AMMM Project

Cerka Patrick - Singez Hugo

25th of May, 2025

Contents

1 Formalization of the problem

We start by giving a formal statement of the problem.

Overview of the problem:
A cooperation of N members shares a computer that they want to use in order to mine cryptocurrencies. Each month, the members declare the days they want to use the computer. For two members i and j, the members i bets the amount mij to have priority over member j and the member with the priority can use the computer (by paying the amount he has bet to the cooperation). Given those bets, the goal is to define the priority matrix such as no loops of priorities are formed, and the income for the cooperation are maximized.

Inputs of the problem:

- The integer number N of members in the cooperation
- The integers matrix m of size $N \times N$ with the associated bets of the members (m[i][j] $= m_{i,j}$ for i,j in $[1; N]$)

Outputs of the problem:

- Boolean matrix x of size $N \times N$ with the priorities of the members. If $x_{i,j} = 1$ then member i has priority over member j
- The integer z, the total income for the cooperation

2 Integer linear programming model

According to the previous formalization of the problem, we can now solve it using integer linear programming model.

Variables

- The priority matrix x of size $N \times N$ such as $x_{i,j} = 1$ if member i has priority over member j of type BOOLEAN
- The INTEGER z corresponding to the total income for the cooperation with the defined priorities
- The vector of INTEGER r of size N where r_i (for integer i in $[1; N]$) corresponds to the rank of the member i in the the topological ordering used to defined by the matrix x

Objective Function

- We want to maximize the revenue for the cooperation for the month. To this extend, the objective function is to **MAXIMIZE** z

Constraints

- The value of the revenue for the cooperation is equal to the sum over all members i and j of the bet of member i if he has priority over member j

$$z \leq \sum_{i=1}^{N} \sum_{j=1}^{N} m_{ij} \cdot x_{ij} \qquad \textit{(1)}$$

- For two distinct members i and j only one member should have priority over the other. Also, a member cannot have priority over himself so $x_{i,i} = 0$. This latest constraint is already implied by the constraint *(1)*

$$x_{i,j} + x_{j,i} \leq 1 \qquad \textit{(2)}$$

  Indeed, with the constraint *(1)* we will have at least $x_{i,j}$ or $x_{j,i}$ not null if there exist a conflict between i and j

- The rank of each member i is lower than the number of players

$$r_i \leq N \qquad \textit{(3)}$$

- For two distinct members i and j, the member with priority has the lowest rank in the topological ordering

$$r_i - r_j + 1 \leq (1 - x_{i,j}) * N \qquad \textit{(4)}$$

  Because of the equality of constraint *(2)* we have that r_i and r_j will be different for two distinct members. Also, this inequality results from the theorem that we were introduced to during the solution of exercises in class.

3 Greedy algorithm

The Greedy Algorithm is based on a simple logic:

- In the **BuildAcyclicPriority** function, We take as input the matrix with the priority values m and the dimension N;
- We build this arrays *node_score* with pairs containing each the index and the sum of all it's outgoing arcs;
- Then we sort *node_score* in descending order and initialize another vector of integers named *order*. This array will contain the sorted indexes of *node_score*;
- We then start to build our solution, where the first index of order will be the start of our "topological order". and we continue like this through all the *order* array;
- We print the matrix and then call the*local_search_order* function passing order.

Algorithm 1: Build Acyclic Priority Matrix

```
Input: Matrix $m$, Integer $N$
Output: Priority matrix $x$
Initialize empty list $node\_score$
for $i \leftarrow 0$ to $N-1$ do
    $sum\_row \leftarrow 0$
    for $j \leftarrow 0$ to $N-1$ do
        $sum\_row \leftarrow sum\_row + m[i][j]$
    end
    Append $(sum\_row, i)$ to $node\_score$
end
Sort $node\_score$ in descending order by $sum\_row$
Initialize array $order$ of size $N$
for $i \leftarrow 0$ to $N-1$ do
    $order[i] \leftarrow node\_score[i].index$
end
Initialize $x$ as an $N \times N$ zero matrix
for $i \leftarrow 0$ to $N-1$ do
    for $j \leftarrow i+1$ to $N-1$ do
        $u \leftarrow order[i]$
        $v \leftarrow order[j]$
        $x[u][v] \leftarrow 1$
    end
end
Print matrix $x$ and score sum of $m[i][j]$ where $x[i][j] = 1$
local_search_order($order$, $m$)
return $y$
```

