Algorithmic Methods for Mathematical Models

Course Project

Oriol Martínez Acón, Imanol Rojas Pérez

1. Introduction

In this report the course project is explained and solved. The statement of the problem, broadly, is the following:

A set S of n codes must be entered in a door lock to unlock the door. The codes are sequences of m digits that can be either 0s or 1s. All codes must be entered exactly once except for the first and last codes which must be a sequence of m 0s, this code is also inside of the S set. The zeros sequence must be entered at the beginning and at the end of the code entering

The codes are entered using a display which has a sequence of m zeros as the first sequence and a cursor that is placed in the leftmost position. The system has three keys:

- *RIGHT is used to move the cursor on the display one step to the right.*
- *FLIP is used to change the value of the position where the cursor is placed to the opposite value.*
- *ENTER is used to enter the code in the system. It can only be used when the cursor is at the rightmost position. Whenever enter is used, the cursor moves to the leftmost position and the entered code remains in the display.*

The objective of the project is to enter the codes in a certain order such that the usage of the FLIP button is minimized and the code entering process accelerated. So, the objective function of the problem is to minimize the use of the FLIP button.

Overview of the report

The second section of the project is the formulation of the model in mathematical form. This is the mathematical model introduced in CPLEX. Since the problem is complex, it will be solved using meta-heuristics algorithms such as Greedy, Greedy + Local Search and GRASP. These different algorithms are explained, and its pseudocode is implemented in the third section. To solve the problem the datasets used in the solvers must be generated as desired to analyze and compare different algorithms. The explanation of the instance generation and the calibration of the parameters used in the problem can be found in the fourth section. In the fifth section the comparisons between the different algorithms are explained using graphs and tables. Finally, the conclusions extracted from the analysis conducted in this report are written in the seventh section.

2. Mathematical formulation of the model and variable explanation

This section contains two different models for the same problem. The first one being the former version of the project which was not optimal in terms of execution and solution time and the last version which improves the execution times taking advantage of the symmetries in the problem and the subtour elimination using the Miller-Tucker-Zemlin formulation (which is applied to the Traveling Salesman Problem that is similar to ours). The second solution. Both versions are divided in five sections: Variables (inputs), decision variables, auxiliar variables, constraints, and objective function.

The first mathematical formulation of the problem implemented was the following one:

Variables (inputs)

- n, number of codes per set.
- m, length of each code of the set.
- $S_{n\times m}$, Set of codes that must be entered with the m 0s code included

Decision variables

- x_{kj} , Variable that is true if the k-th code entered is code j. With $0 \le k \le n-1$, $0 \le j \le n-1$.
- y_{ij} , Variable that is true when after code I the next code is code j. With $0 \le i \le n-1$, $0 \le j \le n-1$.

Auxiliar variable

- F_{ij} , Variable with the number of flips between code i and code j. (With $0 \le i \le n-1$, $0 \le j \le n-1$)

Constraints

- **First code is the 0's sequence.** $x_{00} = 1$.
- **Checking that starts and ends with 0s.** For all z ($0 \le z \le n-1$): $\begin{array}{l} y_{0z} = x_{1z} \\ y_{z0} = x_{(n-1)z} \end{array}$
- **Codes are not repeated (rows).** For all k: $\sum_{j=0}^{n-1} x_{kj} = 1$
- **Codes are not repeated (columns).** For all j: $\sum_{k=0}^{n-1} x_{kj} = 1$
- **Force that the code i precedes the code j.** For all k, i and j: $x_{ki} + x_{(k+1)j} - y_{ij} \le 1$

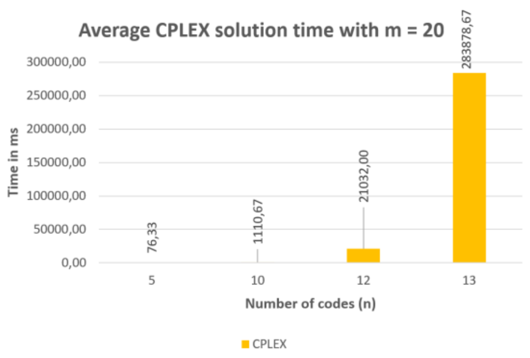
Objective function

The goal of the problem is to minimize the total number of flips. This will be the following:

$$\min \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} y_{ij} \cdot F_{ij}$$

As explained before, this version of the problem had a bad performance. With m = 20, the maximum n with solution under 30 minutes was 13. Here are some results from the executions:

	CPLEX with m = 20			
n	1	2	3	Mean
5	81	79	69	76,33
10	1261	1034	1037	1110,67
12	20464	20267	22365	21032,00
13	319706	272719	259211	283878,67



The new model formulation using the Miller-Tucke-Zemlin method in the following (look at the references to find the webpage where we found the optimization):

Variables (inputs)

The input variables stay the same as in the other model.

