Assignment 1: Evolutionary Process Discovery

1BM120, Decision making with artificial intellegence - Q4 (2025)

Group 13

Full Name	Student ID
Yannick Hikspoors	1441361
Tijmen Mölder	1479482
Celine Zanders	1504177
Tijn Zeelenberg	1456423

Eindhoven, May 9, 2025

1 | Task 1

For this task, a Genetic Algorithm (GA) was implemented using the DEAP (Distributed Evolutionary Algorithms) library in the Python programming language. The goal was to discover the best configuration of a Petri net (PN) that accurately represents the request process.
The GA uses a population of candidate solutions, where each individual represents a potential Petri net configuration. The process starts with initializing the population with random individuals, each containing a list of integers representing places and transitions in the Petri net.
The operations of the GA are implemented by:

- Crossover: The two-point crossover (cxTwoPoint) method is used to combine the genetic material (places and transitions) of two parent individuals to create offspring. The crossover probability used is 0.5.
- Mutation: The mutation operation (mutUniformInt) randomly changes integers in an individual's representation, introducing variation into the population. The mutation probability used is 0.2.
- Selection: The tournament selection method (selTournament) is used to select individuals for reproduction based on their fitness. The tournament size used is 3.

Fitness is evaluated using a fitness function, which measures how well a Petri net configuration aligns with the observed traces of the request process. The algorithm evolves over 50 generations with a population size of 100. The population is updated in each generation using the genetic operations mentioned before. To track the progress of the algorithm, statistics such as the average, minimum, and maximum fitness values are recorded. The average and maximum fitness over the generations can be seen in the plot in figure 1.1. The best individual found by the GA has a fitness of 0.5536.

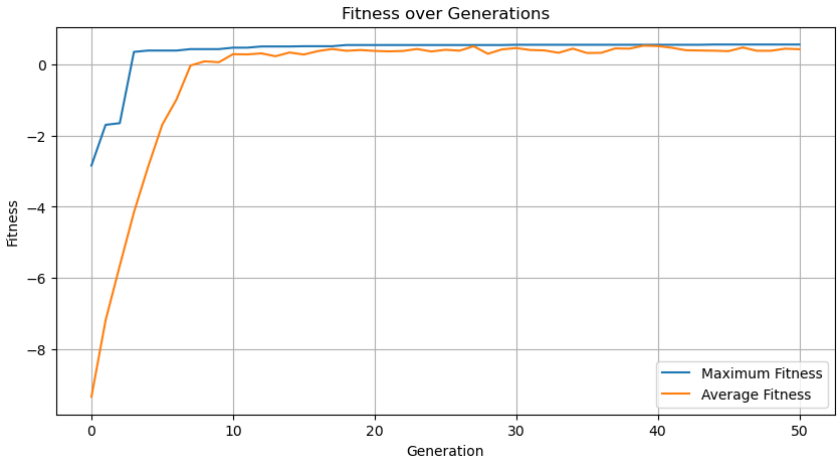


Figure 1.1: The evolution of the average and maximum fitness over 50 generations.

2 | Task 2

In Task 2, we evaluated the performance of various genetic algorithm (GA) variants using different combinations of crossover, mutation, and selection operators. The combinations tested were as follows:

- Crossover: cxTwoPoint and cxOrdered
- Mutation: mutUniformInt and mutShuffleIndexes
- Selection: selTournament and selRoulette

The primary objective was to assess the impact of these operator combinations on both the optimization performance and overall running time of the algorithm. The fitness of the best solution was monitored across 10 runs of 50 algorithmic generations for each combination. The distribution of fitness values was visualized using boxplots in Fig. 2.1. These boxplots show the variability of the final max fitness for the

different combinations of crossover, mutation, and selection. Whereafter, analysis could indicate potential issues with premature convergence. Additionally, each combination's average best fitness (ABF) was tracked across generations and plotted to evaluate the convergence rate of each GA variant.
As shown in Fig.2.2 the results show that the combination of cxOrdered, mutUniformInt, and selTournament outperforms the other combinations in the long run, while being outperformed in the earlier generations. This combination shows a relatively high and early stabilization compared to other combinations. On the other hand, combinations involving cxTwoPoint display more fluctuations in fitness, suggesting possible premature convergence on suboptimal solutions in some runs. The final ABF and running time for each combination can be found in Table 2.1. This Table shows that the combination cxTwoPoint, mutUniformInt and selTournament is the optimal combination of operators.

