discuss its main implementative challenges.

2.1 THE BASIC IDEA

In this work, we aim at constructing a new algorithm that can improve an existing solution of a TSP instance. While similar procedures already exist, like the one described in Subsection 1.4.2, the problem will be tackled by a different perspective, as explained below.

The main idea behind our approach is that, given a good solution of the instance, we can find a better solution by searching the “neighborhood” of it. Hopefully, a good solution already contains a portion of the optimal solution, and needs only a few changes in the set of selected edges to be improved. We can then iteratively exploit such a procedure to lower the cost as much as possible, or until we reach a target cost, or we run out of execution time.

To do so, we need to define the distance between different solutions to a given TSP instance. After doing so, we will define a subproblem, called master, using a particular ILP model that solves a relaxed version of the TSP formulation of Subsection 1.2.3 with some added constraints and a modified objective function to take into account the distance between the two solutions. Since the result is likely to violate one or more of the relaxed constraints, we will then use another procedure for the slave subproblem, which will try to enforce the constraints without going “too far” from the solution of the master. Iteratively solving the master and the slave should produce a succession of solutions which is likely to converge to a both improved and feasible solution, if not optimal.

The approach is somewhat inspired by previous works; in particular, [5] uses a similar approach while looking for feasible solutions instead of optimal ones.

2.2 DISTANCE FUNCTION

A solution of a TSP instance is easily represented as the set of values the binary variables in its formulations assume. For this reason, the most natural way to define a distance function between two different solutions is to use a Hamming-like distance, which is basically equivalent to taking the binary XOR of the two arrays and then summing over all the components.

More formally, given a TSP instance over the graph $G = (V, E)$, with $|E| = n$, and two solutions $x = (x_1, x_2, \ldots, x_n)$ and $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$, the distance between them will be defined as follows in the rest of this work:

$$
\Delta(x, \bar{x}) = \sum_{i=1}^{n} |x_i - \bar{x}_i|
$$

(14)
2.3 THE MASTER SUBPROBLEM

As already mentioned, we construct the Master subproblem as an ILP problem that is based on a relaxation of the TSP general ILP model previously presented. In particular, we will relax the subtour elimination constraints, to obtain a model that can be actually handled by commercial solvers. Then, instead of asking for the cheapest solution using the original edge costs, we want to retrieve the closest solution to the given one (which we will refer as $\tilde{x}$) that also costs less than a “target” parameter, $T$, provided on input to the algorithm.

The ILP model follows.

$$\min \Delta(x, \tilde{x})$$

$$\sum_{e \in \delta(i)} x_e = 2 \quad i \in V$$

$$\sum_{e \in E} w_e x_e \leq T$$

$$x_e \in \{0, 1\} \quad e \in E$$

In other words, we avoid requiring the model to find us the largest cost improvement over the given solution. Instead we require to get the closest one which also happens to fulfill a provided cost requirement. This is done via a modified objective function (15) which completely ignores the weights $w$ and just minimizes the distance function defined in Section 2.2. The additional constraint (17) ensures that any feasible solution of the model satisfies our demand on the cost. The choice of $T$ will be discussed in more detail later.

The solution produced by this ILP is not feasible for the original TSP instance, in general. While being close to an another feasible solution is more likely to fulfill all the SECs even if they are not included in the model, this is not guaranteed, and the solution can contain subtours that prevent it to be feasible. For this reason, we need another procedure that heuristically tries to enforce those constraints without adding them to the ILP formulation.

2.4 THE SLAVE SUBPROBLEM

The task that the slave subproblem performs is to try to remove any sub-tour present in the output of the master. It does not need to necessarily produce a feasible solution for the TSP instance we are processing, it just needs to select a set of edges that makes very unlikely to find subtours (or at least the same subtours that were already present) in the next iteration of the master subproblem.

While there are multiple ways to do so, a simple yet effective method to obtain this kind of effect in our specific application is to find a 1-tree over the given graph, modifying each edge weight to properly mirror its presence on the master output. The fact that a tree must be connected by definition should have the desired effect, forcing the outcome to break each subtour to be connected with the remaining subgraph. Modifying the original weights is also crucial since we do not want to completely destroy the informations contained in the solution of the master.
That being said, the slave subproblem is defined as finding a $1$-tree over the graph underlying the TSP instance we are processing, where each edge $e$ has the following weight:

\[
\hat{w}_e = M(1 - x^*_e) - Mx^*_e + w_e \\
= M(1 - 2x^*_e) + w_e
\]

where $x^*$ is the 0-1 solution returned by the current master subproblem, and $M$ is a constant such that:

\[ M \gg w_e \quad \forall e \in E. \]

This redefinition of the edges weights has the following meaning: always prefer an edge already selected by the solution produced by the master subproblem; if two edges are both selected (or not selected) in the solution of the master, then prefer the one that is cheaper in the original TSP instance. Defining the weights in this way makes “cheaper” for the slave to produce a $1$-tree which contains the greatest possible amount of edges already present beforehand, and thus is as close as possible to the master output while ensuring that all the nodes are reachable one from each other.

To actually solve the SST subproblem, we use a slightly modified version of the Kruskal’s algorithm. This choice is not immediately evident while looking at the subproblem formulation, but becomes very clear to a closer analysis. The slave subproblem repeatedly solves a SST over the same graph, and while the weights indeed depend on the $x^*$ vector produced by the master, the possible values that $\hat{w}_e$ can assume are only two: $M + w_e$ or $-M + w_e$. This is an obvious consequence of the fact that $x^*$ is a binary vector. Since the Kruskal’s algorithm needs to sort the weights each time it runs, and then greedily sweeps the ordered array selecting the best edges that do not violate the tree structure, it would be very valuable to avoid repeating this operation for each iteration of the algorithm. This is possible by sorting the edges by nonincreasing weight $w_e$ first, then at each iteration, create a copy of the sorted array and re-sort it, this time by nondecreasing value of $x^*_e$; this operation can be implemented in linear time quite easily, as $x^*_e \in \{0, 1\}$. This simple trick provides us a list which is sorted exactly as it would be by utilizing the $\hat{w}_e$, but it does not need to compute them, and it also avoids to perform multiple sortings. The computational cost of performing the Kruskal’s algorithm at each iteration becomes then $O(n)$, where $n$ is the cardinality of the edge set $E$, and an initial overhead of complexity $O(n \log n)$ is performed only once at the start-up of our procedure.

### 2.5 A First Version of the Algorithm

In this section we will combine the master and slave subprocedures to obtain a first version of the algorithm we aim at constructing. In addition to the TSP instance and a solution of it that we want to improve, on input we require a target $T$, which is the value we want to reach.

Here is a first pseudocode for our algorithm. The subproblems, master and slave, are described in the previous two sections 2.3 and 2.4; we combine them together as illustrated in Algorithm 1.

---

1 This also removes the problem of choosing $M$, that could be nontrivial if the application is not known beforehand.
**Data:** A graph \( G = (V, E) \),
the weights \( w_e \ \forall e \in E \),
\( \tilde{x} \) solution of the TSP induced by \( G \) and \( w \),
a target value \( T \)

**Result:** An improved solution \( \hat{x} \)
initialize \( \hat{x} \) to \( \tilde{x} \);

repeat
\[
x^* = \text{solve the master with objective function } \Delta(x, \tilde{x});
\]
\[
\hat{x} = \text{solve the slave with weights defined as in (19)};
\]

until \( \sum_{e \in E} w_e \hat{x}_e \leq T \) and \( \hat{x} \) is a feasible solution of the TSP;
return \( \hat{x} \)

Algorithm 1

So, in this first pseudocode, at each iteration we solve the master to find a solution better than the target \( T \) given in input, and then solve the slave, that will return a connected set of edges. We hope that the sequence of solutions found this way can eventually converge to a feasible solution of the initial TSP instance which is cheaper than \( T \).

However, early analysis and testing of this approach allowed us to realize that there are two issues that prevent the algorithm to perform efficiently (and even correctly) if using the implementation given.

1. **Stalling:** It is possible for the algorithm to loop indefinitely on a pair of solutions that solve respectively the master and the slave subproblem. In fact, there is not mechanism that prevents the master from returning multiple times the same solution. This can be a problem, especially if we ask to reach a target \( T \) which is lower than the optimal TSP solution for that instance: the solution \( x^* \) of the master will contain subtours every time (if the subproblem is feasible), and we have no way to detect such a situation.

2. If \( T \) is much lower than the cost of the provided solution \( \tilde{x} \), it is unreasonable to ask the master to return us a vastly improved solution in one iteration; this basically destroys the meaning of our objective function \( \Delta(x, \tilde{x}) \): in fact, the master will be likely forced to choose a solution not that close to \( \tilde{x} \) to fulfill the cost constraint, thus losing the benefits of searching locally for improvements.

3. It is unreasonable to ask a target \( T \) to be reached as a parameter for the algorithm. The user will probably be interested in the best solution we can find within a predetermined amount of time. It would be desirable for the algorithm to not require such a parameter, and then just terminate if it can reach the optimal solution (and prove that is indeed optimal) or if it runs out of time.

In the following sections we discuss ideas to prevent stalling and to preserve locality. We will deal with points 2 and 3 first, since the solution to the first one partially depends on the others.

### 2.6 Step-by-Step Improvement

As already noticed in the previous section, it is not reasonable to ask our ILP formulation to immediately find a solution not greater than \( T \),
which is likely to be very low compared to the cost of our initial solution \( \bar{x} \). Furthermore, requiring the user to provide \( T \) is itself not desirable, so we should find a way to deal with this parameter internally.

To solve this issue, we implement a simple policy that handles the target \( T \) internally. The idea is to lower the target by a tiny bit each time we find an improvement over the current best solution, and to let it unchanged if we do not find any in the current iteration. A first implementation of this idea does the following:

1. Initialize \( T \) to \( (\sum_{e \in E} w_e \bar{x}_e) - 1 \);

2. If the master or the slave finds a solution \( \bar{x} \) that is feasible for the TSP instance we are considering, and is cheaper than the best solution encountered until now, update \( T \) to \( (\sum_{e \in E} w_e \bar{x}_e) - 1 \).

This approach completely removes the need for the user to provide \( T \) externally, and also allows the ILP model to improve more gradually, allowing to find solutions effectively close to the previous one. However, the drawback is that the improvement proceeds slowly, since we only require a minimal improvement for each iteration.

To capture the best properties of both methods, we actually implemented a different update strategy for \( T \), that slightly resembles the one we just presented, but it tries to be a little more aggressive and lower the cost of the solution more steadily if possible. This strategy is explained below:

1. Initialize a variable \( m \) to \( \frac{1}{10} \) that represents the minimum relative improvement that a solution needs to achieve in order to be considered effectively an “improvement”.

2. Initialize \( T \) to \( (1 - m) \sum_{e \in E} w_e \bar{x}_e \);

3. If the master or the slave finds a solution \( \bar{x} \) that is feasible for the TSP instance we are considering, and is cheaper than the best solution encountered so far, update the incumbent \( \bar{x} \) and redefine \( T \) to \( (1 - m) \sum_{e \in E} w_e \bar{x}_e \);

4. If the master becomes infeasible at a certain iteration (i.e. we lowered too much \( T \)), revert \( T \) to the last one that did not cause the master to be infeasible (call it \( T_p \)), then cut \( m \) in half and update \( T \) again as \( (1 - m) \sum_{e \in E} w_e \bar{x}_e \);

5. If \( m \sum_{e \in E} w_e \bar{x}_e \) becomes smaller than one at any given iteration, fall back to the previously proposed improvement strategy.

