Group 24 Assignment Report

An Evolutionary Algorithm for an elevator to deliver as many people as possible to their desired floor

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

October 1, 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 the least amount average waiting time. In large buildings with a lot of floors, efficient elevators are crucial for managing traffic and must serve every one 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/or travel time is the core focus in the report. Different strategies and hyperparameter 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

2.2 Similar published academic work

[1], "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."

Mapping the elevator problem to the swapping problem.

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

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

2.1 Why this evolutionary approach

3 Algorithm

3.1 Evolutionary approach

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

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

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 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 our algorithm's results against previously known results.

To thoroughly evaluate the algorithm we 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 we selected buildings with 10, 50, and 100 floors containing populations of 10, 50, and 100 people respectively. 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 looked 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 we proceeded to run the experiments. Each experiment was conducted on a computer with an AMD Ryzen 7 7800X3D and took roughly ten seconds to a minute. In totally 150 experiments were run.

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 us 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

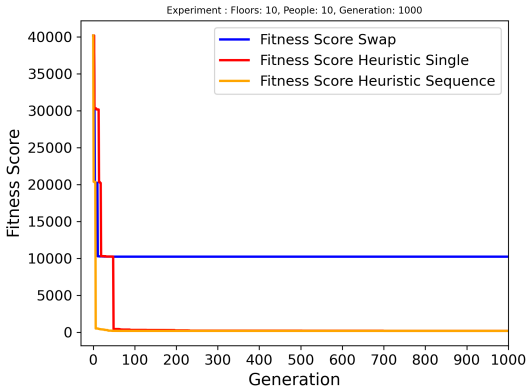
The following results are displayed using graphs to highlight some of the insights we have taken from the experiments we ran. Blue line and text represents our simplest crossover that swaps the last halves of two parents to create two children described in(3. X1). The red represents the second crossover that selects genes from each parent based on their fitness scores, described in (3. X2). The orange represents the third crossover that selects contiguous segments of genes from each parent based on their fitness scores, described in (3. X3).

We started of 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. We then changed fitness function to one that gave huge penalties for all people that wasn't served by the elevator when the route was finished. It also increased the score with one for each person not arriving to its destination for each floor traveled. This means that the lower score the better. It also made it possible for us to analyze the average waiting time and this was also a benchmark that most papers we found used for this kind of problem.

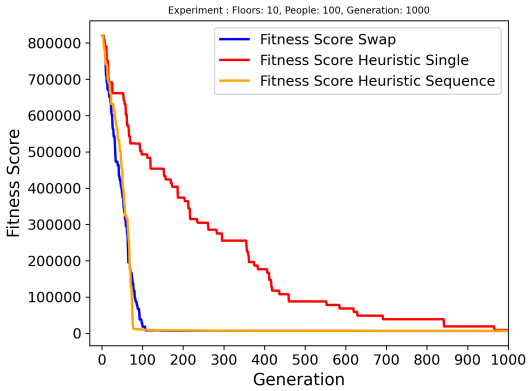
We didn't investigate how different mutation functions would affect the results. We used 5 different ones that was randomly selected when mutation occurred. The mutation functions we used was swapping a random element in a genome, increase the length of a genome, decrease the length of a genome and the combinations of swapping and increasing or decreasing the length of a genome. These are all thoroughly explained in (3. M).

5.1 Building 1

Figure 1 shows results with 100 people in the building and the worst performance from 5 runs for each crossover on the smallest building with 10 people in it. Figure 1a displays the worst case in these runs where the blue crossover failed to serve one person and then received a huge penalty. We used figure 1b to highlight that with more people inside the orange crossover kept getting good results with great consistency. The blue one also performed as good as the orange one, but it wasn't as consistent. The red one was extremely unreliable it performed well sometimes, but it also got stuck in local minimums for long time periods. 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, and it 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.

