University of Padua
Master's Degree in Computer Science
School of Science - Department of Mathematics
METHODS AND MODELS FOR COMBINATORIAL OPTIMIZATION
Academic Year 2023/2024

# MEMOCO REPORT

—

## LAB EXERCISES

**Submitted by: Marco Brugin**

Submission date 11 novembre 2024

Matriculation number: 2122864

# Indice

# 1 Part I

## 1.1 Introduction problem

The problem illustrated in the paper require to determine what is the sequence of all that minimizing the total drilling time to build electric panels, given that the time needed for make a hole is same and constant.
This exercise can be modeling as TSP ($TravellingSalesmanProblem$), a problem compute on graphs $G = (N, A)$ where $N$ are the nodes that indicate the hole to be done and A, given $i$ *and* $j \in N$ are the arches represent the time to move drill between these two hole.
To resolve it in graph model, it is enough to compute a minimum cost Hamiltonian cycle (Figure 1).

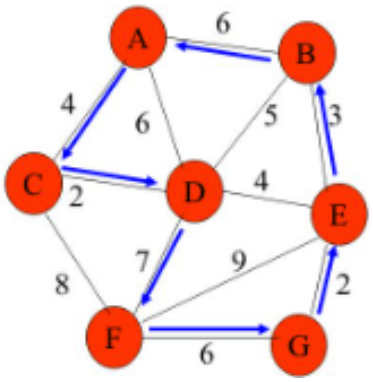


Figura 1: Hamiltonian cycle

## 1.2 Implementation

In the file *main.cpp* is been implemented the model that represent this problem by using Cplex API for C++, proving by prof Prof. Luigi De Giovanni through the file *cpxmacro.h*. Also to compile is been used the automation tool *Make* for also specified the dependecies of library. The structure of *main.cpp* is the following:

1. At the beginning after the included library, it is necessary to develop a function, called *setupLP* to setup the enviroment where the model will be solve and to create it.
   a) Firstable to separate the model from the data file i read all datas by $ifstream\ input$ of C++.

```
ifstream input(file);
input >> N;
double C[N*N];
for(int i=0;i<N;i++){
        for(int j=0;j<N;j++){
                input >> C[i*N+j];
        }
}
```

   b) Create the name for each node: i give the name n at the node in the position n.
   c) As many variables will be needed, to keeping trace all of them, i create a mapping for all variable x and y called $map_x$ and $map_y$ respectively. In this duch that all variables have two indexes it has been neccessary to used a vector inside anotehr one in this way.

```
//mapping
vector<vector<int>> map_x;
map_x.resize(N);
for (int i=0; i<N; i++ ) {
        map_x[i].resize(N);
        for ( int j=0; j<N; j++ ) {
            map_x[i][j] = -1;
                }
}
vector<vector<int>> map_y;
map_y.resize(N);
for (int i=0; i<N; i++ ) {
        map_y[i].resize(N);
        for ( int j=0; j<N; j++ ) {
                map_y[i][j] = -1;
                }
        }
```

d) Create all variable x by the function $CPXnewcols$ and for all of them i set the respectivelly lower bound, upper bound, the type, the name and value the coefficient that they have in the objective function.

e) The some was done for the y variables, paying attention to the x variables are *Continous* instead y variables are binaries.

f) Create the three blocks of equal constraint required by model.

g) Create an addictional block of constraint for the activation of y binary variables by big-M constant (here M = $(\left|N^1\right| - 1)$).

2. Execute the the model by $CPXmipopt$ and compute the optimal value of the objective function.

3. Extract the value obtained for each variable by $CPXgetx$.

4. Extract the name of each variables by $CPXgetcolname$ and print only that have a value greater of zero.

5. Compute the time necessary to compute all.

6. Print all the result.

## 1.3 Result obtained

The model has been tested in local, inside a virtual machine by this configuration:

- Operating system Ubuntu 64 bit
- 5 CPU
- 4580 MB memory.