There's no need to control whether our solution could lead to deadlock. That's why: given that we follow order, which is sorted by default, it is impossible to incur in an arc entering a previous node. To put it simple, if I have an ordered list

$$a_1, a_2, a_3, \ldots, a_n$$

and we only create arc to successive nodes/elements, it's impossible to create a backward arc.

The cost of the greedy part alone is $\Theta(N^2)$, given by the computation of the *node_score* list. In any case we are going to compute the whole m matrix.

4 Local Search algorithm

For the local search, first we need to implement a way to evaluate our solution(s). Thus, we created the evluate function, which simply adds each value of *m[i][j]* if *x[i][j]* is 1.

Algorithm 2: Evaluate Order Score

```
Input: Order array $order$, Matrix $m$
Output: Score integer
1  $N \leftarrow$ length of $order$
2  $score \leftarrow 0$
3  for $i \leftarrow 0$ to $N-1$ do
4      for $j \leftarrow i+1$ to $N-1$ do
5          $score \leftarrow score + m[order[i]][order[j]]$
6      end
7  end
8  return $score$
```

The actual local search is quite simple: it iterates through the x matrix and swaps two values and evaluating the new solution through our function *evaluate*. All this in a while cycle, to iterate as long as we are improving the solution, until we find the local maximum.

Algorithm 3: Local Search on Order Array

```
Input: Order array order, matrix m
Output: Improved order with local search
1  $N \leftarrow$ length of order
2  $improved \leftarrow$ true
3  while $improved$ do
4      $improved \leftarrow$ false
5      for $i \leftarrow 0$ to $N-2$ do
6          swap(order[i], order[i+1])                  // Swap adjacent elements
7          $new\_score \leftarrow$ evaluate_order(order, $m$)
8          swap(order[i], order[i+1])                  // Revert swap
9          $current\_score \leftarrow$ evaluate_order(order, $m$)
10         if $new\_score > current\_score$ then
11             swap(order[i], order[i+1])              // Accept improvement
12             $improved \leftarrow$ true
13         end
14     end
15 end
16 Printing logic...
```

Talking about complexity, the worst case scenario could lead us to a $O(N^4)$, given by the while loop, the for inside it and the two nested loops in the evaluate function.

However, considering that the whole function aims to improve to a **local** maximum, it will not iterate typically the whole m matrix.

This leads us to this complexity: $O(W * N^3)$, with W the duration of the while loop, which will be shorter most of the time than N.

5 GRASP algorithm

The GRASP implementation is this:

- We get the same input as the **BuildAcyclicPriority** function with an addition: α. This parameter determines how many candidates can be chosen randomly for the priority;
- The overall logic is also similar to **BuildAcyclicPriority**: we compute the nodes with the highest sum of values of the outgoing arcs;
- We initialize order. Then we build the solution by randomly picking a value inside the range defined by $[0, N*\alpha]$. We keep track of nodes already chosen and skip them in the successive iterations;
- We print the matrix.

