Group 24 Assignment Report

An Evolutionary approach to elevator time optimization

Oskar Sundberg Linus Savinainen
Samuel Wallander Leyonberg Gustav Pråmell
Joel Scarinius Stävmo

October 2, 2024

Contents

1. Introduction

This report addresses an investigation for optimizing an elevators route to improve the efficiency of picking up and delivering passengers to their desired floors. In large buildings with a lot of floors, efficient elevators are crucial for managing traffic and must serve everyone in a reasonable time. Poorly designed elevator systems will lead to long waiting time, unnecessary stops and unsatisfied users. Elevator technology have made progress during the years, but many elevators still struggle with finding an efficient way to serve passengers.

The hypotheses that evolutionary algorithms will be able to find near-optimal or optimal routes for elevators aiming to maximize the number of passengers served while minimizing travel distance and travel time is the core focus in the report. Different strategies and hyperparameters are experimented with to enable demonstration of how evolutionary algorithms can be used in such a problem.

This paper will focus on a statically defined problem, i.e. there will not be more passengers coming to the elevator as time goes on, unlike a normal elevator.

2. Optimize elevator routing

2.1. Mathematical formulation

Diffing a lower bound:
D_n is the distance needed, D_t is the distance travled, N is the number on people in the building and F is the number of floors, using zero index.

Assumption I:
The distance between any floor J and $J+1$, or J and $J-1$ is 1 and the *time* required is 1.

Assumption II:
Following I, a person traveling from e.g. floor J to floor $J+3$, $D_n = |J-(J+3)|$.

Assumption III:
The elevator has a capacity of E people.

Assumption IV:
For X people that start on the same floor and needs to go to the same floor, the lowest *time* spent waiting is equal to $D_n \cdot X$.

Proof:
To prove IV, the trivial case is a building with $N = 0$. There is no need to move the elevator, resulting in $0 \cdot N = 0$ and $time = D_n \cdot 0 = 0$.

Simple case:
A building with $F = 2$ and $N = 1$, where the person needs to move for floor zero to floor one. It is trivial to see that the best solution is for the elevator to take one person from floor 0 to floor one. Resulting in $D_n = time = 1 = D_t \cdot 1$.

A more general case:
A building with $F = 2$ and P people that need to move from floor zero to floor one. The optimal solution is $P \cdot D_n = P \cdot D_t = time = P$. This is the lower bound of the problem. By proving IV, it can be shown that the lower bound is $D_n \cdot P$.

The upper bound:
To prove the upper bound the following solution can be imagined; in an arbitrary building, move the elevator continously between any two or more floors that does not have any people. Resulting in $1 \cdot \infty = \infty$.

2.1.1. Mapping the elevator problem to the swapping problem.

"Each vertex of a graph initially may contain an object of a known type. A final state, specifying the type of object desired at each vertex, is also given. A single vehicle of unit capacity is available for shipping objects among the vertices. The swapping problem is to compute a shortest route such that a vehicle can accomplish the rearrangement of the objects while following this route."[1]

- Vertices are represented as floors in the elevator.
- The object is defined as a person.
- Initial state consists of N people waiting at the floors.
- The final state is reached when all people are at their destination floor.
- The elevator is a vehicle with a max capacity that moves the people between floors, i.e. a single vehicle shipping objects among the vertices.
- The elevator problem is aiming to reduce the waiting time of the people in the elevator and thereby reduce the distance traveled.

"Even the simple swapping problem is NP-hard."[1]

2.2. Similar published academic work

A paper[2] explains how to solve a similar problem but with multiple elevators (what the paper call cars. Instead this study only looks at one elevator per building. The paper has been the starting point in terms of algorithms and approaches in this study. See 1

