Welfare Navigation Using Genetic Algorithm

David Erukhimovich and Yoel Zeldes
Hebrew University of Jerusalem
AI course final project

Abstract – Using standard navigation algorithms and applications (such as the Waze cellular app) yield good results in many cases. However, a problem involving traffic jams arises when using applications such as Waze: given a traffic jam, Waze does not know what’s the best route for you. But there are traffic jams that can be caused by Waze, such that taking a different navigation approach would have avoided them. Our approach tries to give a good navigation result which avoids creating traffic jams as much as possible.

Index Terms – navigation, genetic algorithm, graph search, A*.

I. INTRODUCTION

Many navigation algorithms and applications exist – some are good, and some are better. One remarkable such application is Waze: it takes the traffic and traffic jams into consideration while calculating the best navigation from point A to point B, and avoids big traffic jams that make the travel time unnecessary big.

A. Problem Description

Taking the traffic and traffic jams into consideration is a main advantage of Waze over other navigation applications. Nevertheless, Waze does incur a weak point (like other navigation applications): it has a narrow view point. When calculating a car’s best route, it only cares for that car’s welfare. If there are hundreds of people using Waze on the same time, Waze will take a greedy approach, and give each user a best route which doesn’t take into account the routes given to the other users. Such approach can cause many traffic jams.

Fig. 1 illustrates such a situation: in this situation there’re two points – A and B – and only two roads connecting them – R1 and R2 – where R1 is shorter than R2. If there are many cars at A trying to get at the same time to B, assuming both roads are empty of cars, Waze will guide all of the cars to go through the shorter road – R1. Assuming R1 is a problematic road and can’t contain all of the cars, a traffic jam will be created (this traffic jam wasn’t predicted by Waze). This would be too late for the cars to go through R2, and they’ll be late for work.

A more suitable navigation would be one that regulates the cars between the two roads such that all of them will get a reasonable travel time, and none will get stuck in jams.

B. Our Approach

We use a novel navigation algorithm which takes into account the welfare of all of the cars in the system.

The core of our algorithm uses genetic algorithm, with the support of graph search algorithm and a simulator which can simulate traffic and arrival times of the cars in the system. In section 2 we describe the model used to describe the problem. In section 3 we describe the various components of our algorithm. In section 4 we show and discuss some results. In section 5 we discuss future work.

II. MODEL REPRESENTATION

We use two layers of data models in our algorithm – one which describes the various locations and the roads connecting them (denoted as map), and one describing the cars (denoted as world). The reasons for such separation (instead of using a single model for both the map and the world) are:

1) Reusing: By separating the map from the world, one can run our algorithm on the same map with different sets of cars with ease (which is like navigating in the real world – the cars can change all the time). On the other hand, the world description can be fixed, and the map can be changing, reflecting the case where there’re roads repairs / new roads, and the cars may take different routes.

2) Simulation: As described later, our algorithm includes a simulator which simulates the cars driving the routes to their destinations. In order to make our code modular and easy to understand, the map part doesn’t change during the simulation, and the world part does (because it contains the car’s locations, which change during their driving).

![Fig. 1 A basic navigation problem of getting from point A to point B.](image)

In the left figure: the navigation calculated by Waze.

In the right figure: a better navigation which reduces the load on the roads.
A. Map Model
The map consists of a graph. Each node in the graph represents a location, and each edge represents a road. Each road contains a weight. This weight, in our implementation, is simply the road’s length. However, one could take the advantage of Waze, and take the traffic loads to be the roads’ weights.
Each road also contains a capacity. This capacity is the maximal number of cars which can simultaneously be on the road without causing a traffic jam. When there’s no traffic jam, all of the cars on the road drive on the maximal speed. When there’s a traffic jam, all of the cars’ speeds decrease exponentially on the number of cars exceeding the road’s capacity. The base of the exponent dictates how bad a traffic jam is: a big base means the decrease of speed is very meaningful, and our algorithm will try to avoid exceeding the capacity.

B. World Model
The world consists of the cars in the system, their source locations, and their destinations. Because the world, as will be described later, is used during the simulation, it also has to store the cars’ locations at each point in time during the simulation.

III. THE ALGORITHM’S COMPONENTS
The main idea behind our navigation algorithm is using genetic algorithm: each car has a vocabulary of possible routes from the car’s source location to its destination location. A routing decision which navigates each of the cars through one of its possible routes is defined to be an individual. More precisely, an individual is a vector with an entry for each of the cars in the system. The i’th entry contains the route chosen for the i’th car (if that entry contains the number n, then the n’th route of the car’s vocabulary is chosen).
A population is a set of individuals. Our algorithm starts with a random initial population. At each step, it creates the next generation of individuals. Each individual in the new generation is created by taking two random individuals (with probability calculated by a fitness function) from the previous generation, making a crossover between them, and a mutation with some probability. This process creates a new vector of routings.
After a sufficient amount of iterations, our algorithm stops, yielding the best routing it found so far (in accordance with the fitness function).
The fitness function gives each individual a score reflecting how good the routing was for all of the cars. This is done by means of a simulator, which takes the cars’ routings dictated by the individual, and simulates their driving and the traffic jams created by them.