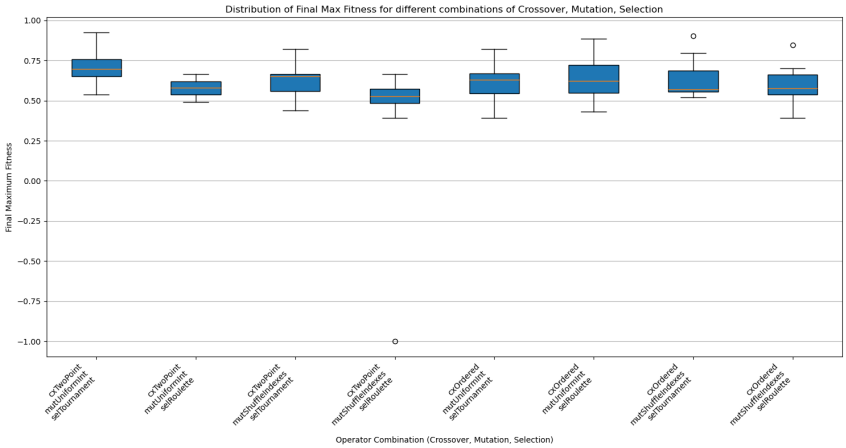


Figure 2.1: Boxplot Distribution of Final Maximum Fitness for Different GA Operator Combinations.

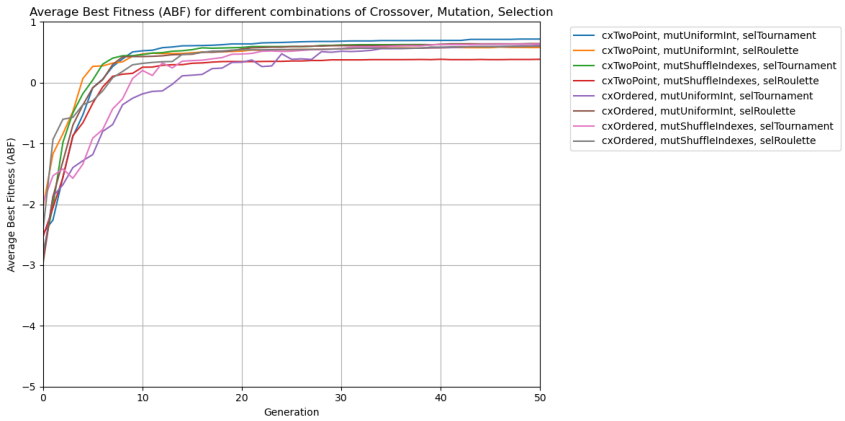


Figure 2.2: Average Best Fitness (ABF) Over Generations for Different GA Operator Combinations.

Crossover	Mutation	Selection	ABF	Time (minutes)
cxTwoPoint	mutUniformInt	selTournament	0.7169	8:33
cxTwoPoint	mutUniformInt	selRoulette	0.5773	8:36
cxTwoPoint	mutShuffleIndexes	selTournament	0.6337	8:19
cxTwoPoint	mutShuffleIndexes	selRoulette	0.3843	7:07
cxOrdered	mutUniformInt	selTournament	0.6170	7:17
cxOrdered	mutUniformInt	selRoulette	0.6429	8:04
cxOrdered	mutShuffleIndexes	selTournament	0.6336	32:30
cxOrdered	mutShuffleIndexes	selRoulette	0.5984	07:21

Table 2.1: Comparison of genetic algorithm configurations on their ABF and runtimes.

3 | Task 3

In Task 3, the goal was to investigate the impact of mutation and crossover probabilities on the performance of the genetic algorithm (GA). We used the optimal operators from Task 2, namely cxTwoPoint, mutUniformInt and selTournament. Different combinations were tested of the probabilities for crossover and mutation and these were evaluated. Their effect on the optimization performance focuses on the average best fitness (ABF) achieved across generations. The task involved running the GA, with the crossover- and mutation probabilities with values of 0.2, 0.4, 0.6, and 0.8.
By adjusting these parameters, we aimed to determine the best configuration for the GA in terms of performance and computational efficiency. The results of these simulations can be found in Table 3.1.

