Train scheduling using genetic algorithm and simulated annealing

Joakim Olsen

October 17, 2018

1 Introduction

In this report, we will discuss the use of a genetic algorithm, and simulated annealing, to search for an effective train schedule, given certain conditions. For the problem investigated in the report, these conditions are given as follows: There are six train stations, with a certain distance between them. There are two types of trains, which travel at different velocities, thus needing different amounts of time to travel between the stations. The first train type leaves every 25 minutes, while the second train type leaves every 30 minutes. From one of the end stations, train one starts at $T = 0$ (T is minutes), and the other train starts at $T = 10$. From the other train station the first train leaves at $T = 5$, and the second train leaves at $T = 13$. These times, as well as the intervals and travel times of the trains, can of course be changed. There is a single train track between the stations. Thus, the trains can only cross on the stations, and they have to wait for each other. The challenge is therefore to find the most effective way of waiting, to minimize the waiting time. Figure 1 illustrates a possible schedule. This problem is solved using rather straight forward versions of the genetic algorithm and simulated annealing. The methods themselves are not discussed much further, since they are considered known for the audience.

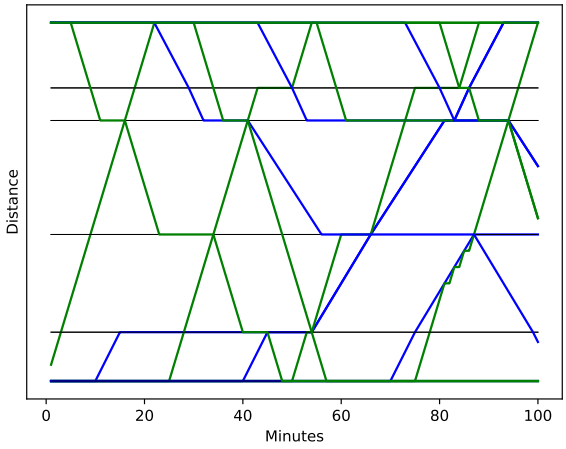

Figure 1: The figure shows the movement of the trains as a function of time. The green lines are train type one, and the blue lines are train type two. The Black horizontal lines are the stations, where the trains can cross and pass each other.

2 Design

The train schedule needs to be represented in a way so that it is possible for the genetic algorithm and simulated annealing to work with the representation. The problem of directly representing the train times is that crossing two individuals in the genetic algorithm might be difficult, and to make it a valid individual, many changes would have to be made. Figure 2 visualizes this.

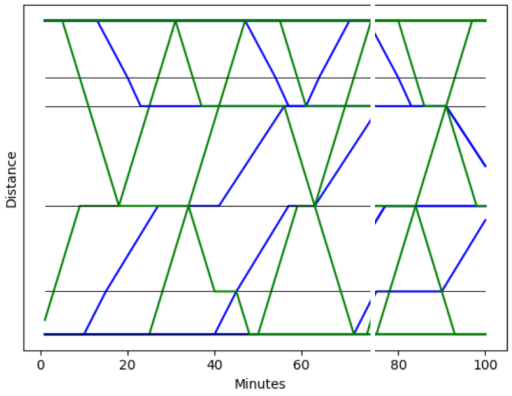

Figure 2: Two different schedules combined. Some of the trains should suddenly be somewhere else, and to change this error might influence the other trains, resulting in a messy clean up job.

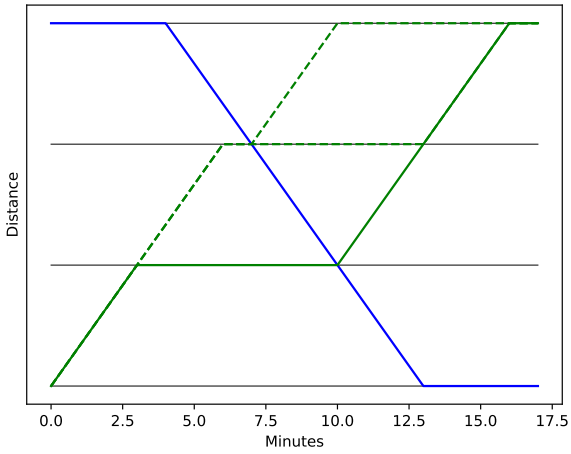

Figure 3: Say the full green line represents one of the best solutions. If the train could have gone to the next station without conflicts (dotted line), it could wait there, giving the same result. Thus, waiting when it could have gone without conflicts is never better, but can be equal.

