Finding the optimal sorting network by harvesting the power of coevolution

Ghiga, Claudiu-Alexandru
claudiu.ghiga@info.uaic.ro

Nistor, Serban
serban.nistor@info.uaic.ro

November 16, 2020

Abstract

Finding the optimal sorting network is still a challenging problem and many different methods exist in order to try and solve it. One particular strategy consists in using genetic algorithms in a coevolution manner. We will see in this paper that this method can be really interesting and that it can be extend in many ways with minimal effort.

Keywords— Optimal Sorting Network, Genetic Algorithm, Coevolution

1 Introduction

Sorting networks are data-oblivious algorithms that are used to sort numbers. In other words, regardless of the form that the numerical input data takes, a sorting network will produce as output a monotonically increasing sequence of the provided data. Sorting networks are composed of wires and comparators.

The problem that we have tried to solve consists in finding an optimal size sorting network for a given number n of wires. Since the problem of finding an optimal sorting network of size k necessitates checking for the existence of all sorting networks with size smaller than k and the number of possible configurations grows

by $\frac{n}{2}$ for each new comparator added, the problem is in co-NP.[1] Because of this, much of the research done in this domain focuses on finding lower and upper bounds for the size of sorting networks with n wires and in using heuristic algorithms in order to reduce the search space sorting networks.

2 Proposed Method

Our proposed method focuses on using a nature inspired method, namely coevolution to solve this problem. We designed our coevolution procedure by thinking at the prey-predator model, and in this case the prey is the input that the sorting networks are receiving, the networks being in this case the predator.[2]

This method has an important advantage in our case, because our input data can be hard to compute if we aim for big sized networks, lets say with a size of 16 for example, so by using the proposed method, we can begin by generating only a part of the input, which will then evolve alongside our sorting networks.[3] As you probably noted, we have two species that need to evolve in order to survive: the input population and the sorting networks population.[4]

The input population is represented using binary encoding, each individual being a binary array with the same size. As for the sorting networks population, an individual is represented as a list of pairs, each pair representing a comparator. The generation of both the populations is random, and as for the actual sizes used, the input population has 100 individuals and the sorting networks population has 300 individuals.

The genetic algorithm uses as the fitness function for the sorting networks population $\frac{\#\text{no\_of\_sorted\_cases}}{\text{input\_population\_size}}$ and the opposite form, $1 - \frac{\#\text{no\_of\_sorted\_cases}}{\text{input\_population\_size}}$ for the input population. In this case, the best sorting network would have a fitness of 1 if it can sort all of the inputs from the input population, and symmetrically an input would be evaluated by the number of sorting networks which could not sort it.

The selection is done by using the rank-based selection in order to try to avoid premature convergence.[5] This method, in general leads to a slower convergence towards a global optima, but in this particular case it helps with not remaining stuck in a local optima, so the slower convergence is not really a problem. We also use elitism as an addition to this selection procedure because we empirically observed that the genetic algorithm needs less iterations to escape local optima in the late stages of the evolution compared to the case when no elitism was in place.[6]

The number of individuals selected by using elitism is 10% from the total population. This procedure helps keeping the best sorting networks in the evolution loop, but also the best testing cases because this selection is used for both of the species. Another important thing to mention is the fact that this selection procedure keeps the number of the individuals in both populations unchanged.

In terms of genetic operators, we use mutation and crossover and we will discuss them separately for each population. For the input population, the mutation is

something standard and it is done by flipping one randomly chosen bit from the individual. The mutation rate is at 5%, which was chosen empirically in order to keep the balance between exploration and exploitation. The crossover for the input population is also pretty standard when it comes to genetic algorithms and it is achieved by randomly selecting an even number of candidates that will go through crossover; the crossover itself it is done by splitting each candidate in half and then combining the pieces from two distinct candidates. The probability that a candidate will undergo crossover is set at 60%.

Now, when it comes to the operators used on the sorting networks population, things are a little bit different. We have implemented three different possible mutations. The first one consists in choosing two random comparators and swapping their contents for a network, the second one will be adding a new comparator to the current individual, at a randomly chosen index and the third one will remove one randomly chosen comparator from the current network structure.

The probability for an individual to undergo mutation is of 5%, and then the probability to apply the swapping mutation is of 80%, the adding mutation of 10% and the removing mutation of 10%.