Average Best Fitness		**Crossover Probabilities**			
		0.2	0.4	0.6	0.8
Mutation Probabilities	0.2	0.5957	0.6429	0.6919	0.6927
	0.4	0.6842	0.6366	0.6931	0.6910
	0.6	0.7029	0.7057	0.7383	0.7514
	0.8	0.6879	0.7140	0.7799	0.6955

Table 3.1: Average Best Fitness across mutation and crossover probabilities.

4 | Task 4

In Task 4 the impact of domain specific constraints on the genetic algorithm was explored. The PETRINAS problem provides certain domain knowledge encoded into the fitness function, which should be adhered to by the candidate solutions or they get penalized. Rather than evaluating invalid solutions and then penalizing them. A tool decorator was applied to automatically fix the candidate solutions after mutation and crossover, as to prevent violations of these constraints.
The key challenge in this task was ensuring that the newly generated candidates comply with 2 specific constraints, namely ”no self-loops”, a transition from one place to the exact same place, and ”no loops/backwards transitions”, a transition from a place with a higher index to a place with a lower index. The tool decorator was used to handle these constraints, making sure that the solutions remained valid according to PETRINAS' domain knowledge. When the decorator encounters a solution with a self-loop, it replaces the destination place with another destination place that comes after the origin place (e.g., [0, 0] becomes [0, 1]. If there is a self-loop in the final place, the first place is changed instead (e.g. [9, 9] becomes [8, 9]. This way, the ”no loops/backwards transitions” constraint is automatically satisfied as well. Solutions that violate that constraint are fixed by swapping the origin and destination (e.g., [4, 3] becomes [3, 4].
Repairing individuals that violate domain constraints before evaluating their fitness should improve the speed of convergence since individuals that violate domain constraints are given a low fitness score, therefore if we have fewer violated constraints, we have higher average fitness and have a pool of better solutions for the next generation. However, this method limits the diversity of solutions and therefore has less exploration, so has a higher chance of finding a local optimum. If and how much this actually matters will be investigated in Task 5.

5 | Task 5

In Task 5 a GA with the best hyperparameters and operators identified in Tasks 2 and 3 gets combined with the tool decorator from Task 4 to run the GA in an optimized way. This then gets compared to the GA with the best hyperparameters and operators, but without the use of the tool decorators. The GAs are run for 500 generations to give both more than enough time to converge. The results can be seen in Fig. 5.1 where both the average and maximum fitness over the generations are plotted for the GA with and the GA without tool decorators. In Fig. 5.2 a zoom in on the first 10 and last 50 generations can be seen. Focusing on the first 10 generations shows clearly that using the decorator increases the speed at which the algorithm initially converges to its (local) optimum, with a higher starting maximum fitness and quicker improvements the first generations. After that the average fitness of the decorator GA always stays higher than that of the ”standard” GA. However, the max fitness / best solution of the ”standard”

GA gets better than the decorator GA around 50-60 generations and ends up with a better max fitness and final solution after the complete run, which can more clearly be seen in figure 5.2b. As discussed in Task 4, this is probably due to the decorators allowing for less exploration and a narrower potential solution space, which makes the algorithm convert to a suboptimal solution compared to the ”standard” GA. The final best individual (”standard” GA) had a fitness of 0.983.