The choice of design is therefore made with easy crossings in mind. First, a kind of a greedy choice is made, namely that if a train can move an entire stretch without having any conflicts, it always will. This does indeed limit the set of possible solutions from the algorithms, but the following argument, which is also illustrated in figure 3, shows that at least one of the best solutions will be in the set of possible solutions. Say there is a best solution in which the train waits for another train at station $S = 1$, but it could have gone to $S = 2$ before having the conflict. Then there is another at least equally good solution in which the train runs straight to $S = 2$, and waits there until it would have arrived there anyways.

Using this assumption, the trains will just go until they have a conflict, which means that we only need a design to resolve conflicts. This is done with a simple $N_t \times N_r$ matrix, where N_t is the number of trains one direction, and N_r is the number of trains in the other direction. The matrix contains zeros and ones, where zero gives one direction right of way, while one gives the other direction right of way. There is also a similar matrix of size $N_t \times N_t$ which similarly holds information on whether a slow train has to wait for a fast train to pass, or if the fast train must follow the speed of the slow train. From these matrices, and the greedy choice of running if there are no conflicts, it is possible to calculate and visualize the schedule of the trains. The code for calculating the visualization, as well as calculating fitness, is found in the trains.py file.

3 Genetic algorithm

The genetic algorithm is implemented in the following way: Fitness is calculated based on the waiting time of the trains put together. In practice, this is done by running the trains, and when a conflict arises, the

design matrices are used to solve them. This results in trains having to wait, and this waiting time is put together, yielding a total waiting time. This is used as fitness, and the goal is then to minimize the fitness, to reduce the combined waiting time for all the trains. The selection has been implemented using both proportional selection, and selection by rank. The final choice will be discussed afterwards. The crossing is done uniformly, with the same probability of getting an element from both parents. Mutation is done by changing a one to zero or zero to one, with a certain probability. Replacement is implemented with different proportions of replacement of the parents, also including replacing all (generational). The stop condition is timer-based, making it easy for the user to see the behavior of the algorithm for as long time as wished.

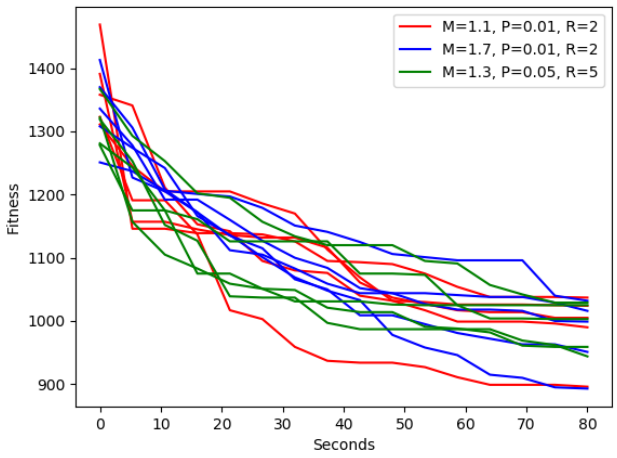

Figure 4: Several runs for 80 seconds for different parameters.

3.1 Parameterization

There are many parameters to determine: Probability of mutation P, max-value in selection by rank M, replacement part R and size of population n. Initially, to get an overview of performance, the convergence plots for five minutes are made with all combinations of the values $P = [0.001, 0.005, 0.1, 0.5]$, $M = [1.05, 1.1, 1.3, 1.7]$, $R = [0.2, 0.5, 0.8, 1]$ with $n = 10$, in addition to using the same values for proportional selection. The plots will not be included in this report, because there are many (20 from this round), and due to randomness they are not that informative, but they give a good indication on where to start. Selection by rank seems to works better than proportional selection. Three of the combinations of parameters which seemed to work well in the initial convergence plots have been further investigated. The result is given in figure 4.

Here, each have had five runs to investigate the variance in the convergence, which we see is quite big. None of the solutions seems to be much better than the others. The solution of $M = 1.1$, $P = 0.01$, $R = 0.2$ and $n = 10$ is chosen as a starting point for further exploration of each parameter. Figure 5, 6, 7 and 8 shows the effects of changing each parameter.

