A Genetic Algorithm for Profit!

Team profit.rocks

Jacob Schäfer (jacob.schaefer@student.hpi.de)
Alexander Sohn (alexander.sohn@student.hpi.de)
Richard Wohlbold (richard.wohlbold@student.hpi.de)

15th January 2023

Abstract

InformatiCup 2023 is about *Profit!*, a type of facility layout problem. This report proposes a theoretical approach using a customized genetic algorithm. We then present our implementation, going into detail on design, software architecture, and software engineering practices. Using data collected from numerous runs on selected and randomly generated problem instances, we show that the algorithm is able to find close-to-optimal solutions in many cases. We then discuss options to further improve the theoretical approach using advanced optimizations for genetic algorithms, and show how to improve the performance of our implementation.

Contents

List of Figures

List of Tables

List of Algorithms

1 Profit!

InformatiCup is a competition for computer science students in Germany, Austria and Switzerland held by the German Society of Computer Science. The challenge of InformatiCup 2023 is *Profit!* [1], a Facility Layout Problem (FLP). According to the definition of Hosseini-Nasab et al., an FLP is defined as the placement of facilities in a plant area, with the aim of determining the most effective arrangement in accordance with some criteria or objectives under certain constraints [2].

1.1 Problem Statement

In *Profit!*, all objects are placed in a rectangular grid, the scenario. Given deposits and obstacles, the task is to place mines, conveyors, combiners, and factories. The evaluation function simulates a specified number of turns. In each turn, resources are mined from deposits, transported to the next conveyor, combiner, factory, or mine, or converted to products in factories. Only products result in points.

The algorithm receives the scenario dimensions $(w, h) \in \mathbb{N}^2, 1 \leq w, h \leq 100$, a number of evaluation turns $t \in \mathbb{N}$, and a product-resource matrix $A \in \mathbb{N}^{8\times 8}$ where A_{ij} specifies the number of resources of type i needed to manufacture product j. It also gets a value vector $\mathbf{v} \in \mathbb{N}^8$ where v_i specifies the number of points obtained by manufacturing product i. Lastly, the algorithm receives a list of input objects and returns a list of output objects.

The following input objects exist:

1. Deposit with dimensions $(w_i, h_i) \in \mathbb{N}^2$ and subtype $s_i \in \mathbb{N}, 0 \leq s_i \leq 7$. The subtype identifies the resource type that can be mined. A maximum of $5 \cdot w_i \cdot h_i$ units can be mined from the deposit.
2. Obstacle with rectangular dimensions $(w_i, h_i) \in \mathbb{N}^2$.

See Figure 1 for the position of ingresses and egresses for input objects. The objects are always completely in bounds and never intersect.

The following output objects exist:

1. Mine with subtype $s_i \in \mathbb{N}, 0 \leq s_i \leq 3$. The mine's subtype specifies its rotation. Figure 2 contains all mine subtypes.
2. Conveyor with subtype $s_i \in \mathbb{N}, 0 \leq s_i \leq 7$. The conveyor's subtype specifies its length and rotation: Short conveyors span three cells, and long conveyors span four cells, each one cell wide.
3. Combiner with subtype $s_i \in \mathbb{N}, 0 \leq s_i \leq 3$. The combiner's subtype specifies its rotation. Figure 3 contains all combiner subtypes.
4. Factory with subtype $s_i \in \mathbb{N}, 0 \leq s_i \leq 7$ and square dimensions $(5, 5)$. The factory's subtype specifies the manufactured product.

There are a set of rules that any solution has to satisfy:

1. All objects are in bounds and do not intersect. There is one exception: Conveyors may intersect with their middle parts as shown in Figure 4.
2. Egresses may only be connected to a single ingress.

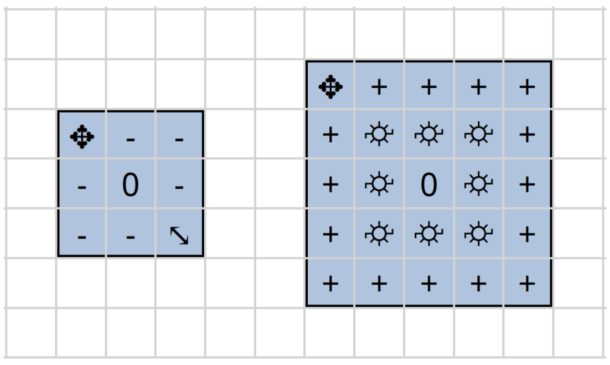


Figure 1: A deposit 3x3 and a factory. Ingresses are marked [+], egresses [−].

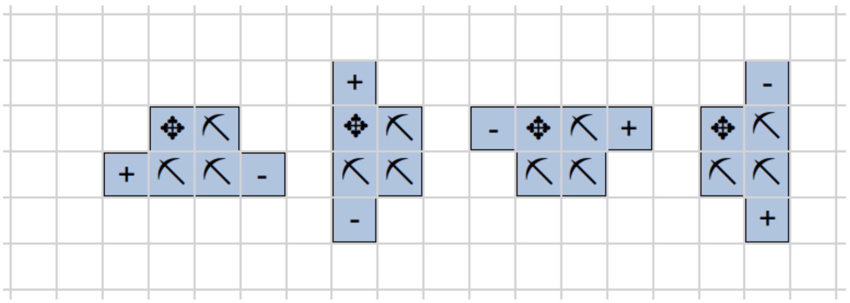


Figure 2: Mines of subtype 0, 1, 2, and 3. Ingresses are marked [+], egresses [−].

