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:

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

$$x_i \text{intero}$$

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.

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.

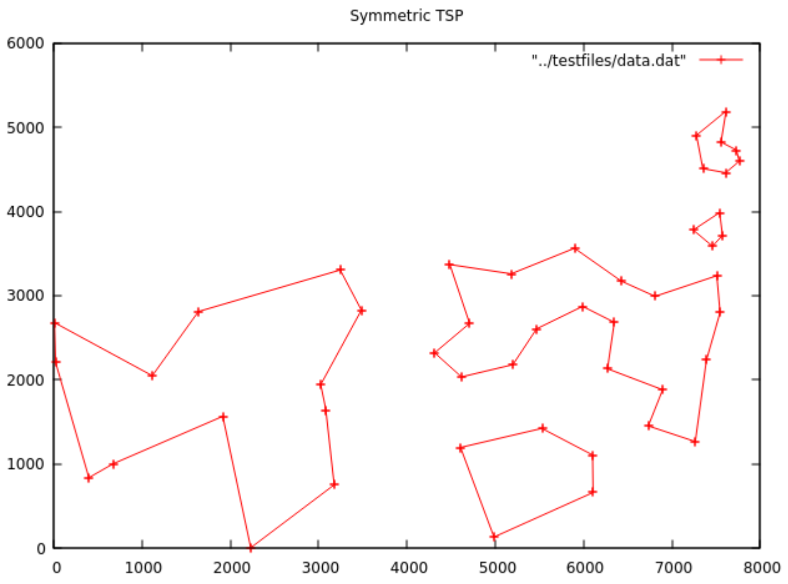

Figure 1: The image represent att48.tsp solved with the problem formulation showed in section 1

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 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.

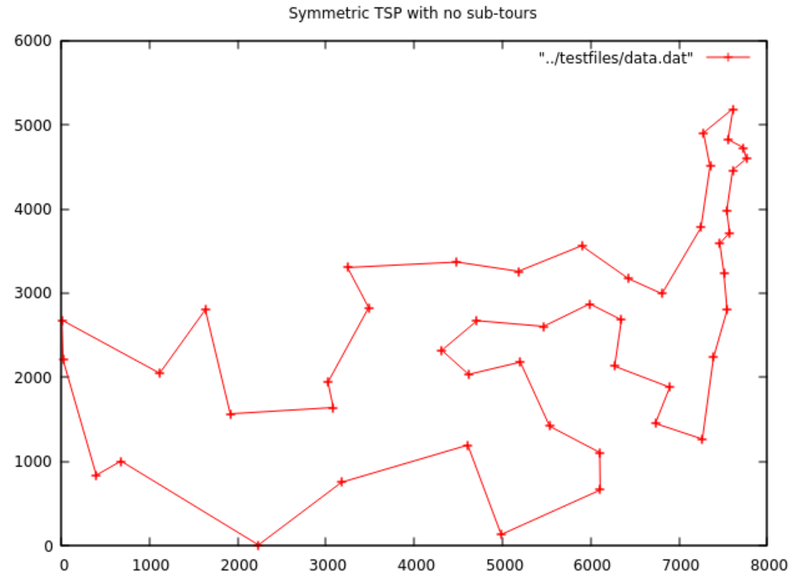

Figure 2: The image represent att48.tsp solved with the loop method described in section 3.1

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 3.

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 4, 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}.

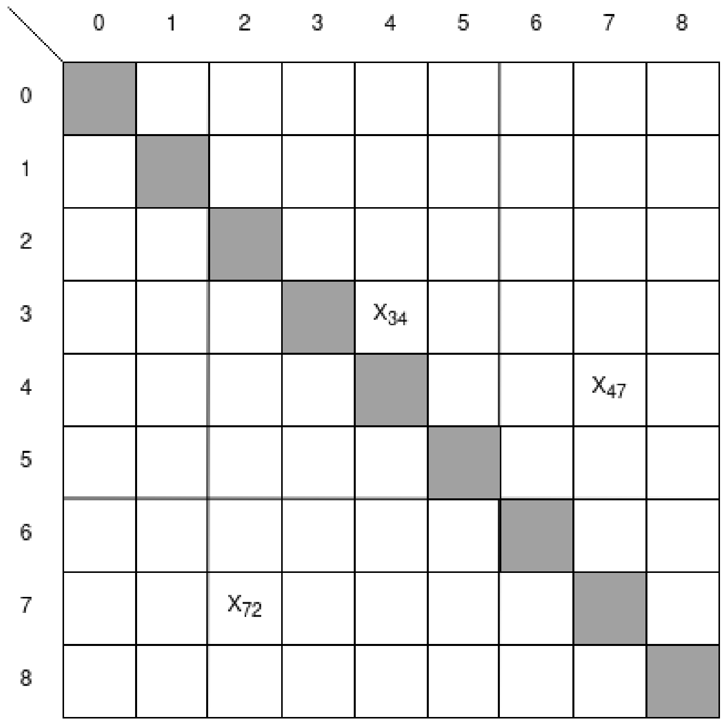

Figure 3: The image represent the matrix method we used to reference the variable inside CPLEX

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):

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

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

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.

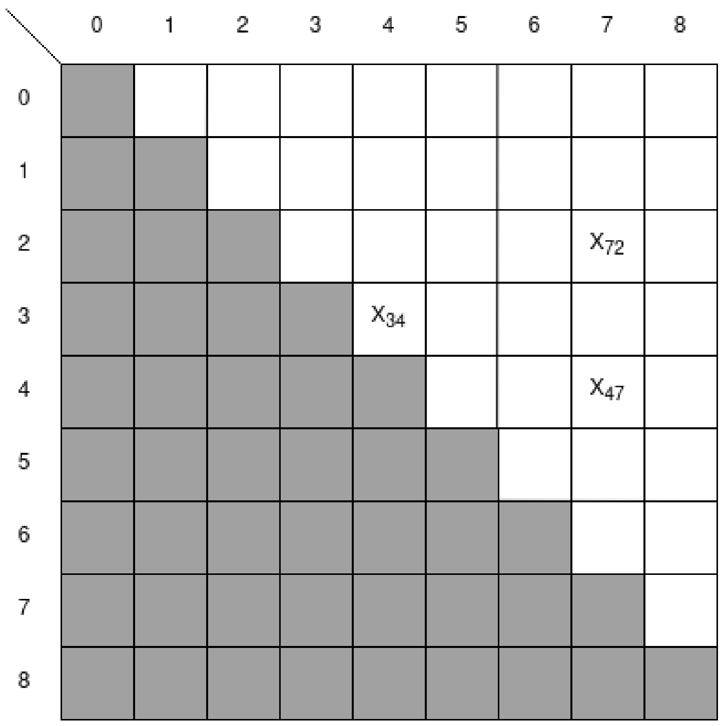

Figure 4: The image represent the matrix method we used to reference the variable inside CPLEX in the case of the simmetric TSP. In this case half of the matrix is not used since the values that were in the gray cells are the same of its "corrisponding white cell" since x_{ij} have the same value than x_{ji}. In fact the value x_{72} in the figure 3 is transposed into x_{27}.

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 $x_i j$ assume the user passed value.

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.