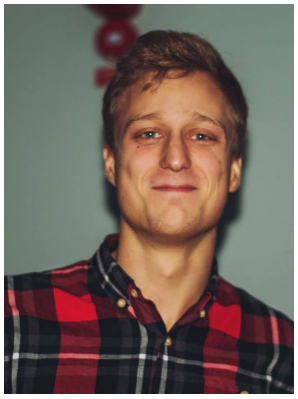
Investigation of planning algorithms characteristics and performance in graph environments

Kaya, Koray

Nagy, Rajmund

Qian, Livia



Stigenberg, Jakob

Yu, Bryan

September 9, 2020

Abstract

This report investigate empirically the contingent planning, online planning and re-planning problem as related to the real world experience of an individual's initial exploration of a new city. The report investigates the performance of three approaches: Q-learning, MDP, genetic algorithms, and their performance across various environments. The results of the investigation suggests that the strength of various approaches reveals themselves in different environments.

1 Introduction

Consider moving to another city where you have found a new job. Although it may be fun and exciting to explore your new home city, there are certain things you will want to optimize. For example, finding your optimal path to work or other locations. Nowadays, the market has a lot of tools to offer for these specific problems, e.g. Google Maps and Waze. However, these rely on data gathered by a lot of users and not specifically one's personal experience. And what if you wanted to challenge yourself – to see how well you can perform at learning the layout and navigating across the dynamics of the city – all on your own? Well, now this sounds a lot like an online contingent planning problem.

2 Background

2.1 Markov Decision Processes

Markov Decision Processes (MDPs) model the environment using a state space $\mathcal{S}$, an action space $\mathcal{A}$. Furthermore, given an action $a \in \mathcal{A}$, the state transition probabilities are given by a matrix T_a. Finally, the rewards are given by the reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$, taking the current state, action and next state.

In a graph structure, each node corresponds to a state, and from every node the set of actions are given by the edges connected to that node. Thus, picking an edge connected to a certain node will advance the agent to the node connected by the chosen edge. In this case, the process is deterministic, i.e., all state transitions have a probability equal to one or zero. Still, the MDP framework is applicable.

2.1.1 Finding the optimal policy

Given a reward function, $r(s, a, s')$, an agent wants to maximize the total reward accumulated when traversing through the graph. Through dynamic programming, the optimal policy is easily achieved. First, introduce $V(s) : \mathcal{S} \to \mathbb{R}$ as the total reward to be accumulated when starting in state s. Then, we can write

$$V(s) = \max_{a \in \mathcal{A}} \Big[r(s, a, s') + \gamma V(s') \Big] \tag{1}$$

for some discount parameter $\gamma \in (0, 1)$. In other words, the total accumulated reward is the sum of the reward collected now and the reward to be collected in the future. Therefore, through back-propagation, all $V(s)$ can easily be learned and the optimal action is simply given by

$$a^* = \arg\max_{a \in \mathcal{A}} \Big[r(s, a, s') + \gamma V(s') \Big] \tag{2}$$

If the rewards are assumed negative, then the back-propagation goes as follows. Note that positive rewards would encourage an agent to traverse in loops.

1. Initialize $V(s) = 0 \ \forall s \in \mathcal{S}$. Add $(s_\infty, 0)$ to a set L where s_∞ is the goal state.

2. While L is not empty:
   (a) pick (s', v') from L.
   (b) for every $s \in \mathcal{S}$ such that there is an action a to go from s to s':
      i. $v = r(s, a, s') + \gamma v'$
      ii. if $v \geq V(s)$, then $V(s) = v$ and add (s, v) to L

When the algorithm terminates, the optimal action in each state is given by Eq. 2.

2.2 Q-learning

Tabular Q-learning is a model-free reinforcement learning algorithm. The agent starts with minimal knowledge and learns through the consequences of its actions under multiple episodes. This is a great algorithm for the problem we are facing, which is a partially observable and dynamic environment. Its setup is similar to that of the MDP's, however, now the state transitions and reward functions are unknown. Instead, the agent must explore the effect of different actions in order to learn.

Q-learning relies on learning the value of the Q-function. Defined as $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, the Q-function gives the total expected reward to be accumulated from state $s \in \mathcal{S}$ if action $a \in \mathcal{A}$ is chosen, assuming optimal play. Similarly to Eq. 1, the Q-values must then satisfy

$$Q(s, a) = \sum_{s'} \mathbb{P}\big[s'|s, a\big] \big[r(s, a, s') + \gamma \max_{a' \in \mathcal{A}} Q(s', a')\big] \tag{3}$$

Note, that if the state transitions and rewards are known (and deterministic), this relationship is equivalent to Eq. 1.