The rationale behind this approach is that we can ask for a fairly large improvement to our MIP model at first, since the initial tour is likely to be far from the optimal solution, then we start lowering the minimum improvement required as soon as we detect infeasible subproblems. When the first infeasible subproblem is detected, the update strategy for \( T \) turns into a kind of binary search, that will lower \( m \) as much as it is needed to regain feasibility of the master subproblem. This should, as hinted previously, allow us to retain the advantages of both strategies, since we are progressing faster than just lowering the solution by 1 every time.

However, we still are not guaranteed to obtain an unfeasible master subproblem if \( T \) is lower than the optimal solution. While such a solution is possible, the most likely scenario is a loop between an unfeasible and
cheaper-than-T master solution, and a feasible, but more expensive than T slave solution. Without a stalling prevention mechanism, both strategies cannot detect when they have to stop, and the second one is likely to fail completely.

2.7 STALLING

The core issue with stalling is that nothing prevents the master to produce the same solution twice. This situation is especially likely to show up if our target value $T$ is close to the optimal solution to our TSP instance. While obtaining an unfeasible master is possible, the more likely scenario is a loop as described in the previous section. A too low $T$ will force subtours in the master solution, and the slave subproblem is likely to be either more expensive than $T$ or not a feasible TSP solution (if $T$ is lower than the optimum, the slave solution cannot have both properties, obviously).

To avoid this problem, we implement an idea that is borrowed from the technique called tabu search. The tabu search is a local search method for mathematical optimization problems, searching for improvements of a current solution in its neighborhood, but are likely to get stuck in local minima. To solve this problem, the tabu search keeps a tabu list, that is, a list of solutions that are “forbidden” for the algorithm to consider again within a short amount of time.

In our case, the approach is slightly different. We will keep a tabu list of subtours, and each time the master subproblem is unfeasible, a subprocedure will extract the shortest subtour and add it to the list. Then, the master subproblem is slightly redefined, with a little abuse of notation, as follows.

\[
\min \Delta(x, \bar{x}) \quad (20)
\]

\[
\sum_{e \in \delta(i)} x_e = 2 \quad i \in V \quad (21)
\]

\[
\sum_{e \in E} w_e x_e \leq T \quad (22)
\]

\[
\sum_{e \in E(S)} x_e \leq |S| - 1 \quad S \in \text{tabu list} \quad (23)
\]

\[
x_e \in \{0,1\} \quad e \in E \quad (24)
\]

In this model we represented the tabu list as a set of subtour, which are sets of nodes that violate the constraints (5) explained in section 1.2.3.

The use of a tabu list approach has two positive effects on the algorithm, as previously mentioned:

* it removes the possibility of loops;

* it allows us to detect unfeasible subproblems.

The first point is straightforward. To better explain the second one, we notice that the only thing that keeps the master subproblem from being infeasible is the fact that its solution can have subtours because of the relaxation of the SECs. By adding the violated SECs to the formulation, we gradually disallow the only way the model has to provide a solution that costs less than T, and thus we force it to produce different values.
This procedure can indeed generate all the SECs in the worst case, which is obviously not tractable since solvers are unable to handle such a large amount of constraints. We allow the user to provide the maximum size of our tabu list, and if more subtours have to be added to it, older ones are replaced with newer ones. The size of the model remains small this way, and the tabu list should fulfill its role in almost all cases (the exception being a particularly long loop of solutions with similar value, however this, while technically possible, is very unlikely and fixable by expanding the maximum tabu size).

2.8 A REVISED VERSION OF THE ALGORITHM

As a reference, we provide the pseudocode of our algorithm (Algorithm 2) after the modifications we illustrated in Sections 2.6 and 2.7.

The code is similar to the one already illustrated in Section 2.5. The key differences are already been explained in the previous sections. In addition, we now avoid solving the slave subproblem if the master subproblem updated the value of \( T \); in that case, the master solution is already a valid solution of the underlying TSP instance, thus there is no need to try to heuristically enforce connectivity. The other slight difference, which is connected to the contents of Section 2.6, is that we check for improvements both solutions (slave and master), not only the output of the slave.

---

**Data:** A graph \( G = (V, E) \),
the weights \( w_e \ \forall e \in E \),
\( \tilde{x} \) solution of the TSP induced by \( G \) and \( w \)

**Result:** An improved solution \( \hat{x} \)

initialize \( \hat{x} \) and \( \xi \) to \( \tilde{x} \);
initialize \( m \) and \( T \) as described in Section 2.6;

while \( \sum_{e \in E} w_e \hat{x}_e \geq T \) or \( \hat{x} \) is not a feasible solution for the TSP do

\( x^* \) = solve the master with objective function \( \Delta(x, \xi) \);
if \( x^* \) is feasible for the TSP and \( \sum_{e \in E} w_e x^*_e \leq T \) then

\( \hat{x} = x^* \);
continue;
else if the master is unfeasible then

update \( T \) and \( m \) as described in Section 2.6;
find the shortest subtour and add it to tabu list;
else

find the shortest subtour and add it to tabu list;
end

\( \xi = \) slave optimal solution with weights defined as in (19);
if \( \xi \) is feasible for the TSP and \( \sum_{e \in E} w_e \xi_e \leq T \) then

\( \hat{x} = \xi \);
update \( T \) as described in Section 2.6;
end

end

return \( \hat{x} \)

---

Algorithm 2
This is the final version of our procedure, and its implementation will be discussed in detail in the next chapter.
In this chapter we will describe the main choices done and issues encountered while implementing the ideas presented in Chapter 2.

The programming language chosen to code the algorithm is C, that is preferred over any other high level programming language because of its better efficiency and performance.

3.1 DATA STRUCTURES

The main data structures used in our implementation are quite simple. The main things we need to represent are the graph underlying the TSP instance we process, and edge sets corresponding to various solutions to the subproblems we solve. These data are stored in global variables, so each function can access them freely and modify their value. This eases implementation a bit, since we do not need to pass all the arrays multiple times to the various function implemented. While this can hurt the reusability of the code, most of it is quite specific to our particular application, and it is hard to imagine a practical situation in which this choice would create problems.

We illustrate briefly how such entities are implemented.

3.1.1 Graph representation

To represent our graph $G = (V, E)$, we implicitly number the nodes from 0 to $|V| - 1$. An edge $e \in E$ has three attributes we want to keep track: the two vertices on which the edge is incident, and the cost of the edge. This is achieved with a simple struct shown below.

```c
typedef struct edge{
    double cost;
    int first_node;
    int second_node;
} edge;
```

This structure allows us to simply define the graph as an array of edges. In addition to this array, we create a data structure (called **nodes in the code) that, for each node, maintains a list of each edge incident at that node. The list is implemented as an array of integers: for example, if nodes[0][a] has a certain value, then edges[nodes[0][a]] is an edge that is incident T node 0; the value of a is arbitrary. This data structure introduces a little overhead in terms of memory, however it saves time, especially for sparse graphs, and allows us for an easier and more understandable implementation.
3.1.2 Solutions representation

Each solution of both the master or the slave subproblem is represented as an array of binary integers of length $|E|$. The following edge sets are kept in memory throughout the execution of the algorithm:

- the solutions to the master subproblem at the present iteration and at the previous one;
- the solutions to the slave subproblem at the present iteration and at the previous one;
- the cheapest solution encountered that is feasible for the underlying TSP instance.

3.1.3 Tabu list

The algorithm needs to keep and update a tabu list, whose details are explained in Section 2.7. We coded this data structure in a slightly different way than the one described previously. The table does not store the edge sets that violate a SEC in the previous iterations, instead it directly memorizes the corresponding SEC. This allow us to use a simple bidimensional array of integers to model the tabu list. The previously mentioned array (called **tabu) basically represents a matrix of coefficients; this eases the interaction with the ILP solver utilized, since such a representation is close to the native one used by it. The list is empty at first, and is expanded whenever a violated SEC is encountered.

3.1.4 Other data structures

There is data structure that we did not illustrate in the previous section, and is worth a brief mention. It is a sorted list of edges in order of nondecreasing cost. This list is necessary for the considerations done in Section 2.4. It is implemented as an array of integers, called *sorted_list; if an integer a is encountered before another integer b while sweeping the array from the first element to the last, this means that edges[a].cost is no greater than edges[b].cost.

Other than that, every parameter that controls the execution of the algorithm (like the time limit iteration limit, the current target formerly referred as $T$, and other miscellaneous variables) are mostly kept global to allow any procedure to access it.

3.2 THE MASTER SUBPROBLEM

The main implementative choice to do while implementing the master subproblem defined in Section 2.3 is the one concerning the solver utilized to obtain a solution of the master ILP model. We considered two main alternatives to cover this functionality:

- Concorde\(^1\), a state-of-the-art TSP solver widely used in many applications;

\(^1\) See Section A.1 for additional informations.
• ILOG CPLEX, a commercial LP/ILP/MILP solver by IBM.

Utilizing Concorde is very desirable for our application: the ILP formulation is only slightly different from a common relaxation of the general ILP model, the only differences being the objective function and the additional constraint (17) Section 2.3. Concorde allows the insertion of custom constraints; however, it handles them internally in hypergraph format, which makes very difficult to do so for our constraint.

For these reasons, we chose ILOG CPLEX as our solver. The CPLEX and CPLEX callable library functions we used are described in more detail in A.2.

We implemented a couple of subroutines to solve the master subproblem:

• solve_master() wraps the initialization of the ILP problem, the calls to CPLEX, and the updates to the target and to the tabu list;

• setproblemdata() initializes all the arrays that we need to pass to CPLEX for it to solve the ILP model;

• get_subtour() searches the current master solution for subtours, and returns the shortest one if found;

• add_to_tabu_list() and new_sec() are used to keep the tabu list up to date; the SEC correspondent to the previously extracted subtour (if any) is calculated and added to the tabu list.

3.2.1 CPLEX parameters setting

Since the calls to the CPLEX solver in the master subproblem represent the vast majority of the computational effort of our algorithm, we spent a fair amount of time trying to lower the time to complete the call, even at expense of finding the optimal solution (as expected from a heuristic algorithm).

The first approach we adopted is the following. Since the whole algorithm is a heuristic method, it is not essential for CPLEX to return us the best solution that fulfills all the constraints; we can be fine with a reasonably good one, too. Since CPLEX exploits heuristic methods before switching to an exact B&B strategy, we implemented the possibility to set to 1 the parameter CPLEX_PARAM_INTSOLLIM. Doing so forces CPLEX to stop after having found one feasible solution, which will not be the closest one to the previous iteration, but will be “reasonably close” while saving a considerable amount of computational time. This approach actually does not affect the optimality of the solution found, since we are just requiring a solution “reasonably close” to the previous one.

The second approach is aimed at avoiding an excessive computational time at each call to CPLEX. We limited the number of nodes of the decision tree CPLEX uses internally to solve ILP problems to a fixed (and reasonably small) amount. If CPLEX exceeds this number, the problem is considered unfeasible, so we either reduce the percentual improvement, if it happens during a binary search, or terminate otherwise. This setting will be differentiated between binary search and the less aggressive approach, that have been presented in Section 2.6. We will describe the particular settings we used for these parameters in Chapter 4.

