UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Master in Innovation and Research in Informatics

Algorithmic Methods For Mathematical Models (AMMM)

Course Project Report

Roger Gili i Coscojuela
roger.gili.i@estudiantat.upc.edu

Edgar Rodríguez de la Calle
edgar.rodriguez.de.la.calle@estudiantat.upc.edu

Roger Gili i Coscojuela
Edgar Rodríguez de la Calle

1. Formal statement of the problem

1.1. Problem statement

The given statement for the course project is the following:

"Secret agent James Bond is on a dangerous mission fleeing from the enemy. In order to get back to his base, he has to enter some codes in the access control of the door. But he has to do it swiftly, because the enemy is treading on his heels.

More precisely, let us assume that the codes are sequences of digits 0 and 1 of length m. To unlock the door, there is a prefixed set S of n different codes that have to be entered. Each of these codes has to be entered exactly once, in any order, with the following exception: the first and the last codes that are entered have to be the sequence consisting of m zeros (which is always included in S).

Unfortunately for agent Bond, the access control is a bit outdated and has a rather awkward interface, which consists of 3 buttons (RIGHT, FLIP, ENTER) and a display. At any time the display shows a sequence of m digits 0 and 1 (initially, the sequence of m zeros). There is a cursor which at the beginning is at the leftmost position, and which can be moved from one position to the next one on the right by pressing the RIGHT button. The digit the cursor is on can be flipped with the FLIP button. Finally, when the sequence is ready and the cursor is at the rightmost position, the ENTER button must be pressed to register it. Then the cursor moves back automatically to the leftmost position, but the last sequence that was entered is kept on the display. The same procedure is applied for the next code, until all codes have been entered.

Agent Bond wants to enter the codes as quickly as possible, to avoid getting trapped by the enemy. The goal is to minimize the number of times that the FLIP button has to be pressed (the RIGHT and ENTER buttons are not included in this count because the number of times they have to be pressed is always the same: note that, in order to enter each code, the RIGHT button will have to be pressed m − 1 times, and the ENTER button once)."

1.2. Data, Inputs and Outputs

From the problem statement we can extract the following input data:

- A set S that contains a fixed *n* number of codes.
- Each code is a sequence of *m* binary digits.

From the problem statement and the input data, we realize that we are actually dealing with a well-known optimization problem, the Travelman Sales Problem (TSP)

In our case, each city is represented by a code. The cost from moving from one code to another is the difference between the digits of each code.

Given 2 different codes, for example 0011 and 0101, the cost of changing from the first code to the second one is 2, because the second code has 2 different digits.
That is, indeed, the number of flips that have to be minimized in our problem.
Knowing that, our output data will be:

- The number of flips f after putting all the codes once
- The order that each code has been entered such that f is minimized.

2. Mixed Integer Linear Programing Model

2.1. Inputs

Our model receives the following input data:

- n : The number of codes.
- m : The number of binary digits that each code has.
- S : The set of n codes with m digits

2.2. Variables

For our model we have used the following variables:

- f *:* Discrete variable that represents the total number of flips performed after entering all codes.
- $d_{i,j}$: Discrete variable that represents the distance or cost matrix that stores the number of flips that represents changing from one code i to another code j.
  The indices i and j ranges from ($0 \le i,j \le$ n)
- $y_{i,j}$: Binary variable which is True (1) if after code i, code j is entered. False (0) otherwise.
  The indices i and j ranges from ($0 \le i,j <$ n)
- $x_{k,j}$: Binary variable which is True (1) if the k-th code entered is j. False(0) otherwise.
  The indices k and j ranges from ($1 \le k,j <$ n)

2.3. Objective function

The objective function is described as follows:

Minimize: f

2.4. Constraints

First Constraint: This constraint gives the intended meaning to *f*. Being the total sum of the costs of changing from one code to another.

$$\sum_{i,j=0}^{n} d_{i,j} y_{i,j}$$

Second constraint: This constraint ensures that from a code *k* only one code *j* is entered next.

$$\sum_{j=1}^{n} x_{k,j} = 1$$

Third Constraint: This constraint ensures that one code *j* is reached after only one code *k*.