Since the rewards are unknown, the training must proceed by exploring the map. The Q-value for each visited state-action pair, (s_t, a_t), is updated according to the following function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)\big] \tag{4}$$

where α is a constant referred to as the learning rate. Note that if $\alpha = 1$, then no averaging occurs and the new Q-value is given exactly by

$$Q(s_t, a_t) = r_t + \gamma \max_{a'} Q(s_{t+1}, a')$$

The discount rate, γ, introduces temporal difference to the equation. In cases where the constant is below 1, the Q-values closer to the present will have a higher value and therefore making good choices right now will be preferred to making good choices in the future. If the value is equal to 1 we will have no temporal preference.

2.2.1 Exploration versus exploitation

At every time step, the *believed to be* best action is given by

$$a^* = \arg\max_{a \in \mathcal{A}} Q(s_t, a)$$

However, the Q-values may, e.g., not have converged yet or be stuck in a local optimum. These issues require additional exploration. There are many approaches to induce an agent to perform more exploration. Two common options are using ϵ-greedy policies and optimistic initialization.

An ϵ-greedy policy chooses the next action as

$$a = \begin{cases} a^* & \text{with probability } 1 - \epsilon \\ \text{Uniformly random from } \mathcal{A} & \text{with probability } \epsilon \end{cases}$$

Therefore, a certain amount of exploration is always present in the algorithm. One may additionally change ϵ during the training, e.g. explore a lot initially and then taper down.

Optimistic initialization uses greedy policies, i.e., the action is always chosen according to what is believed to be the best action. However, by starting with very large Q-values, the algorithm will naturally explore as it will train the Q-values to become smaller.

2.3 Genetic algorithms

Genetic algorithms are metaheuristics that are based on the idea of biological evolution [1]. Their advantage lies in their flexibility and the ability to operate in very big solution spaces where classical search methods are infeasible. The main idea of a genetic algorithm is to store a number of possible (not necessarily valid!) solutions, and continuously try to improve them. Each solution is represented in a compact way to speed up the computation. The exact representation depends on the problem, for example, the states could be binary numbers (useful for the Knapsack problem), integer sequences (for finding the shortest path in a graph) or a permutation of a fixed sequence (in the case of the traveling salesperson problem [2]).

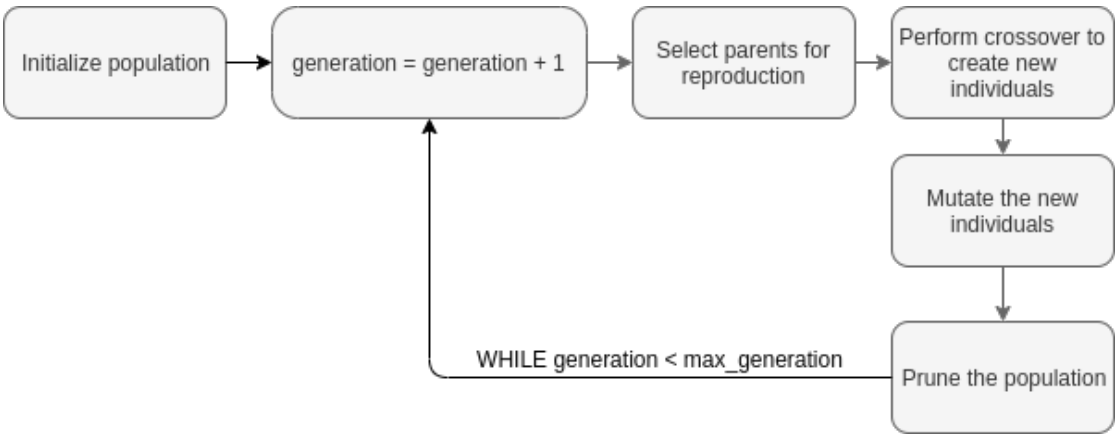


Figure 1: The structure of the algorithm

An encoded solution to the problem is called a chromosome, each element of the chromosome is called a gene, and the set of stored solutions is called the population. For the performance evaluation of the chromosomes, a fitness function has to be defined. This function needs to be fast to compute and it should assign a value to each possible solution in proportion to how close it is to the optimal one. The structure of the algorithm can be seen in Figure 1.

There are several methods for selecting the parents. One of them is the Roulette Wheel Selection, where the probability of choosing a chromosome is proportional to its fitness, therefore the best individuals have the highest chances of reproduction. In some cases it is possible that one chromosome is vastly superior than the others, therefore it will end up dominating the population and the genetic diversity will suffer. To avoid this scenario, the variants of the Rank Selection method can be used [3].

The crossover operation is the central point of every genetic algorithm: it takes at least two chromosomes as its input, then it creates children based on the parents' characteristics, mimicking biological reproduction. The implementation should aim to combine the "good" parts of the parents - this can be achieved by using domain dependent knowledge when designing the operation. Some of the simplest examples are single point crossover and arithmetic crossover. In the former, a gene is chosen randomly, then the offspring are created by combining one parent's allele sequence up until that gene with the other parent's sequence starting from that gene. In the latter, the parent chromosomes are combined by a performing arithmetic operations, e.g. taking the mean of the alleles.

