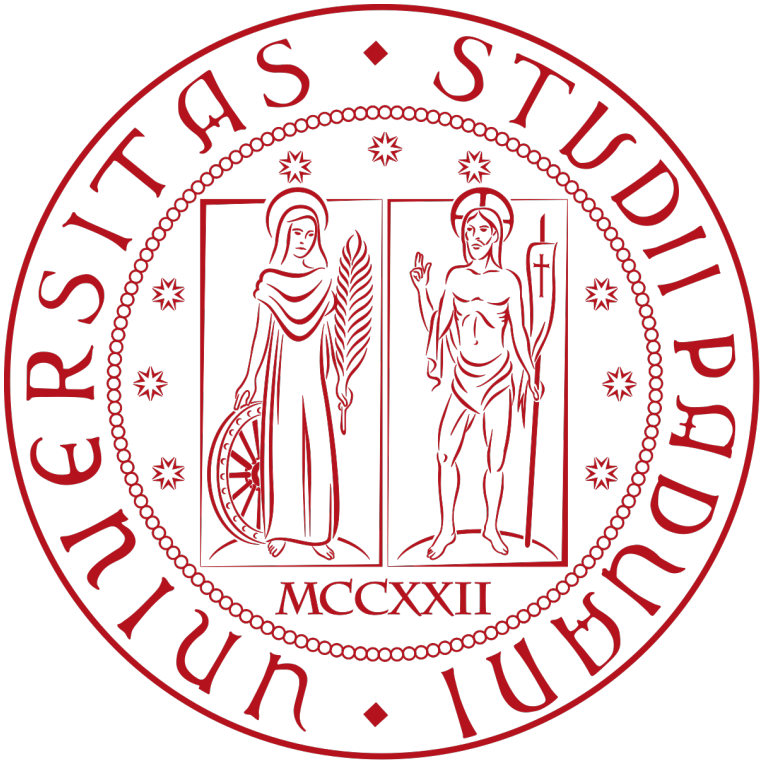
Progress Report

Operational Reseach 2

Thomas Porro

August 2021



School of Computer Engineering

Contents

1 The Problem

In this report we are going to describe, analyze and implement solutions for the Travelling Salesman Problem (from now on it will be called TSP).
Essentially the problem have this type of formulation is the following: "Given a list of cities and the distances between eachother, find the shortest path that connect all the cities". This could be "translated" to find the shortest (or the one with the lowest cost) hamiltonian circuit given an oriented graph $G = (V, A)$, where V are the cities of the problem and A are the paths that connect each city to the other ones.

Matematically the problem is the following. We start numbering all the cities that we have, from now on they will be called nodes. Each couples of nodes can be connected with an edge that is essentially the corresponding of the "street" in the problem, so we introduce a decisional variable x_{ij} where if the direct path from the node i to the node j is chosen its value is setted to 1, 0 otherwise

$$x_{ij} = \begin{cases} 1 & \text{if the arc } (i,j) \in A \text{ is chosen in the optimal solution} \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

Now we can describe the first formulation of the problem:

$$min \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{1.2a}$$

$$\sum_{(i,j)\delta^-(j)} x_{ij} = 1, \quad j \in V \tag{1.2b}$$

$$\sum_{(i,j)\delta^+(j)} x_{ij} = 1, \quad i \in V \tag{1.2c}$$

$$x_{ij} \geq 0 \text{ intero}, \quad (i,j) \in A \tag{1.2e}$$

In these equations we use the value c_{ij} as the cost of the path from the node i to the node j. The equations 1.2b and 1.2c lead to the fact that each node must have only one arc incoming and one arc outgoing.

2 Code setup

In order to implement the models to be solved we decided to use the common and powerful tool IBM CPLEX. Usually this software isn't free but due the academic usage it was made available for all the students that needed it.
CPLEX allow its user to decide which programming language to use between Python and C; in this project we used C.

To visualize the nodes and the paths found by our program we used a Gnuplot which is a command-line driven utility. Its code is protected byt copyright but the download is completely free. The sofware needs to be installed on the machine where the code is executed because Gnuplot is executed as a pipe: in particular before the plotting all the data is wrote to a file (according to the documentation) and than Gnuplot read and create the plot from that file.

To build the performance profiles in this report we used a python program written by D. Salvagnin (2019).

The first thing we did was build a parser capable of interpreting the TSP problems provided by the TSPLIB. The main data to save was the number of nodes, the coordinates of the nodes (they will be or relative coordinates that describe the position of the nodes or real world coordinates), the type of distance function to use (for example when are used real world coordinates the distance function need to consider the sphericity of the world). For each tsp problem we assume that the datafile contains a complete graph, so each node is directly connected with all the others nodes.

In my particular case all the project was developed on a linux machine with Ubuntu 20.04.