As saying in the classes, for obtained a better realistic statics for every number of nodes taken into consideration are been made three different tests with different range of weights that are been assigned dinamically.
Remark that these tests are not exhaustive such that do not cover all the possible cases.
For the execution is necessary the following command.

```
make
./main name_folder_instance/name_of_instance
```

In the following table are summaries the obtained results time for every set of nodes and for every range of weights, for any case is taken the better instance result (Table 1).

| | **Time [s]** | | | |
|---|---|---|---|---|
| **N** | **Range [1,10]** | **Range [1,100]** | **Range [1,1000]** | **Range [1,10000]** |
| 10 | 0,99 | 0,88 | 1,2 | 0,87 |
| 20 | 2,2 | 2,3 | 1,9 | 2,3 |
| 50 | 17 | 23 | 17,9 | 22,4 |
| 100 | Infinity | Infinity | Infinity | Infinity |

Tabella 1: Table of result

## 1.4 Conclusion

In conclusion, analysing the result obtaining, it can say that in the compute of optimal solution we don't obtained a good performance, sometimes we can not obatain a solution. But it is necessary to consider, the problem in exam is **NP-HARD** problem and the instances created about there are not sufficient to make predictions on the real performance that a MILP can have with it, because there is not regularity relation between the performance, the number of nodes and weights. Despite this, on average it can be said that performance decreases as the number of nodes increases.

# 2 Part II

## 2.1 Introduction problem

The second part of the exercise require to implement an ad-hoc optimization algorithm, as an alternative to solving the implemented model with Cplex, by using one of the several meta-heuristic methods seen in classroom.

## 2.2 Implementation

For solve this second part i decided to implement one a meta-heuristic method using as base local-search: Tabu Search.
To implement is necessary to set some parameters:

- **Max_multistart counter**: the algorithm is executed several times, each time starting from a different initial solution. The initial solution is provided by the **Christofides** algorithm, the subsequent ones are obtained by randomizing the initial solution.
- **Max_iterations counter**: represents the maximum number of iterations that the algorithm will perform before stopping. This parameter is essential to control the running time of the algorithm and to prevent infinite loops, especially when the algorithm cannot find an optimal solution in a reasonable time. In this case it is used as a **stopping criterion**.
- **TabuLength**: represents the number of moves or solutions that are temporarily considered "taboo" or prohibited during the process of finding the optimal solution. If TabuLength is too short, prohibited moves remain in the tabu list for a small number of iterations. This could lead the algorithm to quickly re-explore solutions already visited, increasing the risk of loops and reducing the overall effectiveness of the search. If TabuLength is too long, moves remain prohibited for an extended period. This may overly limit exploration, as the algorithm may be forced to explore less promising solutions due to the numerous restrictions imposed by the tabu list. The optimal length depends on the specific problem and must be evaluated through experimentation.
- **DiversificationTabuLength**: is a technique used to broaden the exploration of the solution space, trying to prevent the algorithm from focusing too long on a certain region of the solution space.
- **MaxNonImproving**: is a parameter used to control the maximum number of consecutive iterations in which no improvement of the current solution is observed.
- **MaxDiversifications**: it is a parameter used to control the maximum number of diversification phases that the algorithm can perform during the search process. Avoid excessive exposure and prevent stagnation.

- **TimeLimit**: rappresent a maximum time limit for the execution of the algorithm, in this case it is used as a further stopping criterion.

## 2.3 Christofides

The Christofides algorithm is a well-known approximation algorithm for solving the Traveling Salesman Problem (TSP). The Christofides algorithm guarantees a solution that is at most 1.5 times the optimal tour length.
It procede in these three steps:

1. Compute **Minimum Spanning Tree (MST)**: the algorithm begins by constructing **Minimum Spanning Tree (MST)** (MST) of the graph. An MST is a subgraph that connects all the vertices (cities) together with the minimum possible total edge weight (distance), without forming any cycles.

2. **Minimum Weight Perfect Matching** the algorithm finds a minimum weight perfect matching on the vertices of odd degree in the MST. A perfect matching is a set of edges such that every vertex in the graph is incident to exactly one edge, and in this context, the matching is done only for those vertices that have an odd degree in the MST. This step ensures that the graph can be turned into an Eulerian graph (a graph where every vertex has an even degree).

3. **Eulerian Circuit and Hamiltonian Cycle**: the algorithm then combines the edges of the MST and the minimum weight perfect matching to create an Eulerian graph. In this graph, it is possible to find an Eulerian circuit (a path that visits every edge exactly once). Finally, this circuit is transformed into a Hamiltonian cycle (the solution to the TSP) by bypassing any repeated vertices.

In addition to these properties I used this algorithm to quickly finish an initial solution to Tabu search.
Furthermore, using this technique, an upper bound was obtained for the optimal solution to the problem.
Finally, a vector will be used to store the solution obtained.

