Algorithmic Methods for Mathematical Models Project

Bertille Temple
David Masek

May 25, 2023

1 Formal statement of the problem

For a given number n we want to find a set of different natural numbers such that no two different pairs of numbers in the set are the same distance apart and the difference between the maximum and minimum numbers in the set is minimized.

INPUTS:
n is a given natural number.

OUTPUTS:
X is a set of n natural numbers, where the distance between any two pairs is unique. We can order the elements to index them: $\forall 1 \leq i \leq n$. If we denote D as an array of differences: $d_{i,j} = x_i - x_j$, then we want to find X such that

$$\forall 1 \leq i, j, k, l \leq n : ((i, j) = (k, l)) \vee (i = j \wedge k = l) \iff |d_{i,j}| = |d_{k,l}|$$

$$((i, j) \neq (k, l)) \wedge (i \neq j \vee k \neq l) \iff |d_{i,j}| \neq |d_{k,l}|$$

We can see that:

$$i \neq j \wedge i \neq k \wedge k \neq l \wedge j \neq l \implies ((i, j) \neq (k, l)) \wedge (i \neq j \vee k \neq l)$$

Thus:

$$i \neq j \wedge i \neq k \wedge k \neq l \wedge j \neq l \implies |d_{i,j}| \neq |d_{k,l}|$$

This will be useful in the model implementation in opl.

OBJECTIVE FUNCTION:
The objective is to minimize the distance between the smallest and largest elements of X: $\text{minimize}(\max(X) - \min(X))$. Since there is always a valid solution containing zero we can consider $\min(X) = 0$ and simplify the objective to

$$\text{minimize} \max(X)$$

2 Integer linear program model

INPUT PARAMETERS:
n is the number of integers to consider in the set.

DECISION VARIABLES:

- $d_{i,j}$ is the difference $x_i - x_j$, for $1 \leq i, j \leq n$
- $av_{i,j} = |d_{i,j}| \implies -av_{i,j} \leq d_{i,j} \leq av_{i,j}$ and $av_{i,j} \geq 0$
  To give $av_{i,j}$ the intending meaning in opl, we introduce the binary variable:
  $e_{i,j}$ is 1 if $d_{i,j} - av_{i,j} \geq 0$, which means that $d_{i,j} \geq av_{i,j}$.
  Knowing that $d_{i,j} \leq av_{i,j}$, we deduce that $e_{i,j}$ is 1 if $av_{i,j} = d_{i,j}$.
  $e_{i,j}$ is 0 if $d_{i,j} + av_{i,j} \leq 0$, which means that $d_{i,j} \leq -av_{i,j}$.
  Knowing that $d_{i,j} \geq -av_{i,j}$, we deduce that $e_{i,j}$ is 0 if $av_{i,j} = -d_{i,j}$.
- $dif_{i,j,k,l}$ is the difference $d_{i,j} - d_{k,l}$, for $1 \leq i, j, k, l \leq n$
- $avd_{i,j,k,l} = |dif_{i,j,k,l}| \implies -av_{i,j,k,l} \leq dif_{i,j,k,l} \leq avd_{i,j,k,l}$ and $av_{i,j,k,l} \geq 0$
  To give $av_{i,j,k,l}$ the intending meaning in opl, we introduce the binary variable:
  $g_{i,j,k,l}$ is 1 if $dif_{i,j,k,l} - avd_{i,j,k,l} \geq 0$,
  which means that: $d_{i,j} - d_{k,l} - |d_{i,j} - d_{k,l}| \geq 0$
  $g_{i,j,k,l}$ is 0 if $dif_{i,j,k,l} + av_{i,j,k,l} \leq 0$
  We can make the same reasoning as for $av_{i,j}$ to deduce that:
  $g_{i,j,k,l}$ is 1 if $avd_{i,j,k,l} = dif_{i,j,k,l}$
  $g_{i,j,k,l}$ is 0 if $avd_{i,j,k,l} = -dif_{i,j,k,l}$
- $b_{i,j,k,l}$ is 1 if $avd_{i,j,k,l} \leq 0$, which means that
  $|d_{i,j} - d_{k,l}| \leq 0$. As $|d_{i,j} - d_{k,l}| \geq 0$, we obtain that $|d_{i,j} - d_{k,l}| = 0$,
  which means that $d_{i,j} = d_{k,l}$
  Literally, $b_{i,j,k,l}$ is one if the distance between x_i and x_j is the same as the distance between x_k and x_l, 0 else.
- x_i is the number at position i in the set X, $1 \leq i \leq n$.
- z is the maximum number in X.