2.1 CPLEX environment

In order to work properly CPLEX needs to build his internal data structure to hold all the information needed to solve the problem. So the first thing to do is to create a pointer to the environment of Cplex through the `CPXopenCPLEX(&error)`: this function will return a pointer to the CPLEX environment that will be needed to use his entire library.
Once the environment is build CPLEX needs an additional data structure to hold the constraints of the optimization problem we want to solve, in order to use it we build an empty object using the function `CPXcreateprob(env, &error, "TSP")`: this will return a pointer to the problem where we will write all the constraints that we need.

```
CPXENVptr env = CPXopenCPLEX(&error);
CPXLPptr lp = CPXcreateprob(env, &error, "TSP");
```

2.1.1 Variable and constraint creation

Now that the enviroment is setted up we can start by creating our first variable. CPLEX handle the variable and constraints respectivetely as columns and rows; to unserstand it better let's make an example with a simple minization problem:

$$\begin{array}{lrcrcll} \min & x_1 & + & 3x_2 & + & x_3 & \\ & 2x_1 & & & - & x_3 & \leq 60 \\ & & & 4x_2 & + & 7x_3 & \geq 20 \end{array}$$

$$x_i \text{ integer}$$

Thius problem can be seen as a matrix composed by rows and columns, each column correspond to a variable, each row correspond to a constraint that the problem as to satisfy. Using the callable library of CPLEX we can build the rows and columns easily.
To build the variable we need to add a column to the problem, we can to this using the method `CPXnewcols`; this method allow us to create more than a single column per time through his parameters, in our case we built each variable singularly as shown in the code:

```
CPXnewcols(env, lp, 1, &obj, &lb, &ub, &variable_type, cname)
```

Here `env` and `lp` are respectively the pointers toi the environment and to the problem build in the section 2.1. The arguments `&obj, &lb, &ub, &variable_type, cname` are arrays that contains the values for the objective function, lower bound, upper bound, variable type and name for each variable. The last argument to analyze is the number 1, it represent the number of variables to add during this call to the CPLEX's library (as said before in this project we build the variables singularly).

At the end of this phase we have only the variable used in the objective function, now we will discuss how the constraints are implemented in the code. This time the process is more complicated that the previous one, in fact we need first to build an empty constraint, that means that on the left side there are no variables or numbers, than we change the coefficients for each variable. Let's explain it with and example:

```
CPXnewrows(env, lp, 1, &rhs, &sense, NULL, cname)
```

Here `env, lp, 1` and `cname` corresponds at the same variable as in the case of the columns, `&rhs` is the righthand side term for each constraint to be added to the problem object, `&sense` is the sense of each constraint to be added to the problem object, `NULL` contains the range values for the new constraints, for example it sets a maximum/minimum value to each constraint.
At this point we have something like this (example with random values):

$$\text{*empty*} \qquad \leq 30$$

So a part of the constraint is build, we need to decide what variables we want to put in this inequation. In order to do that we use the function `CPXchgcoef`.

Before seeing this method in detail we will explain how CPLEX reference to the rows and columns. In fact the name that we pass to the callable library are not directely used by CPLEX but are needed to the developer to understand if the problem is built correctely; for its function CPLEX simply enumerates each row and each column, so if we want to reference to a specific variable or constraint we need to now its number; by creating variables and

constraint we know exactely which is built before and whic after, so for example in the optimization problem shown at the begginning of the section to x_1 is given the number 1, to x_2 the number 2 and so on. The same thing happens to the constraints. Now we can fully understand the method previously introduced.

`CPXchgcoef` uses the already known arguments `env` and `lp`, in addition to them in needs the number of the constraint we want to modify, the number of the variable we want to change the coefficent and the coefficient for that variable. For example to build the first contraint of the already used optimization problem we need two calls to this method since we need to modify two coefficients of the variables:

```
// Here the first number is the id of the constraint
// the second is the id of the variable (1 for x1, 3 for x3)
// and the third one id the value of the coefficents
// (2 for x1 and -1 for x2)
CPXchgcoef(env, lp, 1, 1, 2.0);
CPXchgcoef(env, lp, 1, 3, -1.0);
```

Now that we have introduced the code setup and how the models are built we can start by examinating the variuos methods that we have use to solve the TSP optimization problem.

2.2 Solution and data management

In this section we will explain how we managed the data from the .tsp file and how we handled the solution obtained with CPLEX.

The first thing we build was a new struct that contains all useful data read from the .tsp file provided by the TSPLIB. Theese files in fact contains different infromations:

- NAME: the name of the file;
- COMMENT: some comment on the data;
- TYPE: the type of the problem;
- DIMENSION: the number of nodes that are in the file;
- EDGE_WEIGHT_TYPE: the typology of the distance between the nodes;
- NODE_COORD_SECTION: section, usally it goes to the end of the file, in which are described the nodes coordinates.

We decided to save only the fundamentals informations, in our case only the number of nodes, the node coordinates and the type of the distance functions between the nodes. All these data are saved into the struct during the parsing operation of the .tsp file.

After the solution is computed by CPLEX we can access it by using the method `CPXgetx`. This function will return the full array of variables, than by checking their values (1 if is true) we can see if some edge has been selected and if it is valid we save it to an array inside our struct.

Now we have our solution saved its time to check if it is a single tour or if there are sub-tours. To do that we use a simple trick, since the edges are connected with two nodes, we start from the first edge extracted and we go to one of the nodes, now we are there we check at what node this one is connected. If we already visited the next node we end our cycle and we continue the algorithm by checking all the unvisited nodes, if there is any.
To save each subtour we use and additional array that will contain the tour number at which each node belong, see figure 1 for a graphical representation.

3 Problem resolution

The first we have done in this project was testing that our program will actually work. As our first coding task we implemented the problem formulation that we described in the section 1, the problem reported has only the minimization function and it ensures that each node has only one edge incoming and only one edge outcoming. In this first part of this report, we will consider only the symmetric TSP so the resulting graph will be undirected. Until otherwise specified the following methods are testes with the file att48.tsp.

