Minimum Array Partition

Daniil Grbić, Ivan Gogić

September 29, 2023

1 Problem description

Given a $n \times n$ array A of non-negative integers, and a positive integer p, we need to choose $p-1$ horizontal dividers $0 = h_0 < h_1 < \cdots < h_p = n$ and $p-1$ vertical dividers $0 = v_0 < v_1 < \cdots < v_p = n$ partitioning A into p^2 blocks, so as to minimize the following expression:

$$\max_{\substack{1 \le i \le p \\ 1 \le j \le p}} \sum_{\substack{v_{i-1} \le x \le v_i \\ h_{j-1} \le y \le h_j}} A[x, y]$$

2 Methods

2.1 Brute Force (BF)

Firstly, we write a brute force algorithm for the purpose of finding optimal solutions for small test cases, which we will use to evaluate our optimisation algorithms.

BF generates all possible partitions of the matrix, and chooses the best one of them. This essentially means that BF will need an unreasonable amount of time to solve all but the smallest of testcases.

The complexity of this approach is $O(n, p) = \binom{n}{p}^2 * p^2$, indeed $\binom{n}{p}^2$ is the total number of partitions, while for each of them we need to check p^2 blocks. We find the sum of each block in constant time, by utilising a partial sum DP matrix.

2.1.1 Cutting optimization

A small optimization of the previous approach involves abandoning some partitions early when it is clear that they will not yield a better solution than the current known best.

Consider this: we know the sum S of the entire matrix M, and we know there are p^2 blocks in total. According to the Pigeonhole principle, at least one of said blocks will have a sum greater or equal to $\frac{S}{p^2}$.

Alternatively, if we have checked $m < p^2$ blocks and their total sum is $S_m < S$, then the maximum value of the remaining blocks will be greater or equal to $\frac{S - S_m}{m - p^2}$.

If at any point this sum happens to become greater than the best solution we have found beforehand, we can safely skip checking the rest of the partition in question. In practice, this speeds up the algorithm 3 to 5 times.

2.2 Genetic Algorithm (GA)

A genetic algorithm (GA) is a p-metaheuristic inspired by the process of natural selection. Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on biologically inspired operators such as mutation, crossover and selection.

We use a nearly textbook GA implemenation. Nevertheless, given the specificity of the problem, crossover and mutation implementation deserve a closer look.

2.2.1 Crossover

Crossover of two partitions can be done by an arithmetical approach. For simplicity, we will explain the process on a one-dimensional partition. This can be directly applied to two dimensions for the problem in question.

Consider two individuals, A and B, with partitions $[1, 3, 9]$ and $[2, 5, 7]$. Their child C will try to take a middle path in the hopes of bettering their fitness, by taking the average of respective divider pairs: $C = [\frac{1+2}{2}, \frac{3+5}{2}, \frac{9+7}{2}] = [1.5, 4, 8]$. Since dividors have to be whole numbers, we fix C by either rounding the first element down, or rounding it up (chosen randomly).

A problem arises when multiple dividors end up on the same spot after crossover, we solve this by putting the problematic dividers in random unoccupied spots. This guarantees all individuals are feasible, while improving their fitness.

2.2.2 Mutation

Mutation is achieved by trying to make small movements of individua dividers with a given probability to do so, considering there is an available place to move.

2.3 Simulated Annealing (SA)

Simulated Annealing is a probabilistic optimization method inspired by annealing in metallurgy. It's effective for combinatorial optimization problems with large search spaces. SA starts with an initial solution and uses a temperature schedule to accept worse solutions with a probability.

During the SA process, small changes are introduced through random movements of block borders, allowing for exploration of the solution space.

2.4 Variable Neighbourhood Search (VNS)

Variable Neighborhood Search (VNS) is an optimization technique that iteratively explores different neighborhoods of solutions. It begins with an initial solution and systematically varies the neighborhood structure to search for better solutions.

During the VNS process, changes are introduced through random movements of (multiple) block borders.