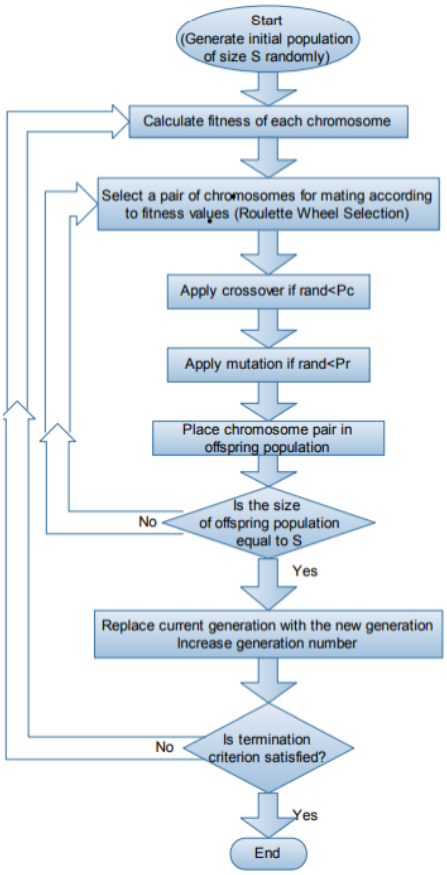


Figure 1: From [3]

The paper describes a 21% improvement with their methods compared to Ghareib paper [4] on the same problem. As this problem is not interlay equal to this study, comparison with this would not give a representee outcome. However, by following the approach in the paper, simulations were executed for the problem space and compare the results to results given with (theoretically) more suited genetic operations.

	Case Study Scenario
Passenger 1	5 → 9
Passenger 2	6 → 7
Passenger 3	3 → 15
Passenger 4	11 → 0
Passenger 5	20 → 8
Passenger 6	10 → 17
Passenger 7	13 → 19
Passenger 8	1 → 14
Passenger 9	16 → 2
Passenger 10	18 → 12

Figure 2: Where the f does this come from Samuel??

A paper[5] investigates for different approaches in terms of algorithm to the problem at hand. One of its algorithm to solve the elevator dispatching problem is a genetic algorithm where the paper use Davis-order crossover, swap mutation and calculate average time for occupants which is used as fitness function. The paper's building (problem space) and predefined settings were as following:
The results from their test show that GA where the best performing algorithm for this problem with the definitions above.
This study presents results that later compare to this study with this study's first implementation as given of the flow chart [2] as well as later implementation with updated genetic operations. Given the paper's result for testing different algorithms on the problem, a genetic algorithm approach seems to contribute with the best result to the elevator dispatching problem.

Another paper[6] shortly describes different types of crossover and mutation operations. The paper describes Wright's heuristic crossover and categorized it as one of the crossover functions how offspring are "in the exploration region near the parents"[6]. This reduce premature offspring by overcoming local minimum. Further investigation into different implementations of heuristic crossover, the concept seemed as a good crossover function for this study problem due to its simplicity and ability to be largely modified to suit the reports problem and fitness function.

2.3. Why this evolutionary approach

The report's starting point was highly inspired from the flowchart above [2]. The report's idea was to implement this as a first approach to have a working code base and to be able to easily switch between genetic operations for experimental reasons. From the papers mentioned in 2.2, similar papers, a conclusion was drawn that a heuristic crossover would suit the report's problem because it could be

implemented to swap larger sections of genes from different parents. This would be important due to the fact that it is certain sequences of genes that will contribute towards a better fitness rather than single genes. This study's fitness function was at first only focused on giving points for delivering a passenger to its desired floor. This was to make sure that every passenger would arrive. However, later this was changed to have the fitness function take the time spent for the passengers instead together with a penalty time for not delivering a passenger to its destination. The reports mutation function plays a fundamental role and is quite aggressive. It swaps genes and can increase or decrease the size of the chromosome. The decision was made for avoiding local maximums found by the crossover function. As the reports implementation of a heuristic crossover function takes larger sequences out of one parent, the thought was that this would not give an explorational function, thus bringing the need for a more aggressive mutation function to compensate for this.

3. Algorithm

3.1. Evolutionary approach

The genetic algorithm implemented in this study is based on the framework presented in a relatively recent paper [3], which explored a similar problem. The details of the implemented genetic algorithm can be found in 1. A key difference in this paper is the absence of a termination criterion; instead, the algorithm is set to execute for a specified number of generations. The flowchart illustrating the original framework is shown in Figure 6.