As we can see the solution that CPLEX found doesn't contain a single tour but a lot of sub-tours. The TSP problem requires that all the nodes must be connected with only one cycle, to do that we implemented different

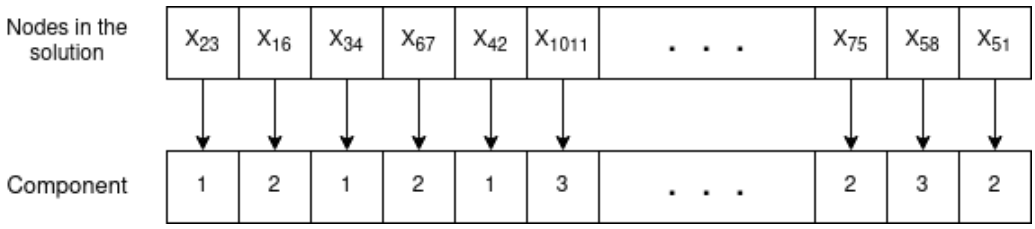


Figure 1: Structure of the solution. Each variable extracted from the solution provided by CPLEX is associated with a number that specify in which tour the variable belongs. In case of a solution with one tour all the array component will be filled with 1's.

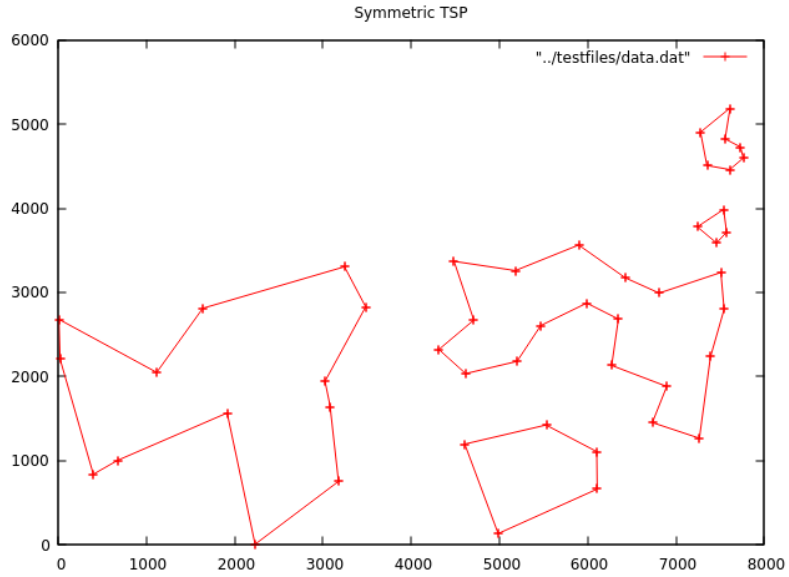


Figure 2: The image represent att48.tsp solved with the problem formulation showed in section 1

solutions that we will see in this report, the following subsection will explore the basic formulation of the loop method (also known as benders) and some compact models (MTZ and GG).
Before the introduction of this methods we need to introduce the most known formulation to the loops problem inside the TSP. Dantzig, Fulkerson and Johnson [**?**] introduced what it can be said to be one of the most efficient approach:

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1; \quad \forall S \subset \{2, \ldots, n\},\ |S| \geq 2 \tag{3.1}$$

Where S is the set of nodes that are in the solution. This equation implies that the number of edges in the solution cannot be greater that $|S| - 1$.

The main problem with this formula is that it produces a great number of constraint ($O(2^n)$) and it makes it unfeasable even for relatevely small amount of nodes.

3.1 Loop model

This method is easier to implement and also easier to understand. This process that we are going to describe is also known as benders decomposition and it is based on the principle of the divide-and-conquer.

The basic algorithm is the following: we give CPLEX the same model as before but this time we check if the solution has some sub-tour, if yes then we apply the loop method to each cycle found. Essentially it creates a new constraint and adds it to the problem, the constraint that is going to block the formation of the cycles in the next call of CPLEX.
Let's make an example, we have a cycle that is composed by five nodes and consequently by five edges that forms the

sub-tour; the constraint we build forces the software to connect those five edges with a maximum of four (number of nodes - 1) edges, this method must be applied to all the sub-tours found in the solution, then we call again CPLEX to solve the problem. If the problem has again some cycles inside we will apply again the loop method until the nodes are connected with only one cycle.

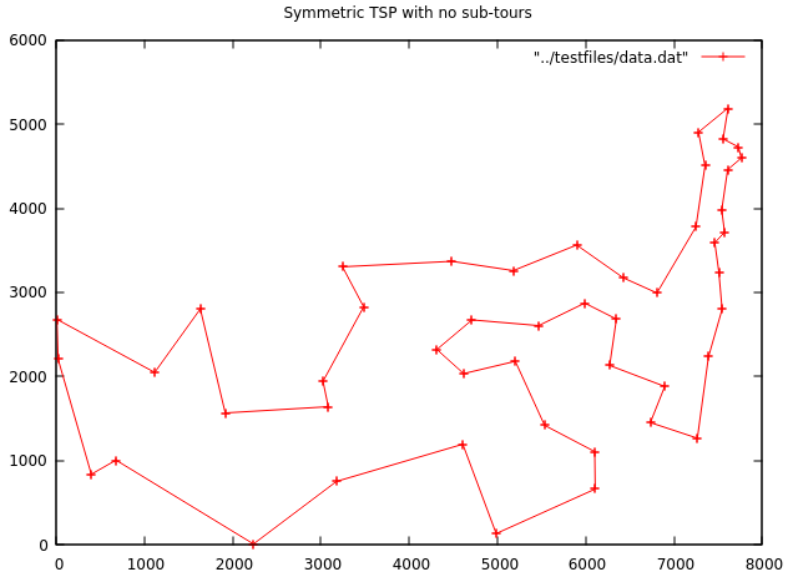


Figure 3: The image represent att48.tsp solved with the loop method described in section 3.1