We also used a third approach, which deserves a separate section to be better explained.
3.2.2 RINS Heuristic exploiting

The CPLEX solver we used in the master subproblem employs a wide set of heuristics. We are in particular interested to its RINS heuristic method. However, this method works best if CPLEX is provided with a starting feasible solution. Obviously we can not provide it, otherwise we would already have a good solution to update our target and go further in the execution. However, we can slightly modify the ILP model used in the master to achieve this result:

\[
\begin{align*}
\min & \quad \Delta(x, \bar{x}) + Mz \\
\text{s.t.} & \quad \sum_{e \in \delta(i)} x_e = 2 \quad i \in V \\
& \quad \sum_{e \in E} w_e x_e - z \leq T \\
& \quad x_e \in \{0, 1\} \quad e \in E \\
& \quad z \geq 0 \quad \text{integer.}
\end{align*}
\]

where $M \gg 0$ (50000 in our implementation).

Using this variation allows us to always provide a feasible solution — the best TSP solution found — since the model now can arbitrarily weaken the cutoff constraint.

Note that this approach is incompatible with CPLEX_PARAM_INTSOLLIM set to one (the algorithm would immediately stop, since the solution we provide to it is already feasible). We instead used another parameter, CPLEX_PARAM_OBJDIF, which allows to do roughly the same: it tells CPLEX to stop when it encounters a solution costing less than $M - 2n$, where $n = |V|$; this means that the first solution with $z = 0$ is returned.

3.3 The Slave Subproblem

The slave subproblem is relatively unchallenging when it comes to implementation. As already explained in 2.4, a slightly modified version of the Kruskal’s algorithm for the Shortest Spanning Tree is the core of the slave subproblem. The only remaining task is to select the two cheapest edges incident on the excluded vertex (which is, in our case, the last one; the choice of it is irrelevant anyway from a theoretical point of view, and doing so eases slightly the implementation), which is trivial.

The functions used for the slave subproblems are:

- `solve_slave()`, that is a wrapper for every other function call used for the solution of the slave, in analogy with `solve_master()``;
- `kruskal_setup()`, that is actually called in the main loop of our algorithm and not from the slave; however, it is described here since it is related to the custom implementation of Kruskal’s algorithm. Its function is to fill the sorted list as described in Subsection 3.1.4, which is done using a standard implementation of a Quicksort algorithm;
- `kruskal()`, that solves the SST problem defined in Section 2.4;
- `has_subtour()`, that is a wrapper for the previously defined `get_subtour()`, it returns zero if the graph is a feasible solution, and a nonzero value otherwise.
3.4 Main Loop

The only relevant aspect to mention about the main loop is the handling of the stopping condition. We require the user to input both a time limit and an iteration limit; if either one is met at the end of any given iteration, the program stops. The only other fact that may cause the algorithm to terminate is bound to the update process described in Section 2.6: if we are lowering the target of one unit at a time, and we obtain an unfeasible master, we terminate. In fact, this means that the current best solution is optimal thanks to the assumption of having integer costs.

3.5 Input and Output

We have two dedicated functions, parseinput() and print_results(). We briefly illustrate input and output formats, and the parameters that need to be passed to the algorithm for the parser to work correctly.

The function parseinput() implements a very basic parser for the input files that represent the graph underlying the TSP instance we are processing, and the initial tour. In the edge file, the first row is required to contain the number of nodes and edges of the graph, then each subsequent line is a triple of integers that represents an edge: the first two integers are the two vertices on which the edge is incident, and the third one represents the cost. For a simple graph with 5 nodes and 7 edges, the file would look like this:

```
5 7
0 1 3
1 2 2
2 3 1
3 4 6
0 4 2
1 4 5
1 3 4
```

In the tour file, each line must contain only one integer, which represents a node. Reading the lines from the first one defines the edges of the starting tour. Obviously, each pair of nodes in the tour file must correspond to an edge on the edge file; if an illegal pair is detected, the parser halts and returns an error message. This is a sample tour file for the previous graph:

```
0
1
2
3
4
```

The function print_results() prints on the file results.dat the best tour encountered in the execution of the algorithm. The format is the same used for the tour file.
Our routine also produces a log, that is extremely useful for monitoring the algorithm’s performance. Various figures are reported, such as the Hamming distances between master and slave and the amount of nodes CPLEX used to solve the master, along with the values of the objective function (an “*” is displayed each time an improvement is found). Below it is reported an execution on a test instance.

![Figure 2: A sample execution of our algorithm](image-url)
In this chapter we describe the testing methodology we used to test the
algorithm, on which instances we did such testing, and the experimental
results we gathered.

4.1 TEST INSTANCES

The TSP instances we used to test the performance of our algorithm are
synthetic ones; they are produced using the following steps:

1. randomly choose a set of k points in $\mathbb{R}^2$ (also called point cloud), where
   k is the number of desired nodes;
2. find a greedy tour in the instance defined by the nodes we just gener-
   ated and their distance on the plane;
3. randomly generate j edges starting from the point cloud;
4. merge the edges in the greedy tour with the randomly generated ones
   to obtain the initial edge list for our instance.

This procedure achieves two valuable objectives while generating an in-
stance to be solved:

- it ensures that the graph has already a tour (the greedy one), so the
  TSP defined on that graph has surely a solution;
- it easily produces a good, but not optimal, tour that we can use to
  initialize our algorithm.

The point cloud, the edge set, and the greedy tour are produced using the
executable edgegen provided in the Concorde solver\(^1\). See also Appendix B
for the code we used to generate our instances.
Table 1 illustrates the characteristics of our test instances.

4.2 PARAMETERS TO BE TESTED

In our tests, we ran our algorithm with a variety of settings, to verify
which set better works towards getting a good solution as soon as possible.
We briefly describe them.

- Use of the slave subproblem: we are interested to see if the slave sub-
  problem effectively helps the algorithm to converge faster towards a
  heuristically good solution. We call this setting “Slave” in the follow-
  ing tables, and a value of 1 denotes an active slave, while a value of 0
  means the opposite.

\(^1\) For more details, see Section A.1.
Table 1: Test Instances

<table>
<thead>
<tr>
<th>Number</th>
<th>Nodes</th>
<th>Edges</th>
<th>Starting Tour</th>
<th>Optimal Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>300</td>
<td>2243</td>
<td>4703</td>
<td>4661</td>
</tr>
<tr>
<td>2</td>
<td>300</td>
<td>4485</td>
<td>4490</td>
<td>4375</td>
</tr>
<tr>
<td>3</td>
<td>300</td>
<td>6728</td>
<td>4491</td>
<td>4311</td>
</tr>
<tr>
<td>4</td>
<td>400</td>
<td>3990</td>
<td>7104</td>
<td>6975</td>
</tr>
<tr>
<td>5</td>
<td>400</td>
<td>7980</td>
<td>7128</td>
<td>6620</td>
</tr>
<tr>
<td>6</td>
<td>400</td>
<td>11970</td>
<td>6836</td>
<td>6587</td>
</tr>
<tr>
<td>7</td>
<td>500</td>
<td>6238</td>
<td>9378</td>
<td>9194</td>
</tr>
<tr>
<td>8</td>
<td>500</td>
<td>12475</td>
<td>9301</td>
<td>9169</td>
</tr>
<tr>
<td>9</td>
<td>500</td>
<td>18713</td>
<td>9941</td>
<td>9117</td>
</tr>
<tr>
<td>10</td>
<td>600</td>
<td>8985</td>
<td>12791</td>
<td>12551</td>
</tr>
<tr>
<td>11</td>
<td>600</td>
<td>17970</td>
<td>13133</td>
<td>12437</td>
</tr>
</tbody>
</table>

- Use of the binary search approach: we want to test if doing a binary search introduces an excessive amount of computational effort or if the execution is sped up by this approach. This setting can be seen from field “PercImp” in our tables, that represent the minimum percentual improvement at the starting iteration. We tested two possible values: 0% (binary search disabled) and 10%. In the first case, we set the number of nodes CPLEX is allowed to explore in the decision tree to 5000, while in the second case we use 50 nodes as long as the binary search is active, and 1000 nodes when it is disabled.

- Number of Integer Solutions limited to one: we want to check if this setting effectively speeds up the execution, as we hope. The parameter is noted as “IntSol” in our tables, and has been set to one if the tables reports an 1, and let to its default value (+∞) otherwise.

- Use of the z – RINS approach: it is very interesting to see if and how much RINS speeds up the computation. Again, this is marked as “RINS” in our tables, and a value of 0 means the normal approach, while a value of 1 refers to the one explained in 3.2.2.

Table 2: Test Results, Instance 1 (300 nodes, 2243 edges)

<table>
<thead>
<tr>
<th>Slave</th>
<th>PercImp</th>
<th>IntSol</th>
<th>RINS</th>
<th>StartTour</th>
<th>OptSol</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>1</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>4703</td>
<td>4661</td>
<td>4661</td>
<td>3</td>
</tr>
</tbody>
</table>
### 4.3 Test Results

We include a number of tables that illustrate the test results we obtained. Note that the fields “StartTour”, “OptSol”, “Output” actually refer to the objective function value, not to the solution itself.

<table>
<thead>
<tr>
<th>Slave</th>
<th>PercImp</th>
<th>IntSol</th>
<th>RINS</th>
<th>StartTour</th>
<th>OptSol</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>28</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>18</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>29</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>32</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>26</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>41</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>16</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>24</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>29</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>33</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>4490</td>
<td>4375</td>
<td>4375</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 3: Test Results, Instance 2 (300 nodes, 4485 edges)

<table>
<thead>
<tr>
<th>Slave</th>
<th>PercImp</th>
<th>IntSol</th>
<th>RINS</th>
<th>StartTour</th>
<th>OptSol</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>4491</td>
<td>4311</td>
<td>4311</td>
<td>60</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>4491</td>
<td>4311</td>
<td>4311</td>
<td>71</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>4491</td>
<td>4311</td>
<td>4311</td>
<td>53</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>4491</td>
<td>4311</td>
<td>4324</td>
<td>31</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>4491</td>
<td>4311</td>
<td>4311</td>
<td>43</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>4491</td>
<td>4311</td>
<td>4311</td>
<td>84</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>4491</td>
<td>4311</td>
<td>4311</td>
<td>59</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>4491</td>
<td>4311</td>
<td>4311</td>
<td>70</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>4491</td>
<td>4311</td>
<td>4311</td>
<td>48</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>4491</td>
<td>4311</td>
<td>4312</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>4491</td>
<td>4311</td>
<td>4311</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>4491</td>
<td>4311</td>
<td>4313</td>
<td>73</td>
</tr>
</tbody>
</table>

Table 4: Test Results, Instance 3 (300 nodes, 6728 edges)