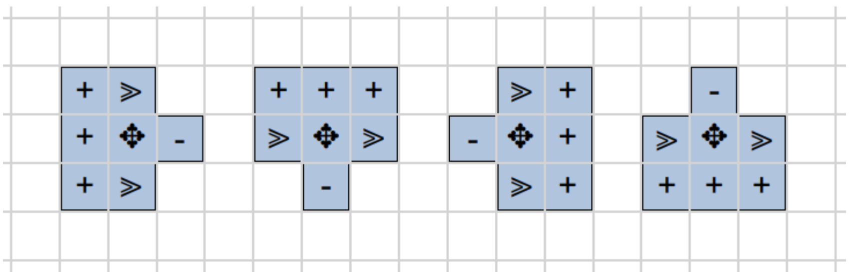


Figure 3: Combiners of subtype 0, 1, 2 and 3

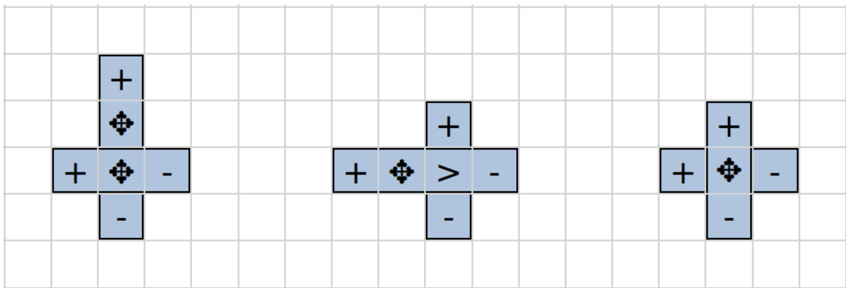


Figure 4: Conveyors that intersect with their middle parts

3. Egresses of mines may not be connected to ingresses of other mines.
4. Only ingresses of mines may be connected to egresses of deposits.

The algorithm's performance on a problem is determined by the number of points at the end of the simulation. If two solutions have the same score, the solution that reaches that score in fewer turns performs better.

1.2 Problem Analysis

The problem is hard for a couple of reasons. Foremost, the search space is very large. When trying to place two factories that produce different products on an empty scenario of size 100x100, there are roughly $100^2 \cdot 100^2 = 100\,000\,000$ possible options to do so. The number of options increases dramatically with more objects. Thus, it is infeasible to cover the entire search space. Further complexity is added by variable parameters like the deadline, number of turns, width, and height, how many objects are placed in a scenario, and the product composition.

1.3 Optimal Score Calculation

Based on the available resources and product composition in a given scenario, an upper bound for the score $\hat{s} \in \mathbb{N}$ can be calculated by solving an *Integer Linear Program* (ILP). An ILP is a general type of optimization problem where all variables are integers.

Note that even though the ILP results in an upper bound, a solution with score $\hat{s}$ may not exist. This is due to the ILP only looking at product composition, fully ignoring the placement of input objects.

Let $\mathbf{x} \in \mathbb{N}^8$ be the product unit vector such that for every product i with $0 \leq i \leq 7$, $x_i \in \mathbb{N}$ represents the number of units manufactured. The upper bound $\hat{s} \in \mathbb{N}$ can then be calculated by taking the inner product with the value vector $\mathbf{v}$: $\hat{s} = \langle \mathbf{v}, \mathbf{x} \rangle$. This expression is maximized in the ILP.

Let $\mathbf{r} \in \mathbb{N}^8$ be defined as follows: Let D_k be the index set of all deposits for resource type $0 \leq k \leq 7$. The number of available units of resource type k, r_k, is then given by $r_k = \sum_{d \in D_k} 5 w_d h_d$. We can therefore formulate an inequality that limits the amount of resources used by the products manufactured to the available resources using the product-resource matrix A:

$$A\mathbf{x} \leq \mathbf{r}$$

Our ILP is thus formulated as follows: Maximize $\langle \mathbf{v}, \mathbf{x} \rangle$ subject to $A\mathbf{x} \leq \mathbf{r}$. Since there are 8 variables and constraints, the ILP can be solved in constant time, which results in a runtime of less than 1 ms in practice. This makes this approach feasible for a couple of things:

- When not optimizing for the number of needed turns as a second criterion, the algorithm can terminate when a solution with score $\hat{s}$ was found. In this case, no solution with a higher score can exist.
- The theoretical upper bound $\hat{s}$ can serve as a benchmark: When finding a solution with score s, the *relative score* can be calculated as $\frac{s}{\hat{s}}$.

2 Genetic Algorithm

Genetic algorithms are known to be able to find good solutions to optimization problems with a large search space. Furthermore, they have been used to solve facility layout problems [3, 4].

A genetic algorithm is an optimization algorithm inspired by the process of natural selection. It works with a population of individuals, representing solutions to a certain problem. An individual has a set of properties called chromosome. A genetic algorithm defines operators, either mutations or crossovers, which alter or combine chromosomes. In an iterative process, genetic operators are applied to the individuals, their fitness is evaluated, and the fittest individuals are selected [4].

The key to a successful genetic algorithm are the exact choices of the algorithm, fitness function, mutation operators, and crossovers. In the following sections, we present important design decisions that contribute to our algorithm's efficiency.

2.1 Design

In our algorithm, a chromosome represents a solution for the given scenario. It includes all output objects. It is checked for validity and then evaluated by our fitness function, which runs a simulation as specified in Section 1.1.

One of the defining aspects of our algorithm is the way it handles conveyors: Instead of placing conveyors randomly, it uses the notion of a *path*.

A path has defined start and end objects and consists of $k \in \mathbb{N}$ conveyors. The ingress of the first conveyor is attached to the egress of the first object, and the ingress of the $i + 1$-th conveyor is attached to the egress of the i-th conveyor, for $1 \leq i < k$. Lastly, the egress of the k-th conveyor is attached to the ingress of the end object.

In our algorithm, paths are not part of the mutation and crossover process. Therefore, the regular mutation and crossover operators for factories, mines, and combiners result in pathless chromosomes. Before evaluating the chromosomes, the algorithm builds paths between the objects using our pathbuilding algorithm.

See Algorithm 1 for a high-level description of the entire genetic algorithm.

2.2 Pathbuilding

When building paths for a chromosome, the following steps are applied:

- For each combiner, a path to a factory of random subtype is built.
- For each mine, a path that connects the mine to a factory, is built. The factory is required to need the mined resource for its product.

When building paths, a factory is considered to be reached if the path is directly connected or if it is connected to an ingress of a conveyor in a path leads to the factory. As a result, the algorithm is able to build paths like in Figure 5.

For the pathfinding algorithm, we interpret the problem as an unweighted directed graph $G = (V, E)$, where each cell of the scenario is a node. Thus, $|V| = w \cdot h$. For each valid placement of a conveyor, we insert edges from its ingress to all positions that are next to its egress and do not intersect with the placed conveyor.