Algorithm 1 Genetic Algorithm

Require: $g \geq 0$
  Initialize a new population
  $g \leftarrow 0$
  while g < generation limit **do**
    for all genomes in population **do**
      Execute the genome on its people
    end for
    Calculate the fitness of the population
    Sort the population
    if want elitism **then** ▷ Start prepairing the next generation
      Bring some of the best genomes to the new population
    end if
    while the size of the new population < population size **do**
      Select two parents from the old population
      if the crossover chance is fulfilled **then**
        Breed two children using a crossover operator
      else
        Select the two parents as the two new children
      end if
      for each child **do**
        if the mutation chance is fulfilled **then**
          Apply a mutation operator
        end if
      end for
      Append the two children to the next population
    end while
  end while

3.1.1. Selection

The chosen selection algorithm is rank-based selection, and the basic idea behind it is to choose two parents based on their ranks rather than their raw fitness values. For example, if the population consists of six genomes, the best genome is given rank six, the second best rank five and so on. The worst genome is given rank one. Each genome is then given a probability P_i to be chosen based on their respective ranks, expressed as:

$$P_i = \frac{\text{rank}_i}{\sum \text{ranks}}$$

That is, the best genome would have a probability of $P_6 = \frac{6}{21}$ to be chosen, the second best $P_5 = \frac{5}{21}$, and the worst genome $P_1 = \frac{1}{21}$.

3.1.2. Crossover

The first basic crossover operator is one that simply swaps the last halves of the parents, while respecting the possibility that the length of the two parents may be of different length. For example, if the two parents are

$$[1, 2, 3, 4], [5, 6, 7, 8, 9, 10]$$

the resulting children would be

$$[1, 2, 8, 9, 10], [5, 6, 7, 1, 2]$$

Two last crossover functions where both heuristic crossover functions, meaning the parents fitness score determines how much of the genes from each parent is transferred to its children with favor for the "dominant" parent.

The first heuristic crossover function works simply by summarizing both parents' fitness scores and then calculating the parents' score differences in percentage. Random single genes are then transferred to the children where the previously calculated percentage gives a higher chance for transferring its genes based on that percentage.

The second crossover function works in a similar way but with the implementation of a crossover point instead of choosing single genes. This crossover point moves further down the dominant parents chromosome based on the difference between the parents fitness score. With the crossover point instead of choosing single genes,

the children get a larger sequence from the dominant parent and therefore more alike the dominant parent.

3.1.3. Mutation

Three basic mutation operators were implemented. Firstly, a swap mutation where two randomly chosen floors are swapped. Secondly, an operator which increases the length of the genome by appending a random floor at the end. Lastly, an operator that decreases the length by removing a randomly chosen index of the genome. With these three operators, an additional two operators were constructed by combining the effects of swap and either length operator.

3.2. Solution representation

Given the nature of this problem, there is not a typical direct solution representation approach as seen in e.g. the knapsack problem, where the solution representation is also directly what is being manipulated in the genetic algorithm. In this paper's elevator problem, the genome is a sequence of floors in which the elevator travels to with the goal to deliver as many people. This situation forced a 'two-way' representation, where the fitness of a genome is reflected upon a list of people. Thus it was decided to encapsulate the list of floors and the list of people under a single class 'Genome' as the solution representation.

3.3. Fitness function

After some basic trial and error, and more importantly, a healthy discussion with the assignment teacher, it was decided to exclusively punish unwanted behaviour of the genome instead of giving rewards or mixing rewards with punishments. As the ultimate goal of the algorithm is to deliver all passangers in a short average time, a punishment-only approach acts like a 'catch-all' sink. Instead of trying to find and giving rewards for all of the different kinds of positive behaviours, it is instead easier to punish any and all negative behaviours. For example, if a passanger wants to travel from floor five to one, that requires an absolute minimum of four floors worth of travel. The implemented fitness function would in that case punish based on the difference in distance needed and the distance that was actually traveled by the passanger. Another simple example would be to punish the genome for everyone that did not arrive at their destinations, rather than rewarding for those that did arrive.