OBJECTIVE FUNCTION:

$$\text{minimize } z$$

CONSTRAINTS:

- X is ordered and contains unique numbers: $\forall 1 \leq i < j \leq n : x_i + 1 \leq x_j$
- The zero is always contained in the set: $x_1 = 0$
- z has the intended meaning of maximum (holds for optimum, since we minimize z): $\forall 1 \leq i \leq n : z \geq x_i$
- $av_{i,j}$ has the intended meaning:
  $d_{i,j} - av_{i,j} >= -2 \cdot 11 \cdot (1 - e_{i,j})$;
  $d_{i,j} + av_{i,j} <= 2 \cdot 11 \cdot e_{i,j}$;
- $avd_{i,j,k,l}$ has the intended meaning:
  $dif_{i,j,k,l} - avd_{i,j,k,l} >= -2 \cdot 11 \cdot (1 - f_{i,j,k,l})$;
  $dif_{i,j,k,l} + avd_{i,j,k,l} <= 2 \cdot 11 \cdot g_{i,j,k,l}$;
- no two different pairs of numbers in the set are the same distance apart which can be written as :

$$\sum_{k=1}^{N} \sum_{j=1}^{N} \sum_{l=1}^{N} b[i,j,k,l] == 2, 1 \leq i \leq n, i \neq k$$

  or

$$\sum_{j=1}^{N} \sum_{l=1}^{N} b[i,j,i,l] == n, 1 \leq i \leq n$$

This model has 3 problems:
first the opl output is wrong for n=4. We think this is due to the last constraint which is not enough to enforce that all distances are different in the set.
Then, when n is larger than 4, the computational time increases a lot, which might be related to the integer property of the problem. To tackle this issue, the problem can be relaxed by removing the integer constraint.
Finally, an **upper bound** should be chosen to give the absolute values variables their intending meaning, and opl does not accept an upper-bound which includes n as n is a decision variable. Here we selected 11 which is the upper bound for n=5 but it should be updated when looking for n larger.

We looked for another way to ensure that no pairs are the same distances apart. We used the following implication found in the first part:

$$i \neq j \wedge i \neq k \wedge k \neq l \wedge j \neq l \implies |d_{i,j}| \neq |d_{k,l}|$$

which we translated in opl into the following quadratic constraint:

$$\begin{aligned} &\forall (i,j \in N : i < j) \\ &\quad \forall (k,l \in N : k < l) \\ &\qquad (i \neq k \wedge j \neq l) \Rightarrow d[i,j] \neq d[k,l]; \end{aligned}$$

We created a new model with this constraint, which reduces a lot the problem complexity because it does not imply to use absolute values nor binary variables(consequence of $i < j$ and $k < l$).

3 Greedy constructive algorithm

The idea of the greedy constructive algorithm is to iteratively select the best candidate from a candidate list and add it to the partial solution, until the partial solution turns to be a complete one. For our problem we use:

- initial solution: an empty set
- the candidates: sets with one more element (non-negative integer)
- feasibility function: checks if the distance between any two pairs of elements is unique
- objective function q: maximum of the set.
- selection function: argmin
- solution function: checks that the set contains n numbers

Due to a nature of the given problem, there is theoretically an infinite number of possible candidates (all non-negative integers). In practice we'll implement a limit on the number of possible candidates. We call this limit M. Further, we add only feasible solutions to the candidate set, since we want our results to be feasible (and unfeasible solutions would never be selected anyway).

The choice of M is not important for the greedy algorithm, as we only pick the best (greedy) solution and in our case we know that this is the lowest feasible number. If we build the candidate set from the lowest number, then $M = 1$ is sufficient for the greedy algorithm. However allowing bigger candidate set will be important for the GRASP algorithm later.