$$\sum_{k=1}^{n} x_{k,j} = 1$$

Fourth Constraint: This constraint ensures that after the code containing all zeros, the first code is entered.

$$y_{0,j} = x_{1,j} : \forall j \ (1 \leq j < n)$$

Fifth Constraint: This constraint ensures that after the last code, the code containing all zeros is entered again.

$$y_{j,0} = x_{(n-1),j} \ \forall j\!: \ (1 \leq j < n)$$

Sixth Constraint: This constraint is ensuring that there are no subtours in the order where the codes are entered.

$$x_{k,i} + x_{(k+1),j} - y_{i,j} \leq 1 \ \forall k\!: \ (1 \leq k < n-2) \ \forall i,j\!: \ (1 \leq i,j < n)$$

3. Heuristic Algorithms

For the correct understanding of the nexts pieces of pseudocode of the different algorithms, we need to record a little bit the different data structures that we have used.

For the representation of each code, we have created a class named *Node* that has as attributes an *id*, representing the code, a *ToId* that represents the next code that will be entered, and *connections,* representing the number of connections that each code has inside the graph.

For the representation of the graph where the node exists, we have created a class *Graph* that has as attributes a list of class *Node* objects and a matrix named *CostMatrix* that stores the associated cost of going from one node to another.
Inside each algorithm a list is created that will store the nodes with all the updated information, such as the *ToId* and the number of *Connections*. This list represents the solution and will be returned once the algorithm finishes.

3.1. Greedy

Here is the pseudo code that describes the Greedy algorithm.

Algorithm 1 GREEDY

```
function GetFeasibleLinks($\omega$)
    Candidates = {}
    for i in $\omega$ do
        if node[i] connections < 2 then
            for each j in N do
                if Node[j] connections == 0 then
                    c = Cost of moving from Node[i] to Node[j]
                    candidate = (c, Node[j])
                    Add candidate to Candidates
                end if
            end for
        end if
    end for
    Sort Candidates by its cost
    return Candidates
end function

function GreedySolver
    Initialize N, the initial set of codes
    $\omega \leftarrow$ {}
    Add to $\omega$ the first node in N
    while $\omega$ is not a solution do
        Candidates = getFeasibleLinks($\omega$)
        Update $\omega$ with first node of Candidates
    end while
    return $\omega$
end function
```

As it can be seen in the pseudo code above, the greedy algorithm checks every iteration for the possible next nodes that can be added, given a partial solution, that starts arbitrarily with the first node representing the first code in the set of codes.
These posible nodes called candidates, are sorted by its associated cost from lowest to highest.
The node with lower cost is added to the solution, then all the related data of the nodes in the solution and the new added node, such the *ToId* and the number of *Connections* is updated.
This is done until all nodes represented in the set of codes have been added to the solution and verified that the given solution is feasible.

3.2. GRASP

Here is the pseudo code that describes the GRASP algorithm.

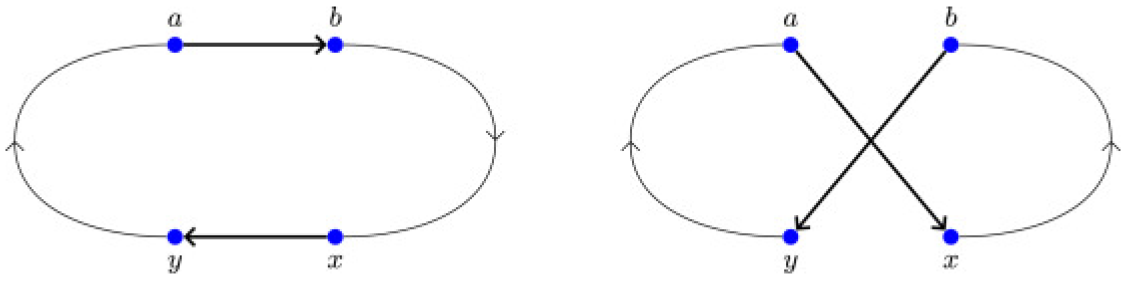
```
Algorithm 2 GRASP
 1: function GRASPSOLVER
 2:     Initialize N, the initial set of codes
 3:     $\omega \leftarrow \{\}$
 4:     Add to $\omega$ the first node in N
 5:     while $\omega$ is not a solution do
 6:         Candidates = getFeasibleLinks($\omega$)
 7:         $q_{min} \leftarrow min\{q(c) \mid c \in Candidates\}$
 8:         $q_{max} \leftarrow max\{q(c) \mid c \in Candidates\}$
 9:         $RCL_{min} \leftarrow \{c \in Candidates \mid q(c) \leq q_{min} + \alpha(q_{max} - q_{min})\}$
10:         Select $c \in RCL$ at random
11:         Update $\omega$ with random selected node form RCL
12:     end while
13:     return $\omega$
14: end function
```

