Ant Colony Optimisation and Reinforcement Learning

Adam Price

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2019

Abstract

Ant colony optimisation and reinforcement learning share a few fundamental similarities, yet as reinforcement learning grows in popularity, we see very few new studies being conducted in ant colony optimisation and other metaheuristic algorithms. In this paper we conduct a detailed look into the workings of ant colony optimisation and reinforcement algorithms. We then move on to problems from the domains of ant colony algorithms to reinforcement learning to increase understanding and to demonstrate ant colony optimisation's potential in Markov decision processes.

Acknowledgements

I would like to acknowledge my supervisor, Dr Michael Herrmann, for our frequent discussions and his help throughout this project. I'd also like to thank my brother, Matt Price, and friend, Daniel Johnson, for proofreading this paper.

Table of Contents

Chapter 1

Introduction

In recent years reinforcement learning has seen a large surge in popularity, primarily due to it's ability to out perform humans in common human tasks like video games or Go [11, 9]. A large reason for this is the deep learning approaches that have become possible in the advent of increased computing power in the recent years. However, many metaheuristic algorithms could also see benefit from increased computing power. For example, genetic algorithms can use parallel computing to run concurrently run fitness evaluations. Fundamental similarities can been seen between certain metaheuristic algorithms and reinforcement learning. Many algorithms have also been derived from mergers of metaheuristic algorithms and reinforcement learning [2, 3] and the question of whether reinforcement learning itself falls under the category of metaheuristic algorithms is up for debate. So, despite many similarities, we are not seeing as much popularity around metaheuristic as we do reinforcement learning.

In this paper we will be looking at the ant colony optimisation family of algorithms. They are popular metaheuristic algorithms that have already been linked with reinforcement learning [7]. We will be comparing mathematical similarities between the the two, and applying them to problems commonly found in the domains of ant colony optimisation and reinforcement learning. Our main aim is to ensure our understanding of how algorithms function and the effects of their hyperparameters. We have experimented with algorithms in an array of different problems and provided discussion of our findings.

This paper continues with some background information on the topics we will cover. Then continues to our experiments and discussion, and ends with a brief conclusion.

Chapter 2

Background

In this section we will give background information on reinforcement learning, ant colony optimisation, and the link between them. We will also state some of the algorithms and equations we will use from these areas. We also would like to note now that we are only concerned with discrete time and episodic problem, so the rest of the paper will not consider continuous time or states.

2.1 Reinforcement Learning

Reinforcement learning (RL), at least in the scope of this paper, is concerned with finding optimum control policies for an agent interacting with an environment. "Reinforcement learning is learning what to do" [12]. RL algorithms are essentially optimisation algorithms, however, unlike algorithms found in other parts of machine learning or optimisation, RL does not use error gradients to find optima, in fact there is no concept of error in RL. Instead, a RL agent must maximise the reward that it receives from it's environment by optimising the decisions that it makes in the that environment.

2.1.1 Markov Decision Process

The problems we can apply RL to are modelled as a Markov decision process (MDP). MDPs are a framework for modelling an environment and the decision that can be made in the environment. It does this with 4 elements:

- S is a finite set of states that an agent can be in within the environment,
- A_s is the finite set of actions that an agent can take at state s,

- $P_a(s,s')$ is the probability that action a taken at s will transition to state s',
- $R_a(s,s')$ is the immediate reward received by the agent when transitioning from s to s'

MDPs use discrete time, meaning time is broken up into distinct points at which a decision has to be made. So, at each time step t an action is taken that can update the agents state:

$$s_t \xrightarrow{a_t} s_{t+1} \xrightarrow{a_{t+1}} s_{t+2} \xrightarrow{a_{t+2}} s_{t+3} \xrightarrow{a_{t+3}} s_{t+4}$$

With this in mind we can think about how RL algorithms solve an MDP. It is summed up by a simple loop:

- The agent takes an action in the environment,
- The agent observes the state it has transitioned to and the reward given by transitioning to the new state,
- The agent uses its observations to try and increase its likelihood of receiving more reward in the future.