<table>
<thead>
<tr>
<th>Slave</th>
<th>PercImp</th>
<th>IntSol</th>
<th>RINS</th>
<th>StartTour</th>
<th>OptSol</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>19</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>13</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>34</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>21</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>26</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>21</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>25</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>7104</td>
<td>6975</td>
<td>6975</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 5: Test Results, Instance 4 (400 nodes, 3990 edges)
<table>
<thead>
<tr>
<th>Slave</th>
<th>PercImp</th>
<th>IntSol</th>
<th>RINS</th>
<th>StartTour</th>
<th>OptSol</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>7128</td>
<td>6620</td>
<td>661</td>
<td>179</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>7128</td>
<td>6620</td>
<td>6620</td>
<td>55</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>7128</td>
<td>6620</td>
<td>6620</td>
<td>67</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>7128</td>
<td>6620</td>
<td>6620</td>
<td>62</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>7128</td>
<td>6620</td>
<td>6620</td>
<td>51</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>7128</td>
<td>6620</td>
<td>6620</td>
<td>102</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>7128</td>
<td>6620</td>
<td>6620</td>
<td>67</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>7128</td>
<td>6620</td>
<td>6620</td>
<td>39</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>7128</td>
<td>6620</td>
<td>6620</td>
<td>52</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>7128</td>
<td>6620</td>
<td>6620</td>
<td>40</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>7128</td>
<td>6620</td>
<td>6620</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 6: Test Results, Instance 5 (400 nodes, 7980 edges)

<table>
<thead>
<tr>
<th>Slave</th>
<th>PercImp</th>
<th>IntSol</th>
<th>RINS</th>
<th>StartTour</th>
<th>OptSol</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>88</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>102</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>79</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>83</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>101</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>131</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>84</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>67</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>77</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>85</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>6836</td>
<td>6587</td>
<td>6587</td>
<td>137</td>
</tr>
</tbody>
</table>

Table 7: Test Results, Instance 6 (400 nodes, 11970 edges)

<table>
<thead>
<tr>
<th>Slave</th>
<th>PercImp</th>
<th>IntSol</th>
<th>RINS</th>
<th>StartTour</th>
<th>OptSol</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>17</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>28</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>11</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>20</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>24</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>9378</td>
<td>9194</td>
<td>9194</td>
<td>21</td>
</tr>
</tbody>
</table>

Table 8: Test Results, Instance 7 (500 nodes, 6238 edges)
### Table 9: Test Results, Instance 8 (500 nodes, 12475 edges)

<table>
<thead>
<tr>
<th>Slave</th>
<th>PercImp</th>
<th>IntSol</th>
<th>RINS</th>
<th>StartTour</th>
<th>OptSol</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>86</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>91</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>114</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>74</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>72</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>120</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>84</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>63</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>132</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>92</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>69</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>9301</td>
<td>9169</td>
<td>9169</td>
<td>120</td>
</tr>
</tbody>
</table>

### Table 10: Test Results, Instance 9 (500 nodes, 18713 edges)

<table>
<thead>
<tr>
<th>Slave</th>
<th>PercImp</th>
<th>IntSol</th>
<th>RINS</th>
<th>StartTour</th>
<th>OptSol</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>9941</td>
<td>9117</td>
<td>9235</td>
<td>282</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>9941</td>
<td>9117</td>
<td>9117</td>
<td>345</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>9941</td>
<td>9117</td>
<td>9117</td>
<td>303</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>9941</td>
<td>9117</td>
<td>9117</td>
<td>204</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>9941</td>
<td>9117</td>
<td>9117</td>
<td>168</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>9941</td>
<td>9117</td>
<td>9118</td>
<td>319</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>9941</td>
<td>9117</td>
<td>9117</td>
<td>244</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>9941</td>
<td>9117</td>
<td>9117</td>
<td>149</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>9941</td>
<td>9117</td>
<td>9117</td>
<td>228</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>9941</td>
<td>9117</td>
<td>9122</td>
<td>168</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>9941</td>
<td>9117</td>
<td>9117</td>
<td>171</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>9941</td>
<td>9117</td>
<td>9117</td>
<td>381</td>
</tr>
</tbody>
</table>

### Table 11: Test Results, Instance 10 (600 nodes, 8985 edges)

<table>
<thead>
<tr>
<th>Slave</th>
<th>PercImp</th>
<th>IntSol</th>
<th>RINS</th>
<th>StartTour</th>
<th>OptSol</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>12791</td>
<td>12551</td>
<td>12643</td>
<td>42</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>12791</td>
<td>12551</td>
<td>12551</td>
<td>25</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>12791</td>
<td>12551</td>
<td>12551</td>
<td>39</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>12791</td>
<td>12551</td>
<td>12551</td>
<td>63</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>12791</td>
<td>12551</td>
<td>12551</td>
<td>45</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>12791</td>
<td>12551</td>
<td>12551</td>
<td>91</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>12791</td>
<td>12551</td>
<td>12551</td>
<td>56</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>12791</td>
<td>12551</td>
<td>12551</td>
<td>36</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>12791</td>
<td>12551</td>
<td>12551</td>
<td>46</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>12791</td>
<td>12551</td>
<td>12551</td>
<td>66</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>12791</td>
<td>12551</td>
<td>12551</td>
<td>38</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>12791</td>
<td>12551</td>
<td>12551</td>
<td>84</td>
</tr>
</tbody>
</table>
Table 12: Test Results, Instance 11 (600 nodes, 17970 edges)

<table>
<thead>
<tr>
<th>Slave</th>
<th>PercImp</th>
<th>IntSol</th>
<th>RINS</th>
<th>StartTour</th>
<th>OptSol</th>
<th>Output</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>13133</td>
<td>12437</td>
<td>12437</td>
<td>125</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>13133</td>
<td>12437</td>
<td>12437</td>
<td>64</td>
</tr>
<tr>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>13133</td>
<td>12437</td>
<td>12437</td>
<td>86</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>13133</td>
<td>12437</td>
<td>12437</td>
<td>149</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>13133</td>
<td>12437</td>
<td>12437</td>
<td>60</td>
</tr>
<tr>
<td>0</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>13133</td>
<td>12437</td>
<td>12437</td>
<td>98</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>0</td>
<td>13133</td>
<td>12437</td>
<td>12437</td>
<td>137</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>13133</td>
<td>12437</td>
<td>12437</td>
<td>69</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>13133</td>
<td>12437</td>
<td>12437</td>
<td>89</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>0</td>
<td>13133</td>
<td>12437</td>
<td>12437</td>
<td>121</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>0</td>
<td>1</td>
<td>13133</td>
<td>12437</td>
<td>12439</td>
<td>60</td>
</tr>
<tr>
<td>1</td>
<td>10%</td>
<td>1</td>
<td>0</td>
<td>13133</td>
<td>12437</td>
<td>12439</td>
<td>85</td>
</tr>
</tbody>
</table>

Figure 3: Evolution of master and slave values on Instance 11, with CPLEX_PARAM_INTSOLLIM=1

Figure 4: Evolution of master and slave values on Instance 11, with the RINS approach active
There are a couple of observations we can draw from the results of these tests. The first one is that our algorithm, even with the limitation to the number of nodes, manages to find the optimal solution with almost all parameter settings, and with almost every instance. This is a positive thing, in a sense, however it may suggest a “not sufficiently heuristic” approach to the problem. It would be interesting to repeat the tests with different settings for the node limits, however this was not possible because of strict time constraints for the completion of our work.

The second one is that it does not emerge a clear winner between all the approaches tested. This is partly expected, since many of the settings can work well for a particular instance and not-so-well for another one. A more extensive testing would help in better outlining which approach, on average, yields the best results, and which one should be avoided. The impression is that both exploiting RINS heuristic and the limit of the number of integer solutions to one can greatly help the algorithm to reduce it execution time, although in some runs this effect is much less prominent. Furthermore, the binary search is probably the less successful expedient to speed up the computation: it is required a large amount of effort to effectively show that the current percentual improvement is too high, and while each update improves a lot the best solution, it is not worth it when compared to small, but very frequent updates by the less aggressive approach.

The plots provided also show how the master subproblem seems to “lead” the computation; the slave often produce edge sets with a very high cost just to enforce connection. This phenomenon is more pronounced if the binary search is active, as one can see in Figure 5, and may suggest a secondary role of the slave in the computation.
In this work, we outlined and described a new approach for a heuristic Tour Improvement Procedure for the TSP problem, based on the concept of proximity. A variety of different parameter settings and slightly different implementations is proposed and evaluated with extensive testing.

The results of our testing suggest that many of the proposed parameter settings help to reduce the execution time of the algorithm, although none of them has a dramatic impact on it. A more in-depth tuning phase could help to further improve the performance of the algorithm, and to better grasp the interactions between our ILP formulation and the heuristics CPLEX employs internally. Since the approach is new, we are confident that the performances of our algorithm can be improved significantly.

The final outcome of our research looks indeed promising, although not very interesting as-is. There exist software tools like Concorde that can solve easily instances with more than 1000 nodes with minimal computational and time effort, so we cannot claim that our implementation should be used over Concorde. However, the downside of such approaches is that they are extremely specialized in the problem they solve; it is impossible, for example, to adapt Concorde for other NP-hard problems, and it is very hard to even adapt it for slight modifications of the TSP problem, as we experienced during this work. Our algorithm does not suffer from this limitation; it is instead a quite general approach, that could be very well extended to other problems that do not have a good solver available.

In light of this consideration, the most interesting hint for future research on this topic is to try the approach applied to other difficult optimization problems, especially the ones that lack a great amount of research (unlike the TSP). Doing so could improve the actual approaches to those problems, hopefully providing a good heuristic to improve already existing solutions. The other main foreseeable development to our work is in the direction of specialization and improvement of the approach used to solve the master subproblem; a dedicated solver would dramatically improve the execution time of our algorithm, as the example of Concorde proves without any doubt.
This chapter illustrates the software packages and applications that were used in the development of this thesis. We briefly describe the goal for the use of each tool, and the main instructions or commands used.

A.1 CONCORDE

Concorde is a state-of-the-art TSP solver for the TSP problem and some related problems. It was written using the C programming language, by David Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook.

Initially, we were interested in using Concorde as a solver for the master subproblem. This was not possible, for the reasons explained in Section 3.2; however the Concorde package provide some useful tools that aided us in the development and testing of our algorithm.

A.1.1 TSP Solver

The Concorde main application, the TSP solver, was useful to us for finding rapidly and efficiently the optimal solution for our test instances. This was helpful to check the correctness of our solutions, and to compare the execution time of our algorithm to the one of a state-of-the-art software. The syntax to use the Concorde solver is extremely simple; assume one wants to solve an instance, contained in the file tspinstance.dat in TSPLIB format, he needs to use the following command:

. /concorde tspinstance.dat

There are a number of useful options that can be used with the program, that control the input data or the heuristics or settings Concorde uses while solving the TSP; they can be easily retrieved by typing . /concorde without any argument.

A.1.2 Edge generation

The other aspect in which Concorde helped us was the generation of the test instances. In the Concorde package, an executable, called edgegen, is provided; its main use is to generate point clouds and edge sets (optionally starting from an existing point cloud). The availability of such a tool was very important to implement the instance generation strategy we illustrated in Section 4.1. We describe briefly how it is possible to execute those steps using edgegen.