The algorithm is described with the folowing pseudocode:

```
n = ... # input, target set size
M = 1 # hyperparameter, size of the candidate set
C = {} # candidate set
X = {} # solution
while |X| < n:
    # update C
    C = {}
    c = max(X) # 0 for empty set
    while |C| < M:
        if is_feasible( X ∪ {c} ):
            C.insert({c})
            c += 1
    # evaluation function
    q(c, X) = max(X ∪ {c}) = c
    # selection function
    c_best = argmin ( q(c, X) for c in C )
    # update solution
    X = X ∪ {c_best}
return q(X), X
```

4 Greedy constructive algorithm with local search procedure

The idea of the local search is to iteratively improve on a feasible solution. We use the greedy algorithm from the previous section to obtain a feasible solution. The main part of the local search is the neighbours generation.

Since only adding/removing numbers from the set would make the solution infeasible we only consider exchanging (removing and adding the same number of elements). Since we want to find better solutions one of the exchanged numbers has to be the largest number in the set. Other numbers can be chosen freely. As for the new numbers to be added to the set, it makes sense to only consider numbers lower than the current maximum, otherwise no improvement will happen. We also discard numbers that would make the solution infeasible.

We need to select reasonable number of exchanges to make. Making more exchanges each step provides more power to the algorithm and a chance for better solutions, however it also increases the code complexity.

Additionally we need to decide if we choose the first improving neighbour or if we first generate all of them.

The algorithm is described with the following pseudo-code:

```
given a solution S and a parameter K
while S is not locally optimal:
    nbs = {}
    for each combination C of K numbers from S:
        # remove K numbers
        S_r = S \ C
        for each combination C' of K numbers from {0..max(S)
            }:
            # add K numbers
            S_ex = S_r ∪ C'
            if |S_ex| = |S| and is_feasible(S_ex) and max(
                S_ex) < max(S):
                nbs.insert(S_ex)

    if empty(nbs):
        break # S is locally optimal
    else:
        S = argmin( q(S_ex) for S_ex in nbs )
return S
```

5 GRASP

The GRASP consist of two steps, a randomised alternative to greedy search and a local search.

The algorithm is described with the following pseudo-code:

```
n = ... # input, target set size
M = ... # hyperparameter, size of the candidate set
alpha = ... # hyperparameter
X_best = None
for k=1 ... max_iterations:
    # construction phase
    X = {} # solution
    while |X| < n:
        # update C
        C = {}
        c = max(X) # 0 for empty set
        while |C| < M:
            if is_feasible( X ∪ {c} ):
                C.insert({c})
                c += 1
        # evaluation function
        q(c, X) = max(X ∪ {c}) = c
        # selection function
        RCL = {c ∈ C | q(c) ≤ q_min + α(q_max − q_min)}
        c_selected = select_random ( RCL )
        # update solution
        X = X ∪ {c_selected}
    # local search phase, same as in previous part
    X = local_search(X)
    if q(X) < q(X_best): X_best = X
return X_best
```

The restricted candidate list (RCL) is constucted using the following formula:

$$\mathrm{RCL}_{\min} = \{c \in C \,|\, q(c) \leq q_{\min} + \alpha(q_{\max} - q_{\min})\}$$

6 Hyperparameter tuning and instance generation

Instance generation for this problem is trivial since we only generate the number n.

We evaluated generated instances with CPLEX using our IP model. The largest instance we could solve under 30 minutes was $n = 9$ so we evaluate and compare the approaches on $2 \leq n \leq 9$.

6.1 K tuning and improvement strategy for local Search

We evaluate first improvement and first generating all neighbours and selecting the best one, for different k between 2 and 4. Generating all neighbours took

more time in our experiments and did not consistently improve the results, thus we chose a first improvement strategy. The difference can be seen in figure 1.

Regarding the the k values, the bigger values generally lead to better results but the time complexity rises quickly. For k=2 the quality of the result was too poor, and for k=4 the computing time increases a lot. We decided k=3 would be a good trade-off between computing time and the solution quality. The figures are in table 1.