Decision variables

- x_{ij} , Variable that is true if the i-th code entered is code j. With $1 \le i \le n$, $1 \le j \le n$.
- u_i , Rank of each node.

Auxiliar variable

- F_{ij} , Variable with the number of flips between code i and code j. (With $1 \le i \le n$, $1 \le j \le n$)

Constraints

- **Is not possible to have code i entered after code i**. For all i ($1 \le i \le n$): $x_{ii} = 0$.
- **Fixing the rank of the first node.** $u_1 = 0$
- **Codes are not repeated (rows).** For all i: $\sum_{j=1}^{n} x_{ij} = 1$
- **Codes are not repeated (columns).** For all j: $\sum_{i=1}^{n} x_{ij} = 1$
- **Avoid subtouring.** For all $1 \le i \le n$ and $1 \le j \le n$ while $j != 1$:
  $u_i + x_{ij} \le u_j + (n-1) * (1 - x_{ij})$

Objective function

The goal of the problem is to minimize the total number of flips. This will be the following:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij} \cdot F_{ij}$$

For comparison in this model having n = 250 and m = 500 gives us times of 7 minutes to have a solution, something that was impossible and unthinkable with the previous formulation where n = 13 and m = 20 already exceeded the 30 minutes maximum execution time given in the project guide.

3. Meta-heuristics: Greedy, Greedy + local search and GRASP algorithms implementation.

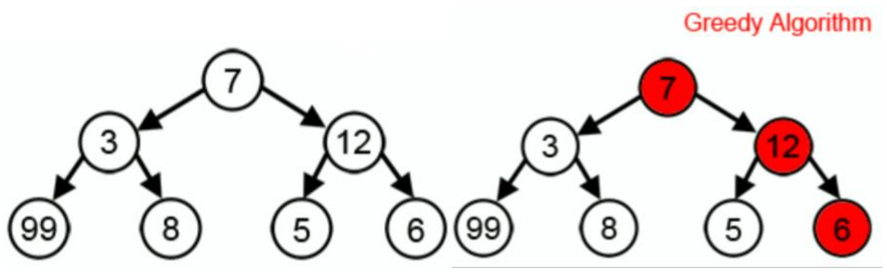
As mentioned in the introduction, the problem of this project is complex, so we can use heuristics to simplify the process of solving the problem. In this section, various meta-heuristics algorithms are explained and, since part of the project has been implementing them on python, their pseudo code is also written and interpreted.

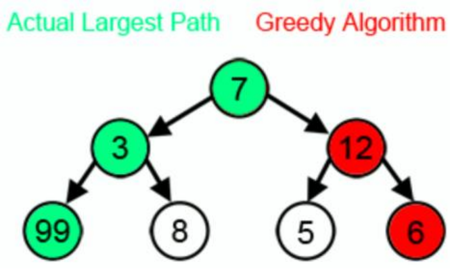
The first implementation is the **Greedy algorithm**. The greedy algorithm solves the problem by selecting the best option available at the moment. The next pseudocode shows how greedy selects a candidate from the candidate set checking on the F matrix (matrix with the flips between codes) which code is the one with the least flips from the node (code) where it starts. The index keeps moving until the F matrix ends (there are no more codes to move to).

```
Algorithm 1: Greedy main function
Initialize flip matrix (F)
Input: instance contains all the values necessary to compute greedy
  algorithm, flip matrix, m and n.
Output: solution is the result of applying greedy with the instance.
path = empty_list
total_flips = 0
row = 0
/* Iterate over the codes to check for candidates                */
for i ← 0 to num_codes do
    new_index, value = _selectCandidate(F[row], path)
    total_flips += value
    add to path(new_index)
    row = new_index
end
/* Set the path to solution                                      */
/* Set total flips to solution                                   */
return solution
```

```
Algorithm 2: _selectCandidateList
Input: candidateList is a row of the flip matrix, containing all the costs
  between a current code to the others.
Input: path is the path followed by the Greedy at the current time of
  the call function.
Output: new_index is the index of the code to add to the path
Output: min_value is the number of flips (cost distance) of the new
  node to add
if length(path) == num_codes then
    new_index = 0
    min_value = candidateList[0]
end
else
    /* Iterate over the path and change to max the elements
       of candidateList to avoid repeating codes                 */
    for element in path do
        candidateList[element] = maxIntValue
    end
    min_value = min(candidateList)
    new_index = get_index(min_value)
end
return new_index, min_value
```