GRASP comes from Greedy Randomized Adaptive Search Procedure. As the names suggest, it is a greedy algorithm with a randomized component.
This algorithm is using the same function shown before for getting the list of candidates, but now, instead of choosing the first candidate, the one with less associated cost, we restrict the candidate list and make a random choice from it.
This Restricted Candidate List will contain more or less candidates depending on the alpha parameter.
If the alpha parameter is 0, the grasp will behave as a Greedy algorithm, and if the alpha parameter is 1 Grasp will behave purely random.

Due to the randomness that Grasp applies to the final solution, this algorithm does not ensure an optimal solution, in fact for different executions will produce different solutions, all of them feasible, but not optimal.
For that reason a second step called Local Search is performed after having the solution created by Grasp. This second step will try to improve the solution, in our case, trying to reduce the total cost of the graph created.

3.3. Local Search

Having in mind that we are dealing with a TSP problem, many Local Search techniques and procedures exist for improving our solution.
In our case we have implemented the 2 Edge Exchange or 2-opt.
This heuristic defines the neighborhood of a solution of all the graphs that have 2 different edges.
It selects pairs of nodes inside the solution and exchanges the link of one with the other.



Visual representation of 2 Edge Exchange

If the newly created links have less cost than the old ones, an improvement has been found and the solution is updated with the new links. If not, we keep searching for pairs of nodes where the exchange produces a better solution.
In our case this exchange check is performed for every single pair of different nodes inside the solution. If something better is found, we start again with the new better solution until there isn't any exchange that is better than the current configuration of the graph. Once Local Search finishes we can say that we found a local optima solution.

But, with the objective of finding the global optimal solution, the combination of Grasp and Local Search is executed several times, in order to explore more of the solution space. After a period of time, we keep the best solution found in all the performed iterations.

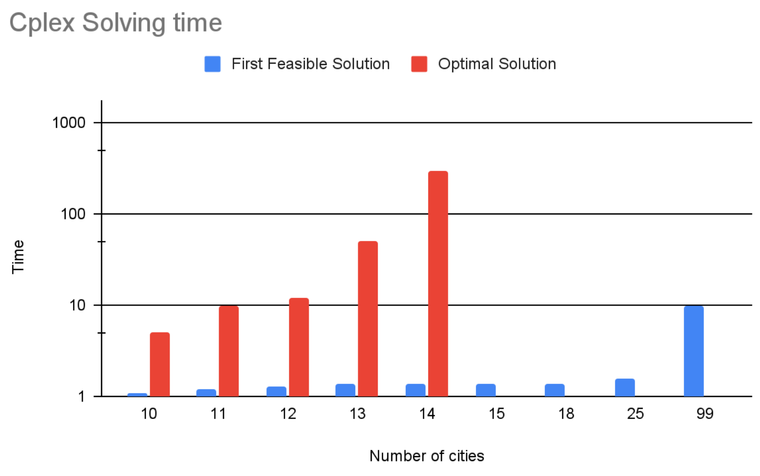
4. Results

The distance between each code is calculated at the start of the problem, because of that, the length of the codes does not impact the performance of our algorithms, instead, what has an impact is the number of codes that our problem has.

The computer in which we ran our test has an Intel i5-3310M mobile processor, with 2 cores and 2 threads per core, and 4GB of ram in total. We haven't used any GPU's accelerators to run our algorithms. In none of the tests, the ram usage hit the computer limits.