3 Improvements

3.1 Parallel evaluation

One of the most restricting aspects of our initial algorithm was its performance. The code profiler revealed that, as is typical of genetic algorithms, the evaluation step was the most costly. Even considering the fact that we work directly with an integer based representation, which allows us to compute the fitness of the populations without decoding and reencoding the genotype, the evaluation step still monopolized the time resources allocated for our algorithm. As a first solution to this problem, we used the we used the multiprocessing capabilities offered by the Python Standard Library[7] in order to split the network population into non-overlaping chunks that can be computed in parallel. This is achieved by creating a matrix that is shared by all the subprocesses, defined as

$$a_{ij} = \begin{cases} 1, & \text{if network } n_i \text{ can sort input } m_j \\ 0, & \text{otherwise} \end{cases}$$

The network relative fitness is obtained by summing along the columns and normalizing by the total number of columns, while the input relative fitness is the complementary of the normalized sum along the rows.

While the multiprocessing module did reduce the cumulative execution time of the algorithm, it did not offer a substantial enough increase and problems for $n > 13$ were still unfeasible. In order to address this issue, we redesigned the representation in order to make it more reliant on the fast low-level C functions of

the numpy library that make efficient use of computational resources and are more ammenable to multithreaded execution. Afterwards, we rewrote the critical parts of the evaluation function in Cython and released the Global Interpreter Lock for the sorting procedure of the networks, allowing us to take full advantage of multicore CPU architectures. The total time spent on the evaluation was reduced by almost two orders of magnitude compared to the original execution time.

By parallelizing the evaluation step of our genetic algorithm, we have made it possible to test the genetic algorithm on larger problem instances while keeping execution times within a reasonable time frame.

3.2 Suplimentary selection strategies

The selection step controls the selection pressure applied to the population, and conversely controls the degree to which the populations conerge. As a consequence, care must be taken when choosing a selection strategy: it must impose enough pressure on the population in order to shift the mean towards better fitness values, but at the same time it must not apply enough in order to induce premature convergence.

Our initial selection strategy was an approach based on ranking and not on the actual fitness scores. The fitness values were preprocessed before the actual selection by mapping them to the exponents of the exponential function a^x, where a is a strategy parameter. The choice of a controls the amount of selection pressure applied to the population: values closer to 0 lead to increased pressure, while values close to 1 decrease pressure.

Since our algorithm performed poorly with rank-based selection because of rapid convergence to highly-specialized populations for the current input sample, we have added two new strategies in order to experiment further with methods that keep more low-performing individuals in the gene pool.

The first one is the commonly-used roulette wheel procedure,[8] where the probability of an individual being selected is proportional to the relative difference between its fitness value and the rest

$$p_k = \frac{f_k}{\sum_{i=1}^{n} f_i}, \text{where } n \text{ is the size of the population}$$

This method can help fight against premature conergence especially in the case of coevolutionary genetic algorithms because it offers individuals high chances of being selected whenever the variance among the fitness scores is low, thus increasing population diversity.

The second selection strategy is named Stochastic universal sampling. This is a variant of the roulette-wheel method, which selects the individuals in the next iteration based on the size the fitness score relative to the entire population, but eliminates possibly damaging effects of random noise.[9] By choosing an initial

pointer in the interval $[0, P]$, where $P = \sum_{i=1}^{N} \frac{f_i}{N}$, which splits the entire interval into evenly spaced subintervals starting from the first pointer, the computed pointers end up choosing individuals both proportionately to their fitness and also by sampling the entire population. Thus, every individual gets a chance of being chosen, thus increasing population diversity over the entire run of the algorithm.

3.3 Spliting network and input hyperparameters

Another important improvement to the algorithm is the separation of the hyperparameter values for the network population and the input population. Now each population has configurable values for the mutation probability, crossover probability and selection strategy. This makes the genetic algorithm more flexible as it opens the possibility for more hyperparameter fine-tuning.

3.4 Automated Hyperparameter Tuning

A very useful algorithm that we implemented alongside our genetic algorithm was one that automatically finds the best hyperparameters that maximize our sorting network fitness. Instead of using an exhaustive searching algorithm like grid search or other similar techniques we have used an algorithm based on bayesian optimization.

Bayesian Optimization is an approach that uses Bayes Theorem to direct the search in order to find the minimum or maximum of an objective function. It is a powerful strategy for finding the extrema of objective functions that are expensive to evaluate, being particularly useful when these evaluations are costly, when one does not have access to derivatives, or when the problem at hand is non-convex.