Since the Greedy algorithm does not take in account all possible resulting paths, it can end up choosing a path that is not the global minimum (is a local minimum) cost path. The following images show how greedy can end up choosing a path which is not the actual objective path. In this example the algorithm is searching for the global MAXIMUM path.





A way of getting a better optimal result partially solving this problem is to use **Local Search**. Local search can make slight changes in the path that greedy finds. These changes are made in the path given from greedy so that it ends up being more optimal. The method that is used in local search is 2-Opt, which searches for 2 noncontiguous changes in a given path.

In this project Local Search is implemented with two policies, First Improving and Best Improving.

- Local Search with **First Improving Policy**: The first improving policy of Local Search iterates the path given from greedy and on each iteration checks the possible 2-Opt changes from the start to the end of the path and performs the changes with a greedy philosophy. This means that as soon as it finds a change that minimizes the path, it performs the change and does not check for other feasible 2-Opt options. The algorithm iterates several times using the former slightly changed path until there is no more changes that can minimize the cost of the path.
- Local Search with **Best Improving Policy**: The Best Improving policy of Local Search iterates over the path given by greedy and on each iteration checks all the possible 2-Opt changes that can be done. From the changes that can be done, the algorithm performs the ones that achieve the overall minimum path. The algorithm keeps iterating the former slightly changed paths until there are no possible changes to do. This algorithm can find a better solution than greedy since it takes in account more possible paths and always gets the minimum of all possible options.

Both Local Search policies can end up getting a local minimum due that the starting point is given by the Greedy algorithm and the 2-Opt feasible changes in that path, which can have a tendency (in the solution tree) that can differ from the global minimum. The pseudocode for 2-Opt and both policies is the following:

```
Algorithm 3: LS_2opt
  Input: solution is the object where all the results will be appended
  Output: solution with the new path followed and the total flips of the
    path
  /* Path obtained from the Greedy algorithm                          */
  /* Iterate over the path to search new better minimums              */
  for $i \leftarrow 0$ to length(path) do
    for $j + 2$ to length(path)-1 do
      /* Taking path costs                                             */
      current_cost = flips[path[i]][path[i+1]] + flips[path[j]][path[j+1]]
      new_cost = flips[path[i]][path[j]] + flips[path[i+1]][path[j+1]]
      if $new\_cost < current\_cost$ then
        /* Create a new circular path                                  */
        result_path[i + 1: j + 1] = path[i + 1: j + 1][::-1]
        if policy == 'FirstImprovement' then
          break all loops
        end
      end
    end
  end
  /* Add path followed to the solution                                */
  /* Calculate the total number of flips with the new path            */
  return solution
```

```
Algorithm 4: Local Search main function
  /* Get values from the Greedy solution                              */
  Input: instance is the results generated in the Greedy algorithm
  Output: incumbent is the solution obtained of the local search
    algorithm
  incumbent = solutionFromGreedy
  incumbentFlips = flipsFromGreedy
  iterations = 0
  while $current\_time < end\_time$ do
    iterations += 1
    neighbor = LS_2opt(incumbent)
    neighborFlips = flipsFromIncumbent
    if $incumbentFlips \leq neighborFlips$ then
      break loop
    end
    incumbent = neighbor
  end
  /* Add iterations to Incumbent instance                             */
  return incumbent
```

Finally, the last algorithm of this section is **GRASP (Greedy Randomized Adaptative Search Procedure) algorithm**.

The Greedy Randomized adaptive search procedure (GRASP) is a metaheuristic algorithm used to solve combinatorial optimization problems. As other metaheuristics the solution found is probably not the best solution but converge to the solution in a faster way.

