Algorithmic Methods for Mathematical Models

Final Project

2022-2023

NIKOLAOS BELLOS

ANASTASIOS PAPAZAFEIROPOULOS

FORMAL STATEMENT

Given a positive integer n, the goal is to find a set of n different natural numbers t0, t1, ..., tn-1 such that:

1) No two different pairs of numbers in the set are the same distance apart.
2) The difference between the maximum and the minimum numbers in the set is minimized.

To elaborate further:

- The problem arises from designing a scientific experiment to determine the period of a sinusoidal sound wave.
- The period must be a multiple of T seconds.
- By measuring the sound pressure at specific instants of time, we can infer the period based on the measured values.
- Equidistant measurements provide information for periods up to 11T, except for periods of 6T or longer.
- The efficient utilization of measurements is achieved by ensuring that no two pairs of numbers in the set have the same distance apart.
- The larger the difference between the maximum and minimum numbers in the set, the more periods will be missed.
- The objective is to find an optimal set of n natural numbers that satisfies the above conditions.
- The example provided demonstrates that the sets {0, 1, 4, 9, 11} and {0, 2, 7, 8, 11} are optimal when n = 5, as they minimize the difference between the maximum and minimum numbers while ensuring no two pairs have the same distance apart.

ILP MODEL

Parameters (Inputs):

The number n, which represents the amount of different natural numbers (number of measurements).

Outputs:

- A set of n natural numbers (without duplicate differences among pairs)

- A fitness score N which is the difference between the first and the last element of the set.

Variables:

The goal is to distinguish which distances there are between the numbers of the set and which not.

- We need a set X, which holds the numbers that make our solution.
  Its size is equal to the amount of numbers we have to find, which is n
  $$X_t[n] \quad , n: 1..N$$
- We also need a 2D array of integers to store all the differences of the pairs from the set, in order to perform comparisons among them.
  For every pair $X_t[i]$, $X_t[j]$, we store their difference in the cell $X_d[i][j]$
  $$X_d[n_i][n_j] \quad , n_i: 1..N, n_j: 1..N$$

Constraints:

1) There is no meaning for the first measurement instance to be greater than 0, because we focus only on the differences between the number pairs in the set. Therefore, we suppose that:
   $$x_1 = 0.$$
2) The measurements take place in chronological order. Therefore, they should be in ascending order. :
   * Anything different would just increase the complexity of the problem without a reason
   $$x_i < x_j : \forall i, j \;\; 1 \leq i, j \leq N \;\; , j < i$$
3) No two different pairs of numbers in the set should have the same distance apart, so the last constraint is:
   $$d_{ij} \neq d_{kl} : \forall i, j, k, l \; , 1 \leq i, j, k, l \leq N \;\; , j < i, l < k, \{i, j\} \neq \{k, l\}$$

<u>* Improvement for CPLEX: (optional)</u>

4) We can apply an upper bound for the last number in the set. In particular, we know that the last number S[n] cannot be greater than $2^{n-1} - 1$. The reason for that is simple and we explain it through the following example. We represent the set S as an array to make the supervision simpler. First, we have the initial set S = {0, 1}. Then, we append every time the next element which is generated always from the the formula:
   $S[n] = 2 \cdot S[n-1] + 1$ (1). By this trick, we achieve not overlapping distances between the elements, because when we append a new element all the new distances belong to another family of distances which are greater than the previous ones. The smaller distance from S[n], is the $d = S[n] - S[n-1]$, using the formula (1):
   $d = 2 \cdot S[n-1] + 1 - S[n-1] = S[n-1] + 1$, where S[n-1] was the greatest distance so far. Therefore, by this way, we achieve not overlapping distances.

We introduce the execution of the algorithm below:
1st iteration:

0	1

2nd iteration:

0	1	3

3rd iteration:

0	1	3	7

4th iteration:

0	1	3	7	15

…
As we can notice, for example, the distance between the elements 7 and 15 is $d_{15,7}$=8, while the greatest distance so far was the $d_{7,0} = 7$.
Inferentially, it is proven that the upper bound of last element S[n] is $2^{n-1} - 1$, where n is the number of measurements.

Objective Function:
We want to minimize the difference between the maximum and the minimum numbers in the set.
In other words, we want to minimize the maximum number (depending on the second constraint), namely x_n, where n is the number of measurements.
* *Another objective function, equal to the above, is to minimize the number of unsatisfied differences between the number pairs in the set.*
(A set of N numbers has N(N-1)/2 possible differences among its pairs. Therefore, the unsatisfied differences are equal to (the maximum number on the list) - (possible differences among pairs).*
So the objective function is :

minimize x_n.