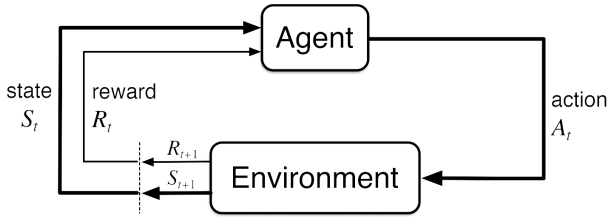

Figure 2.1: Agent-Environment Interaction. From [12]

All the MDPs we will be examining are episodic in nature. This means that they have one or more termination states. When an agent reaches a termination state we consider the episode over and the environment is reset, however, learning conducted by the agent is kept. This lets the agent solve the problem over multiple episodes. In episodic tasks it can be important to consider the set of non-terminal states separately from the full set of states. Often in RL literature S^+ is the set of all states, including the terminal state and S is set of all nonterminal states.

To find solutions to MDPs in RL we need to work out the best actions to take during an episode. It is important to note that an actions value will likely be different

depending on the state the agent is in, so, we have to consider the value of the 'state-action pairs'. A state-action pair is just an action, a at a state s. The value of a state-action pair is often given as $Q(s,a)$. Q is the 'Q-Table', it holds the values of all the state-action pairs, and is used to produce policies for the RL agent (policies tell the environment what actions to take at which states). Through interaction with the environment, the RL agent aims to make their estimate of the Q-Table reflect the actual values of the state-action pairs in the environment.

With all this in mind we can now look at some of the algorithms that try to maximise the reward in an MDP.

2.1.2 Temporal Difference Learning

Temporal difference (TD) learning is at the heart of most of the current reinforcement learning research, and is the closest method to the loop described in section . They are able to update value estimates directly from experience in the environment without waiting for episode termination, and can update estimated values based, in part, on other estimates they have made.

Using other estimated state-action pair values to update the value of a different state-action pair is called bootstrapping. To best understand it we will dissect the general Q-learning update equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R_{t+1} + \gamma \max_A Q(s',A) - Q(s,a)) \tag{2.1}$$

- $\alpha \in (0,1]$, is the learning rate. It controls the step size of the update. It is often best to have the learning rate low (≈ 0.1) to avoid overstepping the actual value of $Q(s,a)$.
- $\gamma \in [0,1]$, is the discount factor. This hyperparameter effects how much weight it gives to future reward. As the algorithm updates during an episode, the reward signal will be able to travel further through the state-action space with higher values of γ. For example, a reward signal of 1 three actions away from the current state can have value $1\gamma^3$ at the current state.
- R_{t+1} is the reward that the agent receives when it transitions between s and s'.
- $\max_A Q(s',A)$ is the highest action value in the next state.

- $R_{t+1}\gamma\max_A Q(s',A)$ is the TD target and can be thought of as an estimate for the true value of $Q(s,a)$. Subtracting $Q(s,a)$ from the the TD target gives the TD error, which is multiplied by the learning rate to give the value of the update.

The algorithm calculates $Q(s,a)$ values by considering the value of the next state. As this is done in all states, the reward signal can propagate through the state-action pairs. This lets TD learning update at every time step.

2.1.2.1 Q-Learning

The first RL algorithm we will look at is Q-learning. Q-learning is described as being off-policy, this is because the estimate it uses in the TD target is not generated by the policy. The version we will be using in our experiments is given by the following pseudo-code algorithm.

Algorithm 1 Q-Learning: off-policy TD control based on [12]

Initialise $Q(s,a)$, for all $s \in S^+$, $a \in A_s$, arbitrarily
for Each episode **do**
  Initialise s
  while s is non-terminal **do**
    Select a using s and ε-greedy policy derived from Q
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma\max_A Q(s',A) - Q(s,a)]$
    $s \leftarrow s'$
  end while
end for

Q is used to generate the policy the agent uses to select an action. Q can be used to generate many different policies, but we will be using ε-Greedy which is described by the following equation.

$$a = \begin{cases} \arg\max_a(Q(s,a)) \text{ (tie settled by random pick)}, & p(1-\varepsilon) \\ \text{random} \in A_s, & p(\varepsilon) \end{cases} \tag{2.2}$$

This policy introduces a new hyperparameter in $\varepsilon \in [0,1]$, it lets the agent take random actions to explore it's environment. It is usually set to be ≈ 0.1 because far more exploitative actions are need to find good solutions.