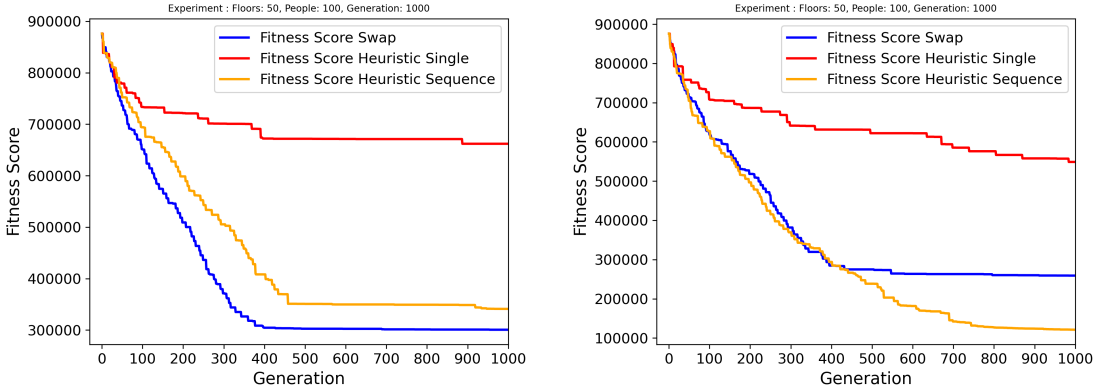
Figure 1: Building 1 comparing worst and best case.

5.2 Building 2

Samuel had some papers he wanted to compare these results with.

5.3 Building 3

As mentioned earlier the red crossover tends to perform worse the more people in the building instances and Figure 2 highlights this phenom again. In figure 2a the blue crossover actually outperform the orange one, that got stuck in a local minimum for a long time. The orange one also was slower than the blue one. In addition, figure 2b then shows a case where the blue one performed similar to how it performed in 2a and the orange one reached an extraordinary low score of 121157 without getting stuck in a local minimum. With other words this pinpoints that the orange crossover have a good worst case and great best case which makes it the most reliable crossover of the three.



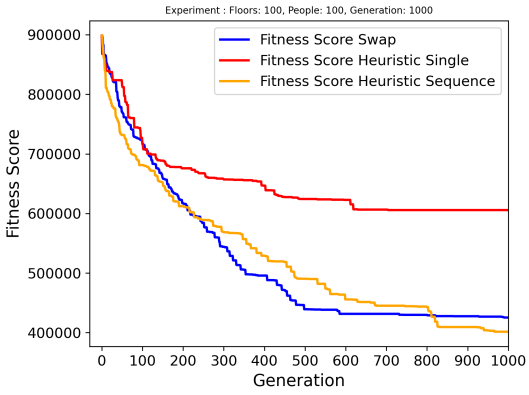
(a) Score/Arrived/Length: 300697/74/69, 661991/35/52, 341167/69/74.

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

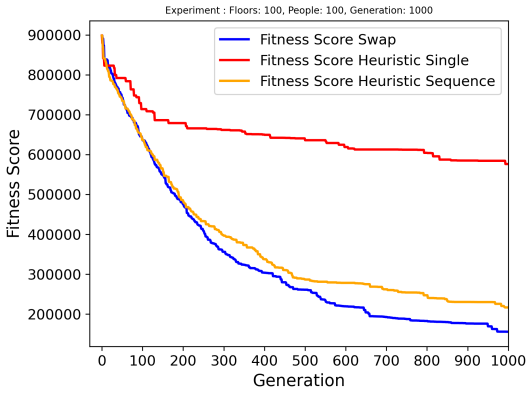
Figure 2: Building 3 comparing worst and best case.

5.4 Building 4

After running all the buildings with mutation rate of 10% we started testing out different values and quickly found out that our algorithm found a solution that serves all or almost all people much faster with a greater mutation rate. We display result from runs with a mutation rate of 60%. 3 shows how much faster our algorithm reach acceptable solutions than before. When mutation rate was low the best solutions gave us fitness scores around 500 000 but with a high rate we were able to get scores around 200 000. The best run served 66 people when mutation was 10%, the best run with mutation 60% served all 100 people. Reason behind this is that to be able to serve as many people as possible the genes have to mutate to become longer than number of floors. With a higher mutation rate this happens in much quicker pace and therefore the algorithm is able to leave local minimums faster. Therefore, a greater mutation rate is beneficial and less computational power is needed compared to increasing the population size or the number of generations. We tested increasing number of generations and received similar results as with a greater mutation rate. If we had more time to experiment with different mutation rates we could probably find an even better rate.



(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 3: Building 4 comparing different mutation rates.

6 Conclusions

The paper mainly looked at different crossover functions impact on our genetic algorithm ability to find solutions for various buildings configurations. For the statistics the conclusions drawn are:

Building	Crossover function
1	Vilken som var bäst
2	Vilken som var bäst
3	Vilken som var bäst

Because the algorithm is not forced to find a solution where all people reach their destination floor, our implementations include a time penalty discussed in (3.x), this 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 and configurations.

A finding from the 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.

References

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