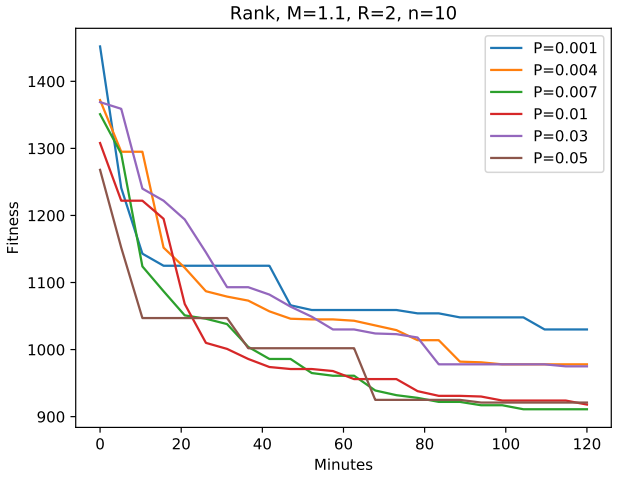

Figure 5: 120 seconds run for different values of probability. Averaged over 5 times.

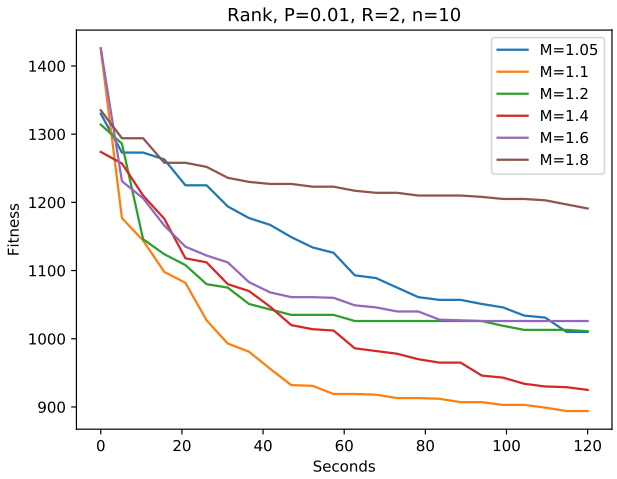

Figure 6: 120 seconds run for different values of max value. Averaged over 5 times.

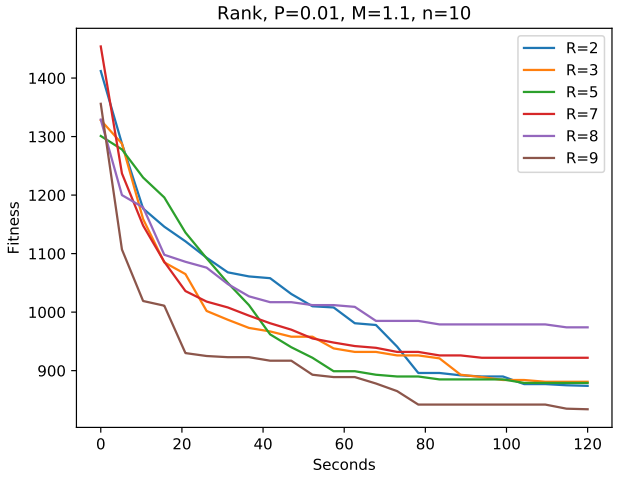

Figure 7: 120 seconds run for different values of replacement amount. Averaged over 5 times.

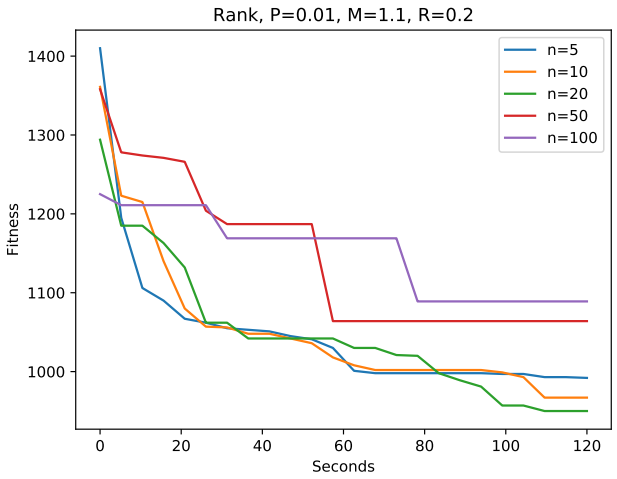

Figure 8: 120 seconds run for different values of individuals. Averaged over 5 times.