The tool is called in a similar fashion to the TSP solver in this way:

. /edgegen [options] [inputfile.dat]

---

1 See Section A.1.3 of this Appendix for more detailed informations.
The file inputfile.dat is optional (it is required or not depending on the option flags supplied), and must be in TSPLIB format once again, if present. The main options that were useful to us were:

- `edgegen -k #nodes -p outfile.dat`: this command randomly creates a point cloud with the specified amount of nodes, and stores it in outfile.dat;
- `edgegen -G -o outfile.dat inputfile.dat`: this command reads the point cloud contained in inputfile.dat (in TSPLIB format), finds a greedy tour between the points using Euclidean distances, and saves the edges of the tour in outfile.dat;
- `edgegen -e #edges -o outfile.dat inputfile.dat`: this command is similar to the previous one, with the only difference that it generates the desired number of random edges using the supplied point cloud; it then stores the results as previously described.

### A.1.3 TSPLIB format

As described in the previous sections, many tools in the Concorde package require the input file to be in TSPLIB format. That is a text format that defines a precise syntax for representing a TSP instance. The complete specification can be found in [10]; we provide an example of a point cloud file for reference.

```plaintext
NAME : concorde_5
COMMENT : 5 nodes, randomly generated by concorde
TYPE : TSP
DIMENSION : 5
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
1 3.000000 4.000000
2 4.000000 0.000000
3 2.000000 1.000000
4 1.000000 4.000000
5 3.000000 0.000000
```

### A.2 CPLEX

ILOG CPLEX is a commercial solver for Linear Programming, Integer Linear Programming and Mixed Integer Linear Programming problems. It is a main part of our implementation of the algorithm illustrated in Chapter 2, used to solve the master subproblem. CPLEX is provided in multiple programming languages, and also includes an Interactive Solver that can be called from the command line. We used the CPLEX Callable Library, which is the implementation of the CPLEX solver in C, alongside with the Interactive Solver as a way to ease the bug fixing process. We briefly list the main commands and parameters we used in this work.

#### A.2.1 Interactive Solver

The Interactive Solver was mainly used to solve ILPs saved by our code via the CPXwriteprob routine, to allow a double check in case of unexpected
results from the application we were developing. The tool has proven invaluable to find and solve troubles, especially some of them that were not apparent at all (usually related to numerical problems).

We provide a short list of the main commands we used in conjunction with the Interactive Solver.

- **read <filename>**: reads an ILP problem from the specified file.
- **set [options]**: allows to control various parameters that affect the execution of CPLEX, for examples the ones that control the number of nodes in the decisional tree or the amount of integer solution found before exiting; more informations on the categories of parameters that one can control are provided when the command is entered.
- **mipopt**: solve the current problem.
- **display [options]**: used to show the current solution of the ILP (display solution variables) or to access to the current parameter settings (display settings).

### A.2.2 Callable Library

The Callable Library provides a C implementation of the CPLEX solver, which is central for our goal, as already described multiple times. The main routines we used to implement our algorithm are listed below.

- **CPXopenCPLEX()**: creates the CPLEX environment that is needed for every other call to it.
- **CPXsetintparam()** / **CPXsetdblparam()**: allows to control the parameter settings that have been described previously in Chapters 3 and 4.
- **CPXcreateprob()**: creates the (I)LP problem that will be initialized and solved.
- **CPXcopylp()**: requires multiple arrays that describe the problem and loads them into it.
- **CPXcopyctype()**: similar to the previous one, it defines which variables are integer and which one are not.
- **CPXaddmipstarts()**: allows to load one or more starting solutions to hopefully ease CPLEX’s computations.
- **CPXmipopt()**: optimizes the current problem.
- **CPXgetx()**: access the values of the decision variables after having solved the (I)LP.
- **CPXgetobjval()**: reads the objective function value corresponding to the current solution.
- **CPXgetnodecnt()**: allows to know how many nodes of the decisional tree were explored by the B&B algorithm CPLEX uses while solving the problem.
- **CPXfreeprob()**: frees the memory used by a problem.
- **CPXcloseCPLEX()**: closes the CPLEX environment after all the computations have been completed.
B | SOURCE CODE

B.1 MAIN PROGRAM

B.1.1 tandem.h

```c
#ifndef EDGE
#define EDGE

/* edge */
typedef struct {
  double cost;
  int first_node;
  int second_node;
} edge;
#endif

int parseinput(int argc, char *argv);
double get_cost(int *sol);
int solve_master();
int add_to_tabu_list(int *size, int **subtour);
int *new_sec(int *size, int **subtour);
static int setproblemdata(char **probname_p, int *numcols_p, int *numrows_p, int *objsen_p, double **obj_p, double **rhs_p, char **sense_p, int **matbeg_p, int **matcnt_p, int **matind_p, double **matval_p, double **lb_p, double **ub_p, char **ctype_p);
int has_subtour(int *working_sol);
int get_subtour(int *size, int **stour, int *working_sol);
void print_current_status();
void print_results();
int evaluate_hamming(int *first, int *second);
void free_and_null(char **ptr);
void print_log_row(int iter, double m_cost, double s_cost, double best, int upd, double tar, double pctint, int hamm, int tabu_s, int cnodes, int maxnodes, double time_ela);
```

B.1.2 tandem.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#include "cplex.h"
#include "tandem.h"
#include "kruskal.h"

#define DEBUG 0

#if DEBUG
#define PRINTVERBOSE(...) printf(__VA_ARGS__)
#else
#define PRINTVERBOSE(...) ((void) 0)
#endif