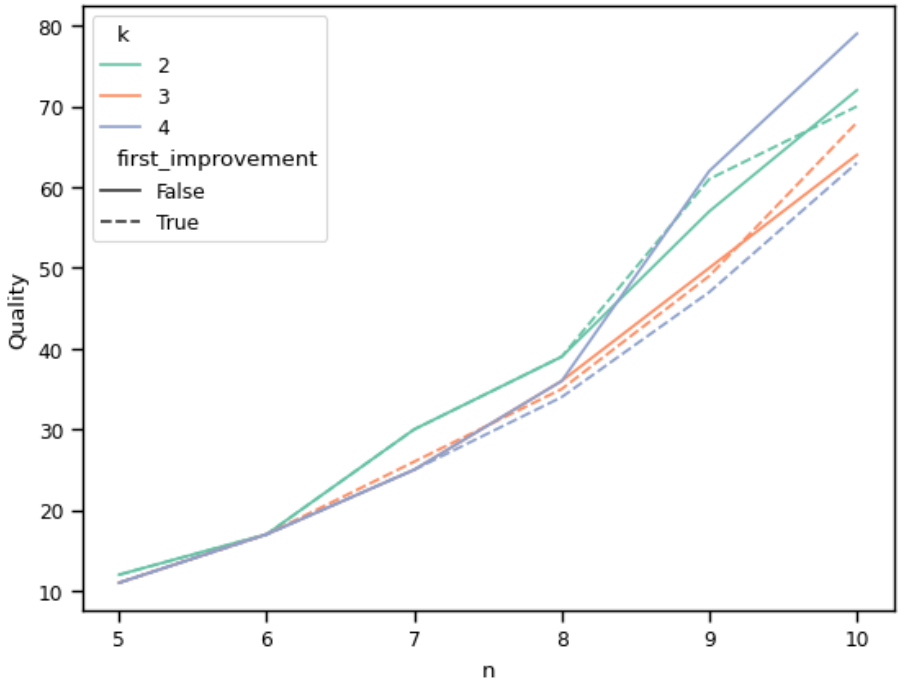

Figure 1: k tuning for local search (from greedy solutions)

n	k	t[s]	q	q*
3	2	0.000039	3	3
4	2	0.000136	6	6
5	2	0.000379	12	11
6	2	0.003189	17	17
7	2	0.007208	30	25
8	2	0.043947	39	34
9	2	0.093353	61	44
3	3	0.000053	3	3
4	3	0.000087	6	6
5	3	0.000967	11	11
6	3	0.011648	17	17
7	3	0.105622	26	25
8	3	0.677661	35	34
9	3	1.923836	49	44
3	4	0.000021	3	3
4	4	0.000031	7	6
5	4	0.000957	11	11
6	4	0.030839	17	17
7	4	0.477426	25	25
8	4	6.429909	34	34
9	4	35.111204	47	44

Table 1: k tuning figures for local search

6.2 Alpha tuning for GRASP

Alpha is the percentage of candidate we will select from the Candidate List to build the Random Candidate List (RCL) in GRASP. When alpha is 0, the GRASP is equivalent to Greedy + Local Search, when alpha is 1, GRASP is completely random. Which alpha should we select ? Here comes the need for tuning alpha. To do so, we will study the relative gap, which is a percentage that evaluates the difference between the optimal solution and the new one found with GRASP. So we computed the optimal gap for different alpha between 0 and 1, and we repeated the process for different n between 3 and 7. Remind that n is the number of elements in the set.