The plots shows that most variations have little effect, there is no consistency in the increase/decrease of the parameters, perhaps except for using a high number of individuals, which performs worse. Randomness seems to dominate also these plots. We therefore finally choose parameters that should not converge too fast, and conclude that the parameters $M = 1.1$, $P = 0.01$, $R = 0.2$ and $n = 10$ give as good results as we can expect to obtain, and therefore define this as our final choice of parameters. Then, our method is defined as follows: The size of the population at all times is $n = 10$. Selection is done from probabilities by the formulas:

$$h(x) = 1.1 - 0.2 \cdot \frac{r(x) - 1}{n - 1}$$

$$P(x) = \frac{h(x)}{n},$$

with $r(x)$ equal to the rank of the individual. Crossing is done uniformly, and in the mutation, bits are changed with a probability of $P = 0.01$. Finally, the $R = 2$ poorest individuals in the population are replaced by children.

4 Simulated annealing

Simulated annealing is implemented in the following way: Fitness is calculated the same way as in the genetic algorithm. A neighbour is defined as a set of matrices with only one bit different from each other. Random neighbours are generated, and the fitness is compared to the previous one. If the solution is better or equal, it is accepted, while if it is poorer, it is only accepted with probability $e^{\Delta f/T}$, where Δf is given by oldfitness−newfitness (negative if the new solution is worse), and T decreases following a formula. Primarily, both decreasing functions

$$\alpha_1(T,k) = kT_i$$

and

$$\alpha_2(T,k) = \frac{T_i}{1+kT_i}$$

are implemented. Stop condition is also here timer based.

4.1 Parameterization

Also here are some parameters to determine: Starting temperature T, decreasing value k, and choice of decreasing function. Initially, to get an overview of performance, convergence plots for five minutes are made with all combinations of the values $T = [100, 1000, 10000]$ and $k = [0.7, 0.8, 0.9, 0.99]$ for α_1 and $k = [0.001, 0.01, 0.1]$ for α_2. The plots will not be included for the same reason as with genetic algorithm, but still they give a good indication on where to start. With T quite low (100), the solutions seems to converge fast, but not towards very good solutions, which is expected with a low temperature. Apart from that, the solutions seems quite equal. The solution of $T = 5000$, $k = 0.9$, using α_1 is chosen as a starting point for further exploration of each parameter. Figure 9 and 10 shows the effects of changing each parameter.

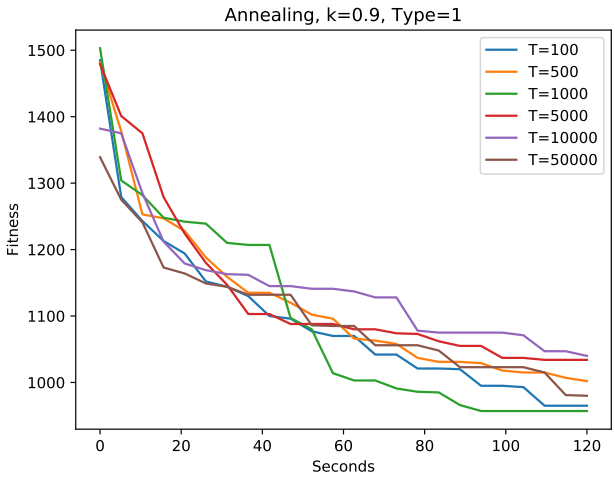

Figure 9: 120 seconds run for different values of T. Averaged over 5 times.

