Assignment 1: Evolutionary process discovery

Max van den Hoven	Dean Bakens	Alwin Verbeeten	Johan de Langen
1750461	1026726	1223096	0995417

17 May 2023

Exercise 1: Genetic Algorithm Implementation

We implemented the genetic algorithm (GA) in Python using the DEAP library[1]. It is important to note that we modified the provided fitness function to stop counting when a transition in a trace cannot be reached from the current place, or when a transition does not lead to a (new) place. Both of these conditions were not included in the provided implementation, causing convergenge towards incomplete petri nets.

Before we tune the hyperparameters of the GA, i.e., the evolutionary operators and mutation/crossover probabilites, we investigate the best fitness at each generation using a set of initial operators and hyperparameters. For this initial run, we use the cxTwoPoint, mutFlipBit and selTournament operators for crossover, mutation, and selection respectively, with a crossover probability of 0.8. For mutation, we use a mutation probability of 0.2, with an independent probability of 0.05 of an attribute being flipped. For selection, we use a tournament size of 5. Furthermore, we run the GA with a population size of 250 for a fixed number of 50 generations. The results of this initial run are shown in Figure 1.

Figure 1 shows that the algorithm has largely converged after 20 generations, meaning that computation time can potentially be halved by reducing the maximum number of generations. Nevertheless, we keep the number of generations at 50 to investigate the rate of convergence for other sets of hyperparameters in subsequent exercises. Also note that Figure 1 is based on a single run of the algorithm, and that randomness can therefore play a role in the evaluation. We do not run the algorithm for multiple iterations however, as this will be done in Exercise 2.

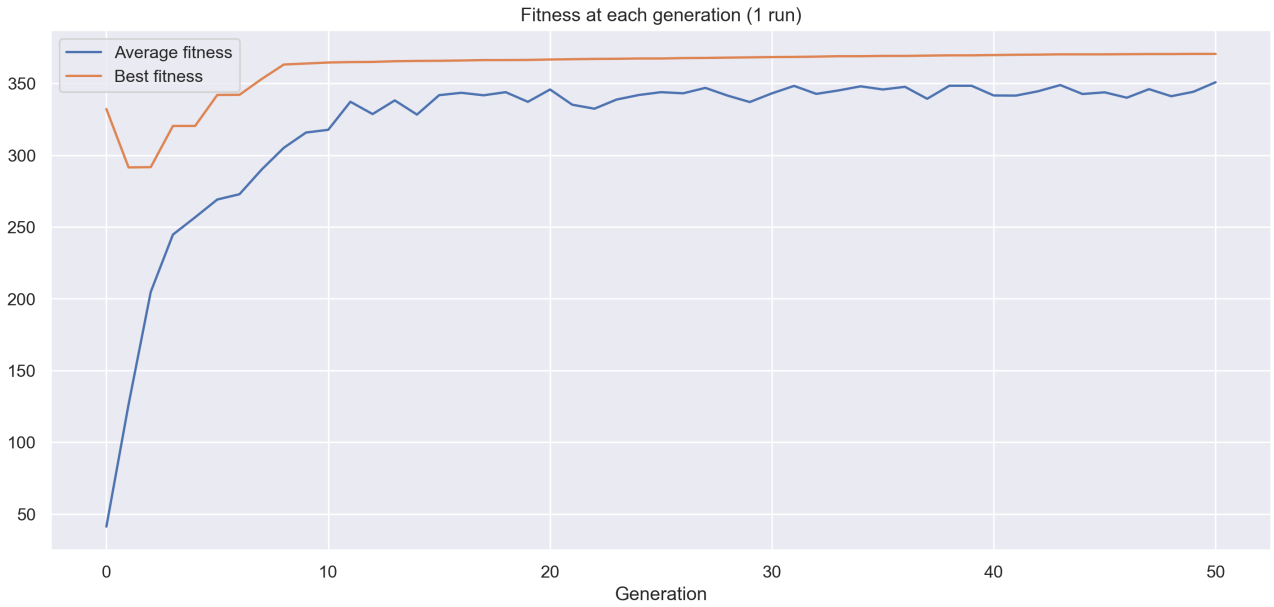

Figure 1: Line plot of the best fitness at each iteration

[1]The implementation is available on GitHub at `https://github.com/maxvandenhoven/evolutionary-petri-net-discovery`