## 2.4 Tabu Search Implementation

In the file *main.cpp* is been implemented the **Tabu Search** using as initial solution the solution provide by Christofides library.
The Structure and some charateristics of it are the following:

- Firstable it is useful to verify if there are all necessary files, in particular if it is presented the istance file of the problem.
- Declare the parameters illustrated before. The way of callibration is large illustrated in the following section.
- Read the data file i read all datas by if stream input of C++. For scalability I suppose that the graph is complete and i create a vector for wweights of size of numNodes * numNodes, where numNodes is the number of nodes.

```
int numNodes;
        instanceFile >> numNodes;
        vector<double> weights(numNodes * numNodes);
        for (double& w : weights) {
            instanceFile >> w;
        }
        instanceFile.close();
```

- Taken the initial solution from the **Christofides**.
- Declare the Tabu List.
- Compute the current value of objective function.

```
    double get_Objective_function(const vector<int>&
    tour, const vector<double>& weights) {
    double totalDistance = 0;
    int nodes = tour.size();
    for (int i = 0; i < nodes; ++i) {
        totalDistance += weights[tour[i] * nodes + tour[(i + 1) % nodes]];
    }
    return totalDistance;
}
```

- After it may be happened two different scenaio:
  1. If it is the first start i compute the **Two_opt** and **Three_opt** and recompute the solution.

```
bool diversify = false;
 while (iter < maxIterations) {
    vector<int> nextSolution;
    if (!diversify) {
        nextSolution = Two_opt(currentSolution, weights, tabuList);
        if (tabuList.size() >= tabuLength) tabuList.erase(tabuList.begin());
```

```
        }
    else {
    nextSolution = Three_opt(currentSolution, weights, tabuList);
    if (tabuList.size() >=
    diversificationTabuLength) tabuList.erase(tabuList.begin());
    diversificationCount++;
     }
    tabuList.push_back(nextSolution);
    double nextCost = get_Objective_function(nextSolution, weights);
```

2. Compute another solution by randoming the previous one and compute **Two_opt** and **Three_opt** and recompute the solution.

```
shuffle(currentSolution.begin(),currentSolution.end(),
default_random_engine(seed));
```

- Verify if the founded solution is best than the previous one and otherwise update the no improving counter

```
if (nextCost < localBestCost) {
            localBest = nextSolution;
            localBestCost = nextCost;
            nonImproving = 0;
        } else {
            nonImproving++;
        }
```

- Compute the time taken and verify if exceed the time limit.

```
elapsedTime = (double)(currentTime.tv_sec - startTime.tv_sec) +
(double)(currentTime.tv_usec - startTime.tv_usec) / 1000000;
  if (elapsedTime > timeLimit) {
      cout << "Time limit exceeded" << endl;
            break;
        }
```

- Verify what is the best restart and print all

```
if (localBestCost < bestOverall) {
            bestSolution = localBest;
            bestOverall = localBestCost;
            bestRestart = restarts;
        }

        restarts++;
    }

    cout << "Best solution found: ";
    printSol(bestSolution);
    cout << "Best solution cost: " << bestOverall << endl;
    cout << "Best solution restart: " << bestRestart << endl;
```

## 2.5 Result obtained

The model has been tested in local, inside a virtual machine by this configuration:

- Operating system Ubuntu 64 bit
- 5 CPU
- 4580 MB memory.

### 2.5.1 Parameters Selection

For obtained a better solution, i dived the instaces in two set:

1. The training set to calibrate the parameters, for it i used the smallest one (10 and 20 nodes).
2. The evalutation set i used the biggest one ( 50 and 100 nodes when possible).
3. I used the instaces just solve in the Part1.

The parameters obtained are the following:

- maxRestarts = 8;
- maxIterations = 90;
- tabuLength = 7;
- diversificationTabuLength = 20;
- maxNonImproving = 3000;
- maxDiversifications = 30;
- timeLimit = 80.

For the execution is necessary the following command.

```
make
./main name_folder_instance/name_of_instance
```

The result obtained in the training set are the following.

| **Instaces** | **user_time [s]** | **Optimal** | **Heuristic Solution** |
|---|---|---|---|
| tsp10_10-1 | 0,30 | 554,615 | 554,615 |
| tsp10_10-2 | 0,40 | 1082,02 | 1082,02 |
| tsp10_10-3 | 0,34 | 507,594 | 507,594 |
| tsp10_100-1 | 0,20 | 878,237 | 878,237 |
| tsp10_100-2 | 0,23 | 598,888 | 598,888 |
| tsp10_100-3 | 0,20 | 902,089 | 902,089 |
| tsp10_1000-1 | 0,27 | 930,476 | 930,476 |
| tsp10_1000-2 | 0,25 | 384,352 | 384,352 |
| tsp10_1000-3 | 0,34 | 528,914 | 528,914 |
| tsp10_10000-1 | 0,32 | 851,625 | 851,625 |
| tsp10_10000-2 | 0,23 | 1079,47 | 1079,47 |
| tsp10_10000-3 | 0,14 | 890,404 | 890,404 |