4. Experimental part

This section describes the setup of experiments:

Each experiment consists of two JSON files; one representing the building which specifies where all people are located, and another containing the population with a set of initial genomes. These experiments are saved to files rather than being generated randomly each time to ensure the ability to run the same experiment multiple times for consistency.

One specific building which already had result data available was used to compare this algorithm's results against previously known results.

To thoroughly evaluate the algorithm it was decided to test it on buildings of different sizes (small, medium, large) and with corresponding population sizes. Given the number of experiments required to be run, buildings with 10, 50, and 100 floors containing populations of 10, 50, and 100 people respectively was selected. Additionally, the specific building mentioned earlier which has 21 floors and 10 people was included.

Buildings were generated with constraints ensuring that each floor could hold between 0 and half of the maximum population. While it was theoretically possible for all people to be on just two floors in practice the distribution was more balanced. For example, in the small 10-floor building with 10 people the distribution was as follows:

From		**To**
0	→	6
0	→	8
3	→	5
4	→	7
5	→	7
6	→	2
7	→	0
8	→	4
9	→	0
9	→	2

After generating and validating all buildings and populations, the experiments were run. Each experiment was conducted on a computer with an AMD Ryzen 7 7800X3D and took roughly ten seconds to a minute. In total 150 experiments were run.

All experiment used the following settings

- **Generations:** 1000
- **Population size:** 100
- **Crossover methods:** Heuristic single, Heuristic sequence, Swap crossover
- **Mutation methods:** Increase, Decrease, Swap, Swap and Increase, Swap and Decrease
- **Elitism rate:** 0.1
- **Crossover rate:** 1.0
- **Mutation rate:** 0.1, 0.6

Due to time constraints the only parameter that was adjusted during the experiments was the mutation rate to limit the number of results to analyze. To ensure the reliability of results each combination of building and population was tested five times allowing to identify potential anomalies. Experiments were conducted with two mutation rates, 10% and 60%, to observe differences in algorithm performance. The thought process behind the very high mutation rate was to semi-simulate the effects of a longer generation limit without significantly increasing runtime.

The experimental procedure is outlined as follows:

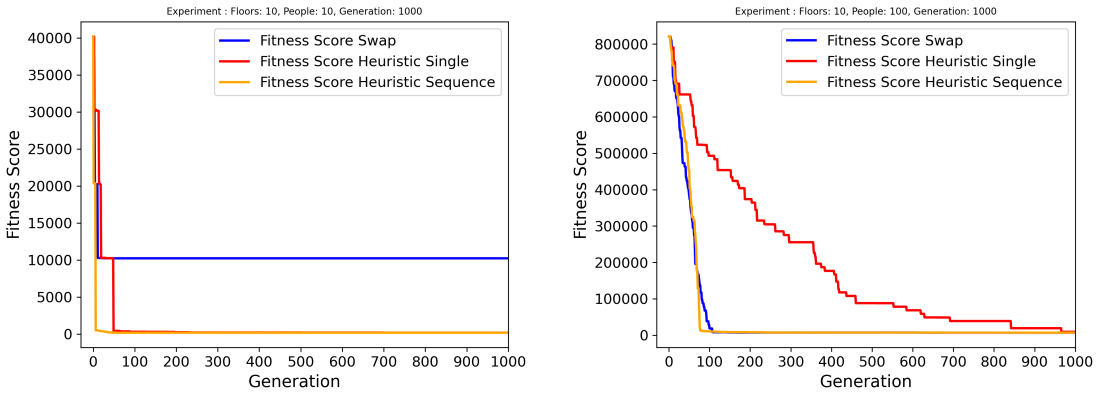
- Generate the building and population, or load a pre-existing experiment.
- Run the experiment generating a CSV file with results and a PNG with a graph.
- Re-run the same experiment with different parameters.
- Analyze the results to evaluate the algorithm's performance.