Our method uses gaussian processes to minimize the function

$$1 - \frac{f_{\text{rel}} + f_{\text{abs}}}{2}$$

where f_{rel} is the relative fitness of the best network and f_{abs} is the absolute fitness of the best network. The method performs a maximum of 100 calls of the GA with the chosen parameters and it keeps track of the parameters that minimize the above function. We added early stopping to the bayesian optimization in order to optimize the computational resources, so the algorithm will stop the search if the value of the above function will reach 0, or in other words if a sorting network from our population reaches a fitness of 1 on both the full dataset and the relative dataset.

The search space that we fed to the meta-algorithm was made of:

- the sorting networks mutation rate with a value between $[0.05, 1.0]$
- the sorting networks crossover rate with a value between $[0.05, 1.0]$

- the input mutation rate with a value between $[0.05, 1.0]$
- the input crossover rate with a value between $[0.05, 1.0]$
- the sorting networks selection method: *Rank Based, SUS, Roulette Wheel*
- the input selection method: *Rank Based, SUS, Roulette Wheel*
- the sorting networks population size with a value between $[100, 10000]$

An important thing to mention is that for SUS selection, elitism was used consisting of 5% from the population size. After five separate runs, this method manages to find the best hyper-parameters on an average of 12 runs, for N=8. In one of the runs, the parameters found were the following:

- sorting networks mutation rate: 0.897
- sorting networks crossover rate: 1.0
- input mutation rate: 0.05
- input crossover rate: 0.522
- sorting networks selection method: Rank Based
- input selection method: Roulette Wheel
- sorting networks population size: 5795

The evolution of the search can be seen in Figure 6

3.5 Adaptive mutation rate

In order to further help the genetic algorithm converge we have implemented an adaptive mutation rate based on the fitness evolution of the best network individual from the population. The idea is that if the GA reaches a slow convergence rate, referring to the absolute fitness value we tweak the mutation rate in order to raise the exploration rate of the algorithm.

The strategy that we've used for updating the mutation rate is the following: we keep a history of the absolute fitness values and we use a window of size 10 in which we check if there was a significant improvement between the first and last iteration from the current window. If the improvement is less than an epsilon (we've empirically chosen a value of 0.03), but higher or equal than 0, than we increase the mutation rate for the network population with a value of 0.01.

If it's the other way around, and the improvement is negative, in other words, there was a decrease in the fitness value, we decrease the mutation rate by the same amount in order to try to re-establish the algorithm's balance.

We tested this technique by starting with the best hyper-parameters for N=8, and we did observe that the algorithm had a tendency to converge faster. On an average we've observed that the GA needs 5% less iterations to converge, but we strongly believe that this method is much more helpful when there is no prior hyper-parameter optimization made.

4 Experiments and discussion

What follows are different runs of the genetic algorithm with small changes in the parameter values that are meant to illustrate the effect a change in one parameter can have on the algorithm as a whole. All the experiments consist of 30 independent runs, each for 500 iterations. The baseline for $n = 8$ wires was arrived at in an experimental fashion, and it consists of the following hyperparameters for the network population:

- a mutation rate $p_m^{net} = 0.01$, of which:
  - 80% are comparator swaps,
  - 10% are comparator insertions and
  - 10% are comparator deletions;
- a crossover probability $p_c^{net} = 0.6$;
- stochastic universal sampling as the selection strategy with no elitism;
- population size of 500 individuals;

and the following values for the input population:

- a mutation rate $p_m^{input} = 0.5$, of which:
- a crossover probability $p_c^{net} = 0.2$;
- roulette wheel as the selection strategy with no elitism;
- population size of 100 individuals;

One thing worth noting is that we only edit the values of the network population's hyperparameters. This is because the settings for the input population only have to be set in such a way that population diversity is maximized and maintained at all times. Any other setting risks leading to the overspecialization of the network population for the input population at that iteration. Since the fitness function is relative to the sampled input population, it does not discriminate between comparator networks that can sort the sample well and networks that can sort the entire input space well. This would lead to networks that have large relative fitness scores, but low absolute (real) fitness scores.

The final results are summarized in Table 1.