Tabella 2: Table of result for Ten Nodes

| **Instaces** | **user_time [s]** | **Optimal** | **Heuristic Solution** |
|---|---|---|---|
| tsp20_10-1 | 0,40 | 1007,77 | 1007,77 |
| tsp20_10-2 | 0,30 | 1016,84 | 1016,84 |
| tsp20_10-3 | 0,3 | 993,678 | 993,678 |
| tsp20_100-1 | 0,38 | 680,892 | 680,892 |
| tsp20_100-2 | 0,40 | 620,46 | 620,46 |
| tsp20_100-3 | 0,48 | 952,076 | 952,076 |
| tsp20_1000-1 | 0,51 | 903,753 | 903,753 |
| tsp20_1000-2 | 0,53 | 754,662 | 754,662 |
| tsp20_1000-3 | 0,3 | 838,491 | 838,491 |
| tsp20_10000-1 | 0,48 | 931,009 | 931,009 |
| tsp20_10000-2 | 0,4 | 634,234 | 634,234 |
| tsp20_10000-3 | 0,7 | 934,947 | 934,947 |

Tabella 3: Table of result for Twenty Nodes

## 2.6 Conclusion

In conclusion, analysing the result obtaining, it can say that the parameters are well calibrated because the error that we obtain is approximately the 0 % from the optimal solution. Also in this training set the performance achieve are better than the optimal solution compute with an exact method with **Cplex**.

# 3 Evaluation and Conclusion

## 3.1 Evaluation

After selection of parameters with set of 50 nodes is been done the comparison between the objective function value obtained and compute the relative error the optimal value and the **Tabu Search** algorithm. I remark that it is not possible to use the set of 100 nodes because,as can be seen in the Table 1, these values are not available.

| **Instaces** | **Optimal** | **Heuristic Solution** | **Relative percentage error** |
|---|---|---|---|
| tsp50_10-1 | 1295,59 | 1295,59 | 0 |
| tsp50_10-2 | 1607,76 | 1609,73 | 0,2 |
| tsp50_10-3 | 1359,76 | 1359,76 | 0 |
| tsp50_100-1 | 1223,94 | 1223,94 | 0 |
| tsp50_100-2 | 1427,65 | 1427,94 | 0,02 |
| tsp50_100-3 | 1415,61 | 1415,83 | 0,015 |
| tsp50_1000-1 | 1305,74 | 1305,74 | 0 |
| tsp50_1000-2 | 1553,63 | 1553,63 | 0 |
| tsp50_1000-3 | 1522,19 | 1522,19 | 0 |
| tsp50_10000-1 | 1271,49 | 1272,44 | 0,004 |
| tsp50_10000-2 | 1496,34 | 1496,34 | 0 |
| tsp50_10000-3 | 1374,67 | 1374,77 | 0,007 |

Tabella 4: Comparison between Optimal solution and Heuristic Solution

By the esperiment execution it can see that until 50 nodes, using the illustrated before the **Tabu Search** approximates the optimal solution to the problem well.

## 3.2 Final Conclusion

After having solved the drilling problem both with a method that ensures and by methodizing an apposition of the same, it can be highlighted that up to 80 nodes both methods converge on the same solution.
A difference to underline is that although the two methods converge towards the same value of the solution, they have clearly different execution times. In particular, the Tabu Search, after having adequately calibrated the parameters, turns out to be a reliable method for approximating the optimal solution.
While the calculation of the optimal solution when the problem tends to become very large and complicated becomes unacceptable in terms of execution time, as an example The execution times of the Tabu Search on 100 nodes are reported (Table 5).

| **Instaces** | **user_time [s]** |
|---|---|
| tsp100_10-1 | 12,56 |
| tsp100_10-2 | 13,79 |
| tsp100_10-3 | 15,18 |
| tsp100_100-1 | 14,06 |
| tsp100_100-2 | 15,04 |
| tsp100_100-3 | 13,35 |
| tsp100_1000-1 | 14,69 |
| tsp100_1000-2 | 15,55 |
| tsp100_1000-3 | 15,25 |
| tsp100_10000-1 | 14,32 |
| tsp100_10000-2 | 14,62 |
| tsp100_10000-3 | 15,58 |

Tabella 5: Table of result for 100 Nodes

Please note, however, that the tested instances do not allow us to make predictions on the behavior of the applied methods.