Exercise 2: Performance Tests with Varying Operators

In order to explore the best combination of evolutionary operators for the GA, we test the performance of eight different setups. Each configuration is run 30 times to decrease the effects of randomness in our evaluation. The performance of each configuration can be seen in Table 1, which shows the average running time and best fitness found over 30 iterations. The best fitness and best time are shown in bold. Figure 2 shows the average best fitness at each generation to investigate possible premature convergence. Figure 3 shows the distribution of the best fitness value to investigate the variability of each configuration.

Crossover	Mutation	Selection	ABF	Time per run (Seconds)
cxTwoPoint	mutFlipBit	selTournament	368.87	4.8749
cxTwoPoint	mutFlipBit	selRoulette	361.59	**4.4756**
cxTwoPoint	mutShuffleIndexes	selTournament	**370.60**	4.7662
cxTwoPoint	mutShuffleIndexes	selRoulette	363.89	4.4911
cxOrdered	mutFlipBit	selTournament	366.72	5.1725
cxOrdered	mutFlipBit	selRoulette	360.12	5.0700
cxOrdered	mutShuffleIndexes	selTournament	370.58	5.0758
cxOrdered	mutShuffleIndexes	selRoulette	363.87	5.1123

Table 1: Resulting average best fitness for varying operators

Figure 2: Line plot of the average best fitness at each generation for each operator configuration

According to Table 1, the highest fitness can be obtained by using the operators cxTwoPoint, mutShuffleIndexes and selTournament. This also results in one of the shortest running times, as cxOrdered seems to increase the running time. Figure 3 shows that this configuration also has relatively low variability, meaning that is consistently finds high quality solutions. It also appears to converge the fastest, as shown by Figure 2.

Furthermore, Figure 2 clearly shows a negative impact of the selRoulette operator, Getting consistently and decisively outperformed by the selTournament operator. Figure 2 also shows that the mutShuffleIndexes operator slightly outperforms the mutFlipBit operator, especially when combined with the selRoulette selection operator. This makes sense, as the default implementation of the mutFlipBit operator in DEAP is based on the NOT operator, which is not well suited towards attribute values of -1 as opposed to the normal boolean values. Figure 3 also shows that the mutFlipBit operator introduces a lot of variability in the fitness distribution.

Following this test, we will use the cxTwoPoint, mutShuffleIndexes and selTournament operators for the remainder of this assignment, as this configuration lead to the best performance in terms of fitness, running time, and variability.