4.1 Selection

Our first observation is related to the differences between selection strategies. Figure 1 presents the evolution of the fitness values of the population over the course of 500 iterations for 4 different selection methods. As can be observed from the graphs, rank-based strategies have the best mean network fitness and an increase in selection pressure (lower values for a) leads to faster convergence towards higher fitness candidate solutions. The fitness proportionate (roulette wheel) selection gives the worst results because of the combined effects of reduced pressure and influence of stochastic noise in the selection procedure.

4.2 Mutation rate

The change in mutation rate , which can be observed in Figure 2, has the expected outcome: if it is too high, than promising potential solutions are destroyed and the final result is a population with high variance and low mean fitness.

4.3 Crossover rate

The coevolutionary algorithm was executed with different crossover rates in order to ascertain the influence of the recombination operator on the convergence rate of the algorithm. The resulting trends can be seen in Figure 3. In this case, faster convergence was obtained for crossover rates $0.2 < p_c^{net} < 0.6$, which points to the fact the crossover rate does have meaningful input when it comes to finding a solution. While this may seem to be the case at first glance, multiple runs of the algorithm with different values for the crossover probability gave no statistically significant differences in the final outcome.

The only statistically significant differnece has been obtained by running the algorithm with $p_c^{net} = 0$, which had a worse final outcome. This at least likely proves that the crossover operator improves the fitness values and that sorting networks that have good mean fitness scores imply the existence of slices/subnetworks that have higher than average fitness scores.

4.4 Elitism

Elitism was also tried as a measure by which to accelerate convergence. By looking at the evolution of the fitness in Figure 4, we observe that the run with no elitism has a more smooth and slow growth while one the which makes use of 5% elitism benefits from faster convergence and sudden changes in fitness. This behaviour is a consequence of introducing single solutions with good fitness that would normally be erased by selection or mutation.

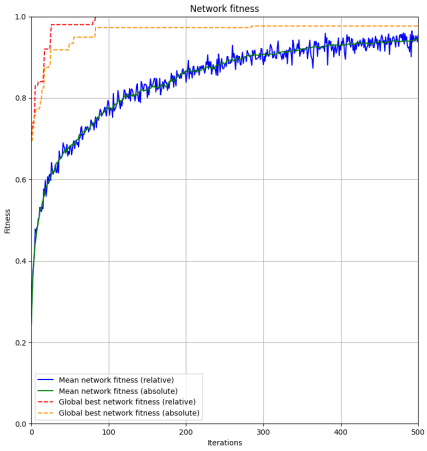
4.5 Larger problem instances

For $n = 13$, networks were initialized with comparators between the current known lower and upper bounds for the 13 wires.[10] The same probability distributions for the mutation cases were kept and a higher population was selected. With only a population that represents 12.5% of all the inputs, the network population managed to reach a mean fitness level of 0.853, a relative global maximum of 0.961 and a size of 44 comparators. The algorithm was stopped early at the 130th iteartion in order to minimize the chances of finding a larger network that could sort all of the inputs. The evolution of the algorithm can be examined in Figure 5 and the resulting network can be found at Appendix C.

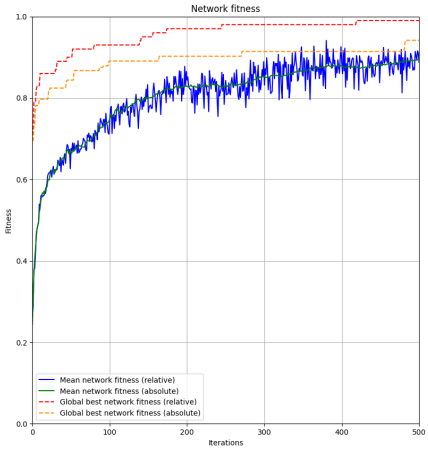
5 Conclusion

As we've already seen, the described method can yield great results, but also some weak ones, depending on the size of the input. Although, the possibilities at which this method can be extended and also the number of different implementations that can be done for the core components: the fitness function, the selection procedure and the genetic operators are limitless, therefore, to conclude, by further experimenting with coevolution on this particular problem, great results may be achieved.