The role of mutation is to introduce exploration to the search in order to avoid getting stuck in local optima and to speed up convergence. Henceforth, this operation typically does not take any domain dependent heuristic into account, it only modifies the chromosomes in a random way. However, the the result should respect the invariants of the representation, meaning that it must not generate repeated alleles for permutation encoding, for example. In some cases, such constraints may be hard to keep without substantially slowing the algorithm down, so careful design is needed.

The last element of a GA is the pruning of the population, i.e., the selection of individuals to keep for the next generation. The most straightforward way to do it is to only keep the best N chromosomes, but in dynamic environments it might make more sense to ensure the survival of a few individuals at the top, then randomly select the rest. This part of the algorithm is not always needed, since it's possible to keep the population size constant during the reproduction phase.

3 Methods

3.1 Model

The traffic environment is modelled using a graph structure where every node corresponds to an intersection and every edge corresponds to a road. The cost of travelling along a road is proportional to the time required which varies depending on the length

Map name	Mean cost	Standard deviation	Type of map
`bigmap`	138	$\sqrt{294} \approx 17.1$	Multiple paths to goal
`LOBSTER`	8740	$\sqrt{42601} \approx 206.4$	Single path to goal with dead-end branches
`Tunnelbana`	1120	$\sqrt{103} \approx 10.1$	Single path to goal with two dead-end branches

Table 1: Mean and variance of the optimal path.

as well as traffic conditions. Using the assumption that there is a hard lower limit of the time required to travel along a specific edge (due to speed limit and road length) and that the likelihood of a certain traffic flow will vary with an exponential decay, the distribution of the edge costs were set to an exponential distribution plus a constant base (b_i), i.e., the distribution of the weight, w_i, of edge i is given by

$$p(w_i) = \begin{cases} \frac{1}{\beta_i} \exp\Big(-(w_i - b_i)/\beta_i \Big) & w_i \geq b_i \\ 0 & w_i < b_i \end{cases} \tag{5}$$

where $\beta_i, b_i > 0$ are specified for each edge.

Given this model, there are a number of possible variations. An initial approach would be to minimize the total cost required to reach a certain destination. However, due to the randomness in the graph, it may be also be of interest to minimize the variance in the acquired cost. In a real life environment, this would correspond to arriving at your destination in a consistent manner. For example, instead of a varying arrival time in the range 5-20 minutes, one may prefer arriving in 15 minutes consistently.

Finally, one may introduce online planning into the model. When arriving at an intersection, it is (usually) possible to observe the true traffic flow of the connecting roads. At that instance, the route may be re-evaluated using the observed values. This scenario is directly transferable to the model, by observing the weight of all edges connected to the node where the agent currently is located.

3.2 Maps

We created a few maps of different sizes in order to get an understanding of the complexity and convergence behavior of each algorithm. The maps can be seen in the appendix under Sec. 5.1. We used mainly three maps, `bigmap` in Fig. 8, `LOBSTER` in Fig. 10 and `tunnelbana` in Fig. 7. The information regarding the optimal path can be found in Tab. 2. These values were easily obtained since the distributions are assumed independent.

3.3 Markov Decision Processes

The state transitions, state space and action space are fully determined by the model. What remains to tune is the reward function. Since MDPs try to maximize the total accumulated reward, the reward function was simply chosen as the negative cost. Further, since the distribution (Eq. 5) allows only positive costs, the rewards will be strictly

negative, thus eliminating the risk of an agent going in loops. However, since the costs are unknown, we will resort to using the expected cost, i.e. the mean of the distribution.

We suggest two versions of the MDP, which have different reward functions. The first, most basic one, uses the true mean as cost, unless the actual cost has been observed. This approach essentially 'solves' the optimal path under the assumption that you know the distribution from which the costs are sampled from. However, this does not reflect our real life problem well enough, in which we are completely unaware of the distributions.

Therefore, another approach would be to continuously keep an estimate of the mean of each distribution and use that as cost. The estimated mean $\hat{\mu}$ can be updated using

$$\hat{\mu}_k = \hat{\mu}_{k-1} + \frac{1}{k}\big[x_k - \hat{\mu}_{k-1}\big]$$

where x_k is the sample at time step k and $\hat{\mu}_1 = x_1$.

Note that using MDPs in this setting is a form of online planning, allowing us to continuously re-evaluate the plan whenever we traverse a node. In order to achieve this, we show the agent the true cost of the neighbouring edges. As discussed in Sec. 3.1, this would correspond to observing the traffic flow on nearby roads at an intersection, before turning onto them.