As we can see this time the solution found have no sub-tours and in particular this one was found in circa 0.3 seconds with a total of 7 iterations where there were added 22 constraints in order to obtain this final solution.

3.1.1 Particularity of the variable creation in the symmetric TSP

In the setup section (2.1) we have seen how the variable and constraints are implemented inside cplex, we have also seen how we can refer to a variable when using the callable library, by simply using its "position" inside the variable array. It is immediately noticeable that the order in which we create them is really important and allow us to know exactly the position of each variable inside that array.

The way we managed the variables is the following: we think them as they were inside a matrix where the rows are the starting node and the columns are the arriving node. We can see an example in the figure 4.

In this way the is it really simple to find the number of the variable we want to reference, for example if we want to use the index of x_{34} we just need to compute this $3 * (\text{number of nodes}) + 4$ and thats it.

In the particular case of the simmetric TSP the path from i to j and j to i is the same so we don't need all the matrix that we have just showed. So in order to easily obtain the number we image a matrix as the one in the figure 5, this time all the cells that are in gray are not used since the useful values are saved in the other cells since x_{ij} have the same value than x_{ji}.

3.2 Miller-Ticker-Zemlin model

As said in section 3 the Miller-Ticker-Zemlin model is a compact model for the resolution of the TSP problem but, instead of the symmetric problem we have faced so far, it can be applied only to the asymmetric one. In fact in this case we will consider x_{ij} and x_{ji} two completely different paths with different lenghts (in our particular case the two path will have the same lenght). This model can delete all the sub-tours in a given problem: the current formulation of the constraints need the construction of an additional variable u_i which is used to denote the position of node i in the tour, here it is its formulation as described in (CITATION):

$$u_i - u_j + nx_{ij} \leq n - 1; \quad i, j \in \{2, \ldots, n\}, i \neq j \tag{3.2}$$

$$u_i \in \Re;\ i \in V : i > 1 \tag{3.3}$$

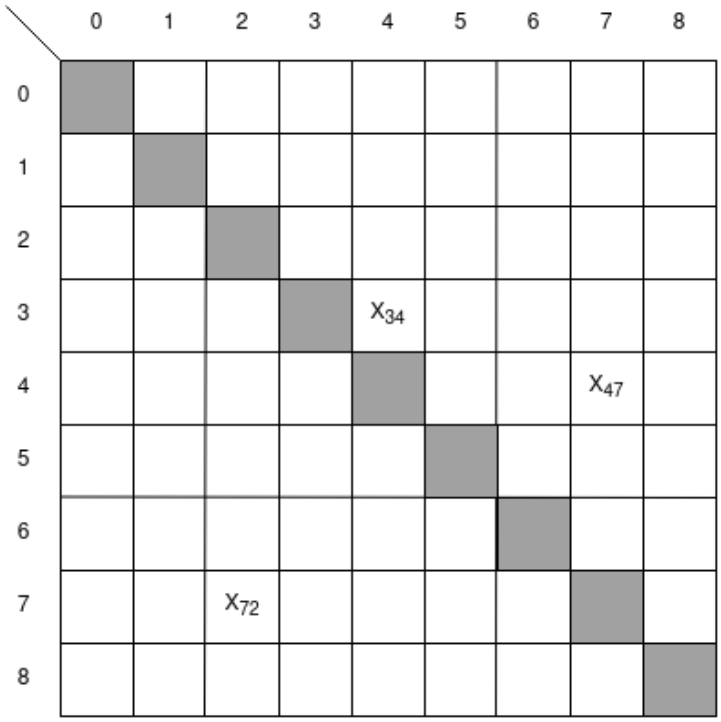


Figure 4: The image represent the matrix method we used to reference the variable inside CPLEX

Where V is the set of nodes. In the original paper the variables u_i were unrestricted but they can be restricted without compromising the solution of the asymmetric TSP, so we can rewrite 3.3 as:

$$u_1 = 1 \tag{3.4}$$

$$2 \leq u_i \leq n;\ i \in V :\ i > 1 \tag{3.5}$$

This formulation compared with [**?**] has a better complexity since the number of constraints used are $O(n^2)$ that is a lot better compered to $O(2^n)$ of the previous model.

3.2.1 Implementation of the model

Essentially the model implies that the nodes in the tour must be numbered in increasing order, from number 1, that is the start node, to n. In fact the formula 3.2 is simply a Big-M constraint that force the node j to have an u greater than node j if the arc x_{ij} is selected. Now that we explained what the 3.2 means we can rewrite it as a Big-M constraint in order to see the reference better:

$$u_j \geq u_i + 1 - M(1 - x_{ij}) \tag{3.6}$$

And than setting M to n and doing some basic algebra we can see the relation between them.