#define PRINTERROR(...) fprintf(stderr, __VA_ARGS__)
#define PRINTUSAGE() printf("Usage: tandem <edgefile> <tourfile> <max tabu size> <iter limit> <time limit> <use slave?> <min percentual improvement> <intsollim = 1 ?> <use z RINS heuristic> <max node count binary> <max node count nonbinary>\n")
```
/* "global" variables */

/* using slave? */
int use_slave;

/* slave at this iteration? */
int slave_curr_iter;

/* solution updated this iteration? */
int sol_update;

/* current target */
double curr_target;

/* old and current master and slave */
int *curr_master;
int *curr_slave;
int *old_master;
int *old_slave;

/* current solution */
int *working_sol;

/* tabu list */
int **tabu;
int num_nz;
int tabu_size;
int max_tabu_size;
int tabu_change;

/* best solution */
int *best_sol;

/* cplex env */
CPXENVptr env;

int cpxnodes;
int maxnodecount_binary;
int maxnodecount_nonbinary;
int maxnodecount_actual;
int intsollim;
int iteration;

/* sorted edge list */
int *sorted_index;

/* use z */
int use_z;

/* stopping conditions */
int max_iter;
double max_time;
time_t start_time;

/* end variables */

/* parses the command line arguments */
int parseinput(int argc, char **argv){
  PRINTVERBOSE("Parsing...\n");
  int status = 0;
  int a, b;
  int node1, node2;
  double edgecost;
  int *temp;
  char *line = malloc(sizeof(char) * 100);
  if (argc < 11){
PRINTERROR("Error: Not enough arguments.\n");
PRINTUSAGE();
exit(1);

FILE *edgefile = fopen ( argv[1], "r" );
FILE *tourfile = fopen ( argv[2], "r" );

if (edgefile == NULL || tourfile == NULL){
    PRINTERROR("Error: cannot open input files.\n");
    PRINTUSAGE();
    exit(1);
}

/* parse edgefile */

/* leggo nnodes, nedges dalla prima riga */

if (fgets(line,100,edgefile)==NULL){
    PRINTERROR("Error: cannot parse nnodes and nedges from edge file\n");
    exit(1);
}
nnodes = atoi(strtok (line, " "));
nedges = atoi(strtok (NULL, " "));

PRINTVERBOSE("%d nodes, %d edges\n", nnodes, nedges);

edges = malloc(sizeof(edge)*nedges);
nodes = malloc(sizeof(int*)*nnodes);
nodes_length = malloc(sizeof(int)*nnodes);

if (edges == NULL || nodes == NULL || nodes_length == NULL) {
    PRINTERROR("Error: out of memory while allocating data structures");
    exit(1);
}

for(a=0;a<nnodes;a++){
    nodes[a] = NULL;
nodes_length[a] = 0;
}

for(a=0;a<nedges;a++){
    if (fgets(line,100,edgefile)==NULL){
        PRINTERROR("Error: not enough edges or edge file format incorrect\n");
        exit(1);
    }
    node1 = atoi(strtok(line, " "));
node2 = atoi(strtok(NULL, " "));
    edgecost = atof(strtok(NULL, " "));
    if (node1>node2){
        int temp = node1;
node1 = node2;
node2 = temp;
    }
    edges[a].first_node = node1;
    edges[a].second_node = node2;
    edges[a].cost = edgecost;

    temp = realloc(nodes[node1], (nodes_length[node1]+1) * sizeof(int));
    if (temp == NULL){
        PRINTERROR("Error: failed to realloc memory in parseinput.\n");
        goto CLEANUP;
    }
    nodes_length[node1]++;
    nodes[node1] = temp;
nodes[node1][nodes_length[node1]-1] = a;
    temp = realloc(nodes[node2], (nodes_length[node2]+1) * sizeof(int));
    if (temp == NULL){
        PRINTERROR("Error: failed to realloc memory in parseinput.\n");
        goto CLEANUP;
    }
    nodes_length[node2]++;
    nodes[node2] = temp;
nodes[node2][nodes_length[node2]-1] = a;
curr_master = malloc(sizeof(int)*nedges);
curr_slave = malloc(sizeof(int)*nedges);
old_master = malloc(sizeof(int)*nedges);
old_slave = malloc(sizeof(int)*nedges);

for(a=0;a<nedges;a++){
  old_master[a] = 0; old_slave[a] = 0;
}

best_sol = malloc(sizeof(int)*nedges);
working_sol = malloc(sizeof(int)*nedges);

/* parse tourfile */
int currentnode, newnode, firstnode;
temp1, temp2;

if (fgets(line,100,tourfile)==NULL){
  PRINTERROR("Error: not enough edges or tour file format incorrect\n");
goto CLEANUP;
}

newnode = atoi(line);
firstnode = newnode;

for(a=0;a<nodes-1;a++){
  if (fgets(line,100,tourfile)==NULL){
    PRINTERROR("Error: not enough edges or tour file format incorrect\n");
    exit(1);
  }
  currentnode = newnode;
  newnode = atoi(line);
  if(currentnode<newnode){
    temp1=currentnode;
    temp2=newnode;
  } else {
    temp1=newnode;
    temp2=currentnode;
  }
  b = 0;
  while(b<nodes_length[currentnode] &&
    (edges[nodes[currentnode][b]].first_node!=temp1 ||
    edges[nodes[currentnode][b]].second_node != temp2)){
    b++;
  }
  if (b==nodes_length[currentnode]-1 &&
    (edges[nodes[currentnode][b]].first_node!=temp1 ||
    edges[nodes[currentnode][b]].second_node != temp2)){
    PRINTERROR("Error! Edge (%dn %d) not found\n", temp1, temp2);
    goto CLEANUP;
  } else {
    working_sol[nodes[currentnode][b]] = 1;
  }
  b = 0;
}

/* last edge of the tour */
if(firstnode==newnode){
  temp1=firstnode;
  temp2=newnode;
} else{
  temp1=newnode;
  temp2=firstnode;
}

b = 0;
while(b<"nodes.length[firstnode] 
(edges[nodes[firstnode][b]].first_node!=temp1 ||
edges[nodes[firstnode][b]].second_node != temp2)){
b++;}

if (b<"nodes.length[firstnode] -1 &&
(edges[nodes[firstnode][b]].first_node!=temp1 ||
edges[nodes[firstnode][b]].second_node != temp2)){
PRINTERROR("Error! Edge (%dn %d) not found\n", temp1, temp2);
}
else{
working_sol[currentnode] = 1;
}

/* init best sol */
for(a=0;a<"nedges;a++){
best_sol[a] = working_sol[a];
}

/* target */
curr_target = get_cost(working_sol)-1;

/* tabu */
max_tabu_size = atoi(argv[3]);
tabu_size = 0;
um_nz = 0;
tabu = malloc(sizeof(int)*max_tabu_size);
for(a=0;a<max_tabu_size;a++){
tabu[a]=NULL;
}

/* time and iteration limits */
max_iter = atoi(argv[4]);
max_time = atof(argv[5]);

/* are we using the slave? */
use_slave = atoi(argv[6]);

/* minimum percent improvement */
min_pct_impr = atof(argv[7])/100;

/* one solution or till optimal one in master? */
intsollim = atoi(argv[8]) == 1 ? 1 : 0;

/* use z */
use_z = atoi(argv[9]);

/* max nodes */
maxnodecount_binary = atoi(argv[10]);
maxnodecount_nonbinary = atoi(argv[11]);
maxnodecount_actual = (min_pct_impr)==0?maxnodecount.binary:maxnodecount.nonbinary;

PRINTVERBOSE("Cost of starting tour: %.2f\n", get_cost(working_sol));
fclose(edgefile); fclose(tourfile);
return 0;

CLEANUP:
PRINTERROR("Error: cleanup still not implemented");
exit(1);
}

/* calcola il costo di una soluzione */
double get_cost(int *sol){
int i;
double to_return = 0;
for(i=0;i<"nedges;i--{

to_return += edges[i].cost * sol[i];
}
return to_return;
}

/* solve the master subproblem */
int solve_master(){
    /* cplex variables */
    char *probname = NULL;
    int numcols;
    int numrows;
    int objsen;
    double *obj = NULL;
    char *sense = NULL;
    int *matbeg = NULL;
    int *matcnt = NULL;
    int *matind = NULL;
    double *matval = NULL;
    double *lb = NULL;
    double *ub = NULL;
    char *ctype = NULL;

    int solstat;
    double *x = NULL;
    CPXLPtr lp = NULL;

    status = setproblemdata (&probname, &numcols, &numrows, &objsen, &obj,
        &rhs, &sense, &matbeg, &matcnt, &matind, &matval,
        &lb, &ub, &ctype);
    if (status){
        PRINTERROR("Error: setproblemdata failed at iteration \"", iteration);
        exit(1);
    }
    x = malloc(sizeof(double)*numcols);
    if (x==NULL){
        PRINTERROR("Failed to allocate memory for solution array
        
    ");
        exit(1);
    }
    for(i=0;i<nedges;i++){
        x[i]=0;
    }
    /* Create the problem. */
    lp = CPXcreateprob (env, &status, probname);
    if ( lp == NULL ) {
        PRINTERROR("Error: Failed to create LP.
        
    ");
        goto TERMINATE;
    }
    /* Now copy the problem data into the lp */
    status = CPXcopylp (env, lp, numcols, numrows, objsen, obj, rhs,
        sense, matbeg, matcnt, matind, matval,
        lb, ub, NULL);
    if ( status ) {
        PRINTERROR("Failed to copy the problem data.\n");
        goto TERMINATE;
    }
    /* Now copy the ctype array */
    status = CPXcopyctype (env, lp, ctype);
    if ( status ) {
        PRINTERROR("Failed to copy ctype\n");
        goto TERMINATE;
    }
    /* add the appropriate mip start */
    if(use_z){
        int mcnt = 1;
        int nzcnt = nedges;
        int *beg = malloc(sizeof(int)*mcnt);
        int *effort = malloc(sizeof(int)*mcnt);
        beg[0] = 0;
        effort[0] = 4;
int *varindices = malloc(sizeof(int) * nzcnt);
double *values = malloc(sizeof(double) * nzcnt);
for (i=0;i<nedges;i++) {
    varindices[i] = i;
    values[i] = best_sol[i];
}

status = CPXaddmipstarts(env, lp, mcnt, nzcnt, beg, varindices, values, effort, NULL);

/* solve MIP */
status = CPXmipopt(env, lp);
if (status) {
    PRINTERROR("Error: Failed to optimize MIP. Status: %d\n",
    status);
    goto TERMINATE;
}

/* getting optimal assignment for decision variables */
status = CPXgetx(env, lp, x, 0, CPXgetnumcols(env, lp)-2);
int status2;
double value = -1;
status2 = CPXgetobjval(env, lp, &value);
if (status || (cpxnodes = CPXgetnodecnt(env, lp)) >= maxnodecount_actual ||
    value >= 50000 -2*nnodes -1) {
    if (status!=0 && status != 1217 /*&& value < 50000 -2*nnodes -1*/){
        PRINTERROR("Error: Failed to get decision variables. Status: %d\n",
        status);
        exit(1);
    } else {
        /* master is unfeasible; updating target and min.impr */
        if((curr_target / (1-min_pct_impr)) - curr_target <=1){
            print_results();
            printf("Error: Master is \"heuristically unfeasible\". Saving best tour in \"results.dat\"");
            printf("Elapsed time: %4.0f\n",
            ((double)time(NULL)) - start_time);
            exit(0);
        }

        slave_curr_iter = 0;
        curr_target = curr_target / (1-min_pct_impr);
        min_pct_impr /= 2;
        curr_target = curr_target * (1-min_pct_impr);
        if((curr_target / (1-min_pct_impr)) - curr_target <=1){
            status = CPXsetintparam(env, CPX_PARAM_NODELIM,
            maxnodecount_nonbinary);
            maxnodecount_actual = maxnodecount_nonbinary;
        }

        for(i=0;i<nedges;i++){
            old_master[i] = curr_master[i];
            curr_master[i] = 0;
        }
    } goto TERMINATE;
}

/* update working sol and master */
for(i=0;i<numcols;i++){
    working_sol[i] = x[i]>0.5 ? 1 : 0;
    old_master[i] = curr_master[i];
    curr_master[i] = working_sol[i];
}

int *subtour;
int size = 0;

/* check if subtours are present */
if(get_subtour(&size, &subtour, working_sol)){
    PRINTVERBOSE("Found subtour (%d): ", size);
    for(i=0;i<size;i++) PRINTVERBOSE("%d ", subtour[i]);
    PRINTVERBOSE("\n");
    add_to_tabu_list(&size, &subtour);
    free(subtour);
} else {
    /* no subtour */
    if (get_cost(curr_master) <= curr_target){
        double new_target = (get_cost(curr_master) * (1-min_pct_impr));
    }
if (get_cost(curr.master) - new.target > 1){
    curr.target = new.target;
} else {
    curr.target = get_cost(curr.master) - 1;
    if (maxnodecount.actual != maxnodecount.nonbinary){
        status = CPXsetintparam(env, CPX.PARAM.NODELIM, 
                              maxnodecount.nonbinary);
        maxnodecount.actual = maxnodecount.nonbinary;
    }
    sol.update = 1;
    slave.curr.iter = 0;
    for(i=0;i<nedges;i++) best_sol[i] = curr.master[i];
}

/* debug check, should never happen */
if(evaluate_hamming(curr.master, old.master)){
    printf("DEBUG: Master and previous one equals. Exiting...

    CPXwriteprob(env, lp, "zerohamming.lp", NULL);
    exit(1);
}

PRINTVERBOSE("Master cost: %.2f\n", get_cost(curr.master));

TERMINATE:
/* free things */
if ( lp != NULL ) {
    status = CPXfreeprob(env, &lp);
    if ( status ) {
        PRINTERROR("Error: CPXfreeprob failed, error code \%d.\n", status);
    }
    free_and_null ((char**) &probname);
    free_and_null ((char**) &obj);
    free_and_null ((char**) &rhs);
    free_and_null ((char**) &sense);
    free_and_null ((char**) &matbeg);
    free_and_null ((char**) &matcnt);
    free_and_null ((char**) &matind);
    free_and_null ((char**) &matval);
    free_and_null ((char**) &lb);
    free_and_null ((char**) &ub);
    free_and_null ((char**) &ctype);
    free(x);
    return (status);
}

/* adds constraint to tabu list */
int add_to_tabu_list(int *size, int **subtour){
    int i, j, count;
    if(tabu.size>max.tabu.size){
        i = 0;
        while (tabu[i]!=NULL){
            i++;
        }
        tabu[i] = new_sec(size, subtour);
        tabu.size++;
        for(j=0;j<nedges;j++){
            num.nz += tabu[i][j];
        }
    } else {
        tabu.change = 1;
        for(j=0;j<nedges;j++){
            num.nz += tabu[i][j];
        }
        free(tabu[i])
        for(i=0;i<tabu.size;i++){
            tabu[i] = tabu[i-1];
        }
        tabu.tabu.size] = new_sec(size, subtour);
        for(j=0;j<nedges;j++){
            num.nz += tabu.tabu.size][];
        }
    }
/* create SEC corresponding to passed subtour */
int *new_sec(int *size, int **subtour)
{
    int i, j;
    int first, second;
    int *to_return = malloc(sizeof(int)*nedges);
    int *is_in_subtour=malloc(sizeof(int)*nnodes);
    if (to_return == NULL || is_in_subtour == NULL){
        PRINTERROR("Failed to allocate memory for solution array\n");
        exit(1);
    }
    for(i=0;i<nnodes;i++)
        is_in_subtour[i] = 0;
    for(i=0;i<*size;i++)
        is_in_subtour[(*subtour)[i]] = 1
    for(i=0;i<nedges;i++)
        if ((is_in_subtour[edges[i].first_node] &&
            !is_in_subtour[edges[i].second_node]) ||
            (is_in_subtour[edges[i].second_node] &&
            !is_in_subtour[edges[i].first_node]))
            to_return[i] = 1;
        else
            to_return[i] = 0;
    free(is_in_subtour);
    return to_return;
}

/* initialize cplex arrays */
static int setproblemdata (char **probname_p, int *numcols_p, int *numrows_p,