3.4 Q-learning

Similarly to MDP, in Q-learning we can only modify the reward function which will be given by the negative cost. However, opposed to MDP, there is no need to estimate the reward beforehand since the update only happens after an edge has been traversed. This also implies that the step to online planning is not straightforward in this case.

We may also alter the learning rate, α, which was chosen to be $\alpha_k = \frac{\alpha_0}{k^{2/3}}$, where α_0 is the initial learning rate, and k is the number of times the specific state-action-pair has been observed. By choosing this type of learning rate, the oscillation of Q-values are reduced but the algorithm will still converge [5].

3.5 Genetic algorithm

The solution space of the problem consists of possible routes one can take between the initial and the terminal nodes. We chose value encoding for the chromosomes, where each gene corresponds to a node's name, with two constraints: first, the sequence must start with the initial node and end with the terminal node, and second, there must not be any repeated states. It might be important to note that we do not forbid the inclusion of invalid state transitions into an individual.

3.5.1 An initial version

The first version of the algorithm initialized the population with completely random paths. Therefore we relied completely on the mutation and the crossover operators for finding the best solution. The mutation operator simply changed one of the nodes in the

path to a different one (while respecting the prohibition of repeated states – we decided that it would be too computationally expensive to also consider the validity of the new state transitions. For crossover, we used a slightly tweaked version of one-point crossover. The idea was to generate viable alternatives by combining two intersecting paths and mixing up their sub-routes, therefore we chose intersection points as the cutting point for the operation. Finally, we sorted the resulting population with a fitness function that simply calculated the cost of the route (with a 10000 penalty for each invalid edge), then kept a fixed number of the chromosomes with the lowest costs.

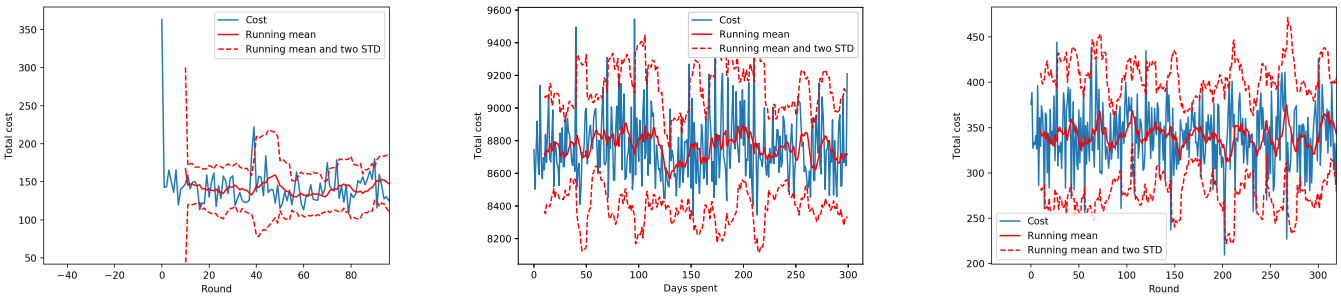
3.5.2 The improved algorithm

Starting with purely random routes was a very bad idea because it caused many invalid edges: since the cost function heavily penalized them, the algorithm always went for short routes with at most one invalid edge (because adding another valid edge would have just increased the cost). Avoiding invalid edges at every step, or performing more sophisticated cost analysis to incentivizes longer routes would have been too computationally expensive, therefore we decided to generate the initial population via random search (choosing random neighbors at every step, starting from the starting node until we run out of unique nodes or reach the end). This way we avoid the need of preventing invalid nodes (for they can only be caused by mutations).

Another change that we introduced was to introduce Roulette Wheel Selection for choosing the parents. Each chromosome is selected with a probability that is proportional to its fitness, compared to the entire population. In principle, doing so will increase the genetic diversity in the new generations, since the lower-ranked individuals are given a chance as well.

4 Results & Conclusions

4.1 MDP Agent



(a) `bigmap` (b) `lobster` (c) `tunnelbana`

Figure 2: MDP results using estimates are look ahead observations.

The MDP that was finally implemented used a reward function equal to the negative cost, where the cost of each edge was estimated using the observed values. Additionally, we continuously replanned the route as we arrived at new nodes and observed the true costs of nearby edges.