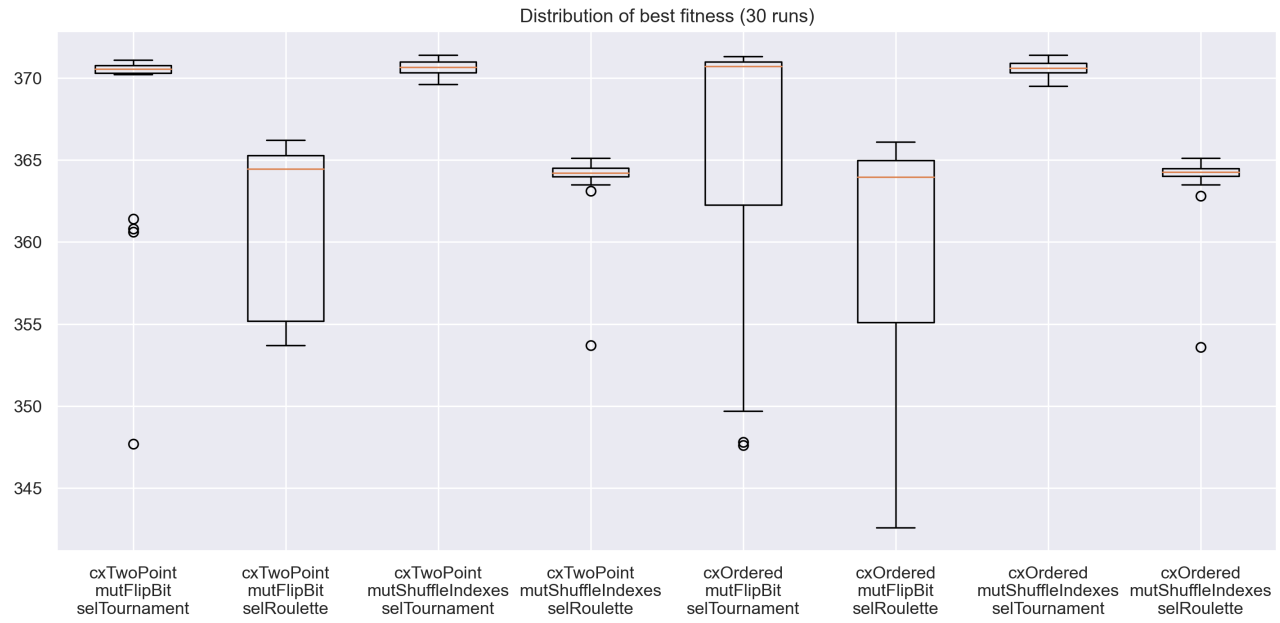

Figure 3: Box plot of the distribution of best fitness values for each operator configuration

Exercise 3: Performance Tests with Varying Hyperparameters

In order to explore the optimal combination of crossover and mutation probabilities for the GA, we test the performance of sixteen different setups. Each configuration is run 30 times to decrease the effects of randomness in our evaluation. The performance of each configuration can be seen in Table 1, which shows the average running time per iteration and best fitness computed over 30 iterations. The best fitness and best time are shown in bold. Figures 4 and 5 again show the average best fitness at each generation and the distribution of the best fitness respectively.

ABF (Time in Seconds)		Crossover Probabilities			
		0.2	0.4	0.6	0.8
Mutation Probabilities	0.2	367.74 (**2.46**)	368.92 (3.25)	369.82 (4.00)	370.63 (4.65)
	0.4	368.50 (3.71)	369.44 (3.60)	370.53 (4.13)	**370.98** (4.67)
	0.6	368.70 (3.57)	369.67 (3.93)	370.48 (4.30)	370.75 (4.61)
	0.8	368.08 (4.16)	368.98 (4.22)	369.42 (4.40)	369.65 (4.65)

Table 2: Resulting average best fitness for varying hyperparameters

According to Table 2, the best combination of hyperparameters is a crossover probability of 0.8 and a mutation probability of 0.4. Furthermore, this combination has a somewhat more consistent results than the other combinations, as seen in Figure 5. In particular, a high mutation probability seems to introduce a high variability, as is to be expected.

Furthermore, Figure 5 shows that a higher crossover probability leads to better results, whereas the mutation probability should not be too high, nor too low. The combination of these two observations results in the observed saw-tooth pattern in the box plot.

Following this test, we will use a crossover probability of 0.8 and a mutation probability of 0.4 for the remainder of this assignment, as this configuration lead to the best performance in terms of fitness and variability. The running time naturally increases, but we find that this trade-off is worth it due to the improved performance in fitness, and the lower sensitivity of running time to mutation probability for high crossover probabilities (final column Table 2). The best petri net found by this configuration is also graphed in Figure 6, which will be discussed in Exercise 5.