    int *objsen_p, double **obj_p, double **rhs_p,
    char **sense_p, int **matbeg_p, int **matcnt_p,
    int **matind_p, double **matval_p,
    double **lb_p, double **ub_p, char **ctype_p)
{
    int i, j;
    cell *iter;
    char *zprobname = NULL;  /* Problem name <= 16 characters */
    double *zobj = NULL;      /* Problem value */
    double *zrhs = NULL;      /* Right-hand sides */
    char *zsense = NULL;      /* Sense of constraints */
    int *zmatbeg = NULL;      /* Beginning of matrix */
    int *zmatcnt = NULL;      /* Matrix count */
    int *zmatind = NULL;      /* Matrix index */
    double *zmatval = NULL;  /* Matrix values */
    double *zlz = NULL;       /* Lower bounds */
    double *zub = NULL;       /* Upper bounds */
    char *zctype = NULL;      /* Type of variable */
    int status = 0;
    int numcols = nedges + use_z;
    int numrows = nnodes + 1 + tabu_size;
    int numnz = 3*numcols + num_nz + use_z;
    zprobname = (char *) malloc ((16 + sizeof(char)));
    zobj = (double *) malloc (numcols * sizeof(double));
    zrhs = (double *) malloc (numrows * sizeof(double));
    zsense = (char *) malloc (numrows * sizeof(char));
    zmatbeg = (int *) malloc (numcols * sizeof(int));
    zmatcnt = (int *) malloc (numcols * sizeof(int));
    zmatind = (int *) malloc (numnz * sizeof(int));
    zmatval = (double *) malloc (numnz * sizeof(double));
    zlb = (double *) malloc (numcols * sizeof(double));
    zub = (double *) malloc (numcols * sizeof(double));
    zctype = (char *) malloc (numcols * sizeof(char));
    if ( zprobname == NULL || zobj == NULL ||
        zrhs == NULL || zsense == NULL ||
zmatbeg == NULL || zmatcnt == NULL ||
zmatind == NULL || zmatval == NULL ||
zb == NULL || zub == NULL ||
zctype == NULL ) {
    status = 1;
    goto TERMINATE;
}

sprintf(zprobname, "Master_%d", iteration);

/* setting zobj */
for(i=0;i<numcols-use_z;i++) {
    if(working_sol[i]) zobj[i] = -1;
    else zobj[i] = 1;
}

/* z */
if(use_z) zobj[numcols-1] = 50000;

int count;
/* setting other things */
for(i=0;i<numcols-use_z;i++) {
    count = 0;
    for(j=0;j<tabu_size;j++) {
        count += tabu[j][i];
    }
    zmatcnt[i]=4 + count;
    if(i==0)
        zmatbeg[i]=0;
    else
        zmatbeg[i]=zmatbeg[i-1]+zmatcnt[i-1];
}

zmatind[zmatbeg[i]]=edges[i].first_node;
zmatind[zmatbeg[i]+1]=edges[i].second_node;
zmatind[zmatbeg[i]+2]=nnodes;

zmatval[zmatbeg[i]]=1;
zmatval[zmatbeg[i]+1]=1;
zmatval[zmatbeg[i]+2]=edges[i].cost;
/* tabu */
for(j=0;j<tabu_size;j++) {
    if(tabu[j][i] == 1){
        zmatbeg[i]=0;
    } else {
        zmatbeg[i]=zmatbeg[i-1]+zmatcnt[i-1];
    }
}

zmatind[zmatbeg[i]-edges[i].first_node];
zmatind[zmatbeg[i]-edges[i].second_node];
zmatind[zmatbeg[i]+2]=nnodes;

zmatval[zmatbeg[i]-1];
zmatval[zmatbeg[i]-1]+1;
zmatval[zmatbeg[i]+2]=edges[i].cost;
/* tabu */
for(j=0;j<tabu_size;j++) {
    if(tabu[j][i] == 1){
        zmatind[zmatbeg[i]+3+count]=nnodes+1+j;
        zmatval[zmatbeg[i]+3+count]=-1;
        count++;
    }
}

/* binary integer vars */
zb[i]=0;
zub[i]=1;
zctype[i]="I";
}

/* z */
if(use_z){
    zmatcnt[numcols-1] = 1;
    zmatbeg[numcols-1] = zmatbeg[numcols-2] + zmatcnt[numcols-2];
    zmatind[zmatbeg[numcols-1]] = nnodes;
    zmatval[zmatbeg[numcols-1]] = -1;
    zlb[numcols-1] = 0;
    zub[numcols-1] = CPX_INFBOUND;
    zctype[numcols-1] = 'I';
}

/* rhs's */
for(i=0;i<nnodes;i++) {
    zsense[i]="E";
    zrhs[i]=2;
}
zsense[nnodes]="L";
zrhs[nnodes]=curr_target;

for(i=nnodes+1;i<numrows;i++) {
    zsense[i]="G";
    zrhs[i]=2;
}

TERMINATE:
if ( status ) {
    free_and_null ((char **) &zprobname);
}
free_and_null((char **) &zobj);
free_and_null((char **) &zrhs);
free_and_null((char **) &zsense);
free_and_null((char **) &zmatbeg);
free_and_null((char **) &zmatcnt);
free_and_null((char **) &zmatind);
free_and_null((char **) &zmatval);
free_and_null((char **) &zlb);
free_and_null((char **) &zub);
free_and_null((char **) &zctype);
}
else {
    *numcols_p = numcols;
    *numrows_p = numrows;
    *objsen_p = CPX_MIN; /* The problem is minimization */
    *probname_p = zprobname;
    *obj_p = zobj;
    *rhs_p = zrhs;
    *sense_p = zsense;
    *matbeg_p = zmatbeg;
    *matcnt_p = zmatcnt;
    *matind_p = zmatind;
    *matval_p = zmatval;
    *lb_p = zlb;
    *ub_p = zub;
    *ctype_p = zctype;
}
return (status);
}
/* solves the slave subproblem */
int solve_slave(){
    int i,j;
    if(use_slave && slave_curr_iter){
        int *sorted_after_m = malloc(sizeof(int)*(nedges-nnodes_length[nnodes-1]));
        int count = 0;
        for(i=0;i<nedges;i++){
            if(working_sol[sorted_index[i]] && edges[sorted_index[i]].second_node!=nnodes-1){
                sorted_after_m[count] = sorted_index[i];
                count++;
            }
        }
        for(i=0;i<nedges;i++){
            if(!(working_sol[sorted_index[i]]) && edges[sorted_index[i]].second_node!=nnodes-1){
                sorted_after_m[count] = sorted_index[i];
                count++;
            }
        }
        int *tree = NULL;
        kruskal(nedges, nnodes-1, edges, sorted_after_m, &tree);
        int best, secbest;
        if(edges[nodes[nnodes-1][0]].cost > edges[nodes[nnodes-1][1]].cost){
            best = nodes[nnodes-1][1]; secbest = nodes[nnodes-1][0];
        } else {
            best = nodes[nnodes-1][0]; secbest = nodes[nnodes-1][1];
        }
        for(i=2;i<nodes_length[nnodes-1];i++){
            if(edges[nodes[nnodes-1][i]].cost < best){
                secbest = best;
                best = nodes[nnodes-1][i];
            } else if (edges[nodes[nnodes-1][i]].cost < best){
                secbest = nodes[nnodes-1][i];
            } } // for
        tree[best] = 1;
        tree[secbest] = 1;
for(i=0;i<nedges;i++)
    working_sol[i] = tree[i];
old_slave[i] = curr_slave[i];
curr_slave[i] = tree[i];
}

free(sorted_after_m);
free(tree);

if((!has_subtour(curr_slave)) && get_cost(curr_slave) <= curr_target){
double new_target = (get_cost(curr_slave) * (1-min_pct_impr));
if (get_cost(curr_slave) - new_target > 1){
curr_target = new_target;
} else {
curr_target = get_cost(curr_slave) - 1;
if (maxnodecount_actual != maxnodecount_nonbinary){
    int status = CPXsetintparam (env, CPX_PARAM_NODELIM,
                              maxnodecount_nonbinary);
    maxnodecount.actual = maxnodecount.nonbinary;
}
}
sol_update = 1;
}

for(i=0;i<nedges;i++) best_sol[i] = curr_slave[i];
}
}

/* wrapper */
int has_subtour(int *working_sol){
    int *sub = NULL;
    int a = 0;
    int to_return = get_subtour(&a, &sub, working_sol);
    free(sub);
    return to_return;
}

/* returns 1 if a subtour is found, 0 otherwise*/
int get_subtour(int *size, int **stour, int *working_sol){
    int *subtour = malloc (sizeof(int)*nnodes);
    int *shortest_subtour = malloc (sizeof(int)*nnodes);
    int *visited = malloc (sizeof(int)*nnodes);
    int len_shortest_subtour = nnodes;
    for(i=0;i<nnodes;i++)
        visited[i] = 0;
    while(1){
        for(i=0;i<nnodes;i++){
            subtour[i] = -1;
        }
        for(j=0;j<nnodes;j++){
            if (!visited[j]){ 
                cur = j;
                visited[cur] = 1;
                for(k=0;k<nnodes.length[cur];k++){
                    if(working_sol[nodes[cur][k]] ||
                        visited(edges[nodes[cur][k]].first_node) ||
                        !visited(edges[nodes[cur][k]].second_node)){
                        if(cur == edges[nodes[cur][k]].first_node{
                            next = edges[nodes[cur][k]].second.node;
                        }
                    }
                }
            }
        }
    }
}
if (i == len_shortest_subtour) {
    len_shortest_subtour = i;
    for (j = 0; j < nnodes; j++) {
        shortest_subtour[j] = subtour[j];
        subtour[j] = -1;
    }
}

i = 0;
while (shortest_subtour[i] != -1) {
    i++;
}

int *temp = realloc(shortest_subtour, sizeof(int) * i);
if (temp == NULL) {
    PRINTERROR("Error: failed to realloc memory in subtour detection\n");
    exit(1);
} else {
    *stour = temp;
    *size = i;
    free(subtour);
}
free(visited);
return 1;
}

/* debug functions, prints lots of infos about the status of the algorithm */
void print_current_status(){
  for(i=0;i<nedges;i++){
    PRINTVERBOSE("%d: %d, %d, %.2f, %d\n",
                i, edges[i].first_node, edges[i].second_node,
                edges[i].cost, working_sol[i]);
  }
  PRINTVERBOSE("\n");
  for(i=0;i<nnodes;i++){
    PRINTVERBOSE("%d: %d, %d, %.2f, %d\n",
                 i, nodes[i].first_node, nodes[i].second_node,
                 edges[nodes[i]].first_node, edges[nodes[i]].second_node);
  }
  PRINTVERBOSE("\n");
}

/* prints final tour */
void print_results(){
  FILE *out = fopen("results.dat","w");
  int i, j, k, cur, prev, next;
  cur = 0;
  fprintf(out, "%d\n", cur);
  for(i=0;i<nnodes-1;i++){
    prev = cur;
    cur = next;
    fprintf(out, "%d\n", cur);
    /* trovo il nuovo next */
    k = 0;
    while(k<nodes_length[cur] && !(best_sol[nodes[cur][k]]
                   && edges[nodes[cur][k]].first_node != prev
                   && edges[nodes[cur][k]].second_node != prev )){
      k++;
    }
    /* aggiorno next */
    if (edges[nodes[cur][k]].first_node == cur){
      next = edges[nodes[cur][k]].second_node;
    } else{
      next = edges[nodes[cur][k]].first_node;
    }
    fprintf(out, "%d\n", cur);
  }
  fclose(out);
}

/* evaluates hamming distance between two solutions */
int evaluate_hamming(int *first, int *second){
  int i, j;
  int return = 0;
  for(i=0;i<nedges;i++){
    if(first[i] != second[i]){ return++;
    }
  }
  return return;
}
int main(int argc, char **argv) {
    iteration = 0;
    tabu_change = 0;
    sol_update = 0;
    cpnodes = 0;
    slave.curr.iter = 1;
    int status;
    env = CPXopenCPLEX(&status);
    if (status) {
        PRINTERROR("Error: failed to initialize CPLEX environment\n");
        exit(1);
    }
    parseinput(argc, argv);
    if (use_z)
        status = CPXsetdblparam (env, CPX_PARAM_OBJDIF, 50000 - 2*nnodes -1);
        status = CPXsetintparam (env, CPX_PARAM_RINSHEUR, 50);
        intsolim = 0;
    else if (intsollim)
        status = CPXsetintparam (env, CPX_PARAM_INTSOLLIM, 1);
        status = CPXsetintparam (env, CPX_PARAM_NODELIM, maxnodecount.actual);
        if (use_slave)
            kruskal_setup(nedges, nnodes, edges, &sorted_index);
        else
            sorted_index = NULL;
    if (DEBUG>0) print_current_status();
    PRINTVERBOSE("Initial tour cost is: %.2f\n", get_cost(working_sol));
    if (!DEBUG)
        printf("Initial graph: %d nodes, %d edges\n", nnodes, nedges);
        printf("Initial tour cost: %.2f\n", get_cost(working_sol));
        printf("Starting minimum percent improvement: %3.2f\n", min_pct_impr);
        printf("Iteration limit: %d, Time limit: %.0fs\n", max_iter, max_time);
        printf("Slave is %s; CPLEX_PARAM_INTSOLLIM is set to %s\n", use_slave?"ON":"OFF", intsollim?"1":"default" );
    start_time = time(NULL);
    while(iteration<max_iter && ((double)(time(NULL)) - start_time<max_time)){
        iteration++;
        tabu_change = 0;
        sol_update = 0;
        cpnodes = 0;
        slave.curr.iter = 1;
        if (DEBUG && iteration%15==1){
            printf("Iter Master Slave Best Target PctImp Hamm\n");
            printf(" TabuSiz Nodes NodLim Elaps(s)\n");
        }
        PRINTVERBOSE("Iteration %d started.\n", iteration);
        PRINTVERBOSE("Calling solve.master...\n");
        solve.master();
        PRINTVERBOSE("solve.master done\n");
        if (DEBUG) print_current_status();
        PRINTVERBOSE("Calling solveslave...\n");
        solve.slave();
        PRINTVERBOSE("solve.slave done\n");
        if (DEBUG) print_current_status();
        PRINTVERBOSE("Iteration %d done\n", iteration);
        PRINTVERBOSE("Cost of current solution: %.2f\n", get_cost(working_sol));
    }
}
if (!DEBUG) print_log_row(iteration,
    get_cost(curr_master), get_cost(curr_slave), get_cost(best_sol),
    sol.update, curr.target, min pct.impr,
    evaluate_hamming(curr.master, curr.slave),
    tabu.size, cpxnodes, maxnodecount.actual,
    ((double)time(NULL)) - start_time);
}

PRINTVERBOSE("Stopping condition reached.

if (1)
    PRINTVERBOSE("Found a tour of cost less than the target
    PRINTVERBOSE("Saving the tour in "results.dat"
    print_results();
}

PRINTVERBOSE("Exiting...

}

CPXcloseCPLEX(&env);

void print_log_row(int iter, double m_cost, double s_cost, double best, int upd, double tar, double pctint, int hamm, int tabu.s, int cnodes, int maxnodes, double time_ela){
    if (m_cost)
        printf("%3d: %s %s %7.1f%s %7.1f %3.2f %s",
            iteration, "-", ",", best, upd="*: "", tar, pctint, " -");
    printf("%2d %4d %4d %4.0f
", tabu.size, cnodes, maxnodes, time_ela);
    else if (m_cost == s_cost)
        printf("%3d: %7.1f %s %7.1f%s %7.1f %3.2f %s",
            iteration, m_cost, ",", best, upd="*: "", tar, pctint, " -");
    printf("%2d %4d %4d %4.0f
", tabu.size, cnodes, maxnodes, time_ela);
    else{
        printf("%3d: %7.1f %7.1f %7.1f %7.1f %3.2f %5d",
            iteration, m_cost, s_cost, best, upd="*: "", tar, pctint, hamm);
        printf("%2d %4d %4d %4.0f
", tabu.size, cnodes, maxnodes, time_ela);
    }
}

void free_and_null(char **ptr){
    if (*ptr!=NULL)
        free(*ptr);
    *ptr = NULL;
}

B.2 KRUSKAL IMPLEMENTATION

B.2.1 kruskal.h

#ifndef EDGE
#define EDGE
/* lato del grafo */
typedef struct edge{
    double cost;       //costo del lato
    int first.node;    //primo nodo
    int second.node;   //secondo nodo
} edge;
#endif

typedef struct data{
    double cost;
    int index;
}
```c
#include <stdio.h>
#include <stdlib.h>
#include "kruskal.h"