Algorithm 1 Genetic Algorithm

```
$C \leftarrow$ emptyChromosomes($PopulationSize$)
while time not over do
    $C \leftarrow$ selectBest($C, PopulationSize$)
    for $1 \leq i \leq NumCrossovers$ do
        $c_1 \leftarrow$ randomElement($C$)
        $c_2 \leftarrow$ randomElement($C$)
        $d \leftarrow$crossover($c_1, c_2$)
        buildPaths($d$)
        evaluateFitness($d$)
        append(C, d)
    end for
    for $1 \leq i \leq NumMutatedChromosomes$ do
        $c \leftarrow$ randomElement($C$)
        $A \leftarrow$ applyMutations($c, NumMutations$)
        for $c' \in A$ do
            buildPaths($c'$)
            evaluateFitness($c'$)
        end for
        append($C, c'$)
    end for
end while
return selectBest($C$, 1)
```

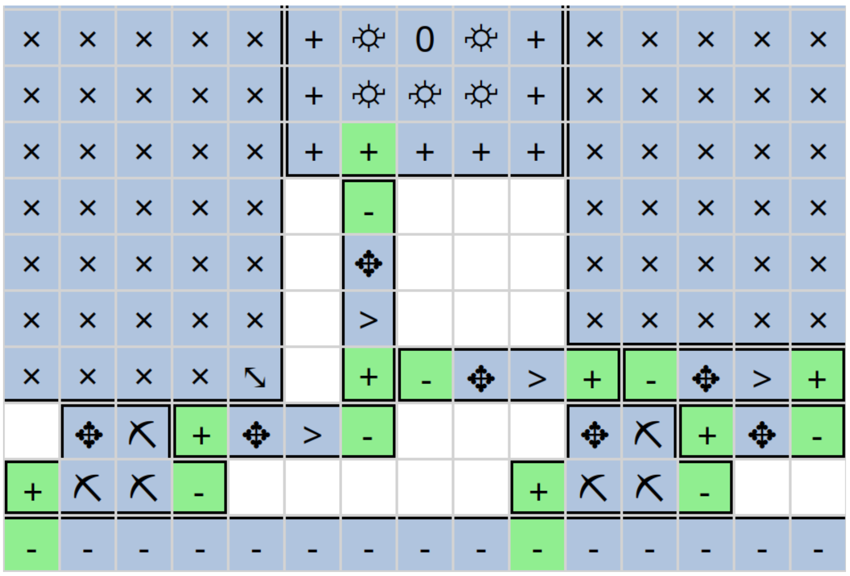


Figure 5: A conveyor is used by two paths

We use a Single-Source-Shortest-Path-algorithm (SSSP) with a starting position s and a set of end positions P. P consists of ingresses of factories or paths that lead to these factories.

Since the graph is unweighted, using a breadth-first search with backtracking is sufficient, running in $\mathcal{O}(|V| + |E|)$. This is significantly faster than SSSP algorithms on weighted graphs like Dijkstra's algorithm [5].

Note that our pathfinding algorithm does not check if conveyors of the resulting path overlap or violate the rule that an egress may only be connected to a single ingress. However, integrating this check would result in a worse run time. For this reason, we accept this inaccuracy and check the validity of a chromosome during the evaluation instead.

2.3 Mutation

Before applying a mutation, all paths of the chromosome are deleted. Our algorithm uses 10 different mutation operators which are listed and explained in Table 1. We choose mutations uniformly from all valid mutations. This is done by iterating over all mutations in a random order, breaking out of the loop when a mutation was applied successfully.

2.4 Crossover

Before performing a crossover, all paths are deleted for both initial chromosomes. Then, the algorithm uses the following rules:

- If two factories of the same subtype overlap, the new chromosome only includes one of the factories.
- If two factories of different subtypes overlap, one of the factories is placed somewhere else.
- If two mines are overlapping and providing resources for the same product, only one mine is kept.
- If two mines are overlapping and providing resources for different products, one of the mines is placed somewhere else. However, the mine stays connected to the same deposit.

2.5 Analysis

When searching for a solution in a large search space, it is important to have a good ratio between exploration and exploitation. The concept of genetic algorithms as used in our approach provides a promising baseline. The described genetic algorithm does not use advanced techniques like varying population size, duplicate elimination, or different sub-populations for exploration and exploitation. However, there is a lot of research in this field and the algorithm can easily be adapted [6].

By abstracting conveyors into paths, the explored search space is limited to the most promising parts. Other optimizations, like placing mines only next to deposits also contribute to limiting the search space. Further, building paths in a separate step allows the mutation operators and the crossover to explore more object placement possibilities.

One downside of our approach is that, as described in Section 2.2, the pathfinding algorithm may return invalid paths. Additionally, paths are built one after the other which keeps the

Mutation name	Description
`addFactory`	Add a factory at a random position from all positions that are available for factories.
`removeFactory`	Remove a random factory.
`moveFactories`	Each factory is removed with a certain probability. Then, for each removed factory a new random factory is added.
`changeProduct`	Randomly change the product of a random factory.
`addMine`	Add a mine at a random position from all positions that are available for mines. Only positions next to deposits are considered to determine available positions.
`removeMine`	Remove a random mine.
`moveMines`	Each mine is removed with a certain probability. Then, for each removed mine a new random mine is added.
`addCombiner`	Add a combiner with a random subtype at a random position from all positions that are available for combiners.
`removeCombiner`	Remove a random combiner.
`moveCombiners`	Each combiner is removed with a certain probability. Then, for each removed combiner a new random combiner is added.

Table 1: Overview of mutations

algorithm from finding optimal path configurations in some scenarios, e.g. when a path cuts off a mine from all factories.

In the future, a theoretical approach to tune the control parameters of the genetic algorithm could improve the results. Possible methods include using another meta-level genetic algorithm, or Relevance Estimation and VAlue Calibration (REVAC) [7, 8].

To conclude, we use a well-established theoretical approach that can be improved by controlling exploration and exploitation with advanced techniques and tuning hyper-parameters. Further, the approach to pathbuilding is crucial to the effectiveness of the approach while introducing a couple of drawbacks.

3 Implementation

For our implementation, we collected a few requirements:

- Easy prototyping of new concepts: We only have limited time that we can dedicate to our implementation. Our choice of tools must not slow us down.
- Good performance: The software has to find a solution in a fixed timeframe, it should therefore run with little overhead.
- Modular architecture: We want to have the ability to recombine functionality to try out new ideas.
- Tooling: Software should be testable, follow coding conventions and be able to run in a container environment.

We chose Go as the programming language for our implementation [9]. Since it is garbage-collected, it allows for quick prototyping. Still, the software runs much faster than interpreted languages, e.g. Python. We also knew Go from previous projects, allowing us to get started more easily. Lastly, Go's tooling is very mature which helped us concentrate on the implementation itself.