In the GRASP algorithm the boundary is the way to set which will be the value for those candidates to be selected. The boundary is defined by the following equation:

$$Boundary = \min(x) + (\max(x) - \min(x)) * \alpha$$

The α is the parameter to be defined before using the metaheuristic. The value of it can be a percentage or just a value saying how much elements of the candidate list will consider for the selection.

After having set the boundary, the GRASP algorithm will analyze the elements of the cost matrix, searching for those that their values are lower than the boundary to create a new Restricted Candidate List (RCL).

Finally, the algorithm chooses randomly one of the elements in the restricted candidate list and calculates the cost by following the selection. The following images are the pseudocode of the GRASP algorithm.

```
Algorithm 5: GRASP main function
  /* Initialize flip matrix (F)                                       */
  Input: instance contains all the values necessary to compute greedy
    algorithm, flip matrix, m and n.
  Input: alpha is the value that sets how much will be the boundary for
    the new restricted candidate list
  Output: solution is the result of applying greedy with the instance.
  path = empty_list
  total_flips = 0
  row = 0
  /* Iterate over the codes to check for candidates                   */
  for $i \leftarrow 0$ to num_codes do
    new_index, value = _selectCandidate(F[row], path)
    total_flips += value
    add to path(new_index)
    row = new_index
  end
  /* Set the path to solution                                         */
  /* Set total flips to solution                                      */
  return solution
```

```
Algorithm 6: _selectCandidateList
  Input: candidateList is a row of the flip matrix, containing all the costs
    between a current code to the others.
  Input: path is the path followed by the Greedy at the current time of
    the call function.
  Input: alpha is the value that sets how much will be the boundary for
    the new restricted candidate list
  Output: new_index is the index of the code to add to the path
  Output: selection is the number of flips (cost distance) of the new
    node to add
  if length(path) == num_codes then
    new_index = 0
    min_value = candidateList[0]
  end
  else
    /* Iterate over the path and change to max the elements
       of candidateList to avoid repeating codes                      */
    for element in path do
      candidateList[element] = maxIntValue
    end
    minF = min(candidateList)
    maxF = max(candidateList)
    boundaryF = minF + (maxF - minF) * alpha
    /* Iterate over the candidateList to search if any value
       is less or equal of the boundary and add to restricted
       candidate list                                                 */
    for candidate in sortedCandidateList do
      if $candidate \leq boundaryF$ then
        rcl = candidate
      end
    end
    selection = random(rcl)
    new_index = get_index(selection)
  end
  return new_index, selection
```

4. Tuning of the parameters and instance generation.

The solvers have data files as input with the generated S matrix and the m and n parameters. This data files are generated using an instance generator. This instance generator takes the values of the n and m parameters as input, generates a set of codes S of n codes of size m, and saves this matrix in a .dat file with the m and n parameters introduced by the user.

The instances have been generated to compare the impact of n in the time that takes to calculate a solution and to check what solutions do the different methods give. The group instances are generated with m fixed to 500 in all of them and n takes values from 25 to 250 (25, 50, 75, …, 250). The n values have been chosen taking in account the solving time of the CPLEX solver since it is the most restrictive (the one that takes more time to solve for bigger n values). It is expected that the other solvers (heuristic solvers) have times close to 0. The data files are named as nXX_m500.dat.

All the instances accomplish the following constraints related to the set of codes S.

- The set S has n codes.
- The codes are composed of m bits that only take binary values (0 or 1).
- A code composed of m zeros is always in the set S as the first code.
- All the codes are generated randomly (besides the 0 code which is forced to appear in all sets).
- All codes appear exactly once in the set, the codes are not repeated.

Optimization of F matrix calculation (preprocessing)

The F matrix calculation has a huge time impact on the program execution time. The solver time is not affected but the overall execution time is highly impacted when n (number of codes in S) takes high values. This can be solved in three steps.