2.1.2.2 SARSA

We will also be using the SARSA algorithm. SARSA (State, Action, Reward, State, Action) is very similar to Q-learning, but with one slight difference. SARSA is an on-policy method, meaning it uses its policy to generate the estimate used in the TD target. The differences between on and off policy methods will be explored more in section 4.1.

The SARSA algorithm we will be using is given by:

Algorithm 2 SARSA: on-policy TD control based on [12]

Initialise $Q(s,a)$, for all $s \in S^+$, $a \in A_s$, arbitrarily
for Each episode **do**
  Initialise s
  Select a using s and ε-soft policy derived from Q
  while s is non-terminal **do**
    Take action a, observe r, s'
    Select a' using s' and soft policy derived from Q
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$; $a \leftarrow a'$
  end while
end for

Our SARSA algorithm uses what we are calling a 'ε-soft policy' which is described by the following.

$$a = \begin{cases} \text{weighted random where } p(a|s) = \frac{Q(s,a)}{\sum_{a' \in A_s} Q(s,a')}, & p(1-\varepsilon) \\ \text{random} \in A_s, & p(\varepsilon) \end{cases} \tag{2.3}$$

This action selection keeps ε exploration, but also lets actions be chosen from a weighted selection where the most valuable actions are more likely to be chosen.

2.2 Ant Colony Optimisation

We will now present ant colony optimisation (ACO) and introduce the algorithms that we will be using from ACO in this paper. ACO is often described in the context of graphs or the traveling salesman problem, but for easier understanding in this paper

we will be laying out ACO in similar terminology to that found reinforcement learning research.

2.2.1 Formation and Natural Inspiration

ACO is a population based metaheuristic algorithm. Metaheuristics are problem-independent techniques, meaning they can be applied to many problems without needing to be adapted to the problem. ACO was inspired by how real life ants lay pheromones to create paths between their nest and food sources [8]. A single ant is partially blind, but many ants together manage to use a pheromone trail to efficiently navigate their environment.

All ACO algorithms follow the same basic steps:

- Construct a set of solutions with ant population.
- Uses solutions to update pheromones.

Updating the pheromones after building the solution puts ACO in the Monte Carlo family of algorithms (along with some RL algorithms). The differences between ACO algorithms are found in how they carry out these two steps.

2.2.2 Ant Systems

The first ACO algorithm we will look at is the ant system (AS) algorithm [6], it was the first ACO algorithm and is the most simple. In AS the ant population is controlled, in part, by a pheromone matrix tau_{sa}. There is pheromone at each state s that 'attracts' an ant toward taking action a. The ants update the pheromone matrix by the following equation:

$$\tau_{sa} \leftarrow \rho \cdot \tau_{sa} + \sum_{k=1}^{m} \Delta\tau_{sa}^{k} \tag{2.4}$$

- $\rho \in (0,1]$ is the evaporation rate. It reduces the amount of pheromone throughout the table each iteration. This lets the ant system effectively forget about bad decisions it made in the past, as only often walked paths will persist.
- m is the number of ants in the population
- $\Delta\tau_{sa}^{k}$ is the amount of pheromone to be laid at state-action pair (s,a) by ant k. How this value is determined will depend on the problem the algorithm is applied to.

Sometimes we will want to use a slightly different update:

$$\tau_{sa} \leftarrow \rho \cdot \tau_{sa} + \Delta\tau_{sa}^{best} \tag{2.5}$$

The difference here is $\Delta\tau_{sa}^{b}est$. Instead of using all ants in the update, only the best solution in an iteration is used.