#define MAX_LEVELS 1000

int quick_sort(data *arr, int elements) {
  data piv;
  int beg[MAX_LEVELS], end[MAX_LEVELS], i=0, L, R;
  beg[0]=0; end[0]=elements;
  while (i>=0) {
    L=beg[i]; R=end[i]-1;
    if (L<R) {
      piv=arr[L];
      if (i==MAX_LEVELS-1) return 1;
      while (L<R) {
        if (arr[R].cost>=piv.cost && L<R) R--;
        if (L<R) arr[L]=arr[R];
        if (L<R) arr[R]=arr[L];
      }
      arr[L]=piv; beg[i+1]=L+1; end[i]=end[i]; end[i++]=L;
    } else {
      i--;
    }
  }
  return 0;
}

/* sorts the list, basically just a mergesort */
int kruskal_setup(int nedges, int nnodes, edge *edges, int **sorted_index){
  data *temp_list = malloc(sizeof(data)*nedges);
  int i;
  for(i=0;i<nedges;i++){
    temp_list[i].cost = edges[i].cost;
    temp_list[i].index = i;
  }
  int status = quick_sort(&temp_list[0], nedges);
  if(status){
    printf("Error: quicksort failed\n");
    exit(1);
  }
  *sorted_index = malloc(sizeof(int)*nedges);
  for(i=0;i<nedges;i++){
    (*sorted_index)[i]=temp_list[i].index;
  }
  free(temp_list);
  return 0;
}

int kruskal(int nedges, int nnodes, edge *edges, int *sorted_index, int **tree){
  int i;
  int *prev = malloc(sizeof(int)*nnodes);
  *tree = malloc(sizeof(int)*nedges);
  for(i=0;i<nnodes;i++){
    prev[i]=0;
  }
  for(i=0;i<nedges;i++){
    (*tree)[i]=i;
  }
  return 0;
}
```

B.2.2 kruskal.c
B.3 INSTANCE GENERATION

B.3.1 script.sh

```bash
#!/bin/bash

#$1=nnods, $2=nedges, $3=outfile

./edgegen -k $1 -p ${3}_pointcloud_nontsp.dat
rm -f ${3}_pointcloud_tsp.dat ${3}_pointcloud_numbered.dat
awk '{printf "%d %s\n", NR-1, $0}' < ${3}_pointcloud_nontsp.dat
>>> ${3}_pointcloud_numbered.dat
echo "NAME : concorde_$1" >>> ${3}_pointcloud_tsp.dat
echo "COMMENT : $1 nodes, randomly generated by concorde" >>> ${3}_pointcloud_tsp.dat
echo "TYPE : TSP" >>> ${3}_pointcloud_tsp.dat
echo "DIMENSION : $1" >>> ${3}_pointcloud_tsp.dat
echo "EDGE_WEIGHT_TYPE : EUC_2D" >>> ${3}_pointcloud_tsp.dat
echo "NODE_COORD_SECTION" >>> ${3}_pointcloud_tsp.dat
tail -n +2 ${3}_pointcloud_numbered.dat >>> ${3}_pointcloud_tsp.dat
./edgegen -G -o ${3}_tour_toconvert.dat ${3}_pointcloud_tsp.dat
./converter ${3}_tour_toconvert.dat ${3}_tourfile.dat
./edgegen -e $2 -o ${3}_randomedges_tomerge.dat ${3}_pointcloud_tsp.dat
./merger ${3}_tour_toconvert.dat ${3}_randomedges_tomerge.dat $2 ${3}_edgefile.dat
rm ${3}_tour_toconvert.dat ${3}_randomedges_tomerge.dat ${3}_pointcloud_nontsp.dat
${3}_pointcloud_numbered.dat
exit
```

B.3.2 converter.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv){
```
FILE *in = fopen(argv[1], "r");
int nnodes, nedges;
 fscanf(in, "%d", &nnodes);
 fscanf(in, "%d", &nedges);
int *edges = malloc(sizeof(int)*2*nedges);
int i, j, dummy;
for(i=0;i<nedges;i++){
    fscanf(in, "%d", &edges[2*i]);
    fscanf(in, "%d", &edges[2*i+1]);
    fscanf(in, "%d", &dummy);
}
fclose(in);
FILE *out = fopen(argv[2], "w");
int cur, prev, next;
fprintf(out, "%d
", edges[0]);
cur = edges[0];
next = edges[1];
//printf("%d %d
", cur, next);
for(i=0;i<nedges-1;i++){
    prev=cur;
    cur=next;
    fprintf(out, "%d
", cur);
    for(j=0;j<nedges;j++){
    }
}
close(out);

# include <stdio.h>
# include <stdlib.h>

int main(int argc, char *argv){
    int max_edges = atoi(argv[3]);
    FILE *in = fopen(argv[1], "r");
    int nnodes, nedges;
    fscanf(in, "%d", &nnodes);
    fscanf(in, "%d", &nedges);
    int *edges = malloc(sizeof(int)*2*nedges);
    int *weights = malloc(sizeof(int)*nedges);
    int i, j, dummy;
    for(i=0;i<nedges;i++){
        fscanf(in, "%d", &edges[2*i]);
        fscanf(in, "%d", &weights[i]);
    }
fclose(in);
FILE *in2 = fopen(argv[2], "r");
int nnodes2, nedges2, nondupedges;

fscanf(in2, "%d", &nnodes2);
fscanf(in2, "%d", &nedges2);

nondupedges=0;

int *edges2 = malloc(sizeof(int)*2*nedges2);
int *weights2 = malloc(sizeof(int)*nedges2);

int first, second, weight, duplicate;
for(i=0;i<nedges2;i++){
    duplicate = 0;
    fscanf(in, "%d", &first);
fscanf(in, "%d", &second);
fscanf(in, "%d", &weight);
    /* elimino duplicati */
    for(j=0;j<nedges;j++)
        if(first==edges[2*j] && second==edges[2*j+1]) {duplicate = 1; break; }
    if (duplicate == 1) { continue;}
    else{
        edges2[2*nondupedges]=first;
        edges2[2*nondupedges+1]=second;
        weights2[nondupedges]=weight;
        nondupedges++;
    }
    if (nondupedges + nedges == max_edges) break;
}
}
fclose(in2);

FILE *out = fopen (argv[4], "w");
fprintf(out, "%d %d
", nnodes, nedges + nondupedges);
for(i=0;i<nedges;i++) fprintf(out, "%d %d %d
", edges[i*2],
edges[i*2+1], weights[i]);
for(i=0;i<nondupedges;i++) fprintf(out, "%d %d %d
", edges2[i*2],
edges2[i*2+1], weights2[i]);
fclose(out);
}
BIBLIOGRAPHY


[10] TSPLIB Specifications. url: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/DOC.PS.