Eventually, the ILP is the following:

min x_n, where n is the number of measurements

s.t. $\forall\, i, j,\ i < j\colon\ x_i < x_j$

$\forall\, i, j, k, l,\ j < i,\ l < k\colon d_{i,j} \neq d_{k,l}$

$x_1 = 0.$

HEURISTICS (Pseudocode)

GREEDY

```
# IMPLEMENTATION 1: Exponential steps
# * Why this works is explained in the report

# 1. Create a list of length n
# 2. Set the first element of the list to 0
# 3. Increment the rest of the elements by exponential of 2

# numbers: list of length n to hold the numbers (solution)

function greedy():
    numbers[0] = 0
    for (i=0 to n-1):
        numbers[i+1] = numbers[i] + 2^i

    return numbers


# IMPLEMENTATION 2: Iterative approach

# 1a. Create list of length n
# 1b. (diffMatrix) Create a 2D matrix to store the difference between the
numbers
# 1c. (diffCounters) Create a list to store the counters of the differences
occurrences
# 2. Set the first element of the list to 0
# 3. Until the list is complete:
# 4. Get the smallest unused difference from the diffCounters list (first 0
position)
# 5. Add this difference to the last element of the list and check if it is
feasible
# 6. If it is feasible, add it to the list
# 7. Update the diffCounters, diffMatrix and the pointer
# 8. If it is not feasible, get the next smallest unused difference from the
diffCounters list (second 0 position)

# numbers: list of length n to hold the numbers (solution)
# diffMatrix: 2D matrix to hold the differences between the numbers
# diffCounters: list to hold the counters of the differences occurrences

function greedy():
    numbers[0] = 0
    smallestDiff = getSmallestUnsusedDifference()
    # Until the list is complete:
```

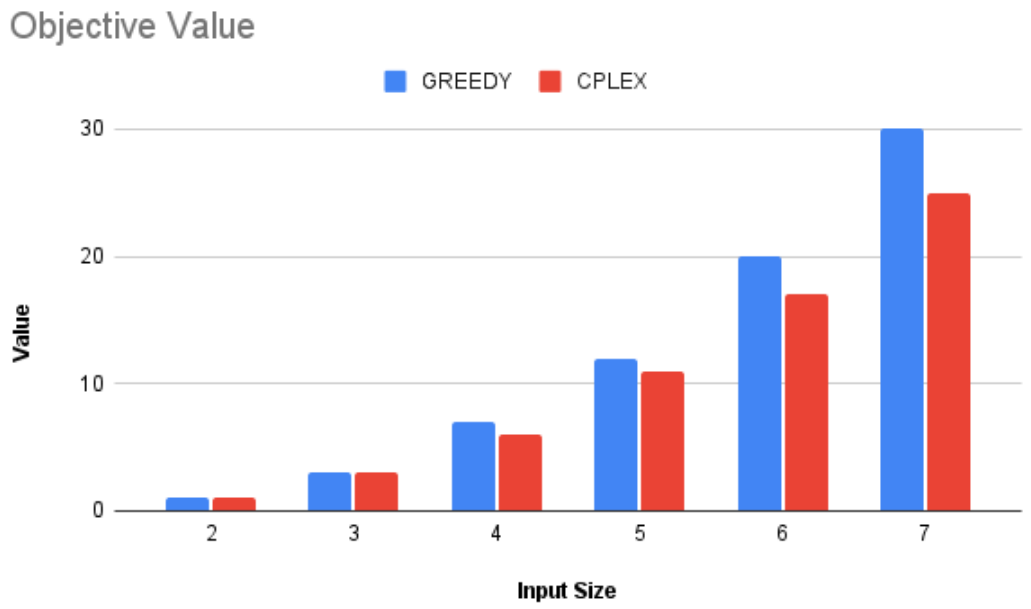
```
    while length(numbers) < n:
        # Get last element + smallest unused difference
        candidate_number = numbers[-1] + smallestDiff
        # Check if it is feasible (no conflicting differences)
        if (isFeasibleToAdd(candidate_number)):
            # Add it to the list
            numbers.add(candidate_number)
            # Update the diffCounters, diffMatrix
            updateDiffCounters(candidate_number)
            updateDiffMatrix(candidate_number)
        # Get the next smallest unused difference
        smallestDiff = getSmallestUnsusedDifference(smallestDiff)

    return numbers
```