To test our algorithms we created instances of the problem, each one with a bigger number of codes. We've solved each instance with our Cplex, greedy, greedy + local search, grasp and grasp + local search algorithms, stopping them if they take more than 30 minutes to solve. For every algorithm, we've recorded the time every time a new best solution was found.

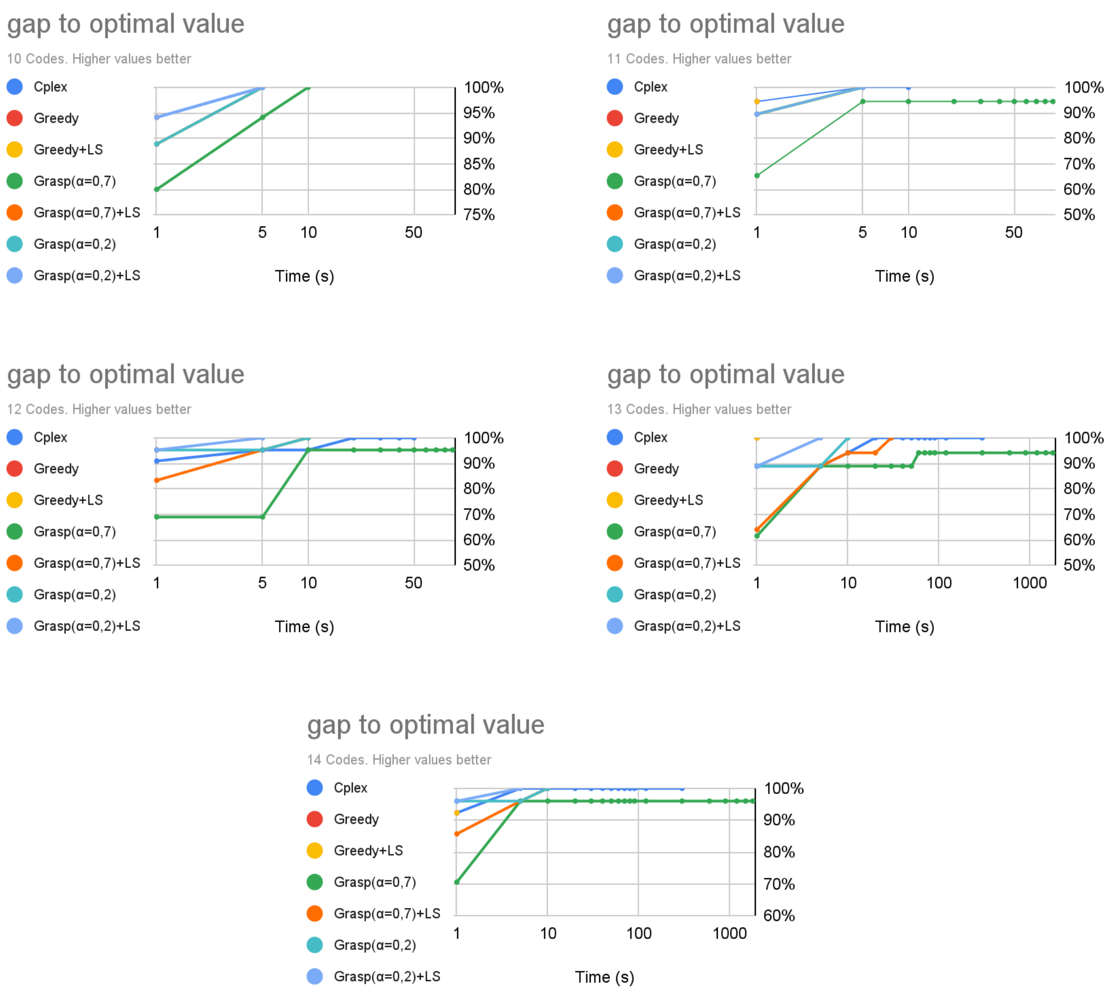
We first started by testing how much time the Cplex implementation takes to find an optimal solution. The results shown below denote that our implementation in Cplex is not capable of giving an optimal solution in less than our time limit when our problem has more than 15 codes. We can see that the resolution time grows exponentially. At the same time, we observe that it is capable of giving us a possible solution in less than two seconds for instances smaller than 25 codes. In the case of the instance with 99 codes, this time is increased to 10 seconds.