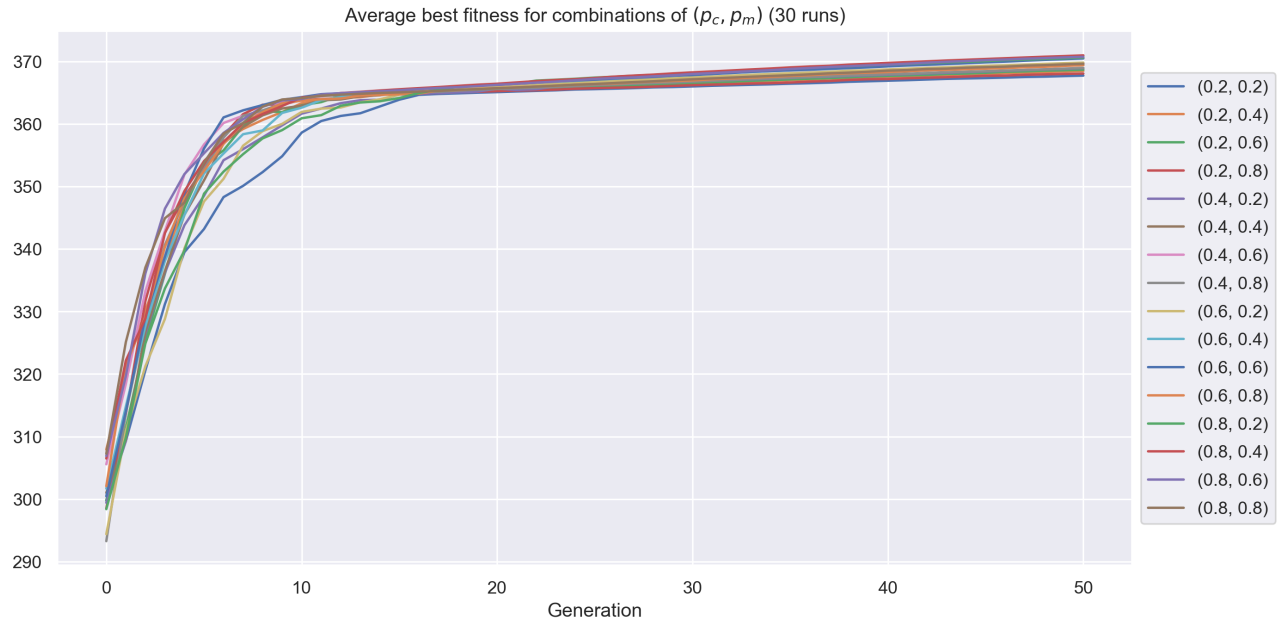

Figure 4: Line plot of the average best fitness at each generation for each operator configuration