Now that we have explained the formula we can start explaining the three implementation of the the MTZ model:

- standard constraints: in this case all the constraints wrote in 3.2 are directely saved into the problem and than the problem is solved by CPLEX;
- lazy constraints: here the constraints are not always applied to the problem, as the name can suggest they are applied lazily, so CPLEX use them only when necessary or not before needed;
- indicator constraints: in the first two cases the Big-M trick is used to trigger a constraint when a particular variable assumes a predermined value, but this method (Big-M) is not always preferable since can behave in unstable ways. Thats why a good implementation of 3.2 is the usage of the indicator constraints: CPLEx automatically activate the constraint $u_j \leq u_i - 1$ when the x_ij assume the user passed value.

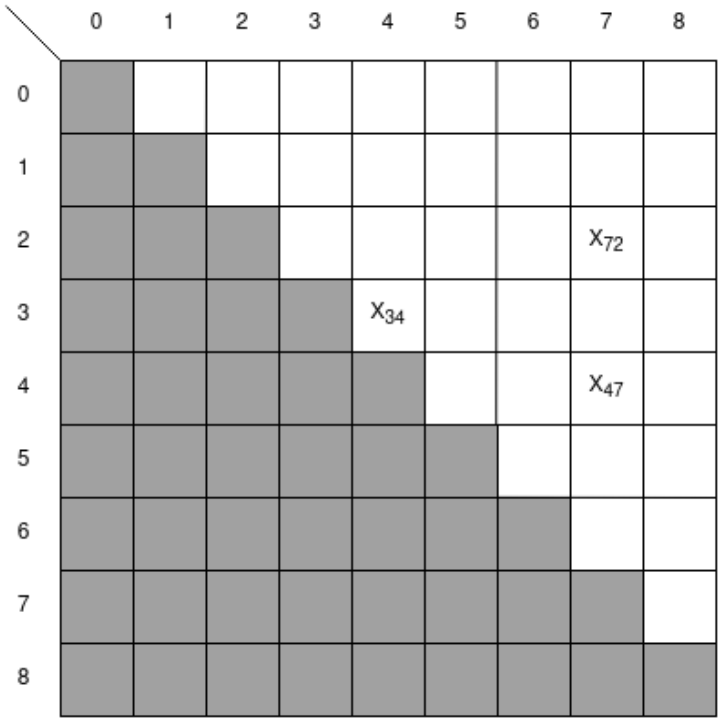


Figure 5: The image represent the matrix method we used to reference the variable inside CPLEX in the case of the simmetric TSP. In this case half of the matrix is not used since the values that were in the gray cells are the same of its "corrisponding white cell" since x_{ij} have the same value than x_{ji}. In fact the value x_{72} in the figure 4 is transposed into x_{27}.

3.3 Gavish and Gaves model

The second compact model we have implemented in our project is the one proposed by Gavish and Gaves (this model will be called from now on GG) and it is based on the single commodity flow: the arcs in fact will be considered as pipes. As the MTZ model also this one can only be applied to asymmetric problem instead of the symmetric one.
Based on [Put here gavish and gaves model] the additional constraints to put in the model are the following:

$$y_{ij} \leq (n-1)x_{ij} \tag{3.7}$$

$$\sum_{j;j\neq 1} y_{1j} = n-1 \tag{3.8}$$

$$\sum_{i;i\neq j} y_{ij} - \sum_{k;i\neq k} y_{jk} = 1 \tag{3.9}$$

Where the variable y_{ij} is a variable created to represent the flow of the arc. The equations 3.8 and 3.9 say that from the first node starts a flow of $n-1$ and than for each node of the path find the flow needs to lower its value by 1. The first constraint we introduced (3.7) is the one that allows the flow to take place only if the arc is selected.

As in section 3.2 we used the Big-M method to create the constraint 3.7, so its implementation is the same as before.

3.4 Performance Analysis

In this section we will analyze the performance of the models presented in the pages above. The test were run over 30 random generated datasets with the models: loop, MTZ, MTZ with lazy constraints, MTZ with indicator constraints and GG. We also imposed a timelimit of 1 hour, if some method does not provide the solution before the time expires it will be considere as a fail.

QUESTO POSSO METTERLO DOPO AVER MESSO IL PRIMO GRAFO DOVE LOOP STRAVINCE

From the first tests we observed that comparing the loop method with the other ones it is unuseful because the computation of the solution is quite instantaneous, so in the following graphs the loop method is removed.

4 Resolution with the callback method

In section 3 we introduce the fastest resolution method we've seen so far, but its implementation is a little bit tricky, and most of all not really "easthetic" CAMBIARE.
In this section we are going to explore the same loop method (so we will reject the solutions with some sub-tours) but this time we will implement its better version, also known as branch and cut.

4.1 The callbacks

Let's dive in in the main argument of this section: the callbacks.

As the name can suggest this type of function are not the classical one we have seen in section 2 and 3, in fact every method of the callable library we have used is applied or before or after the optimization computed by CPLEX. The callback functions give to the user significant additional capabilities because they allow to intervene during the optimization, this upgraded abilities permit the user to work with the internal data structure of CPLEX so the developer must be aware of what he is doing.