Cplex solving time for different instances of the problem

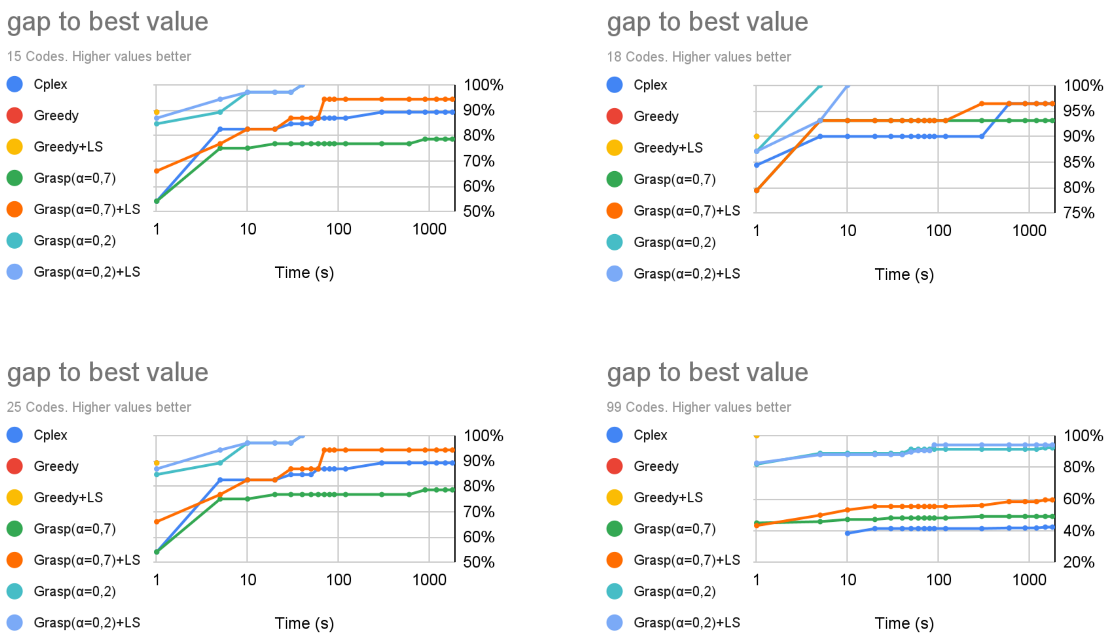
After this test, we ran our algorithm implementations with the same instances and then we calculated the gap between the best value found over time for each algorithm and the optimal value. Because after 14 codes our ILP mode doesn't give us an optimal result in our time limit, in the graphs for more than 14 codes the 100% value corresponds to the best value found by any of the algorithms.

In the following image instances between 10 and 14 codes are compared. Note that the greedy algorithm always ends in less than 1 second, and therefore is always displayed as only one dot at the left side of the graph. Comparing the performance of the different algorithms, we can see that the greedy algorithm gives us outstanding results, taking into account that it finishes its execution in less than a second. We can also see that the grasp algorithm performs better with lower values of α. This behavior is also seen in larger instances of the problem.



Optimal Gap Vs Time in different instances

Below is the comparison using instances of 15, 18, 25 and 99 codes. Because we don't have the optimal solution for these problems, the 100% value is the best value obtained. In this case, we can see that using smaller values of α produces better solutions than using larger values. We can also see that the usage of the localsearch optimization also provides better results compared to not using it in all cases.



Different alpha values for high instances

For our final conclusions, we think that the decision of choosing an algorithm or another in a production environment will depend on the time constraint and the gap to optimal value that we can afford to have. If an optimal solution is needed, the usage of Cplex is mandatory, but we must keep in mind that the time to solve the problem rises exponentially. The greedy algorithm provided good solutions overall, with the worst solution being 89% of the optimal value, finally, the grasp algorithm performed well with lower values of alpha. Because of all the above, a proper approach for a future work can be creating a grasp algorithm that starts with a value of alpha of zero (greedy) and increases this value for each iteration until the max value that we want to try is used.