3 Method evaluation

3.1 Test generation

We wrote a helper script for generating testcases. The script takes 3 command line arguments: n, p, and *type*; where *type* refers to how the matrix M is generated:

Ones fills M with ones, mainly used for debugging

Random fills the matrix with numbers from $(1, n/2)$

Linear , $M_{i,j} = i + j + 1$

Squared , $M_{i,j} = i^2 + j^2 + 1$

For the purpose of evaluating our optimization algorithms, we created 3 sets (random, linear, and squared) of 4 testcases of different dimensions. The dimensions can be categorized as:

Tiny $n = 50$, $p = 3$ (BF friendy)

Small $n = 80$, $p = 6$

Medium $n = 100$, $p = 4$

Large $n = 500$, $p = 10$

Each optimization algorithm is run 5 times for every testcase, and we record the best, worst, and average results and execution times.

We only test BF on tiny testcases to compare the results from optimization algorithms. We run BF once per testcase, since there is no randomness involved.

3.2 Results

Let us begin with testcases representing the *random* type:

```
Testcase: tests/random_n50_p3.in
  BRUTE FORCE
    Result   : 3769
    Time (s) : 2.46
  GENETIC ALGORITHM (GA) [pop_size=75, num_iters=450, elitism_size=15]
```

```
    Results  : BEST=3769       WORST=3769       AVG=3769.00
    Time (s) : BEST=2.11       WORST=2.28       AVG=2.20
  SIMULATED ANNEALING (SA)
    Results  : BEST=3769       WORST=3769       AVG=3769.00
    Time (s) : BEST=0.28       WORST=0.29       AVG=0.28
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=2500]
    Results  : BEST=3769       WORST=3769       AVG=3769.00
    Time (s) : BEST=0.46       WORST=0.48       AVG=0.47

Testcase: tests/random_n80_p6.in
  BRUTE FORCE - skipping due to speed
  GENETIC ALGORITHM (GA) [pop_size=100, num_iters=600, elitism_size=20]
    Results  : BEST=3964       WORST=4072       AVG=3985.60
    Time (s) : BEST=7.86       WORST=8.07       AVG=7.94
  SIMULATED ANNEALING (SA)
    Results  : BEST=3964       WORST=3964       AVG=3964.00
    Time (s) : BEST=2.18       WORST=2.27       AVG=2.21
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=6400]
    Results  : BEST=3964       WORST=3964       AVG=3964.00
    Time (s) : BEST=8.26       WORST=8.45       AVG=8.35

Testcase: tests/random_n100_p4.in
  BRUTE FORCE - skipping due to speed
  GENETIC ALGORITHM (GA) [pop_size=100, num_iters=600, elitism_size=20]
    Results  : BEST=17061      WORST=17061      AVG=17061.00
    Time (s) : BEST=5.06       WORST=5.22       AVG=5.11
  SIMULATED ANNEALING (SA)
    Results  : BEST=17061      WORST=17061      AVG=17061.00
    Time (s) : BEST=0.68       WORST=0.68       AVG=0.68
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=10000]
    Results  : BEST=17061      WORST=17061      AVG=17061.00
    Time (s) : BEST=4.75       WORST=4.85       AVG=4.80

Testcase: tests/random_n500_p10.in
  BRUTE FORCE - skipping due to speed
  GENETIC ALGORITHM (GA) [pop_size=100, num_iters=600, elitism_size=20]
    Results  : BEST=326181     WORST=338514     AVG=330243.80
    Time (s) : BEST=24.44      WORST=25.95      AVG=24.95
  SIMULATED ANNEALING (SA)
    Results  : BEST=360022     WORST=552450     AVG=424930.40
    Time (s) : BEST=9.21       WORST=10.39      AVG=9.89
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=10000]
    Results  : BEST=323192     WORST=417608     AVG=362644.00
    Time (s) : BEST=24.83      WORST=28.67      AVG=26.29
```

For this first type of matrices, we can see that SA tends to find optimal (or close to optimal) solutions at a fraction of the time for tiny, small, and medium tests. However, SA falls behind

on the large test case, displaying a big difference between its best (360022) and worst (552450) runs.