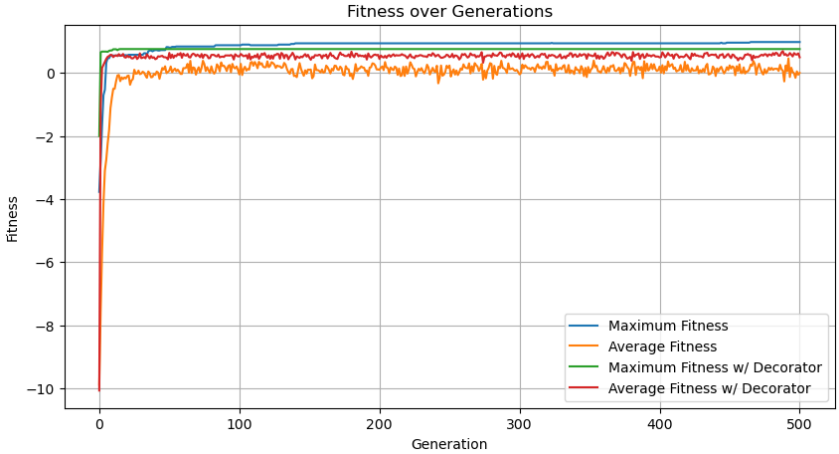
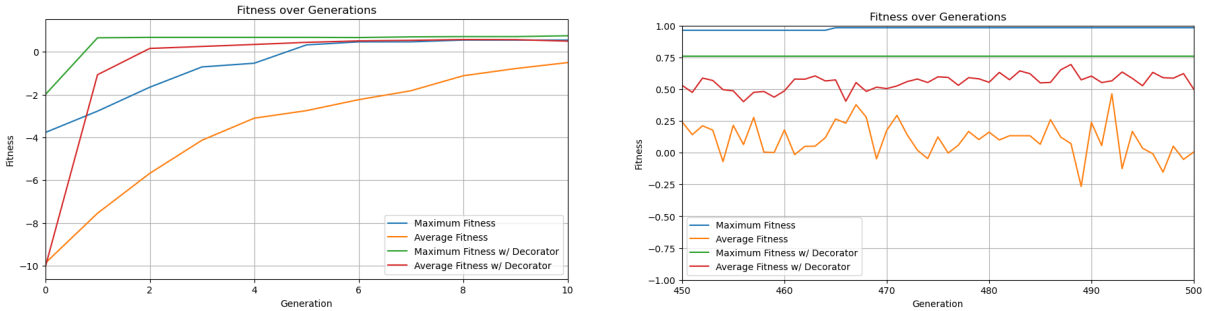


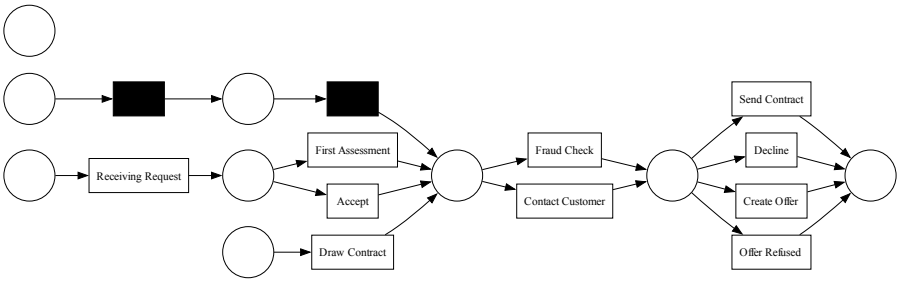
Figure 5.1: Max and average fitness over generations for GA with and without decorators - Complete run



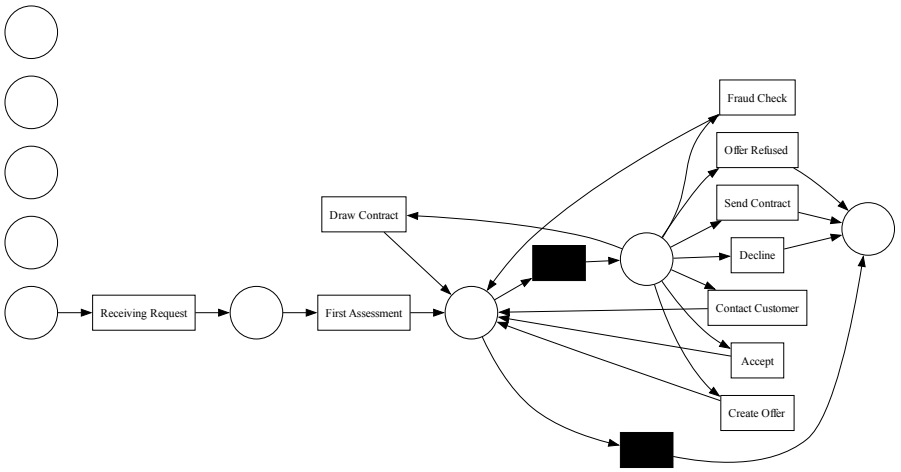
(a) Generations 0–10 **(b)** Generations 450–500

Figure 5.2: Max and average fitness over generations for GA with and without decorators

Lastly, we can compare the two methods by analyzing the Petri nets of the final solutions. The petri net with decorators has no loops, or self loops, while the petri net generated by the GA without decorators, does have loops. Even though the second has a higher fitness, if it was paramount that the constraints are not violated, the first solution might still be better.



(a) Final Petri net with decorators



(b) Final Petri net from Q1

Figure 5.3: Comparison of the final Petri-net solutions using the two methods