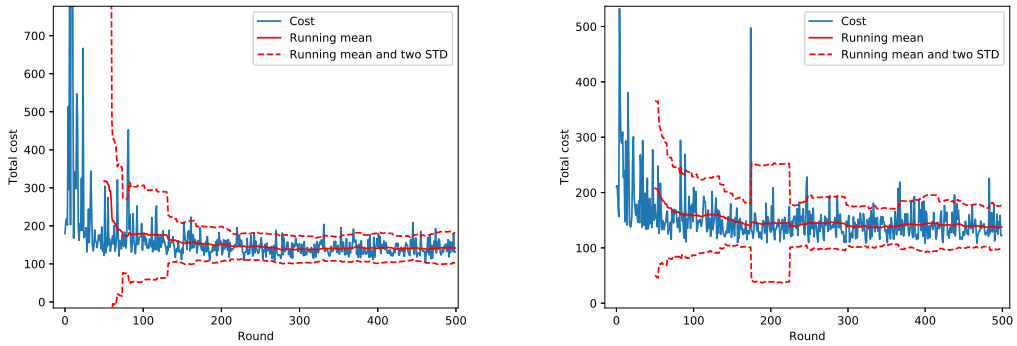
In Fig. 2a the performance of the MDP agent showed relatively fast convergence in the range of 10 days. This result is reasonably in line as the agent begins with a tendency to move towards the goal. The dynamic programming can essentially help prune the tree of unreasonable paths.

When faced with the tunnelbana map, the MDP agent performance was affected by the daily variance. Upon deeper inspection, the tunnelbana map is markedly similar to a line graph with one fork. The MDP agent's back-propagation is expected to capture the optimal route from start to goal independent of the cost and therefore such daily variations along these paths are captured.

The results of tunnelbanna was extended with the investigation of a map with increased number of nodes and forks. The results align with expectations as set by tunnelbana.

Unfortunately, we were unable to observe any behavior where the agent would backtrack due to a future edge having a larger than expected cost although the maps used did not include any realistic opportunities for that to happen.

4.2 Q-learning



(a) Using initial Q-value of 100 and $\epsilon = 0.0$

(b) Using initial Q-value of 0 and $\epsilon = 0.05$

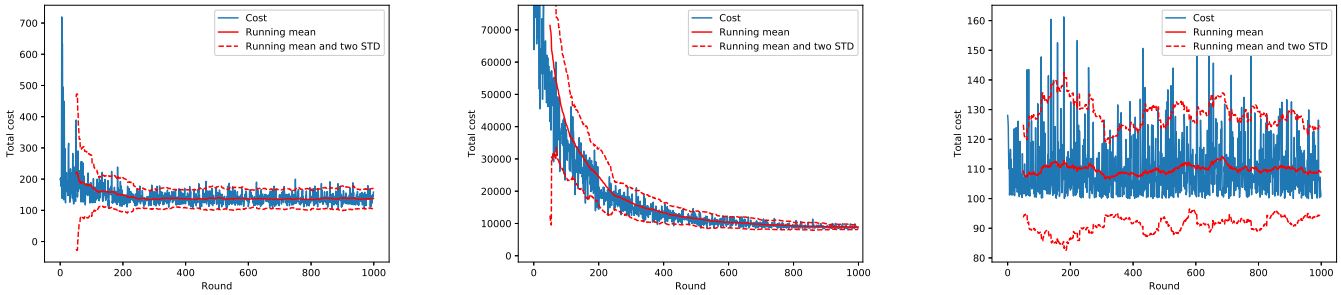
Figure 3: Q-learning on `bigmap` using $\alpha_0 = 1.0$, $\gamma = 0.999$ and varying exploration versus exploitation techniques.

In Fig. 3 the convergence of Q-learning on `bigmap` is shown using the two different exploration vs. exploitation techniques. It appears as if the optimistic initialization (Fig. 3a) converges quicker than using and ϵ-greedy policy (Fig. 3b), however the exploration the first few days takes **a lot** of time. The initial peak of optimistic initialization is approximately double that of ϵ-greedy when initializing a value of 100 (outside the view

Map name	Population size	# of generations	# of parents	P_c	P_m	iter_c
`bigmap`	10	300	5	0.9	0.7	30
`LOBSTER`	30	300	5	0.9	0.3	10

Table 2: GA parameters used in testing

of Fig. 3) and seem to grow unbounded in some cases. On the other hand, ϵ-greedy policies continue to explore with a certain probability, thus the algorithm does not converge as nicely. To counteract this, since Q-learning is an off-policy algorithm, one could introduce a decay on the value of ϵ or using a cutoff to stop the exploration after a certain number of steps [4].



(a) `bigmap` (b) `lobster` (c) `tunnelbana`

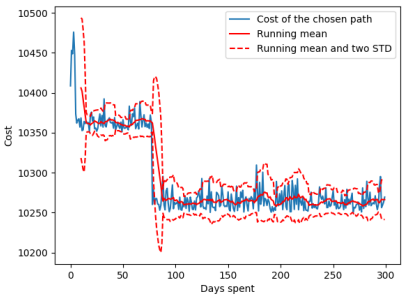
Figure 4: Q-learning on the three maps using $\alpha_0 = 1.0$, $\epsilon = 0.2$, $\gamma = 0.999$, and epsilon cutoff at 200, i.e. $\epsilon = 0$ after 200 rounds.