Both GA and VNS seem clearly better than SA, however our genetic approach has the better average result across all 5 runs, indicating it is much less reliant on chance to find a good solution.

Next, let us compare the results achieved on the *linear* test type.

```
Testcase: tests/linear_n50_p3.in
  BRUTE FORCE
    Result   : 15080
    Time (s) : 3.18
  GENETIC ALGORITHM (GA) [pop_size=75, num_iters=450, elitism_size=15]
    Results  : BEST=15080      WORST=15318      AVG=15127.60
    Time (s) : BEST=2.15       WORST=2.24       AVG=2.19
  SIMULATED ANNEALING (SA)
    Results  : BEST=15080      WORST=15080      AVG=15080.00
    Time (s) : BEST=0.28       WORST=0.30       AVG=0.29
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=2500]
    Results  : BEST=15080      WORST=15318      AVG=15270.40
    Time (s) : BEST=0.40       WORST=0.46       AVG=0.42

Testcase: tests/linear_n80_p6.in
  BRUTE FORCE - skipping due to speed
  GENETIC ALGORITHM (GA) [pop_size=100, num_iters=600, elitism_size=20]
    Results  : BEST=16055      WORST=16055      AVG=16055.00
    Time (s) : BEST=7.88       WORST=8.10       AVG=7.98
  SIMULATED ANNEALING (SA)
    Results  : BEST=15936      WORST=16055      AVG=15983.60
    Time (s) : BEST=2.23       WORST=2.42       AVG=2.33
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=6400]
    Results  : BEST=16055      WORST=16055      AVG=16055.00
    Time (s) : BEST=8.13       WORST=8.32       AVG=8.24

Testcase: tests/linear_n100_p4.in
  BRUTE FORCE - skipping due to speed
  GENETIC ALGORITHM (GA) [pop_size=100, num_iters=600, elitism_size=20]
    Results  : BEST=67760      WORST=68355      AVG=67879.00
    Time (s) : BEST=5.16       WORST=5.34       AVG=5.23
  SIMULATED ANNEALING (SA)
    Results  : BEST=67760      WORST=68355      AVG=67879.00
    Time (s) : BEST=0.71       WORST=0.71       AVG=0.71
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=10000]
    Results  : BEST=67760      WORST=72500      AVG=69065.00
    Time (s) : BEST=4.03       WORST=4.87       AVG=4.38

Testcase: tests/linear_n500_p10.in
```

```
  BRUTE FORCE - skipping due to speed
  GENETIC ALGORITHM (GA) [pop_size=100, num_iters=600, elitism_size=20]
    Results  : BEST=1371552    WORST=1621573    AVG=1454183.80
    Time (s) : BEST=23.79      WORST=26.55      AVG=24.81
  SIMULATED ANNEALING (SA)
    Results  : BEST=1383025    WORST=2073344    AVG=1565182.40
    Time (s) : BEST=9.15       WORST=11.14      AVG=9.76
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=10000]
    Results  : BEST=1378620    WORST=2168270    AVG=1566251.40
    Time (s) : BEST=21.42      WORST=29.76      AVG=25.80
```

Here the story is very similar to what we observed with the previous test type on smaller test cases. Once again SA is able to find a competitive solution in a fraction of the time. In fact, it even beats GA and VNS on the small test, abeit only by a small margin.

Once again, SA has a big difference between its best and worst runs on the large test, similarly to VNS. GA displays the best performance, and the lowest dependance on chance.