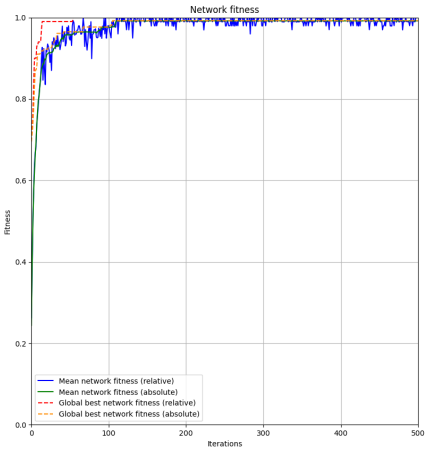
A List of figures



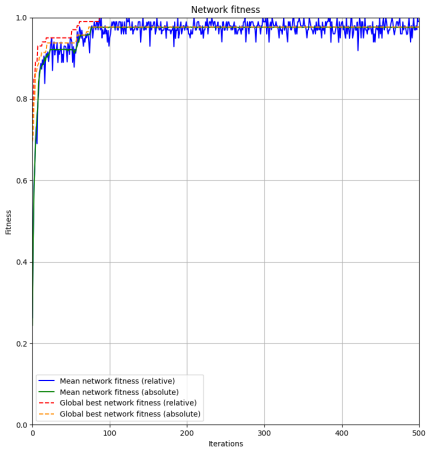
(a) stochastic universal sampling



(b) roulette wheel

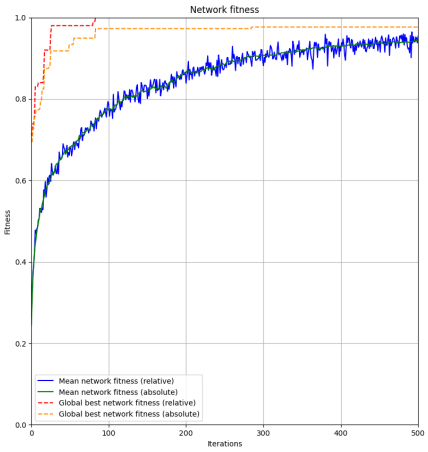


(c) rank-based ($a = 0.99$)

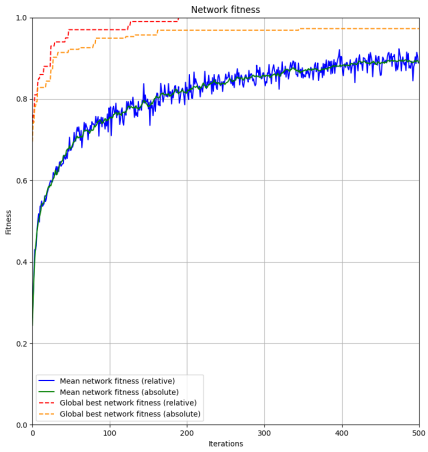


(d) rank-based ($a = 0.98$)

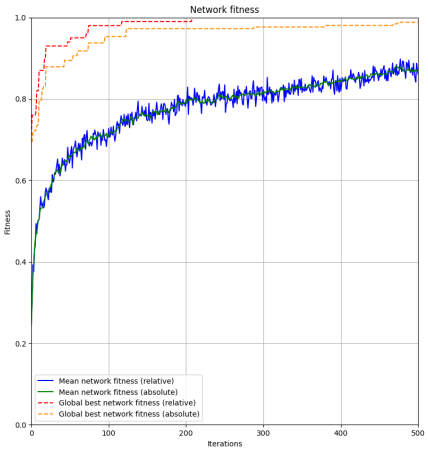
Figure 1: Comparison of differnet selection strategies. (a) is the baseline performance.