- First, a 0s F matrix is created with size $n \times n$. For example, for n = 3:

$$F = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

- Since the F matrix is symmetric, i.e., $a_{ij} = a_{ji}$, the upper triangle flip calculations are the same as the lower triangle calculations. This happens because the differences between one code are the same no matter of the order of the comparison. So, in the preprocessing of the F matrix, only the upper triangle of the matrix must be calculated, avoiding the diagonal, because the flip comparison of a code with itself will be always 0. This value is saved in a variable called f and assigned to the upper triangle coordinates of the matrix. For example, for n = 3 the calculated coordinates are:

$$F = \begin{pmatrix} 0 & a_{12} & a_{13} \\ 0 & 0 & a_{23} \\ 0 & 0 & 0 \end{pmatrix}$$

- Finally, to fill all the matrix, the upper triangle values must be replicated in the lower triangle. This is done by assigning the calculated value of flips to the opposite position in the matrix (F[i][j] = f and F[j][i] = f). So, the final matrix is:

$$F = \begin{pmatrix} 0 & a_{12} & a_{13} \\ a_{12} & 0 & a_{23} \\ a_{13} & a_{23} & 0 \end{pmatrix}$$

5. Algorithm comparison

The next step on the project is to run the algorithms once they are implemented, using the instances generated in the last section. During the execution, the time that takes to the solver to get a solution is measured and the solutions given by each solver from the first group of instances are saved. In this section, these times and solutions are used to compare the algorithms between them.

Time comparison of the algorithms

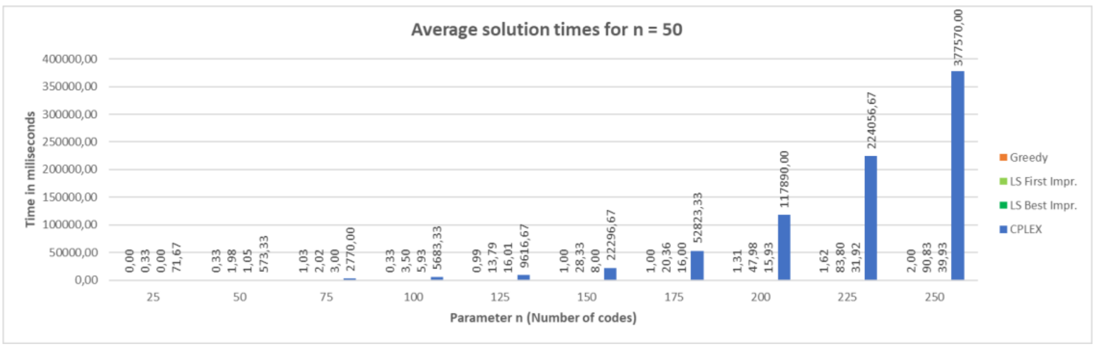
The first solver to be analyzed is CPLEX. CPLEX is an optimization software developed by Robert E. Bixby now owned by IBM that solves integer programing problems, very large programming problems, convex and non-convex problems quadratic programming problems, and convex quadratically constrained problems. It is named for the simplex method as implemented in C (C simplex -> CPLEX). The simplex is the mathematical algorithm that CPLEX uses to solve the optimization problems. The results that CPLEX finds are always the global minimum/maximum even when local minimums/maximums exist. This algorithm evaluates all possible solutions before taking a decision on the most optimal one.

The first parameter of the problem analyzed is n. For that, we use the generated instances, where the m is maintained at 500 and the n is the parameter that takes several values. The results in terms of solving time (in ms) for the CPLEX and the other algorithms where:

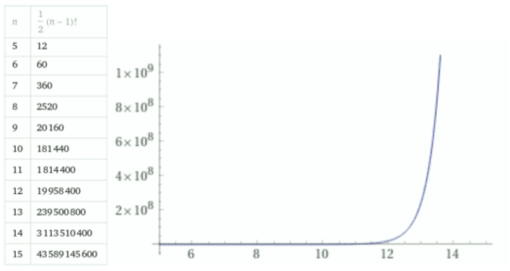
	Average solution time in miliseconds			
n	CPLEX	Greedy	LS First Impr.	LS Best Impr.
25	71,67	0,00	0,33	0,00
50	573,33	0,33	1,98	1,05
75	2770,00	1,03	2,02	3,00
100	5683,33	0,33	3,50	5,93
125	9616,67	0,99	13,79	16,01
150	22296,67	1,00	28,33	8,00
175	52823,33	1,00	20,36	16,00
200	117890,00	1,31	47,98	15,93
225	224056,67	1,62	83,80	31,92
250	377570,00	2,00	90,83	39,93

GRASP is not in the table since it will be analyzed separately. The time parameter of grasp works different with respect to the other algorithms, and we think that is not quite interesting to evaluate GRASP in terms of time since its random. We analyze it in terms of solutions depending on alpha further on this same section.