CPLEX has three different types of callbacks: informational callbacks, query/diagnostic callbacks, and control callbacks. The first ones gives the user additional information on the current optimization without affection the performance or interfering with the solution search space. The second ones access to more detailed informations respect to the informational callbacks but can affect the overall performance of the problem resolution; the query/-diagnostic callbacks are also incompatible with the dynamic search and deterministic parallel functions. The last ones are the one we are going to use and they allow the user to alter and customize how CPLEX perform the optimization.

In order to use them we need to declare their usage before calling the optimization function, we can do this through the function of the callable library:

```
CPXcallbacksetfunc(env, lp, CPX_CALLBACKCONTEXT_CANDIDATE, sec_callback, inst)
```

In this piece of code we can see the variables `env` and `lp` are the well known environment and problem of CPLEX. The third argument (`CPX_CALLBACKCONTEXT_CANDIDATE`) is what the API documentation call contextmask, usually this value must be one of the constants described the callable library; the value passed in the function tells CPLEX to call the callback function (the fourth argument, `sec_callback`) everytime it finds an integer solution of the problem. The last argument is the user data that are passed to the callback function.

4.2 Callback with an integer solution

The first callback we analyze are the one that will be called when CPLEX finds an integer solution. Once the function is called we can perform some operations with the informations we can retrieve from the problem. The operations we perform are similar to the ones descripted in section 2.2.

First we retrieve the solution found by CPLEX with the function `CPXcallbackgetcandidatepoint`, than we perform the same operations we have done when we described the `build_solution` function, so we find all the components of the solution. When there are more than one componentes, so some sub-tours are present, we add to the problem the reason why the solution is infeasable for us through the function `CPXcallbackrejectcandidate`. The meaning of the function is that the current solution found during the optimization is not feasable since it violetes the costraints that we pass as arguments (in fact are similar to the one used when we used the lazy constraints in the MTZ model).

4.3 Callback with a fractional solution

The management of this solutions are really more complex than the one previously analyzed since as all the values in the solution are not integer so we can't handle as we have done before (with the method of the components). Because the operations to perform in order to compute if the solution is composed by sub-tours we decided to use an external static library called Concorde. This library is one of the most powerful available in the market, it has even its own iOS application that allow to solve pretty complex instance of the problem.

The problem with this kind of solution its that can contains some nodes where the number of incidence edges is greater that 2 but their values are weighted in such way to respect anyway the degree costraint.

The methods appleid by Concorde to solve the fractional problem require that the graph is connected (using the function `CCcut_connect_components`), if its true we can call another function that allow us to implement the addition of a cut to the problem (performed by `CPXcallbackaddusercuts`).

4.4 Performance Analysis

SCIRVERE QUI LE PERFEORMANCE

5 Heuristics

Sometimes the complexity of the problem is too much to be solved by a machine (either for its power or the problem complexity). In that conditions we decided to renounce to obtain the optimal solution in favor of a fast solution, even if it is suboptimal.

The algorimths that implement this concept are called heuristics and their strategies are as simple as they are quick.
We are going to explore some of this heuristics, in particular:

- greedy;
- extra-mileage;
- k-opt;
- variable neighboorhood search.

5.1 Greedy algorithm

The greedy algorithm is the easiest to understand. It starts from a random node and than it execute a greedy choice by choosing the arc with the lower cost between the ones that are not connect to a node that is already in the solution. The algorithm performs this operation until all the nodes are in the solution.

This simple idea as a negative effect, in fact the algorithm prefers the nodes that are close to each other and leaves alone the node that are more distant; since all the nodes must be in the solution even the fartest node needs to be chosen at some point, by delaying their choice the greedy method is forced to connect theese nodes provoking the usage of really long edges that surely are not in the optimal solution.

In our code we decide to perform a particular implementation, we start execution the algorithm starting from each node and than we choose the best solution found through the iterations. The problem with this approach is that we slow down the procedd by a factor n because we need to search the best solution among all the nodes used as start.

5.2 Extra-mileage algorithm

This algorithm is more complex respect to the previous one but in favor of a better solution.

The idea in which the extra-mileage is the following: we start connecting the fartest nodes with both edges (like a cycle), than we search for the closest node to the edges, once it has been identified the edge is substitute with a new couple that insert the the node found in the solution. Let's explain it with the example shown in figure 7.

We start connecting with a cycle the fartest nodes (in this case c and d), than we search the node that is closest to theese edges (c), once is found we substitute the edge with two more that allow the new node to enter the solution (in this case in order to allow c to enter the solution one of the edges that connect b and c is removed and the edge x_{bc} and x_{cd} are added). This process is repeated until the solution is formed.

We can see the result of the algorithm in figure 8. Respect to the greedy algorithm we can see that is more ordered with less crossed edges.