3.1 Software Architecture

For the software architecture, our goal was to design it for genetic algorithms, but keep it flexible enough so that the algorithm can be adapted with few changes to the implementation.

Therefore, we implemented a few primitives like *Position* and *Rectangle* with geometric operations, like *Intersect*. On top, we built the game objects *Deposit*, *Mine*, *Obstacle*, *Factory*, *Combiner* and *Conveyor*. Since our approach does not directly work with conveyors, we introduced *Path*s as an additional abstraction. We then partitioned our game objects into input objects (*Scenario*) and output objects (*Chromosome*). This minimizes the amount of copying needed since we can always work on the same *Scenario*, modifying only the current *Chromosome*. Figure 6 shows a UML Class Diagram of our architecture.

Overall, this software architecture allows for a quick feedback cycle. New mutation operators can easily be implemented as the core system provides a rich interface to the game objects. The genetic algorithm itself is very short, making it easy to exchange it for an improved version.

3.2 Software Engineering

3.2.1 Automated Testing

One goal for the implementation was to create automated tests that verify the core logic. For example, we use tests to ensure that the implementation of `(c Conveyor).Egress()` returns the correct egress positions for all conveyor lengths and orientations. We focused on the core logic because bugs have a big impact on the correctness of our solution there.

One metric to measure the amount of code that is covered by automated tests is *statement coverage*. While there are some problems associated with the metric, it is reasonably good at showing which units are well-tested and which ones are not [10]. Our statement coverage results are available in Figure 7. Right now, our implementation has a total statement coverage of 55.5%.

3.2.2 Reviews

Software that was reviewed is less likely to introduce bugs and has significantly higher readability [11]. Therefore, we performed code reviews for all new features and bug fixes: In total, we created, reviewed, and merged 40 pull requests. Apart from contributing to better software quality, we noticed that reviews helped us to share knowledge within the team.

3.2.3 Coding Conventions

As our coding conventions, we follow the Google Style Guide for Go [12]. Our code is formatted automatically using `gofmt` to ensure consistent spacing, indentation, and improve overall readability.

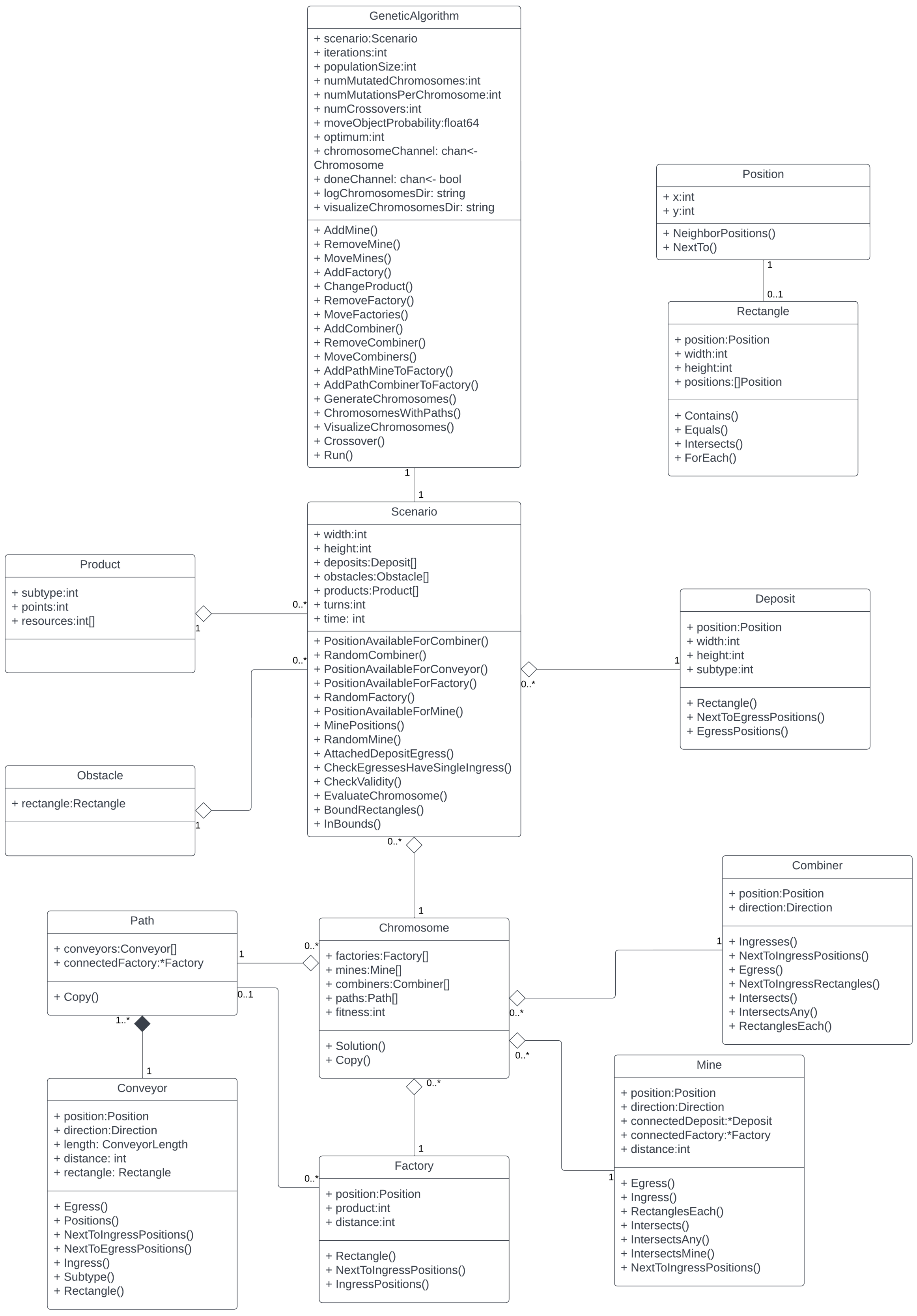


Figure 6: UML Class Diagram of our implementation (some associations to *Rectangle* and *Position* were left out to improve readability)

Source file	Statement coverage
`combiner.go`	65.6%
`conveyor.go`	74.1%
`deposit.go`	100.0%
`evaluation.go`	98.4%
`factory.go`	93.2%
`genetic_algorithm.go`	1.0%
`geometry.go`	100.0%
`json.go`	66.7%
`main.go`	0.0%
`mine.go`	88.0%
`optimum.go`	88.9%
`rng.go`	100.0%
`scenario.go`	100.0%
`visualization.go`	0.0%

Figure 7: Statement coverage of source files in our implementation

3.2.4 Continuous Integration

To make collaboration more efficient, we wanted our `main` branch to always contain a working copy of the software. Therefore, we followed the practice of *Continuous Integration* by using GitHub Actions to check our coding conventions and to automatically build and test our software.