5. Results and Analysis

The following results are presented using graphs to illustrate key insights from the experiments conducted. The blue line and text represents the simplest crossover method, that swaps the last halves of two parents to create two children described in 3.1.2. The red represents the second crossover method, which selects genes from each parent based on their fitness scores, detailed in 3.1.2. The orange represents the third crossover method, that selects contiguous gene segments from each parent based on their fitness scores, outlined in 3.1.2.

5.1. Building Small

Figure 3 presents the results for 100 people in the building, as well as the worst performance from five runs for each crossover on the smallest building configuration with 10 people. Figure 3a displays the worst case in these runs where the blue crossover failed to serve one person and incurred a large penalty. To highlight that with more people in a building, the orange crossover kept getting good results with great consistency Figure 3b was used. The blue acted comparably, but it wasn't as consistent. The red one was highly unreliable, occasionally achieved positive results on the other hand, it frequently got stuck in local minimums for extended durations. Besides that it took more generations for all crossovers to reach an acceptable solution because of the increased number of people. The orange and blue crossover often reached a good solution quicker than the red one, which also tends to get worse the more people it has to serve.



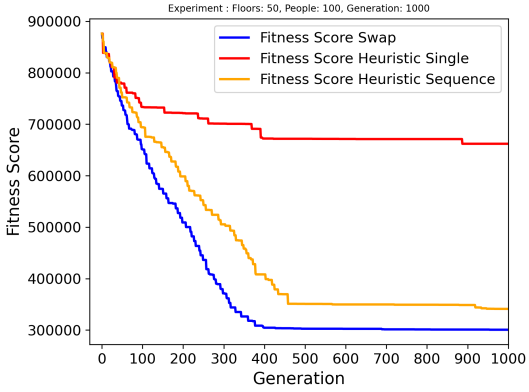
(a) Score/Arrived/Length: 10227/9/9, 182/10/11, 172/10/11.

(b) Score/Arrived/Length: 6727/100/45, 9113/100/61, 6880/100/52.

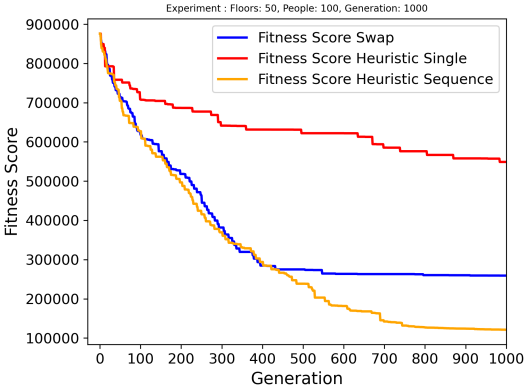
Figure 3: Comparing worst and best case.

5.2. Building Medium

As previously noted, the red crossover tends to perform worse the more people in the building instances and Figure 4 illustrates this once again. In Figure 4a the blue crossover outperformed the orange one, that got stuck in a local minimum for an extended period. The orange crossover was also slower than the blue one. Conversely, Figure 4b then shows a scenario where the blue one performed similarly to how it performed in 4a and the orange one reached an extraordinary low score of 121157 without getting stuck in a local minimum. This demonstrates that the orange crossover have both strong worst case and great best case. This further underlines that the orange crossover is the most reliable crossover of the three.



(a) Score/Arrived/Length: 300697/74/69, 661991/35/52, 341167/69/74.



(b) Score/Arrived/Length: 259130/79/71, 548857/48/59, 121157/94/124.

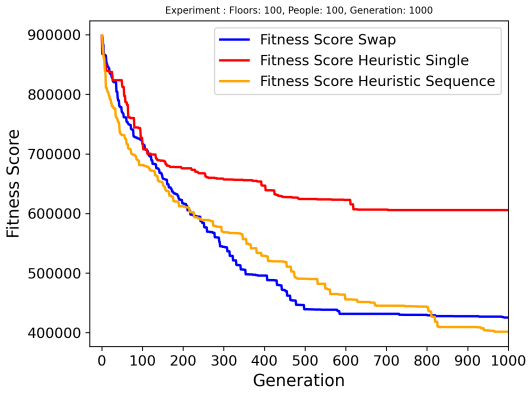
Figure 4: Comparing worst and best case.