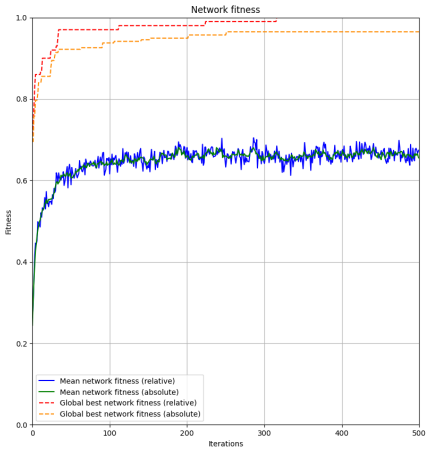
(a) $p_m^{net} = 0.01$



(b) $p_m^{net} = 0.05$

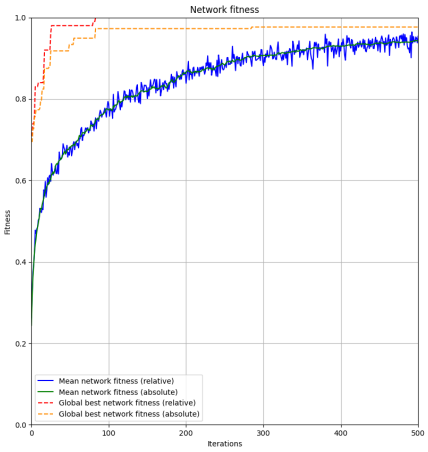


(c) $p_m^{net} = 0.1$

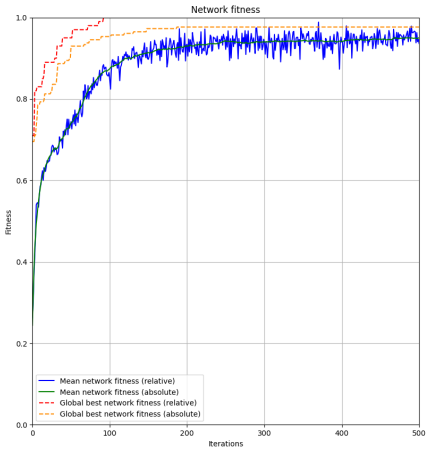


(d) $p_m^{net} = 0.5$

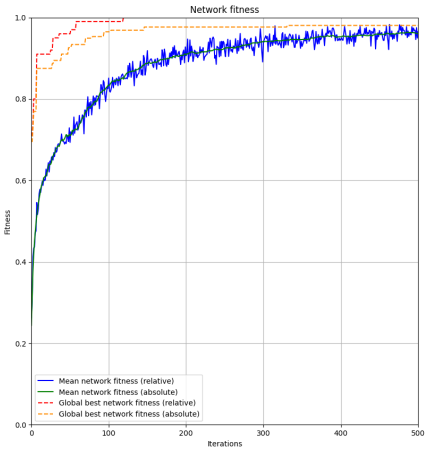
Figure 2: Comparison of differnet mutation rates. (a) is the baseline performance.



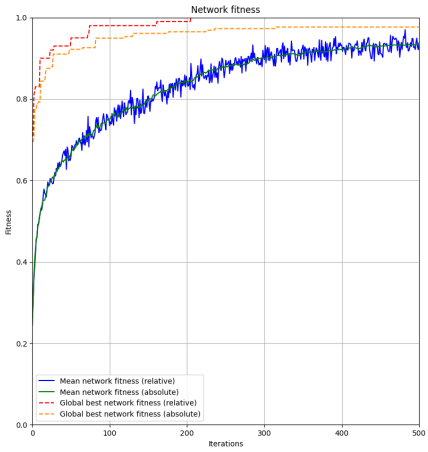
(a) $p_c^{net} = 0.6$



(b) $p_c^{net} = 0.2$

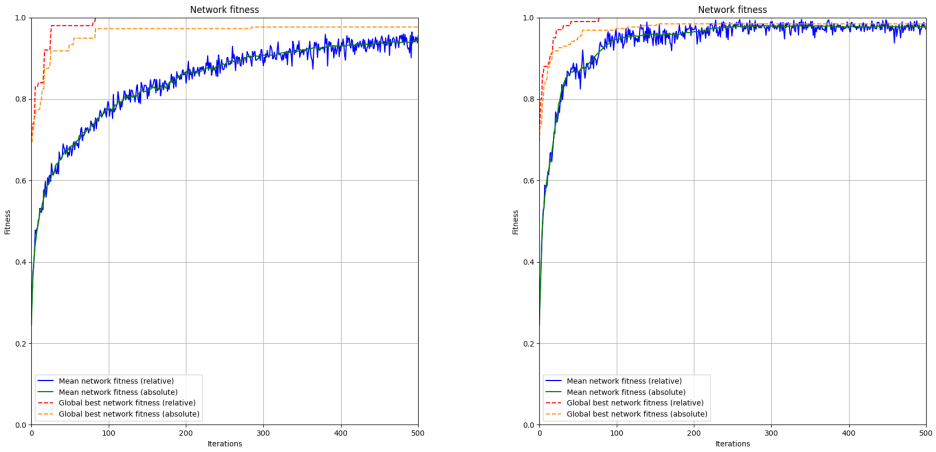


(c) $p_c^{net} = 0.4$



(d) $p_c^{net} = 0.8$

Figure 3: Comparison of differnet crossover rates. (a) is the baseline performance.