A. The Routes Vocabulary
As defined before, an individual is a vector where each entry represents a car’s route which takes that car from its source location to its destination location. We have to define for each car its legitimate routes which take it from the source to the destination. Theoretically, we could have taken into consideration an infinite number of routes (if there’re circles in the map for instance). Even if we took into consideration only a finite number of routes (for instance – all the “meaningful” routes), our algorithm’s performance will be bad both in the aspect of time to finish, and in the quality of the result routings: every vector’s entry could have many values, so there could be a vast number of different individuals. The nature of genetic algorithms doesn’t guaranty a best solution. It could find a near best solution, but because the search space is huge, it’ll take a lot of iterations to find one.

Using a small search space (which includes some good results) will be better. This is why we limit the routes vocabulary size, and give a heuristic for the routes that should be included in the vocabulary, such that the search space will include good individuals.

While designing a good heuristic for which routes to include in the vocabulary, we should keep in mind that although we choose a car’s routes independently from other cars, the result search space must include good combinations of routings so there won’t be any traffic jams.

We chose the following heuristic: given a car, we use A* search to find the shortest route to its destination (this best route is actually what Waze will give us, if we took the roads’ weights to be the traffic loads). We initialize the car’s vocabulary with this route. If this was the only route in the vocabulary, our algorithm result will be identical to Waze’s.

Next, we take one of the route’s roads, and we temporarily give it a weight of infinity, and run A* again. This yields us a different route (unless, of course, there’s no route to destination that bypasses that road). We do this process again for each of the best route’s roads, and add the result routes to the vocabulary.

The reasoning behind this approach is to think in advance what route to take if there’ll be a traffic jam in one of the best route’s roads.

The set of routes we calculated until now still isn’t enough: in some maps, a traffic jam in one of the best route’s roads will always result in a specific route – no matter where the traffic jam is. The result vocabulary will be too small, and the search space might not include good individuals. This is why we continue and enlarge the vocabulary in the following way: every time we add a road to the vocabulary, we remember where there’re virtual traffic jams (weight of infinity). We now choose a random route from the vocabulary, choose a random road within it, and add a virtual traffic jam to this road as well as the previous virtual traffic jam on the previous road/s. We calculate A* again, and add the result to the vocabulary. In the final vocabulary there could be routes with variable number of virtual traffic jams. This increases the probability of having many different routings in the vocabulary.
B. The Initial Population

The genetic algorithm starts with an initial population of individuals. Both its size and the individuals that participate in it influence the final result. The probability that a good individual will appear in one of the generations will be small if the initial population’s size is small (because there’ll be a small number of individuals throughout the algorithm’s execution). On the other hand, if the initial population’s size is too big, the time performance will be bad: the algorithm simulates the driving of the cars according to each of the individuals. We want to perform as little work as possible. If we want to make the calculation time fix, a big size of initial population will result in fewer generations that will be created, which impacts the result.

We use the following heuristic to decide which individuals to include in the initial population: in order to create an individual, we go over each of its entries independently, and randomly choose a route from the vocabulary with weighted probability based on the routes’ lengths. This process results with individuals chosen only by the quality of the entries independently. It would be better to choose them by looking at the quality of all of the individual’s routes combined together. This would require us to run the simulator for each of the possible individuals, which would be ridiculously costly.

C. Crossover

The crossover phase in our algorithm is straight-forward: given two parent individuals, a child is created by randomly selecting entries from its parents: for each entry, the child takes with probability 0.5 the corresponding route from its father, and with probability 0.5 – from his mother. The result is a valid individual (for every car there’s a route which takes it from its source to its destination).

D. Mutation

Each entry of the child is independently mutated with a small probability. If mutated, the corresponding entry is replaced with some other route from the vocabulary.

We implemented several kinds of mutators, each is different by the way it chooses an alternative route from the vocabulary:

1) Uniform mutator: as the name implies, this mutator chooses an alternative route uniformly.

2) Best Route mutator: simply chooses the best available route in the vocabulary. This is done by iterating over all of the possible routes, temporarily changing the car’s route accordingly, and running the simulation all over again. This mutator chooses the route which gave the best result (calculated using the fitness function).

3) Simulated annealing mutator: this mutator randomly chooses a route. It then runs the simulation all over again. If the result fitness is better than the original fitness, the changing occurs. If the result is worse, the changing occurs only with probability of \( e^{-\text{ratio}} \) where ratio is the ratio between the new fitness and the original fitness. Note: the reason we don’t use \( e^{\text{new} - \text{original}} \) as used in similar simulated annealing searches is because we don’t know the range of the results the fitness function might give us. So we want to normalize it, using ratio instead.