In Fig. 4, the ϵ-greedy policy using cutoff is run on the three maps. Although the algorithm does converge to the optimal path, considering that each training iteration can be interpreted to reflect one full human day, a convergence in the range of 250 days suggests that the actor will require the larger portion of a calendar year to consolidate his knowledge base. However, keeping in mind that these results are obtained by running over a graph on which there is no prior knowledge what so ever, it is still reasonable. If the user had access to prior information, then a first step would be to prune the graph of unreasonable paths to consider before exploring. In that sense, MDP is favorable.

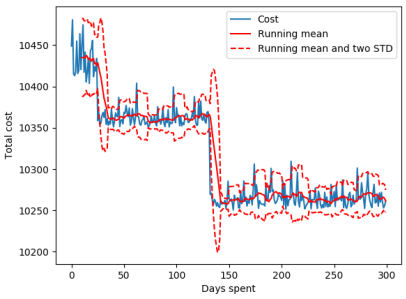
4.3 Genetic algorithm

We tested the GA performance on the `LOBSTER` and the `bigmap` environments. In Fig. 5a, it can be seen that the algorithm quickly found a path with an average cost around 10350, then it managed to improve it again at the 80th generation or so. Fig. 5b exhibits similar results. While the exact speed of the convergence can vary, in our experiments the algorithm usually reached the final performance under 200 generations.

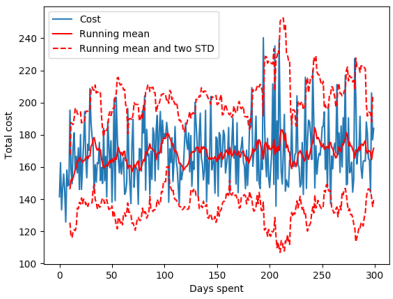
As for the `bigmap` environment, the number of edges is so small that the algorithm already finds the best path at the population initialization, therefore we reduced the



(a) `Performance on lobster`



(b) Another run on the `lobster` map with similar tendencies



(c) Performance on `bigmap`

Figure 5: GA results

population size to 10 in order to get comparable results with the MDP/Q-learning performances (Fig. 5c).

Ultimately, we concluded that genetic algorithms come with considerable caveats. First, they would be more suited for a grid representation of the map, because with the graph representation, we need to deal with invalid edges all the time - in bigger maps, exploration will be disencouraged unless we spend considerable resources on handling them. Second, the algorithm is very dependent on the initialization: in a dynamically changing environment, this can mean that we will never converge to the correct solution if the first state of the map is unusual enough. Nevertheless, the algorithm works reasonably well, achieving faster convergence than the Q-learning approach at the expense of computational resources.

References

[1] John H Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

[2] Jihene Kaabi and Youssef Harrath. Permutation rules and genetic algorithm to solve the traveling salesman problem. *Arab Journal of Basic and Applied Sciences*, 26(1):283–291, 2019.

[3] Noraini Mohd Razali, John Geraghty, et al. Genetic algorithm performance with different selection strategies in solving tsp. In *Proceedings of the world congress on engineering*, volume 2, pages 1–6. International Association of Engineers Hong Kong, 2011.

[4] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[5] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):279–292, May 1992.

5 Appendix

5.1 Maps

We created a number of maps for testing; some of them were created manually (Figs. 6, 7, 8), some were generated (e.g. Fig. 9,10,11).