5.3. Building Large

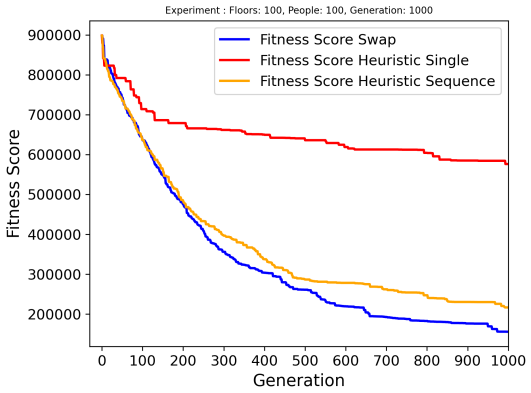
After running all the buildings with a 10% mutation rate, numerous experiments were executed with different rates. It was quickly determined that an increased mutation rate enabled the algorithm to find solutions that served all or almost all people much faster. Results in Figure 5 shows the best runs with mutation rates of 10% and 60%.

With a low mutation rate as in Figure 5a, the best solutions achieved fitness scores around 500 000, whereas a higher rate tested in Figure 5b yielded scores around 200 000. In the best run with a 10% mutation rate, 66 people were served and the best run with a 60% mutation rate served all 100 people. Not only that, it is clear that the algorithm reached acceptable solutions in a greater pace than before. Reason behind this is that to be able to serve as many people as possible the genomes have to mutate to become longer than number of floors. The higher mutation rate allowed the algorithm to reach this objective and escape local minimums faster. Further experimentation with mutation rates could potentially yield even better outcomes.

Similar results were obtained by increasing the number of generations and population size. The downside with this is that it requires more computational power and takes longer time to run. Therefore, a greater mutation rate was beneficial for fast feedback and limited hardware resources.



(a) Score/Arrived/Length: 425352/67/80, 605508/44/102, 401562/66/75.



(b) Score/Arrived/Length: 155619/100/151, 576322/48/117, 216508/94/139.

Figure 5: Comparing the best case for 10% and 60% mutation rate.

5.4. Building Special

This special building used in experiments replicates the setup from a prior investigation on optimization techniques for the elevator dispatching problem [4]. The people in this building has identical starting floors and destinations as the referenced study. Parameters for elevator population and number of generations are also aligned with values used for experimentation in the paper. This was done to ensure a fair comparison between the research. A minor difference is that the fitness function used in this study also gives an extra penalty for each extra floor a person travels in the elevator, as detailed in 3.3. Furthermore, the mutation used in this study also allows the genome length to increase and decrease.

The genetic algorithm in [4] utilized Davis-order crossover, swap mutation, elitism, and a fitness function that measured the average time for all passengers to reach desired floor. The average result reported from running described algorithm, based on 5 runs was 279.1 seconds per traveler. In comparison, the average result in this study for 5 runs, was 305.1 seconds. Thus, this algorithm was approximately 26 seconds ($\approx 8.5\%$) slower than the result in the paper. This difference could be explained by the extra penalty given in the fitness function. Additionally, it is possibly a coincidence, but one factor might have been that Davis-order crossover divides the chromosome into more segments while this crossover only splits it into two larger segments. Why this might affect the result is because when splitting the parent chromosome into two larger segments, parts of them will probably not contribute to a better result, and they will have a larger risk of remaining throughout more generations.

6. Conclusions

This paper mainly looked at different crossover functions impact on genetic algorithm ability to find solutions for various buildings configurations.

It started off with a fitness function that gave 10 points for each person served, which in the end promoted the algorithm to serve all people in the building. The problem with this was that it tended to give long genomes because it was more beneficial to serve all people than to serve them quickly. It was then changed to a fitness function that gave huge penalties for all people that weren't served by the elevator. It also increased the score by one for each additional unnecessary floor a person traveled in the elevator, worsening the overall result since a lower score is better. This also made it possible to analyze the average waiting time and this was also the benchmark that most papers used for results in this kind of problems.