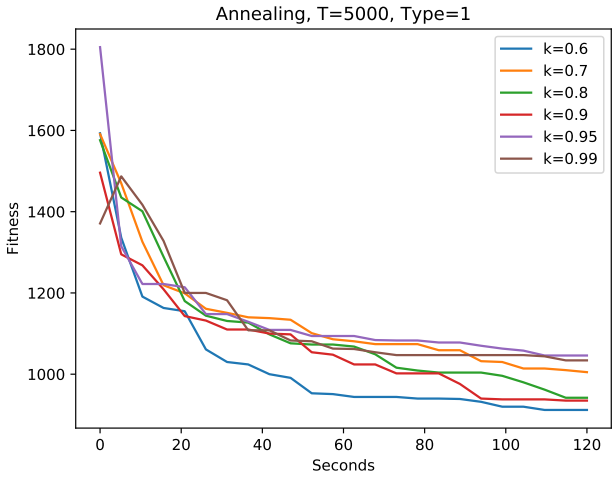

Figure 10: 120 seconds run for different values of k. Averaged over 5 times.

Again, there is no consistency in the increase/decrease of parameters, and the choice is therefore based on not prematurely converging towards local solutions. We choose $T = 10000$ and $k = 0.9$ for our final method, using α_1.

5 Performance

To look at the performance of the final algorithm, it is nice to look at problems with known solutions. It is not so easy to know the best solution to this problem, but by constructing the problem in the right way, it is not so difficult to find a problem with a solution that very likely is the best. One of these are shown in figure 11.

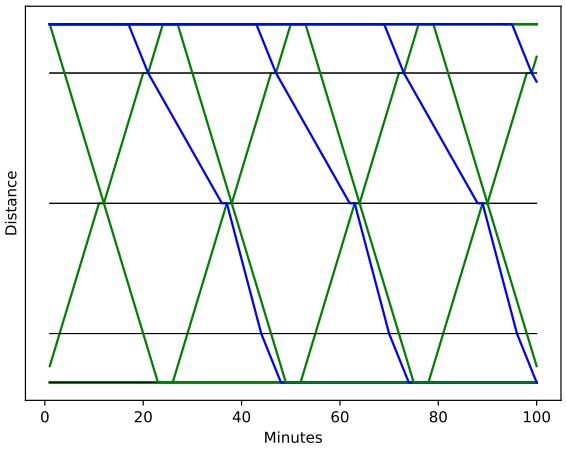

Figure 11: A constructed problem with a solution which is likely the best one.

When the two methods are run on this problem, we get the convergence shown in figure 12.

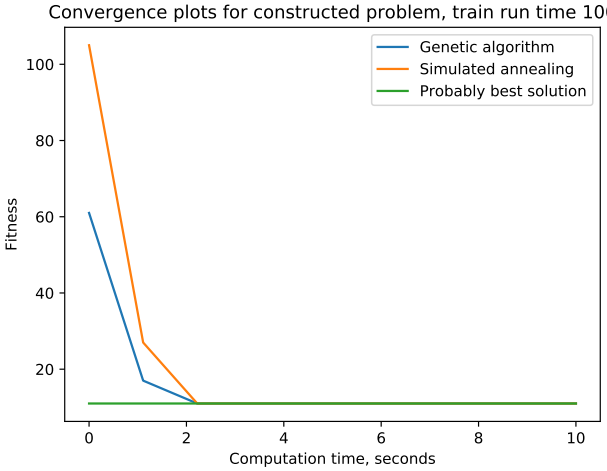

Figure 12: Convergence plot to the constructed problem with train run time 100 minutes.

We see on the plot that both algorithms converge very fast towards the assumed best solution when the problem is not so big. Since the constructed problem is periodic, it can be run for a longer time to see how the algorithms work on a bigger problem. The result is shown in figure 13.

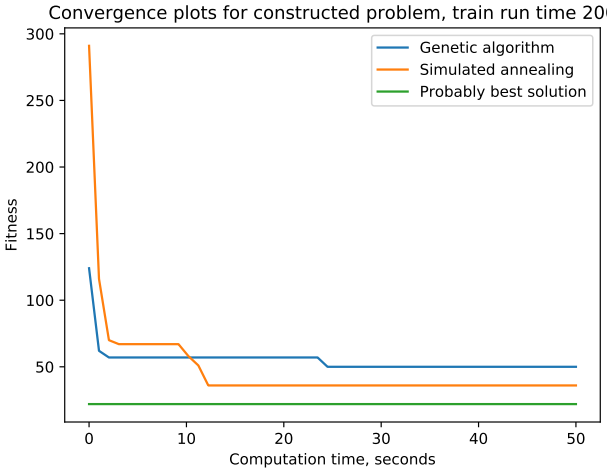

Figure 13: Convergence plot to the constructed problem with train run time 200 minutes.