The last group of test cases belong to the *squared* group:

```
Testcase: tests/squared_n50_p3.in
  BRUTE FORCE
    Result   : 520720
    Time (s) : 4.03
  GENETIC ALGORITHM (GA) [pop_size=75, num_iters=450, elitism_size=15]
    Results  : BEST=520720     WORST=520720     AVG=520720.00
    Time (s) : BEST=2.13       WORST=2.22       AVG=2.17
  SIMULATED ANNEALING (SA)
    Results  : BEST=520720     WORST=520720     AVG=520720.00
    Time (s) : BEST=0.28       WORST=0.29       AVG=0.28
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=2500]
    Results  : BEST=520720     WORST=525369     AVG=521649.80
    Time (s) : BEST=0.47       WORST=0.48       AVG=0.47

Testcase: tests/squared_n80_p6.in
  BRUTE FORCE - skipping due to speed
  GENETIC ALGORITHM (GA) [pop_size=100, num_iters=600, elitism_size=20]
    Results  : BEST=902840     WORST=968898     AVG=916051.60
    Time (s) : BEST=8.06       WORST=8.19       AVG=8.12
  SIMULATED ANNEALING (SA)
    Results  : BEST=901215     WORST=922944     AVG=905560.80
    Time (s) : BEST=2.25       WORST=2.35       AVG=2.30
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=6400]
    Results  : BEST=902840     WORST=902840     AVG=902840.00
    Time (s) : BEST=7.94       WORST=8.32       AVG=8.13

Testcase: tests/squared_n100_p4.in
  BRUTE FORCE - skipping due to speed
```

```
  GENETIC ALGORITHM (GA) [pop_size=100, num_iters=600, elitism_size=20]
    Results  : BEST=4808345    WORST=4808345    AVG=4808345.00
    Time (s) : BEST=5.10       WORST=5.16       AVG=5.14
  SIMULATED ANNEALING (SA)
    Results  : BEST=4807152    WORST=4807152    AVG=4807152.00
    Time (s) : BEST=0.65       WORST=0.70       AVG=0.67
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=10000]
    Results  : BEST=4808345    WORST=5033479    AVG=4853371.80
    Time (s) : BEST=4.16       WORST=4.76       AVG=4.61

Testcase: tests/squared_n500_p10.in
  BRUTE FORCE - skipping due to speed
  GENETIC ALGORITHM (GA) [pop_size=100, num_iters=600, elitism_size=20]
    Results  : BEST=522518392  WORST=619537980  AVG=571390189.00
    Time (s) : BEST=23.39      WORST=26.13      AVG=25.28
  SIMULATED ANNEALING (SA)
    Results  : BEST=533205270  WORST=1038957839 AVG=680165623.40
    Time (s) : BEST=9.39       WORST=11.87      AVG=10.50
  VARIABLE NEIGHBOURHOOD SEARCH (VNS) [num_iters=10000]
    Results  : BEST=551863221  WORST=851774417  AVG=660668487.00
    Time (s) : BEST=19.00      WORST=25.25      AVG=22.11
```

We find the results for *squared* test the most interesting.

The results are very similar to the previous group when we look at the smaller test dimensions.

Interestingly, GA had one particularly bad run on the small squared test (968898 compared to its 902840 best). Looking at the average (916051.60), we conclude that only 1 out of 5 runs exhibited such behavior. Despite displaying the smallest reliance on chance before, it is important to note that all optimization algorithms do still depend on random number generation, and as such can be prone to individual bad peformances.

The largest test once again exhibits the results we have observer beforehand, with GA having the smallest result interval, and the lowest average by a significant margin.

4 Failed ideas

4.1 Linear Programming

We initially had an idea to solve the problem using linear programming, as it seemed easy to define all the conditions that needed to hold true. The problem arose when defining the cost function, as it required block borders as matrix indices, and it appears that it's not possible to index the array with LP variables.

5 Conclusion

In conclusion, the Minimum Array Partition problem has been established as NP-hard, making it challenging to find optimal solutions efficiently. While there exist polynomial algorithms that can provide approximations within a constant factor, their implementation complexity often outweighs the benefits (our approximations factor is better).

As demonstrated in our study, optimization algorithms can be successfully used to solve the Minimum Array Partition problem, with genetic approach displaying the most consistent results, followed closely by VNS. SA remains as a good alternative for when we need to find a decent solution quickly.

These methods strike a balance between computational efficiency and solution quality, making them valuable tools for addressing the Minimum Array Partition problem.

6 References

Efficient array partitioning - Sanjeev Khanna, S. Muthukrishnan & Steven Skiena
On the complexity of the generalized block distribution - Michelangelo Grigni & Fredrik Manne