3.2.5 Continuous Benchmarking

To detect performance regressions during implementation, we added automated benchmarks to our continuous integration pipeline. A benchmark includes the score and the needed time for a set of selected scenarios. We included the tasks provided in the InformatiCup repository [13], some small custom examples as well as more sophisticated tasks. Benchmarks are run for every commit on the main branch and every commit on open pull requests. We deployed the results using GitHub Pages to make them easily accessible to us. Figure 8 contains example diagrams for our benchmarks.

Often, regressions were caused by performance issues. There, we used Go's integrated CPU profiler to get an insight into where our software was spending too much time.

3.3 Tooling

To get better insights into how our alogrithm works, we integrated some tooling into our implementation. We added a command line argument that allows us to export intermediate solutions for every iteration. Further, we implemented a visualization for populations. For every type of object, a heatmap for the placement of those objects is exported. Figure 9 shows an example of such a heatmap for mines after ten iterations.

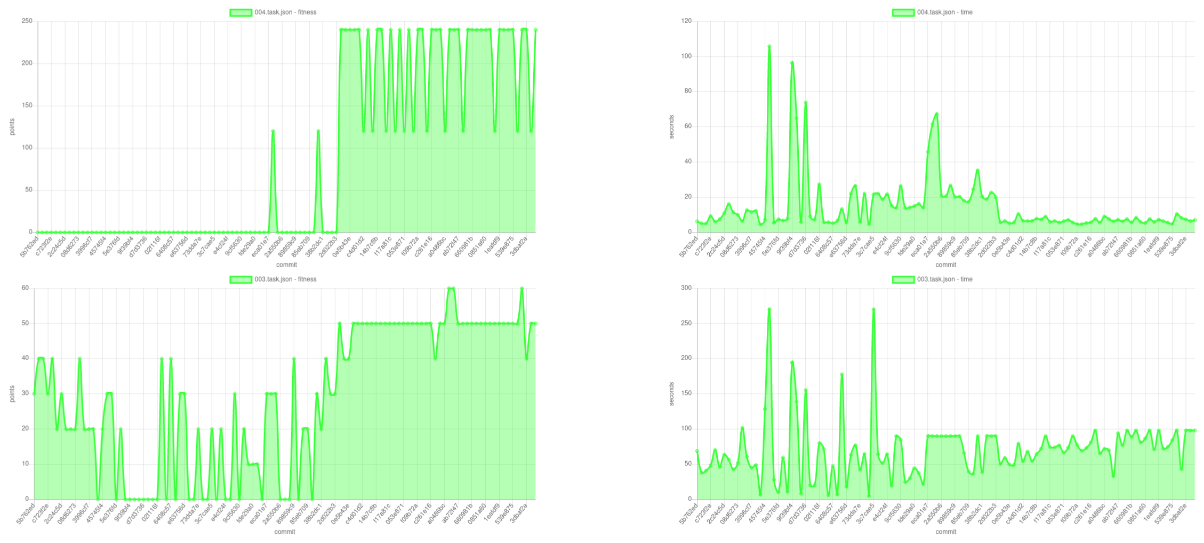
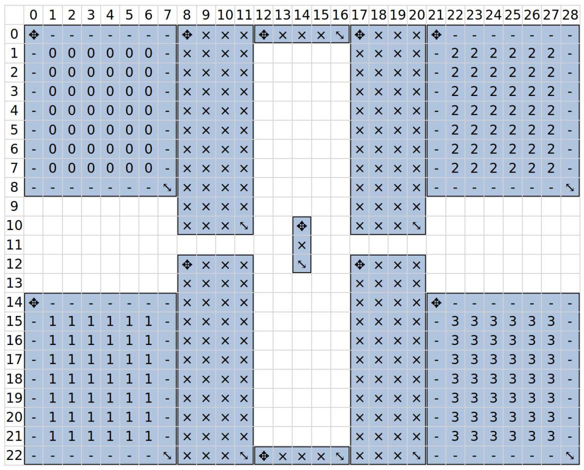
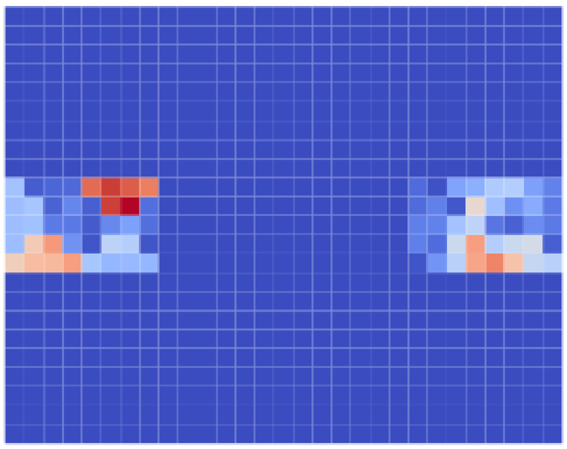