LOCAL SEARCH

```
# 1. Get the initial solution (Greedy)
# 2. Decrease the last element by 1 (fitness score)
# 3. Until you find a feasible solution OR time limit is exceeded:
# 4a. Increase the element before the last one by 1
# 4b. If the element becomes equal to the next one on the list
#   - reset it to the initial value (greedy solution)
#   - increase the previous element by 1
# 5. Update the diffCounters, diffMatrix tables
# 6. Check if solution if feasible
# 7. If it is feasible, update the fitness score + Go to step 3
# 8. If it is not feasible, go to step 5

# solution: the initial solution that the algorithm starts from
# diffMatrix: 2D matrix to hold the differences between the numbers
# diffCounters: list to hold the counters of the differences occurrences

function local_search():
    solution = greedy()
    # copy of the initial solution
    neighbor = solution
    while (time < time_limit) OR existsAvailableDifferences(solution): #
Decrease the last element by 1 until time limit is exceeded
        neighbor[-1] -= 1
        # try different combinations starting from the element before the
last one
        combination_idx = -2
        # Try all different combinations until you find a feasible solution
        while True:
            neighbor[combination_idx] += 1
```

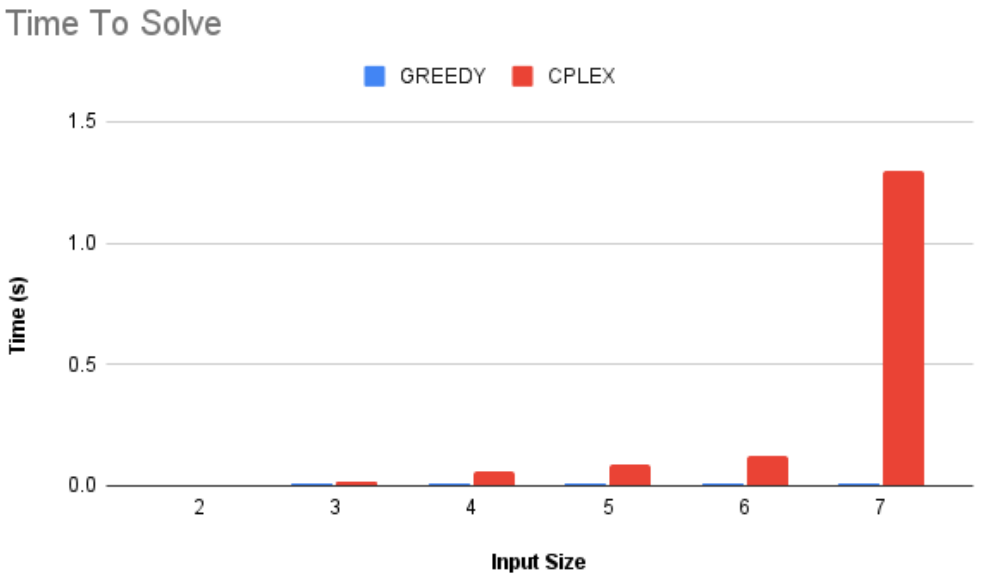
```
            # If the element becomes equal to the next one on the list
            if neighbor[combination_idx] == neighbor[combination_idx+1]:
                # reset it to the initial value (greedy solution)
                neighbor[combination_idx] = solution[combination_idx]
                combination_idx -= 1
                continue
            else if (combination_idx < -2):
                combination_idx += 1
            updateDiffMatrix(neighbor)
            updateDiffCounters(neighbor)
            # If solution is feasible, it is better than the previous one
because of the dicreased last element
            if isFeasible(neighbor):
                solution = neighbor
                fitness = neighbor.fitness
                break

    return solution
```

COMPARISON OF HEURISTICS



EVALUATION

Regarding the objective function / fitness score, it is obvious that the CPLEX outperforms the default Greedy algorithm in terms of the quality of the solutions. While the difference is not big, when it comes to big datasets it could get much larger.



EVALUATION

Regarding the time it takes for each algorithm to produce results, it is obvious that CPLEX takes increasingly more time, as it explores exponentially more solutions. In contrast, Greedy uses an almost linear constructive algorithm that produces results fast (even though they are not the most qualitative ones).

HOW TO RUN THE CODE

CPLEX

1. Create an OPL Project in CPLEX and use the files that are contained in the following folder :
   (Project Folder) > CPLEX
2. Create a configuration inside the OPL Project and place the main.mod file inside it
3. To run the program, Run the configuration (right click > run this)

- To change the input file:
  Change the data source inside the main.mod file
  'var data = new IloOplDataSource("P1.dat"); '

HEURISTICS

1. Run the Main.py file inside the Project Folder
   'python Main.py'

* no configuration needed (ex. for pyCharm)
* **Default Algorithm:** Greedy (without local search)

- To change the input file / heuristic algorithm / algorithm parameters: Change the appropriate data inside the following configuration file: (Project Folder) > config > config.dat