(a) no elitism (b) 5% elitism

Figure 4: Comparison between a genetic algorithm with no elitism and one elitism where 5% of the best solutions are kept in the next iteration. (a) is the baseline performance.

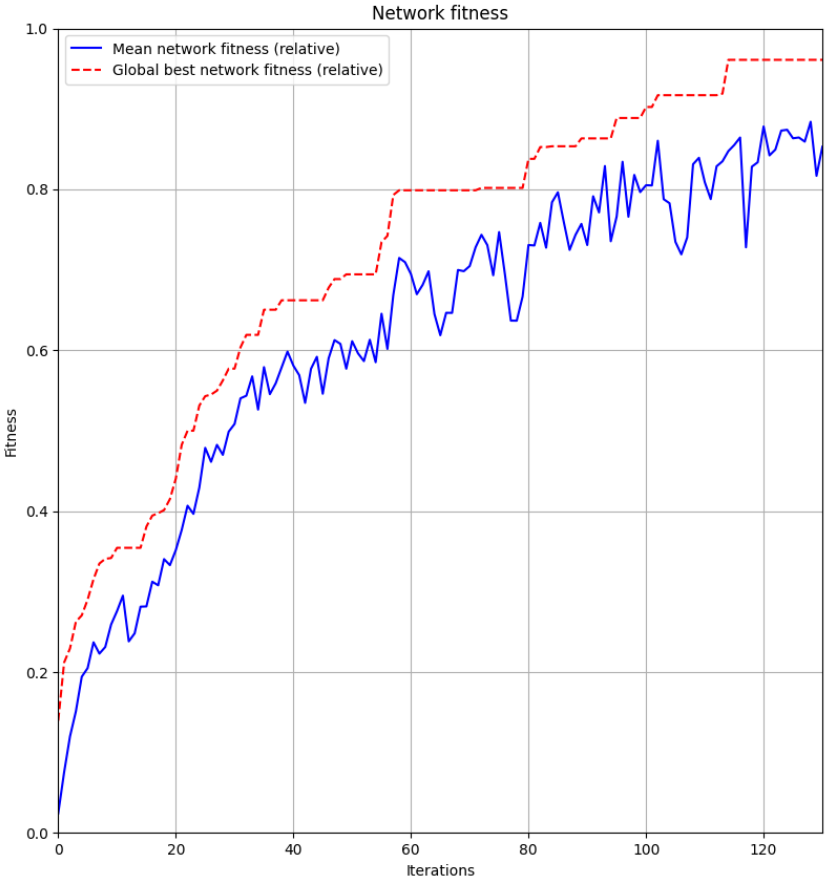


Figure 5: The results for obtained by the coevolutive genetic algorithm for $n = 13$.

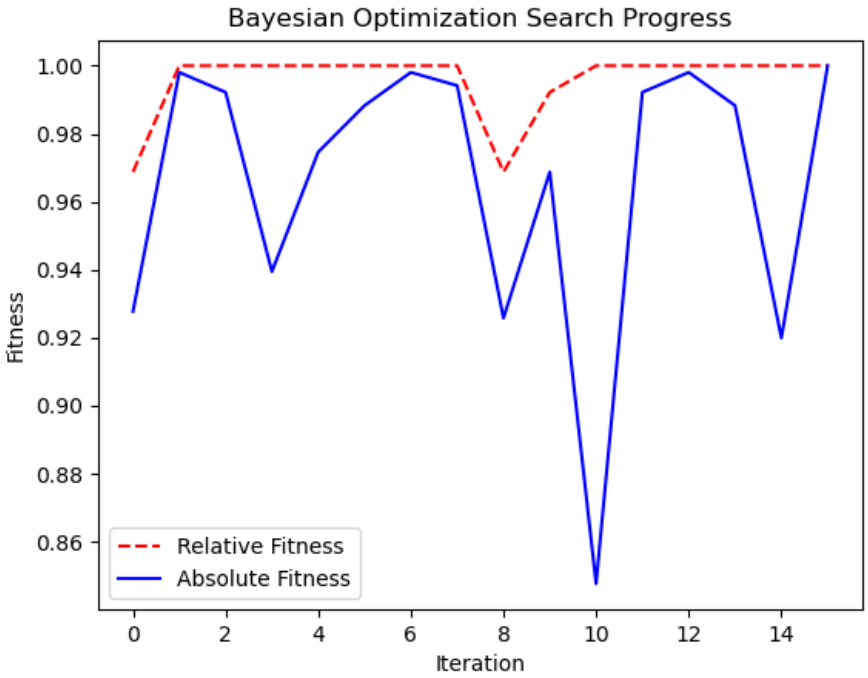


Figure 6: The results of Bayesian Optimization for $n = 8$.