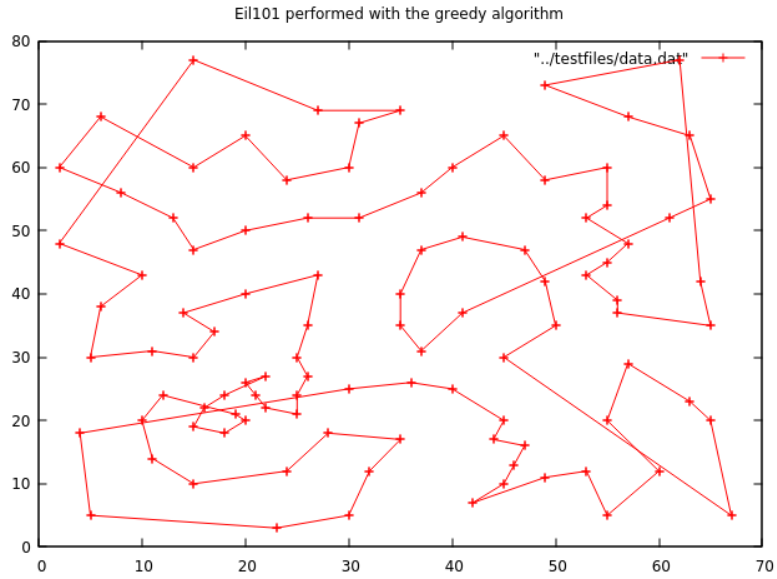
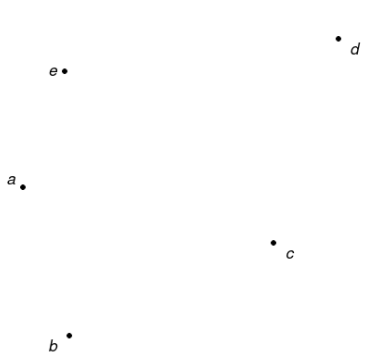
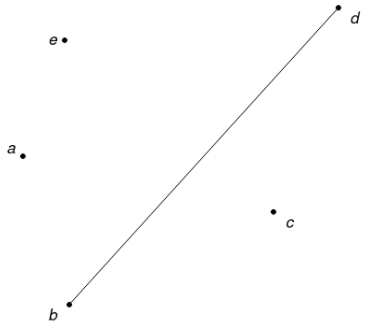


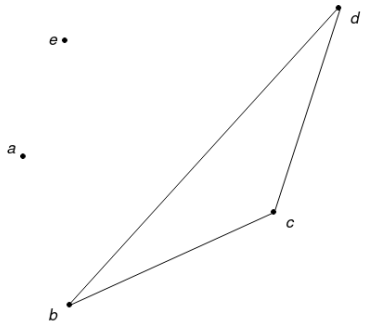
Figure 6: In this figure we can see the application to eil101.tsp of the greedy algorithm. In particular we can see the long edges that are chosen since the procedure prefers the nodes that are close to each other.



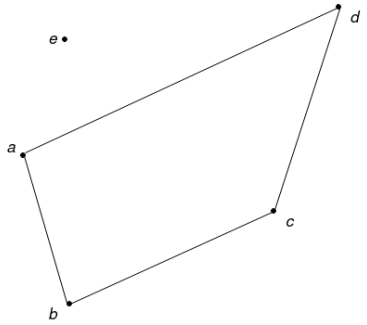
(a) No edges



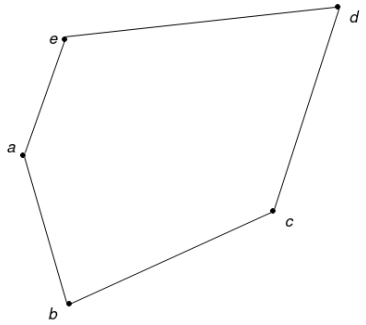
(b) First two edges added



(c) One edge is substituted with other two connecting one more node



(d) Added one more node



(e) All nodes are connected

Figure 7: In this image we can see the process done by extra-mileage to find the solution.

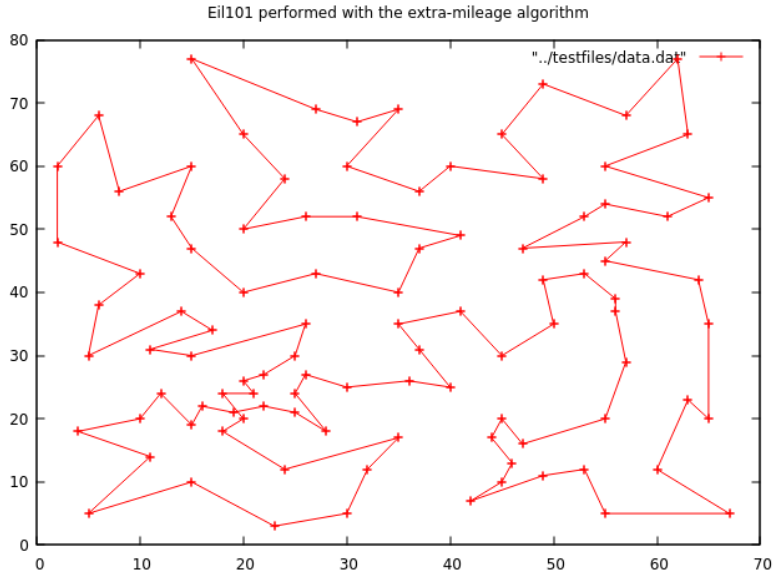


Figure 8: In this figure we can see the result of the application of the extra-mileage algortihm to eil101.tsp

5.3 k-*opt* refining

This section is not about an algorithm to solve the TSP problem but is about to refine (so improve) a solution that is already been found.
The process is based on the triangle inequality, in fact we are going to remove all the edges that cross each other. We can see an example of this operation in the image **??**, where the two edges that cross each other are sostitute by other to edges.

From a theoretical point of view the k-opt operation is a local search on the current solution, is not guaranteed to reach an optimal solution, but the objective value of the solution is going to always improve (if possible), in the worst case the solution will not change.