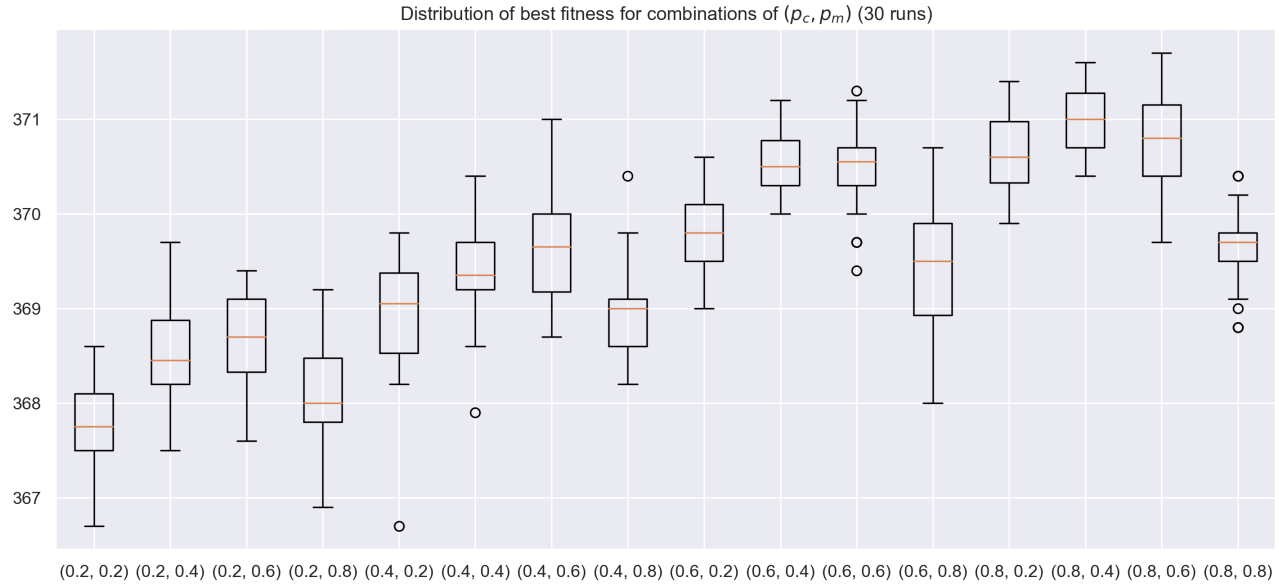

Figure 5: Box plot of the distribution of best fitness values for each configuration

Exercise 4: Flow Constraint with Tool Decorator

The requirement that a transition is connected to at most one place results in the constraint that each column has at most two non-zero entries. To fix any candidate solutions that do not fulfil this constraint, first any violations must be asserted. To this end, the amount of non-zero entries in each column is counted and added to a list. If a violation is asserted, this list of non-zero indices is shuffled and all but the first two entries are chosen to be deleted and set to zero.

A downside of this process is that it may take longer to arrive at an optimal individual. Due to trimming random entries it is possible that otherwise successful individuals will lose fitness after the constraint is enforced. This loss of fitness may be addressed by having a larger amount of generations that can still evolve towards suitable graphs of high fitness. Another option is a more deterministic method of fixing violating columns, but this is at the risk of introducing bias towards certain places.

Exercise 5: Visualized Petri Net

The graph represented by the optimal individual generated using the operators and hyperparameters determined in Exercises 2 and 3 can be seen in Figure 6. This individual has a fitness value of 370.98. As can be seen, all transitions are connected to the graph except for the two invisible actions. Furthermore, this graph also has several cycles. These cycles model certain actions being taken multiple times in the process. An improvement would be to create a cycle by connecting the fraud-check succeeding node to the fraud-check preceding node by one of the unused invisible transitions.