B Tables

Selection	**Crossover**	**Mutation**	**Elitism**	**t(s)**	σ**(s)**	μ_{rel}	σ_{rel}	μ_{abs}	σ_{abs}	k
SUS	0.6	0.01	no	7.581	0.084	0.937	0.020	0.936	0.010	19
Roulette	0.6	0.01	no	7.519	0.093	0.921^{**}	0.025	0.923^{***}	0.022	19
Rank $a = 0.99$	0.6	0.01	no	7.691	0.102	0.968^{***}	0.026	0.972^{***}	0.013	19
Rank $a = 0.98$	0.6	0.01	no	7.666	0.040	0.965^{***}	0.022	0.966^{***}	0.012	20
SUS	0.6	0.01	yes (5%)	7.667	0.182	0.964^{***}	0.020	0.969^{***}	0.012	19
SUS	0.6	0.05	no	7.758	0.093	0.904^{***}	0.016	0.901^{***}	0.010	19
SUS	0.6	0.1	no	7.814	0.077	0.863^{***}	0.016	0.862^{***}	0.010	18
SUS	0.6	0.5	no	7.900	0.556	0.660^{***}	0.014	0.658^{***}	0.010	20
SUS	0.2	0.01	no	7.970	0.515	0.937	0.020	0.937	0.014	19
SUS	0.4	0.01	no	7.581	0.305	0.933	0.019	0.937	0.011	19
SUS	0.8	0.01	no	8.323	0.557	0.927	0.018	0.932	0.012	19

Table 1: Comparisons between the means of the fitness values for different settings of the hyperparameters. The optimal number of comparators for $n = 8$ is $k = 19$. First row is the baseline against which all other configurations are tested. For comparisons, two-tailed paired t-tests were used in order to establish differences between experiments. $^{***}p < 0.01$, $^{**}p < 0.05$, $^{*}p < 0.1$

C Output sample

- $n = 8$

```
[(5 6), (0 7), (1 7), (2 4), (3 4), (2 5), (1 3), (4 7), (0 2),
(1 5), (1 2), (2 3), (3 5), (4 6), (4 5), (3 4), (2 3), (3 4),
(5 6)]
```

- $n = 13$

```
[(5 9), (0 12), (9 12), (3 6), (3 5), (4 8), (1 5), (6 11), (5
6), (8 11), (2 4), (6 12), (0 7), (0 10), (4 7), (2 5), (1 4),
(6 10), (6 9), (4 8), (8 9), (1 6), (4 6), (6 11), (1 2), (6
8), (5 6), (6 7), (1 3), (0 3), (3 6), (2 4), (6 10), (6 8), (3
5), (4 5), (2 3), (7 10), (9 10), (0 2), (10 11), (7 8), (0 1),
(3 4)]
```

References

[1] Parberry, I. 1991. *On the computational complexity of optimal sorting network verification.*

[2] Cooperative coevolution
`https://en.wikipedia.org/wiki/Cooperative_coevolution`

[3] Hillis, W.D. 1990. *Co-evolving parasites improve simulated evolution as an optimization procedure.*

[4] Coevolutionary Principles
`https://www.cs.tufts.edu/comp/150GA/handouts/nchbmain.pdf`

[5] Booker, L. 1987. *Improving Search in Genetic Algorithms.*

[6] Baker, J.E. 1985. *Adaptive Selection Methods for Genetic Algorithms.*

[7] The Python Standard Library Documentation - multiprocessing module
`https://docs.python.org/3/library/multiprocessing.html`

[8] How to perform Roulette wheel and Rank based selection in a genetic algorithm?
`https://medium.com/@setu677/howtoperformroulettewheelandrankbased-selectioninageneticalgorithmd0829a37a189`

[9] Baker, J.E. 1987. *Reducing Bias and Inefficiency in the Selection Algorithm*

[10] Codish M., Cruz-Filipe L., Frank M. and Schneider-Kamp P. 2014. *Twenty-Five Comparators is Optimal when Sorting Nine Inputs (and Twenty-Nine for Ten)*