Since CPLEX solves the problem searching for the global minimum path of the problem and for that it needs to check all possible solutions of its search space. The problem of this project is a combinatorial problem, and its search space grows bigger with the parameter n. The search space is (n-1)!/2. To see the impact of n clearly, a graph of the table of times has been plotted, the graph is the following:



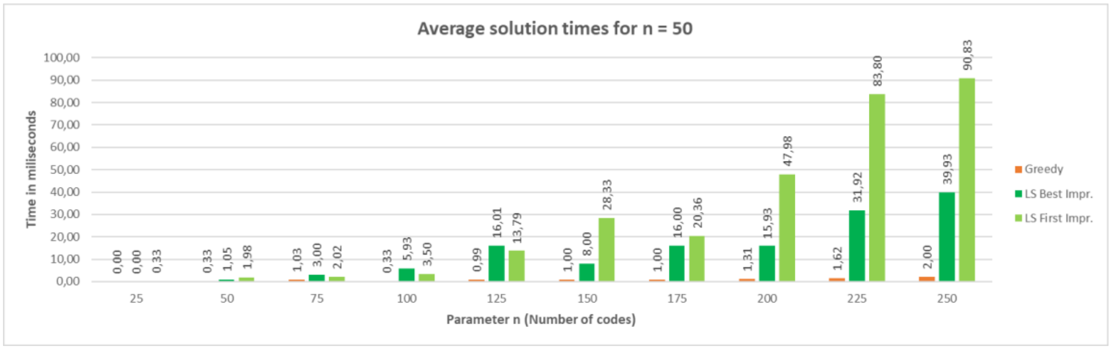
The impact of the n parameter is clear. The shape of the graph is close to the shape of the search space, which makes sense since the search space must be entirely evaluated by CPLEX to get an optimal solution and the more search space, the more time that it takes to reach a solution. Let's look to the graph of the search space evolution with n and see if there are similarities with the time graph obtained manually using the CPLEX solver. The graph and the table (generated with WolframAlpha) of the search space evolution with respect to n being increased is the following:



n	$\frac{1}{2}(n-1)!$
5	12
6	60
7	360
8	2520
9	20160
10	181440
11	1814400
12	19958400
13	239500800
14	3113510400
15	43589145600

The evolution of the search space is like the time graph obtained manually in CPLEX so it can be concluded that the n parameter has a great impact in the solution time because the amount of search space of the problem depends on n and CPLEX must process all the search space to get a solution.

Due that this is a NP-Hard problem it can be solved with heuristics. The next graph shows the times of the heuristic algorithms with respect to n but without CPLEX to see the differences between them.



The Greedy algorithm is the fastest among all due to its way of working because it does not check all the possible solutions. The Local Search policies have different times, and the Best Improvement for bigger n values is the one with lowest times. This happens because with each iteration does a check of all the possible 2-Opt changes and finds the most optimal ones whereas the First Improving algorithm just takes the first minimal 2-Opts that finds. This makes BI arrive to the most optimal path (the one with no more possible 2-Opt changes to do) earlier than FI. For lower values of n, FI has better times since is more probable to find the minimum in less 2-Opt comparisons. Also, both algorithms follow a similar shape as CPLEX wen increasing n. This is because both depend on n when they search for the 2-Opt. More n means more nodes and thus, more possible path changes. Greedy does not follow that shape because of how the algorithm works.

Solution comparison of the algorithms

The next table shows which are the results obtained by each algorithm using the same instances. In this subsection the comparison is focused in how optimal are the solutions that the algorithms get. The parameters used to ger the GRASP solutions are the different alphas shown in the column titles and 200 seconds of execution time.

	Solutions found by the solvers							
n	CPLEX	Greedy	LS First Impr	LS Best Impr	GRASP (α = 0)	GRASP (α = 0.2)	GRASP (α = 0.5)	GRASP (α = 0.8)
25	5788	5832	5794	5804	5832	5788	5832	5832
50	11454	11572	11506	11494	11572	11512	11572	11572
75	17024	17158	17078	17078	17158	17158	17158	17158
100	22572	22614	22584	22576	22614	22614	22614	22614

As expected, the most optimal results are obtained with CPLEX. This is because the algorithm checks all possible solutions to always get the global minimum one. The other algorithms get better times but at the cost of ending up with less optimal results.

The algorithm that consistently gets the worst results is Greedy. As explained in the Greedy algorithm introduction, one of the most common problems of greedy is ending with a solution path that is not the mot optimal due to its way of working. When searching the best path greedy can end up in a branch of the solution tree that tends to a worst solution even if the starting node was the minimum one.