Figure 8: Some examples of benchmarks



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	✥	-	-	-	-	-	-	-	✥	×	×	×	✥	×	×	×	⤡	✥	×	×	×	✥	-	-	-	-	-	-	-
1	-	0	0	0	0	0	0	-	×	×	×	×						×	×	×	×	-	2	2	2	2	2	2	-
2	-	0	0	0	0	0	0	-	×	×	×	×						×	×	×	×	-	2	2	2	2	2	2	-
3	-	0	0	0	0	0	0	-	×	×	×	×						×	×	×	×	-	2	2	2	2	2	2	-
4	-	0	0	0	0	0	0	-	×	×	×	×						×	×	×	×	-	2	2	2	2	2	2	-
5	-	0	0	0	0	0	0	-	×	×	×	×						×	×	×	×	-	2	2	2	2	2	2	-
6	-	0	0	0	0	0	0	-	×	×	×	×						×	×	×	×	-	2	2	2	2	2	2	-
7	-	0	0	0	0	0	0	-	×	×	×	×						×	×	×	×	-	2	2	2	2	2	2	-
8	-	-	-	-	-	-	-	⤡	×	×	×	×						×	×	×	×	-	-	-	-	-	-	-	⤡
9									×	×	×	×						×	×	×	×								
10									×	×	×	⤡			✥			×	×	×	⤡								
11															×														
12									✥	×	×	×			⤡			✥	×	×	×								
13									×	×	×	×						×	×	×	×								
14	✥	-	-	-	-	-	-	-	×	×	×	×						×	×	×	×	✥	-	-	-	-	-	-	-
15	-	1	1	1	1	1	1	-	×	×	×	×						×	×	×	×	-	3	3	3	3	3	3	-
16	-	1	1	1	1	1	1	-	×	×	×	×						×	×	×	×	-	3	3	3	3	3	3	-
17	-	1	1	1	1	1	1	-	×	×	×	×						×	×	×	×	-	3	3	3	3	3	3	-
18	-	1	1	1	1	1	1	-	×	×	×	×						×	×	×	×	-	3	3	3	3	3	3	-
19	-	1	1	1	1	1	1	-	×	×	×	×						×	×	×	×	-	3	3	3	3	3	3	-
20	-	1	1	1	1	1	1	-	×	×	×	×						×	×	×	×	-	3	3	3	3	3	3	-
21	-	1	1	1	1	1	1	-	×	×	×	×						×	×	×	×	-	3	3	3	3	3	3	-
22	-	-	-	-	-	-	-	⤡	×	×	×	⤡	✥	×	×	×	⤡	×	×	×	⤡	-	-	-	-	-	-	-	⤡

(a) Task 4



(b) Placement of mines after 10 iterations

Figure 9: Heatmap visualization for the placement of mines

3.4 Performance

An issue with our implementation is performance on large scenarios for which we identified a key reason: There are few references between the game objects. For example, to find a conveyor that is attached to a mine, the algorithm has to iterate over all conveyors in a list. This creates the problem that frequently, the algorithm performs $\mathcal{O}(nm)$ lookups where $n, m \in \mathbb{N}$ are the numbers of objects in the scenario or chromosome. Therefore, mutations and the evaluation become unnecessarily slow at increasing object counts. Using back-references and index structures, these lookups could be significantly reduced or eliminated.

We already implemented this optimization in the evaluation: When building a path from a mine to a factory, the algorithm stores which factory can be reached and how long the path is. This makes it much easier to simulate the rounds later in the evaluation: Now, only the deposits, mines, and factories are needed. If a resource is mined, it is immediately known when the resource will arrive at the factory. This optimization simplified our evaluation logic and resulted in a considerable performance improvement. We are convinced that optimizations through back-references can result in performance gains in more parts of the implementation.

4 Evaluation

A good algorithm for *Profit!* has to find solutions with high scores for a variety of different types of tasks.

We randomly generated scenarios and performed benchmarks for different deadlines and numbers of turns.

To make the scenarios more comparable, we chose a set of parameters from which scenarios were generated:

- 10 predefined shapes for obstacles and deposits (see Table 2)
- Varying scenario height and width between 10 and 100 with a step size of 10
- Each product requires between 0 and 10 units of available resources in the scenario and gives up to 7 points
- No constraints on the placement of input objects, except that it must be valid according to the rules
- $\frac{1}{5}$ of the placed objects are obstacles, $\frac{4}{5}$ are deposits

For a given product composition and placement of input objects, we tested our implementation with 10 to 90 turns with a step size of 20 and each on deadlines from 20 to 80 seconds with a step size of 20.

Scenarios generated in this way do not guarantee that there will be a solution that scores points. A simple heuristic to filter obviously unsolvable scenarios is to check if no factory can be built, in which case the scenario is discarded.

In addition to evaluating our solution on random scenarios, we chose one scenario and benchmarked it with different deadlines from 20 to 160 seconds with a step size of 20. The scenario has a size of 100x100 with 60 objects on it, 54 deposits, and 6 obstacles. To further increase the complexity, we added eight products and eight different resources. All products use at least two different resources.

Width	Height
1	20
2	2
3	3
6	6
7	7
20	1
3	6
6	3
10	2
2	10

(a) shapes of obstacles

Width	Height
1	1
3	3
5	5
5	10
10	5
7	3
3	7
10	10
12	2
2	12

(b) shapes of deposits

Table 2: Shapes of objects in our scenarios

4.1 Data Analysis

For our random scenarios, we looked at the relative score described in Section 1.3. Although there may not exist a solution with the optimal score for a given scenario, this metric is normalized and allows a comparison between scenarios of different sizes and different amounts of available resources.

Figure 10 shows a histogram of the relative score that was reached, grouped by scenario size with a fixed deadline of 80 s and 90 turns.

We observe that our solution performs very well on scenarios up to 50x50, where our algorithm is able to reach more than 50% relative score on almost all random scenarios. For the 70x70 scenarios, the performance has a larger variance. Our hypothesis is that on these larger scenario sizes, the time needed to find a good solution increases beyond our 90s deadline. This is also consistent with the results for the 100x100 scenarios, where no solutions with over 60% relative score were found.

Therefore, we look more closely at the specific scenario described above. Because of its size, we do not expect our solution to perform close to optimal at shorter deadlines. Figure 11 contains boxplots of the scores of 14097 runs on the test scenario with varying deadlines. On the positive side, we can observe that on average, our algorithm profits from large deadlines, increasing the average score even in the last interval. This reinforces the hypothesis that our program takes a long time to find solutions in large scenarios. We also notice another problem: Performance seems to have a large variance in these types of scenarios. Therefore, we further analyze the score distributions.

Figure 12 contains histograms of the scores on the test scenario, varying by deadline. We can see that with short deadlines, score distribution is right-skewed: There are few runs where early mutations resulted in exceptional performance, with most runs performing unexceptionally. The score distributions get more symmetrical for larger deadlines, as runs with exceptional starts make less progress and previously unsuccessful runs find increasingly better solutions.