E. Extinction

At each step when we create a new generation, we add it to the current population, resulting in a population with individuals of different ages. We keep old generations alive for a few generations until they get too old. When they’re old enough – we remove them from the population. One exception is the best individual (the one which gets the highest value from the fitness function): whenever a generation passes, we let the best individual drink from the eternity fountain, which prevents him from aging. When a better individual is created, the previously-best individual resumes getting old.

We use this heuristic because we don’t want that by chance the best individual won’t participate in crossover, which will result with his lineage disappearing from the world. If he’s the best right now, there’s a chance he’ll be the best even later, so we want to keep him. We note that this doesn’t causes the algorithm to get into a local maxima, because we keep only one individual, while there’re still a significant number of other individuals in the population, which might lead us to a better maxima.

F. Iterations

We stop running the algorithm after a certain amount of iterations. The more iterations we run, the greater the probability that we find a good individual, but the time performance will decrease as well.

G. The Fitness Function

Given an individual, we need to give it a score. We chose to implement a simulator which simulates the cars’ routings that the individual dictates. During the simulation, traffic jams may appear, and the result arrival times will be affected. Finally, the fitness function takes as input the arrival times, and combines all of them into a single number.

We implemented two kinds of fitness function: one which returns the inverse of the longest arrival time, and one which returns the inverse of the mean of the arrival times.

The simulator is implemented using an event-driven approach: it starts with all of the cars being at their source locations. It then calculates which of the cars will complete its current road first. To do so, it calculates the speed at each of the roads (using the roads’ capacities as described earlier). After calculating this time, the simulator changes the clock to that time. During this time all of the cars are driving on the same road, and therefore – their speed doesn’t change. After the time passes, one of the cars (or more) gets into its next road, and all of the cars’ speeds may change. The simulator calculates each of the cars’ speed again, and repeats the process, until all of the cars reach their destinations.
H. GUI

We implemented a simple GUI which can control all of the algorithm’s parameters. The GUI is illustrated in Fig. 2. Once all of the parameters are decided, the algorithm starts running. When the algorithm finishes, the GUI runs a simple A* algorithm, which gives each car the best routing based on the roads’ weights alone (just like Waze). It then makes a comparison between the two, to show which was better. To conclude, an animation of the simulation is shown to illustrate the result.

Another feature of the GUI is the ability to load a script file. This file describes several runs of the algorithm with different parameters. Using this script capability enabled us to conclude the results in the following section.

IV. RESULTS

In this section we compare our algorithm performance against that of simple A* (the result of a naïve navigation application). We also try to understand how the various algorithm parameters affect both the fitness of the results and the time performance of the algorithm.

We do so by examining each parameter separately: for each parameter, we fix the value of all the other parameters, and then run the algorithm with changing values for the parameter under inspection. The algorithm is run on a fixed map which can demonstrate how well the algorithm performs. In this map there’re eight cars which try to get from point A to point B. This is shown in Fig. 2.

We then create a graph showing the fitness of the result: the highest the graph – the better the result (the higher the fitness function is). The fitness function used is the inverse of the average arrival time. In the same graph we show the fitness of the result returned by the naïve A*. We also show the running time of our algorithm. The left y-axis is the fitness value, and the right y-axis is the time it took to execute the algorithm (in seconds).

Because the nature of our algorithm is random, we run the same test several times and calculate the mean of the results. This way, our results are more robust and reliable. We discuss the graph results and try to summarize how the parameters affect the algorithm.

A. Number of Iterations

The simplest parameter of our algorithm is how many generations to create. Fig. 4 demonstrates how it affects the algorithm’s result and runtime.

We can see that the runtime is linear by the number of iterations – an obvious result, since the time to create each generation is constant (more or less).

The more interesting thing to notice is that the fitness of the result gets better and better, until we get to 7 iterations. In the map used in this test, 7 iterations are enough to get good results. A different number of iterations might be needed in more complicated maps.

B. Speed Penalty

As stated previously, when there’re more cars on a road than the road’s capacity, traffic jams arise. The speed on the road of all the cars drops exponentially on the number of cars exceeding the road’s capacity. The base of the exponent dictates how bad a traffic jam is: a big base means the decrease of speed is very meaningful, and our algorithm will try to avoid exceeding the capacity.

In this test, we try to find out how the base of the exponent affects the results.

We can see in Fig. 5 that the fitness of the result of A* also drops exponentially. This is trivial, since A* routings don’t change as a result of the change of speed on the roads: all of the cars are routed through the shortest path, and a traffic jam is created.