In our case we implemented two k-opt algorithms, the 2-opt and the 3-opt. We have decided to do that since as the k grows the complexity grows with it. Just the implementation of the 3-opt algorithm is quite expensive from a computational point of view.

In the figure 9 we can see the differences between the 2-opt and 3-opt refining. In this images we applied theese algorithms starting from the solution provided by the greedy method to kroA200.tsp.
In the 2-opt application we started from a solution cost of 34543.0 and than by the refining algorithm we reached the value 30189.0 with a very little computational time, in fact it needed only 0.65 seconds to reach the final solution.
In the 3-opt application we started from a solution cost of 34543.0 and than by the refining algorithm we reached the value 25379.0 in 77.37 seconds.

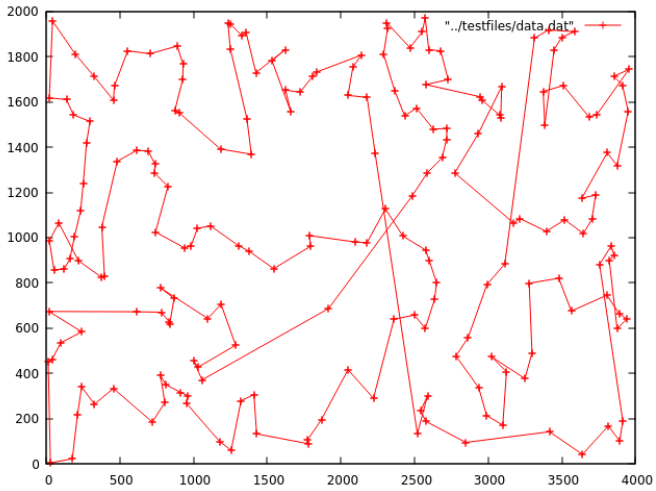
5.4 Heuristics based on the branch and cut

This section will explore the application of two heuristrics in the branch and cut method that we have seen in section 4. Using these methods we can allow to the branch and cut method to solve even big instances but in a way that they probably never reach the optimal solution. The starting point of this section is basically the reduction of the search space.

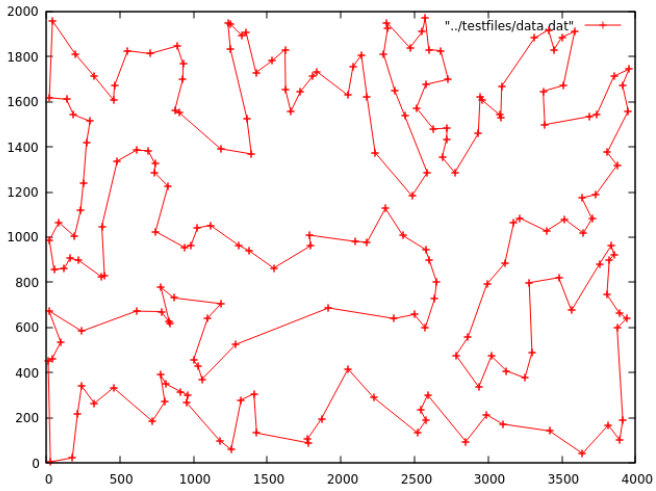
5.4.1 Hard-fixing

The method of the hard-fixing expect to block the value of some variables in the problem in order to get the search space reduce in order to compute more easily a solution for it. The choice of which variable to block is due the user, but eventually can be completely random.

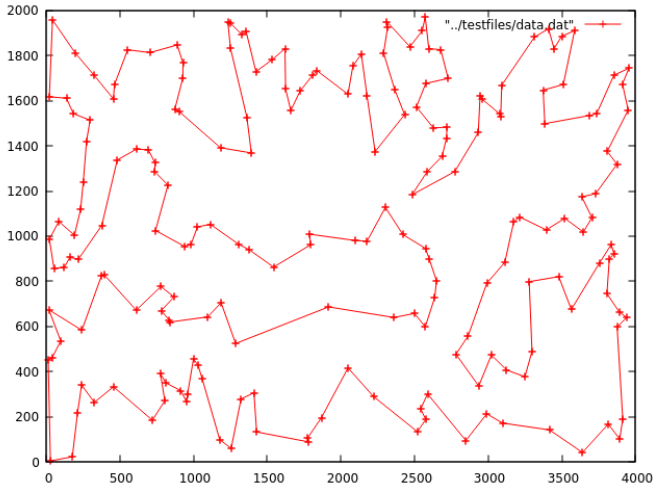
The most important setting for this method is the starting solution, in fact since we are blocking the varibles so if the satarting solution is completely wrong the process of improving it will be more difficult. The process to



(a) The instance kroA200.tsp solved with the greedy algorithm



(b) Greedy + 2-opt refining



(c) Greedy + 3-opt refining

Figure 9: Comparison between the 2-opt and 3-opt refining

retrieve a final solution is pretty easy: we start from that solution, we apply the branch and cut method for a predermined amount of time (or until it finds the optimal solution in the seach subspace), at the end of it we check if the new solution found is actually better than the previous one, if not we reduce the number of blocked variables and repeat the process.

In our implementation we decided to put a percentage on the number of blocked variables, in particular we started from a value of 90% and than for each time the solution found was not better than the previous one we reduced it by a 20%. The solution chosen as start was simply computed by the branch and cut method with the same time limit setted in the algorithm, in this way the starting point should never be a worse one.

5.4.2 Soft-fixing