We see from the convergence plots that the solution is not able to reach the assumed best solutions

for bigger problems, though it is quite close. This indicates that the algorithms can improve solutions, but they are unlikely to encounter best solutions for bigger problems. Figure 14 shows the solution for simulated annealing.

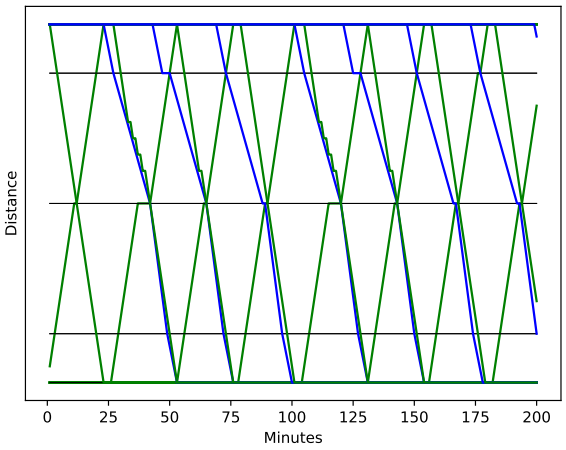

Figure 14: Solution of simulated annealing for constructed problem and train run time of 200 minutes.

An interesting observation here is that some of the changes that could improve this solution are done by changing a single bit. Thus, by the definition of a neighbour, this solution could have been improved by a local search. This observation will be discussed more in the conclusion. Finally we take a look at the bigger main problem, to see how the final convergence is. Plot is shown in figure 15.

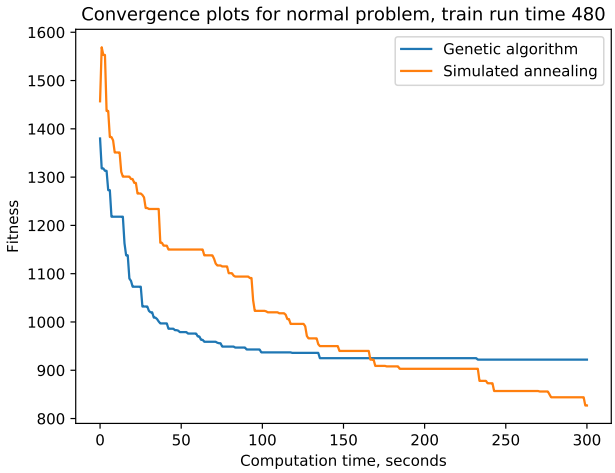

Figure 15: Convergence plot on main problem.

We see from the plot that the genetic algorithm seems to prematurely converge towards a local solution. Simulated annealing seems to still be improving, thus better results might be obtained with longer computation time. The final train schedule of the convergence plot of simulated annealing is shown in figure 16.

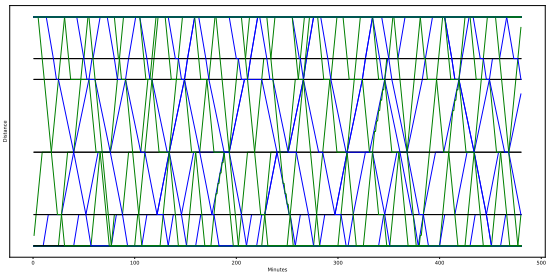

Figure 16: Solution of main problem from simulated annealing.

6 Conclusion

Both algorithms can improve random solutions a lot. This is on top of the greedy choice which makes random solutions fairly adequate. Thus, the algorithms can find good solutions, but is very unlikely to find

best solutions. The performance of the genetic algorithm and simulated annealing seems to be quite similar, but some plots indicate that simulated annealing might be better off with a longer run time. This fits well with observations of solutions which could have been improved by local searches, since simulated annealing might have a higher probability of encountering these solutions than the genetic algorithm. The conclusion is therefore also that implementing other methods utilizing some form of local search (like GRASP) might work well with this design. Another observation with the design is that it has matrices which store a lot of unnecessary information, since most trains will never have a conflict. This also in the end makes it more difficult for both implemented algorithms to find random improvements. The matrices could for example be exchanged with a list which handled conflicts chronologically, thus removing the storing of unnecessary information.