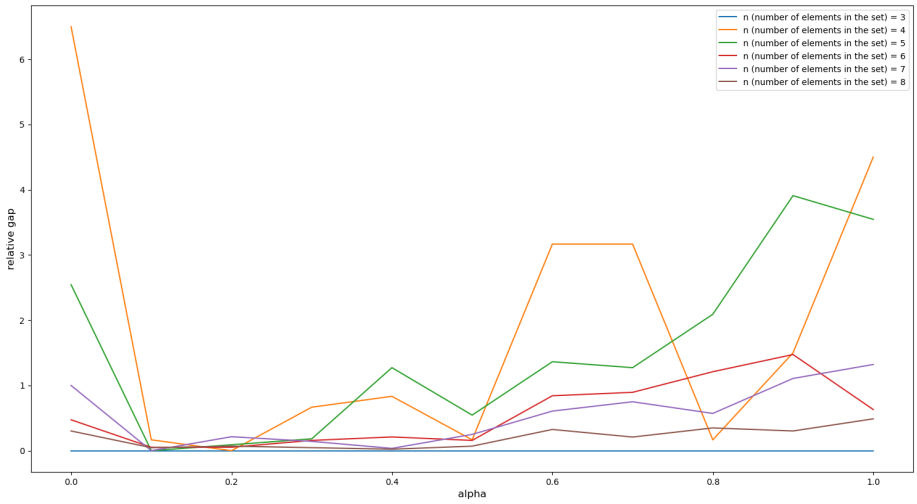

Figure 2: Relative gap between grasp and optimal solution for different alpha

As a result, we obtain the figure 2. As the goal is to minimise the relative gap between the new solution found and the optimum, the best alpha seems to be 0.1 or 0.2. Note that we do not take into account the computational time but only the quality of the solution.

7 Comparison

We compare the CPLEX IP solver, Local Search and GRASP algorithms. The largest n we can solve using CPLEX under 30 minutes is $n = 9$, so we evaluate on n between 2 and 9. We compare solution quality (table 2, figure 3) and required time (table 3, figure 4).

We can see that the GRASP algorithm achieves good results, the difference from optimal solution is only 1 for the largest instances and the solution is optimal otherwise. Results for Local Search are similar to GRASP, but for the largest instance we can see it is noticeably worse and we can assume the difference would increase for larger n.

The required time rises quickly for bigger n for all algorithms. For the largest instance $n = 9$ the GRASP is approximately ten times faster than the CPLEX. The Local Search is approximately thirty times faster than the GRASP. We can see that the time differences between the algorithms are significant.

Algorithm n	CPLEX	GRASP	Local Search
2	1	1	1
3	3	3	3
4	6	6	6
5	11	11	11
6	17	17	17
7	25	25	26
8	34	35	35
9	44	45	49