Lastly, we analyzed the number of iterations performed on the test scenario throughout the runs. We plotted our results in Figure 13. We can see that our solution performs very few iterations at this scenario size, a maximum of 6 with a deadline of 160 s. This result explains the cutoff in the score on large scenarios in Figure 12: In six iterations, our solution will mutate

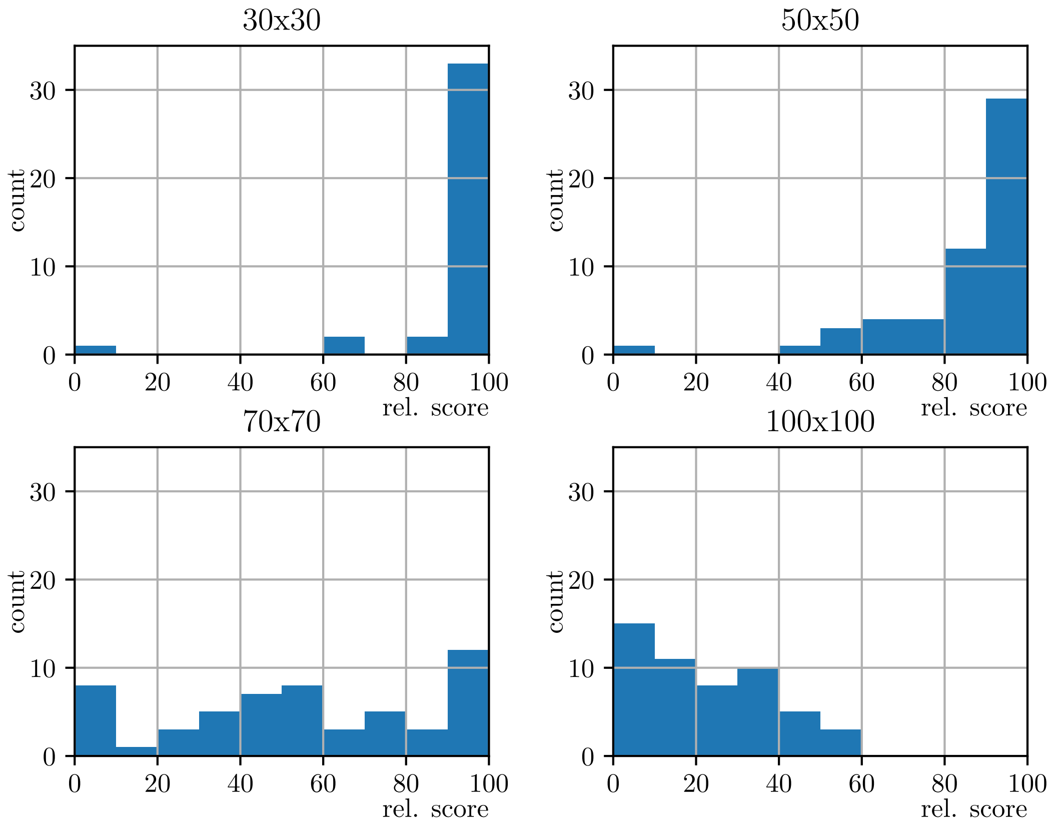


Figure 10: Score distribution on random scenarios of given width and height, 80 s deadline, 90 turns. N=40 in each category.

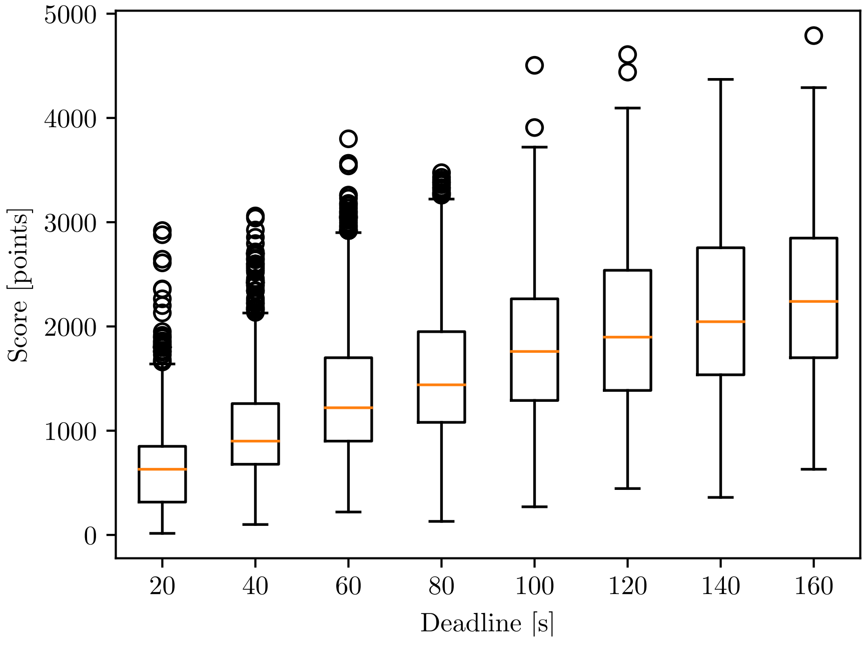


Figure 11: Deadline vs score distribution on the test scenario with a variable deadline, N=14097