Our result also decreases, but not as fast as A*. It’s obvious that no matter what, the results must get worse (because decrease of speed doesn’t do any good to the results). The fact that our result’s fitness decrease slowly is good: it tells us that the algorithm successfully avoided big traffic jams (small traffic jams are unavoidable).

One nice thing to observe is when the base is 1. In this case, there’s actually no speed penalty at all, and the optimal solution is that of A*. The reason our algorithm’s result is worse is that it’s a random algorithm, so an optimal result isn’t guaranteed. The probability to find that result was higher if we used more iterations / bigger initial population size / etc.

![Image](image-url)

Fig. 2 The GUI of the genetic algorithm

![Image](image-url)

Fig. 3 The map used to conclude the results.
C. Initial Population Size

The first generation size also affects all of the following generation sizes. We can verify from Fig 6 that the bigger the initial population is, the better the results get. This is due to the fact that bigger populations have more probability to contain good (and maybe even optimal) results. In conjunction with the number of iterations performed in this test, we managed to get those good individuals.

We can also notice that the runtime increases linearly. This is because our algorithm needs to calculate for each individual its fitness, which is done by running the simulation.

D. Vocabulary Size Factor

This factor determines how many routes will be in a car’s vocabulary: after calculating a car’s first route using A* search, our algorithm counts how many roads are there in the route. Given this number is $n$, the final vocabulary size will be the vocabulary size factor times $n$.

The vocabulary is one of the most important things in the algorithm. Given a poor vocabulary, the search space will contain poor individuals. We can verify in Fig. 7 that the results get better as the vocabulary size factor grows.

There’re two reasons why the time performance gets worse: one is that the calculations of the vocabulary get more time consuming. The second is that running a simulation over a simple individual (for instance - all of the cars get the same route) is faster, since there’re fewer events to process in our event-driven simulator. Bigger vocabulary means complicated individuals, which are more time consuming to simulate.

E. Mutator

We compare the results of different mutators. They differ one from the other in the way they choose an alternative route (consult the previous section to see the different kinds of mutators). In Fig. 8 we can see that although the different mutators do
give different results, this is negligible compared to the runtime impact. We can see that the best route mutator gives the best results, but it has the biggest running time. Following him (both in quality and performance) is the simulated annealing mutator, and the last one is the uniform mutator.

**F. Mutation Probability**

We can see in Fig. 9 that if there’s no mutation at all, the results are the worst. Even a small probability of mutation is enough to make the results better. We can also see that probability of 0.2 is enough, and bigger probabilities don’t really make any difference. The reason the runtime is linear with the probability is that in this experiment we chose the best route mutator. This mutator runs the simulation many times whenever a mutation is made. The bigger the mutation probability is, the more times the simulator is run.

**G. Conclusion**

The results shown here give us a good rule of thumb on the impact of each parameter. However, their impact can depend on the actual map and world used in the experiment. One can learn good configurations for various real-life scenarios (e.g. – a good configuration of parameters to use in maps of the size of Israel), and use them as needed.

We should note that no matter what configuration of parameters we used, we almost always got significantly better results than the naïve A* algorithm.

**V. Future Work**

Although we managed to get good results in our implementation, there’re still things to be done which can improve our algorithm:

1) **Performance:** The runtime of our algorithm is a little slow, especially on big maps such as Israel. A lot of the running time is spent on the simulator: although each simulation takes a little time to complete, there are a lot of simulations done throughout the algorithm. Optimizing this part will considerably increase the performance. Our algorithm is written in Python. Although it’s great for getting good results with ease and speed of development, a good starting point for optimization would be to rewrite parts of it in C or C++.

2) **Hierarchal maps:** Another significant improvement to the performance would be to use hierarchal maps. The first layer would define the main cities and the roads connecting them. After finding the best route between the cities (with a heuristic function of the travel time within the cities driven through in the route, such as the time Waze calculated), a second layer map would be used to find the best route within the city.

This performance improver can really improve the runtime, although it might give us not as good results as the current implementation. But it might be necessary even after optimizing the implementation of various parts of the algorithm, in order to handle real life big maps such as Israel map.

3) **Components implementations:** The algorithm implementation is general enough so it’s very easy to implement different kinds of components which have the same responsibility. An example for that is the different kinds of mutators. However, more components could be designed in the future, such as fitness functions, vocabulary calculators, initial population creators, and the way we terminate the process of creating new generations. Writing new components might give better results generally, or it could give better results in specific map configurations.

4) **Updates to the route:** In real life situations things can’t be predicted as was done in our algorithm, such as real driving speed of drivers and car accidents. These things can create dynamic traffic jams which the algorithm can’t predict. A solution to this would be to update the calculated route during the driving. We could even optimize this updating by using a different vocabulary calculator which takes into account only a few alternative roads to the road the driver is currently driving through.