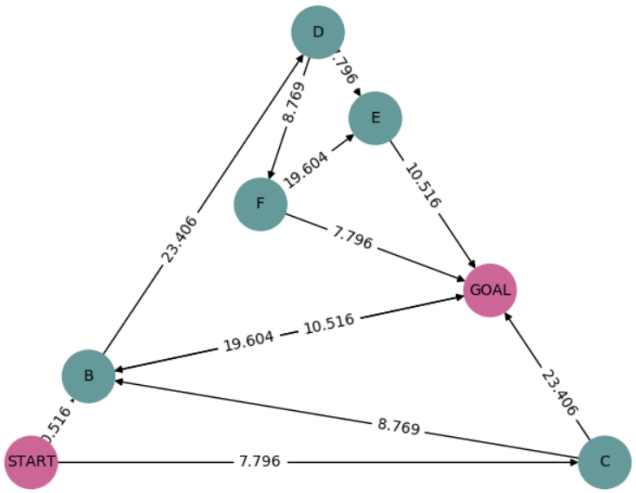


Figure 6: The original test map

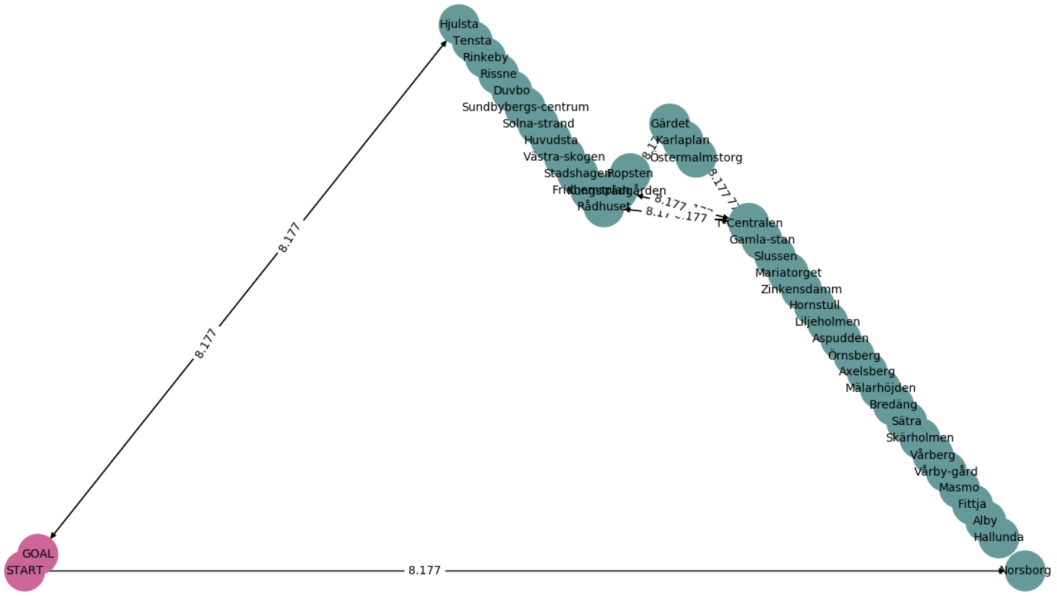


Figure 7: Tunnelbana map (line 10 and 13)

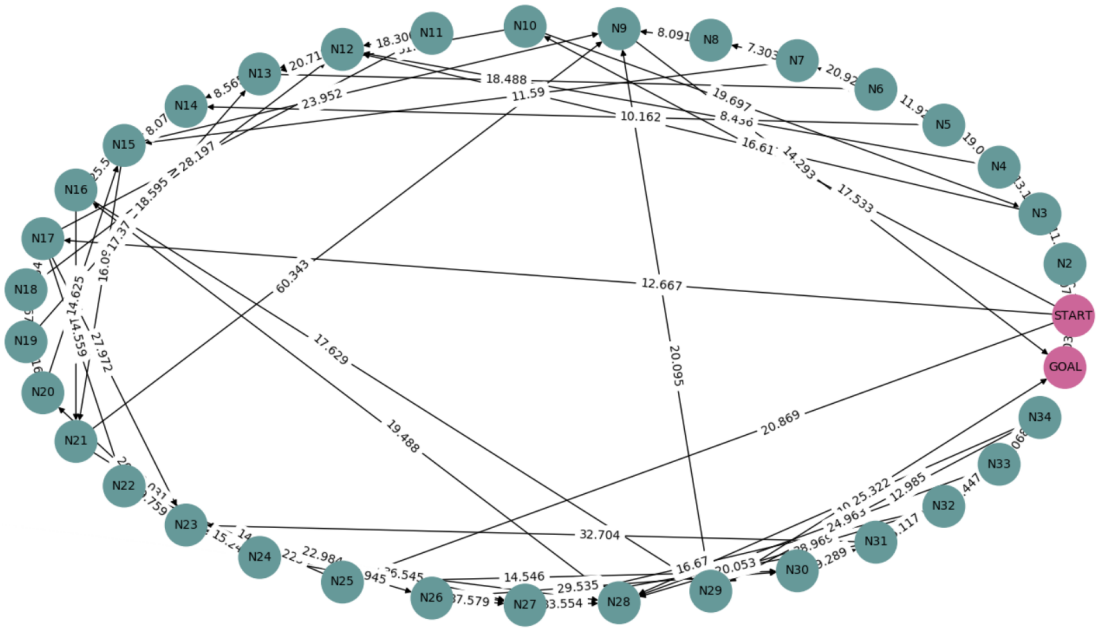


Figure 8: Big map (handmade)