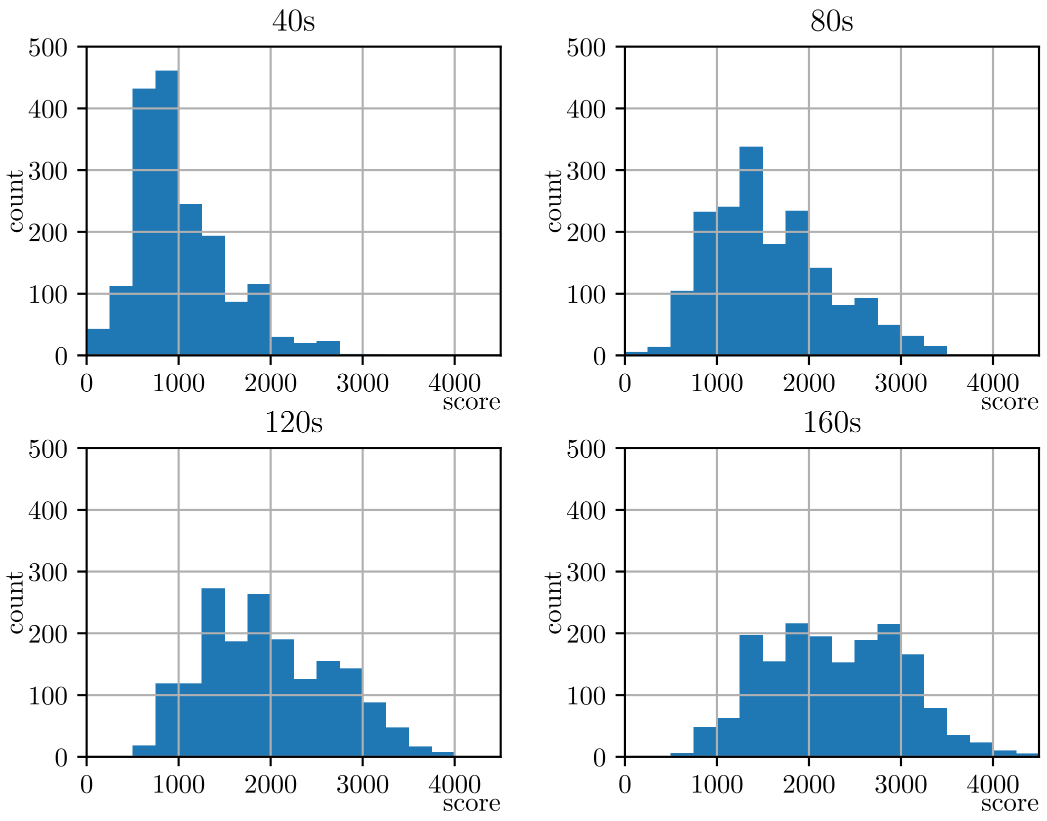


Figure 12: Scores on the test scenario with variable deadlines

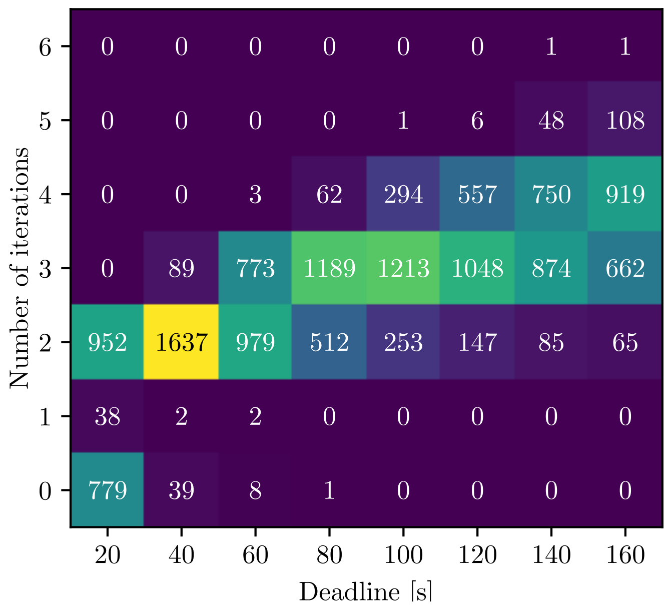


Figure 13: Number of iterations before the deadline on the test scenario

a chromosome at most 120 times, which is not enough to get close to an optimal score. This supports our performance analysis (see Section 3.4). We believe that with a more efficient implementation, the theoretical approach will be able to deliver much better results at large scenario sizes.

5 Conclusion

To build an algorithm that solves *Profit!*, we analyzed the problem and did research on similar problems and their solutions. Our resulting theoretical approach based on genetic algorithms is conceptually simple. The work still contributes to the existing literature on facility layout problems by showing that our simplifications lead to close-to-optimal solutions in limited time-frames. We also provide a modular implementation of the algorithm that can serve as a testbed for new ideas in the future. Since there is extensive research on genetic algorithms, there are many potential improvements to be made.

References

[1] *Profit! - Challenge Description.* URL: https://github.com/informatiCup/informatiCup2023/blob/main/informatiCup%5C%202023%5C%20-%5C%20Profit!.pdf.

[2] Hasan Hosseini-Nasab et al. "Classification of facility layout problems: a review study". In: *The International Journal of Advanced Manufacturing Technology* 94.1 (Jan. 2018), pp. 957–977. ISSN: 1433-3015. DOI: 10.1007/s00170-017-0895-8. URL: https://doi.org/10.1007/s00170-017-0895-8.

[3] Mariem Besbes, Marc Zolghadri, and Roberta Costa Affonso. "A method to solve 2D Facility Layout Problem with equipment inputs/outputs constraints using meta-heuristics algorithms". In: *Procedia CIRP* 104 (2021). 54th CIRP CMS 2021 - Towards Digitalized Manufacturing 4.0, pp. 1698–1703. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2021.11.286. URL: https://www.sciencedirect.com/science/article/pii/S2212827121011847.

[4] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. "A review on genetic algorithm: past, present, and future". In: *Multimedia Tools and Applications* 80.5 (Feb. 2021), pp. 8091–8126. ISSN: 1573-7721. DOI: 10.1007/s11042-020-10139-6. URL: https://doi.org/10.1007/s11042-020-10139-6.

[5] *Difference Between BFS and Dijkstra's Algorithms.* URL: https://www.baeldung.com/cs/graph-algorithms-bfs-dijkstra.

[6] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. "Exploration and Exploitation in Evolutionary Algorithms: A Survey". In: *ACM Comput. Surv.* 45.3 (July 2013). ISSN: 0360-0300. DOI: 10.1145/2480741.2480752. URL: https://doi.org/10.1145/2480741.2480752.

[7] Guicheng Wang et al. "Optimization of Controller Parameters Based on the Improved Genetic Algorithms". In: *2006 6th World Congress on Intelligent Control and Automation.* Vol. 1. 2006, pp. 3695–3698. DOI: 10.1109/WCICA.2006.1713060.

[8] S. K. Smit and A. E. Eiben. "Parameter Tuning of Evolutionary Algorithms: Generalist vs. Specialist". In: *Applications of Evolutionary Computation.* Ed. by Cecilia Di Chio et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 542–551. ISBN: 978-3-642-12239-2.

[9] *The Go Programming Language.* URL: https://go.dev/.

[10] Michael Ellims, J. Bridges, and D.C. Ince. "Unit testing in practice". In: Dec. 2004, pp. 3–13. ISBN: 0-7695-2215-7. DOI: 10.1109/ISSRE.2004.44.

[11] Gabriele Bavota and Barbara Russo. "Four eyes are better than two: On the impact of code reviews on software quality". In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME).* 2015, pp. 81–90. DOI: 10.1109/ICSM.2015.7332454.

[12] *Go Style.* URL: https://google.github.io/styleguide/go/.

[13] *informatiCup 2023.* URL: https://github.com/informatiCup/informatiCup2023.