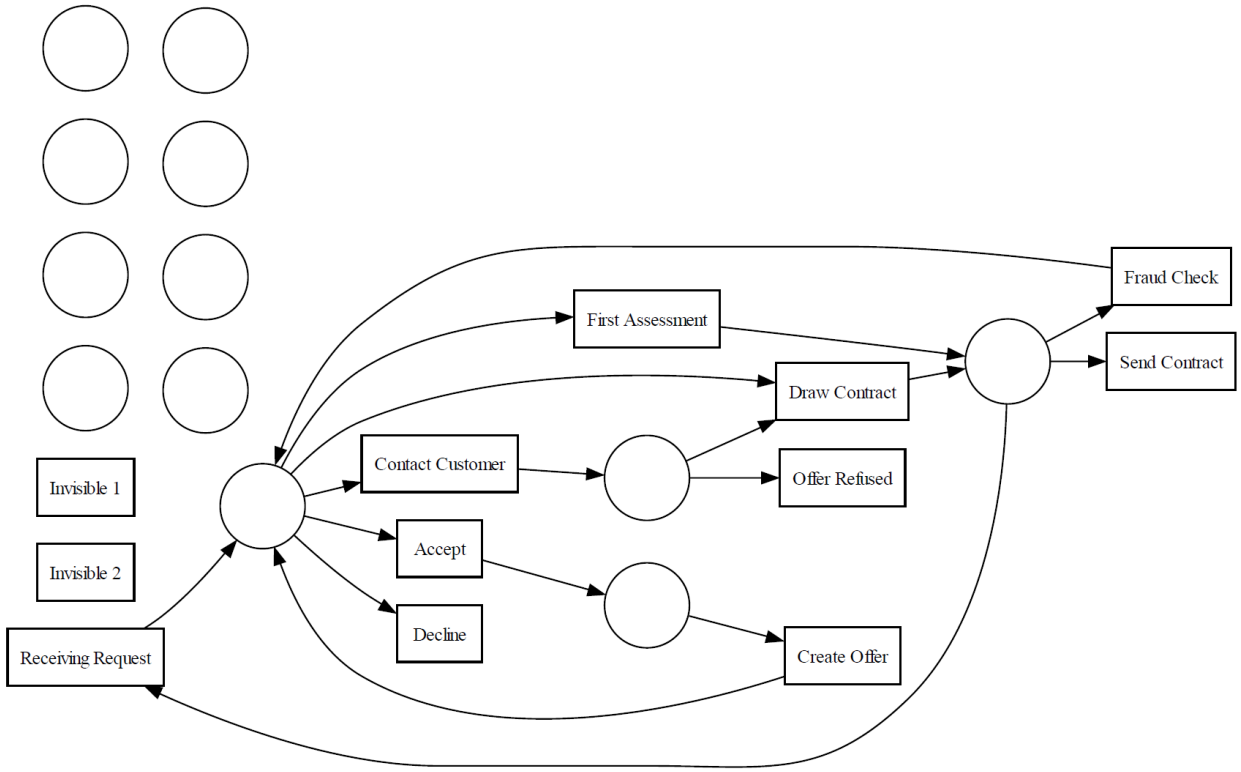

Figure 6: Petri net represented by the optimal individual without using the decorator

The graph represented by the optimal individual when using the decorator to enforce the constraints is shown in Figure 7. This is a graph with a fitness value of 351. Note that four transitions are not connected, thus there are traces that can never be simulated by this graph. Furthermore, this graph does not allow for multiple successive fraud-checks. A possible improvement might again be using the invisible transition to allow multiple consecutive fraud-checks.

A comparison between Figures 6 and 7 clearly shows the difference the decorator makes. When the algorithm is not using the decorator, the resulting graph is far more complex, while only one transition is connected to by multiple places, which is undesirable under the constraint the decorator enforces. This individual also has a far higher fitness despite containing more edges, meaning it simulates far more traces accurately. All in all, the use of the decorator with this setup does not seem undeniably beneficial, although further testing with runs with larger populations and more generations might prove fruitful.

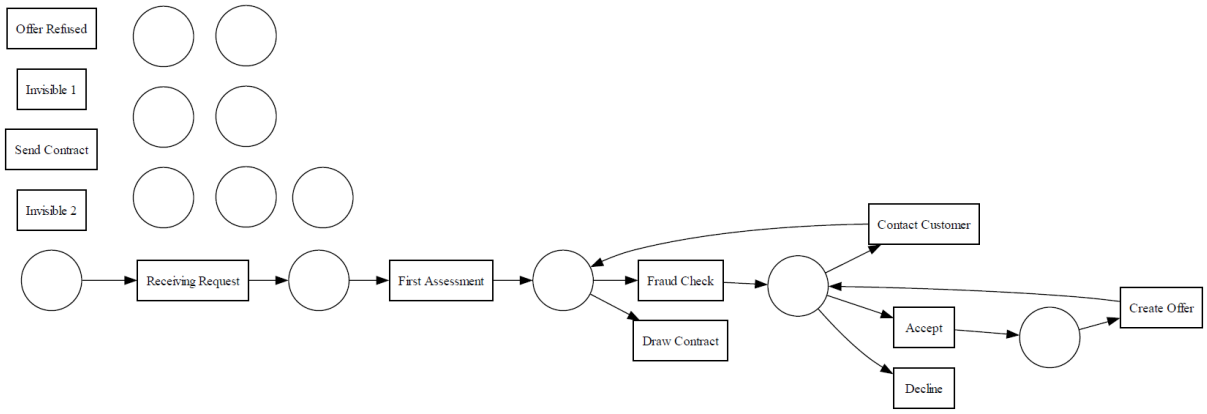

Figure 7: Petri net represented by the optimal individual using the decorator