Table 2: Comparison of algorithms: quality of the found solution

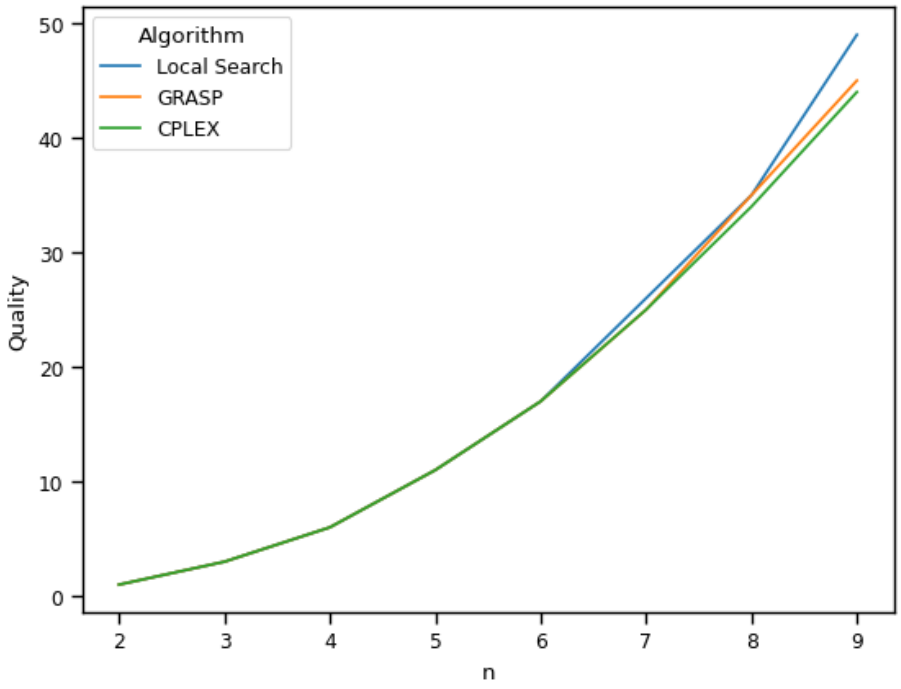

Figure 3: Comparison of algorithms: quality of the found solution

Algorithm n	CPLEX	GRASP	Local Search
2	0.00	0.00	0.00
3	0.02	0.00	0.00
4	0.01	0.00	0.00
5	0.04	0.02	0.00
6	0.14	0.19	0.01
7	1.50	2.44	0.11
8	24.66	12.19	0.68
9	610.09	58.46	1.92

Table 3: Comparison of algorithms: time in seconds

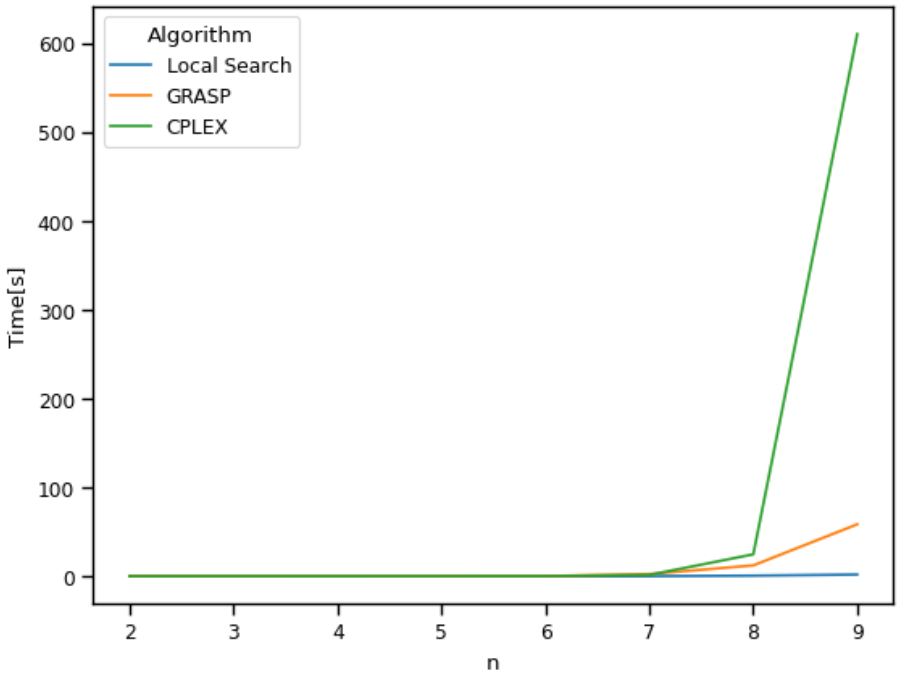

Figure 4: Comparison of algorithms: time in seconds

8 Conclusion

The starting point of this report was to optimise the time when should be taken a measurement of the sound pressure for a given number of measures, to obtain the maximum of information about the period of a sinusoidal sound wave. The final goal is to compare different optimisation algorithms to solve this problem. The first step to tackle this optimisation problem was to formalise it, which means to identify the input parameters, the variables, the constrains and the objective function. Then, we modelled the problem as an integer linear problem, but unable to find an accurate linear constraint, we finally implemented the problem in opl as a quadratic problem. Then we implemented it in python with 2 meta-heuristics algorithms: greedy+local search and GRASP. In this step, we had to choose strategies and parameters. To do so, we tested several ones and made the choice which seems to be the best trade off between computational time, quality of the solution and complexity of the model. So in local search, we selected a first improvement strategy and k = 3 (neighbours to exchange). In GRASP, we selected alpha = 0.1. At the end of the day, CPLEX is the best in term of solution quality, but does not scale well, and is bad in term of computational time. At the opposite, Greedy+ Local Search obtains the relatively worse solutions (but in absoulte, they are quite good) and the best computaional time. GRASP is between them, with good solutions, but not the best, and good computational time, but not the best.