How different mutation functions would affect the results were not investigated in this paper. The implementation used 5 different ones that are randomly selected when a mutation occurred. The mutation functions used were swapping a random element in a genome, increase the length of a genome, decrease the length of a genome and the combinations of swapping and increasing or decreasing the length of a genome. These are all thoroughly explained in (3.1.3).

An approach that would have been interesting to test is to implement elements of heuristic operators into Davis-order crossover. This would allow for more segments being made but also have the more dominant parent transfer more of its genes into the next generations.

Because the algorithm is not forced to find a solution where all people reach their destination floor, this implementation includes a time penalty discussed in (3.3), this is to incentivize the algorithm to serve all people to their destination floor. Further research on the impact of this time penalty in relation to the number of people in the building and their configurations would be interesting.

A finding from this paper is that a higher mutation rate seems to be better. To determine this further research in this area is needed. Is a higher mutation rate better? If so, what is the optimal range?

In addition, this report does not focus on the selection algorithm for the problem. Further research on which selection algorithm will perform best for the problem at hand would be of interest. Also, it could be interesting to investigate how other mutation functions could impact the results.

A. Appendices

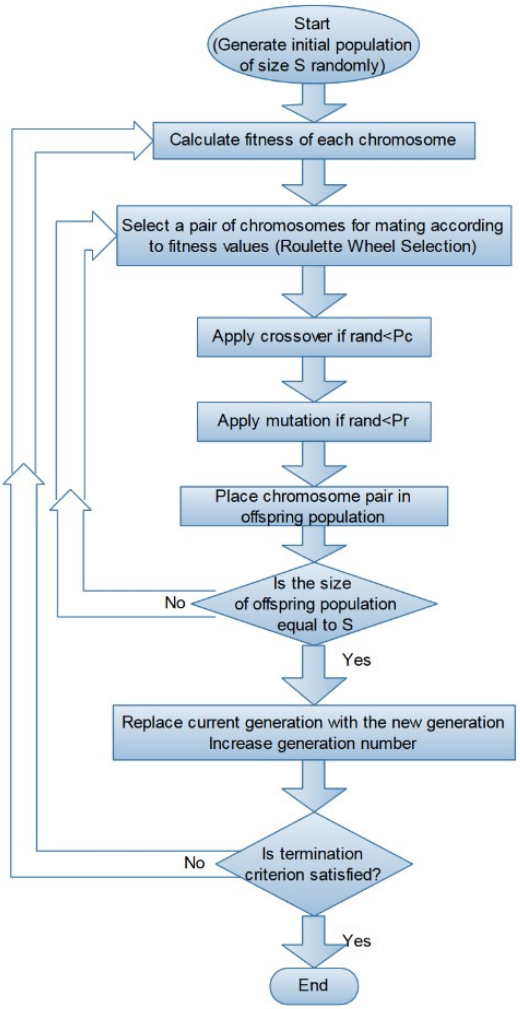


Figure 6: Flowchart from [3]

References

[1] S. Anily and R. Hassin, "The swapping problem," *Networks*, vol. 22, no. 4, pp. 419–433, 1992.

[2] E. O. Tartan and C. Ciftlikli, "A genetic algorithm based elevator dispatching method for waiting time optimization," *IFAC-PapersOnLine*, vol. 49, no. 3, pp. 424–429, 2016.

[3] E. O. Tartan and C. Ciftlikli, "A genetic algorithm based elevator dispatching method for waiting time optimization," *IFAC-PapersOnLine*, vol. 49, no. 3, p. 425, 2016.

[4] W. Gharieb, "Optimal elevator group control using genetic algorithms," *Computer and Systems Engineering Dept., Faculty of Engineering Ain Shams University*, 2005.

[5] S. Ahmed, M. Shekha, S. Skran, and A. Bassyouny, "Investigation of optimization techniques on the elevator dispatching problem," *arXiv preprint arXiv:2202.13092*, 2022.

[6] S. M. Lim, A. B. M. Sultan, M. N. Sulaiman, A. Mustapha, and K. Y. Leong, "Crossover and mutation operators of genetic algorithms," *International journal of machine learning and computing*, vol. 7, no. 1, pp. 9–12, 2017.