Algorithm 4: GRASP Construction of Priority Matrix

```
Input: Matrix $m$, Parameter $\alpha$
Output: Priority matrix $y$
$N \leftarrow$ size of $m$
Initialize empty list $order$
Initialize boolean array $chosen$ of size $N$ with false
Initialize array $scores$ of size $N$
for $i \leftarrow 0$ to $N-1$ do
    $scores[i] \leftarrow \sum_{j=0}^{N-1} m[i][j]$
end
while $size\ of\ order < N$ do
    Initialize empty list $candidates$
    for $i \leftarrow 0$ to $N-1$ do
        if $chosen[i] = false$ then
            Append $(scores[i], i)$ to $candidates$
        end
    end
    Sort $candidates$ in descending order by score
    $rcl\_size \leftarrow \max(1, \lfloor \alpha \times \text{size of candidates} \rfloor)$
    Randomly pick index $pick$ in $[0, rcl\_size - 1]$
    $selected \leftarrow candidates[pick].index$
    Append $selected$ to $order$
    $chosen[selected] \leftarrow$ true
end
Initialize $y$ as an $N \times N$ zero matrix
for $i \leftarrow 0$ to $N-1$ do
    for $j \leftarrow i+1$ to $N-1$ do
        $u \leftarrow order[i]$
        $v \leftarrow order[j]$
        $y[u][v] \leftarrow 1$
    end
end
Print matrix $y$ and score sum of $m[i][j]$ where $y[i][j] = 1$
local_search_order($order$, $m$)
return $y$
```

The complexity of grasp is a bit trickier, given that we use a function of the c++ library, sort. Based on documentation found online, we can assume that it uses quicksort (or mergesort), which are $O(N * log(N))$.

This, together with the while loop of dimension N, leads us to $O(N * (N * log(N))) \rightarrow O(N^2 * log(N))$.

6 Comparing the results

Now that we have established the previous methods to solve the problem, it now makes sense to compare them on different size of solution to check firstly which give the best results in term of **objective function** and **execution time**. To do so, we have generated instances for the problem of size 10, 20, 30, 40, 45 and 50 (3 different instances for each size of input that you can find in the atatched files) and we ran the different method for each one.

For each input, we observed that the CPLEX algorithm gave us the best result for each time and so we will use it as the comparison point. Doing so, we obtain the following graphs:

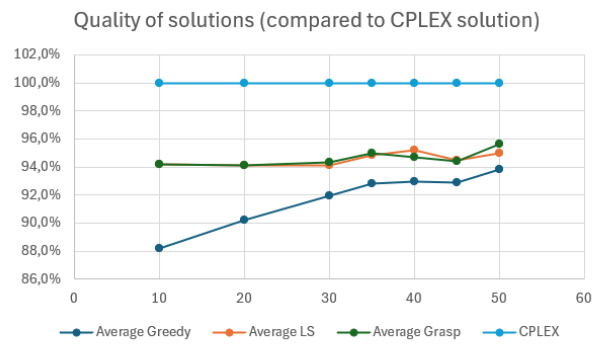

Figure 6.1: Comparison of the results for different instance sizes

As we can see, we clearly have that the greedy solution gives worse results than the other algorithms. Then, come the Greedy + Local Search and the GRASP (including local search) methods. Indeed, the use of the local search allow to obtain way better solutions. The GRASP method seems to give better results than the classical Greedy + Local search for higher set of data, which would make sense given that alpha is tuned to 0,1.

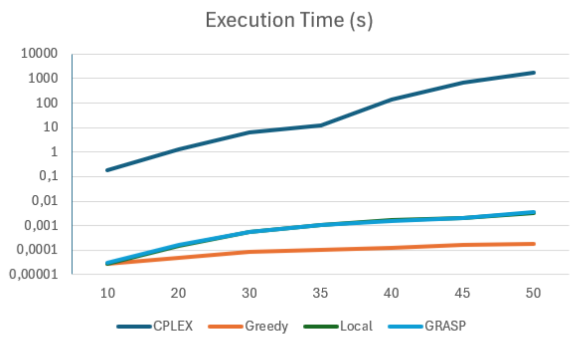

Figure 6.2: Comparison of the results for different instance sizes

We also got an expected (logarithm) graph when plotting the execution time. The CPLEX resolution is much more time consuming than the other methods used (more than 1000 times). Then we also see that both algorithms using local search have very similar execution time (the line lines for GRASP and Local Search are barely discernible on the graph).

To sum up, it seems more efficient and rational to use a local search algorithm to compute a solution, despite the solution obtained is around 95% of the best solution that CPLEX could compute. Indeed, the time difference is such that CPLEX algorithm quickly reach their limit even when defining a 1% tolerance gap.