The Local Search algorithm gets better results than Greedy in both policies because it iterates over the greedy solution path, and it makes slight changes to get a more optimal solution. The best policy is the Best Improvement one. The fact that it checks for all the possible 2-Opt changes makes the final path more optimal whereas the First Improvement is a “Greedy” iteration of the 2-Opt algorithm.

In the table there are different results from the execution of the GRASP algorithm. The alphas set are 0, acting very similar to the Greedy algorithm. This is because in some cases if there is more than one minimum cost values of the flip matrix with same value, GRASP algorithm will take one of the cost values randomly while the Greedy algorithm will take the first min value starting from the left in the vector. The alpha chosen as the best one for executing the GRASP has been the 0.2. With this value the alpha shows a better result than the Greedy algorithm when executing the program with n=25 and m=500. Alphas with a lower value do not generate a lot of randomness when selecting the candidate element, the RCL list is smaller, while bigger values for alpha generates more randomness when selecting the candidate element, the RCL list is bigger. In conclusion, if we had a huge amount of time to solve a problem with huge m and n, a big value of alpha could give us a good solution because it can explore all the tree of possibilities without falling into a local minimum; however, in most of the situations the CPLEX will performs better in terms of time, and it will find the global solution. So, the best idea is to choose low values for alpha and the best one it seems to be 0.2.

6. Conclusions

From the problem modeling, and the comparison of the algorithms in solution time and solution values we extract the next conclusions:

Mathematical modeling conclusions

- The performance of the MTZ method is several thousand times faster than the one we first implemented. This happens because of the constraint simplification, the use of the problem symmetry and the subtour avoidance.
- The matrix of flips F can be calculated taking advantage of the symmetry that the problem has, and this enhances the performance in the preprocessing step.

Algorithm time comparison conclusions

- The n parameter has a huge impact in the solution time of CPLEX since it must check all the solutions in the search space, and it increases with n. The search space of the problem is (n-1)!/2.
- The shape of the time graph of the CPLEX algorithm is similar to the (n-1)!/2 graph because the search space it is strongly related with the solution time as mentioned before.
- CPLEX is many times slower in finding a solution in comparison with the heuristic algorithms. That is the purpose of the heuristic algorithms. In a situation like this NP-Hard problem they can achieve lower solving times.
- The Greedy algorithm is the fastest among all due to its way of working because it does not check all the possible solutions.
- With low values of “m” and “n”, the First improvement shows a better performance, in terms of time, than Best Improvement. That is because the value returned by the First improvement is the best solution that you can find in Local Search algorithm without exploring the whole path; on the other hand, the Best Improvement is doing more comparisons useless to converge into a solution. However, if the size of the “m” and the size of “n” is big, the Best Improvement is converging into a result faster than the First Improvement. That is because the Local Search algorithm in First Improvement does more iterations due to the presence of good changes in the flips result while the Best Improvement iterates less times.

Algorithm solution comparison conclusions

- CPLEX algorithm always gets the most optimal solution. In our case the minimum path cost.
- The heuristic algorithms achieve better times at the cost of having less optimal solutions.
- The Greedy algorithm gets the less optimal solutions because of the way that it works.
- The Local Search algorithm gets better results than the Greedy because it starts with the Greedy path and iterates over it to slightly change it into a new path that has less cost.
- The Local Search policy that gets the most optimal solutions is the Best Improvement since it checks all possible 2-Opt changes whereas the First Improvement is a "greedy" iteration of 2-Opt
- Big problems (m and n large) could take a lot of time if we are trying to solve by using the GRASP. Big values of alpha create more possibilities to explore the whole tree of solutions but paying with a lot of computation time, because it is more random. Small values of alpha create less possibilities to explore the tree of solutions, but it converges faster into a good solution, because it is less random.

7. References & links of interest

- Lecture slides
- Course Labs
- CPLEX: https://en.wikipedia.org/wiki/CPLEX
- Subtour elimination and MTZ: https://co-enzyme.fr/blog/traveling-salesman-problem-tsp-in-cplex-opl-with-miller-tucker-zemlin-mtz-formulation/
- Wolfram Alpha: https://www.wolframalpha.com/
- Project GitHub: https://github.com/oriolmartinezac/AMMM-project