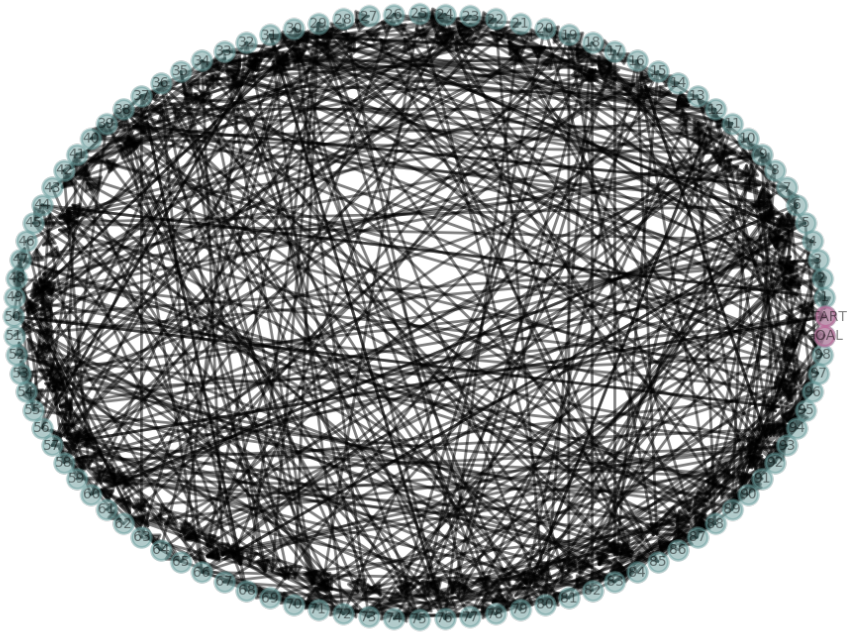


Figure 9: Generated map (100 nodes, 500 edges)

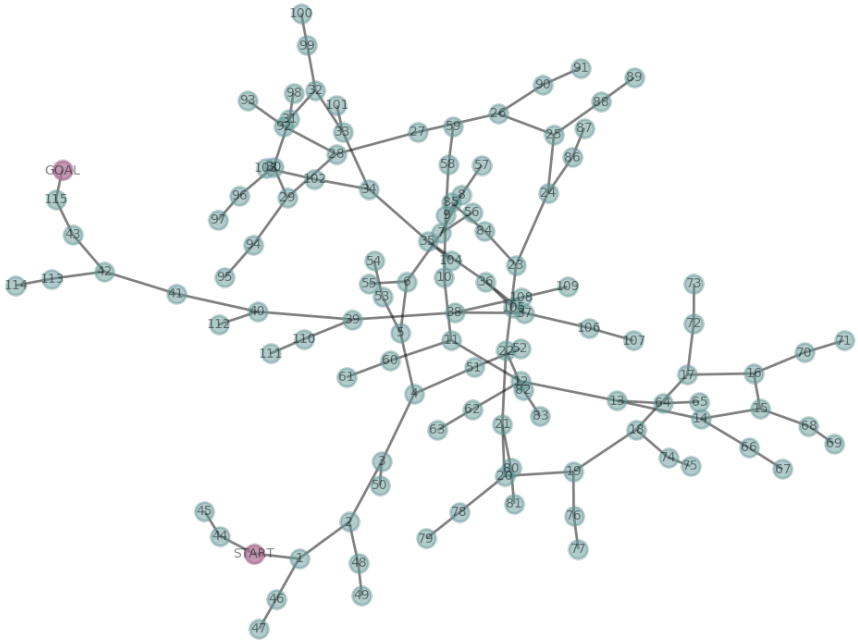


Figure 10: Generated map (115 nodes)

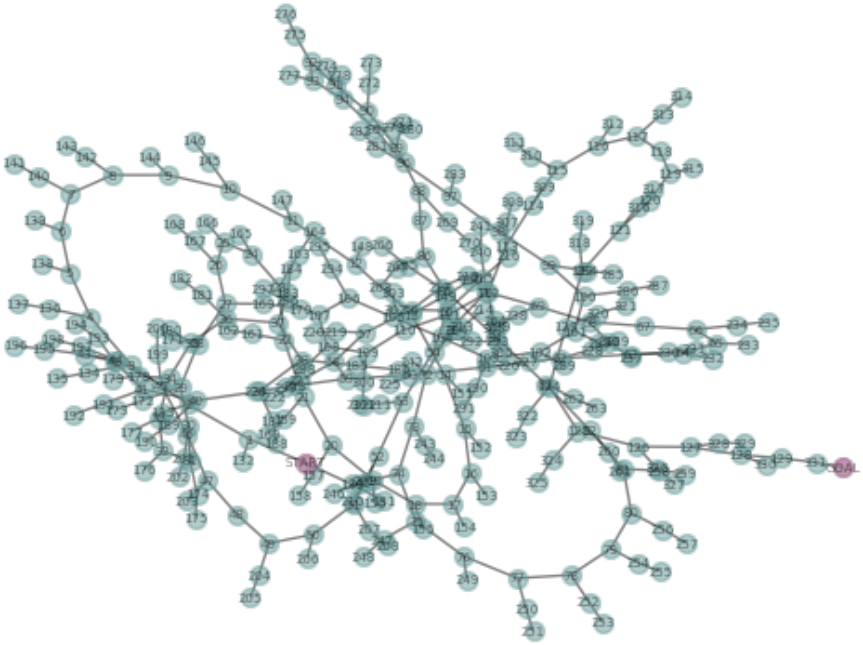


Figure 11: Generated map (331 nodes)