The ants construct solutions by conducting a tour of the environment. At each state their probability of selecting an action is given by:

$$p(a|s) = \frac{\tau_{sa}^{\alpha}\eta_{sa}^{\beta}}{\sum_{a' \in A_s}\tau_{sa'}^{\alpha}\eta_{sa'}^{\beta}} \tag{2.6}$$

- η is a matrix of values at each state-action pair. It is a heuristic that contains values that helps the ant make good decisions. How these values are calculated depends on the problem.
- α is a control parameter that effects the exploration of the ant. High values of α exaggerate the relative size difference between state-actions values, making the ants more deterministic.
- β is like α, but for the heuristic η

2.2.3 Ant Colony Systems

The ant colony system (ACS) was proposed as an improvement to AS [5]. A key difference between ACS and AS is the action selection rule. Called the pseudorandom proportional rule it favors exploitation of the τ matrix:

$$a = \begin{cases} \arg\max_{a}(\tau_{sa}^{\alpha}\eta_{sa}^{\beta}), & p(q_0) \\ \text{Use equation 2.6}, & p(1-q_0) \end{cases} \tag{2.7}$$

The other main difference is in the update rules. In ACS the update is split into two parts. The global update rule:

$$\tau_{sa} \leftarrow (1-\phi) \cdot \tau_{sa} + \phi \cdot \Delta\tau_{sa}^{best} \tag{2.8}$$

where $\phi \in (0,1]$ is the pheromone decay coefficient.

This global update is performed after all of the ants have constructed a solution. It is intended to give a greater amount of pheromone to the best solutions. The 'local' updating rules have a slightly different effect. It is applied at every step of an ants tour:

$$\tau_{sa} \leftarrow (1-\rho)\cdot\tau_{sa} + \rho\cdot\Delta\tau_{sa} \tag{2.9}$$

The effects of the local update can vary on what $\Delta\tau_{sa}$ is set to. It is often set to τ_0 (the initial values on the pheromone matrix). Doing this decays the connections in the pheromone matrix being used in solution building. This causes more exploration for the ants as the ant population becomes less likely to take the same actions.

2.3 ACO and RL

Several papers have been written with the aim of combining RL with ACO however non of them are very recent.

Ant-Q [7] uses, in part, the Q-learning update equation in ACS to form a hybrid ACO and RL algorithm that can be applied the traveling salesmen problem. We look at this algorithm in more detail in section 3.5.

Phe-Q [10] combines multi-agent Q-learning with the concept of the pheromone trail to allow communication between the agents. As we are only focusing on problems that a single agent can solve this lies outside of the scope of this paper.

2.3.1 Mathematical Examination

The way we have formed our SARSA algorithm shares many similarities with the AS algorithm. First let's look at the action selection method used in both. Let's consider a situation in which $\varepsilon = 0$ and $\eta = 0$ at all s, a:

$$\begin{aligned} p(a|s) &= \frac{Q(s,a)}{\sum_{a'\in A_s} Q(s,a')} && \text{SARSA} \\ p(a|s) &= \frac{\tau_{sa}^{\alpha}}{\sum_{a'\in A_s} \tau_{sa'}^{\alpha}} && \text{AS} \end{aligned} \tag{2.10}$$

In a case where $\alpha = 1$ and the values in Q and τ are equal, the action probabilities will be the same. Similarities are also seen in the update equations:

$$\begin{aligned} Q(s,a) &\leftarrow (1-\alpha)\cdot Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)] && \text{SARSA} \\ \tau_{sa} &\leftarrow (1-\rho)\cdot\tau_{sa} + (1-\rho)\cdot\Delta\tau_{sa} && \text{AS} \end{aligned} \tag{2.11}$$

A big difference in these update equations is AS, being a Monte Carlo update, is applied at the end of an episode. However, because SARSA is on-policy, the next TD difference value is generated from the policy in the same way the ant's solution is generated throughout by the policy. Depending on how $\Delta\tau_{sa}$ is calculated, we could have situations in which these algorithms give identical results.

Let's consider a straight line walk problem in which the an agent has to walk from a starting state forward towards the termination state. Once the agent leaves a state it cannot return to it. At the termination the agent is given a reward of 1.

If we use the following equation in AS as the update equation:

$$\Delta\tau_{sa} = r\gamma^{i}, \tag{2.12}$$

where $\gamma \in [0,1]$ is the discount factor, r is the reward given for episode termination, and i is the index of the state-action pair in the solution.

We will get updates that yield identical results to SARSA on a our straight line walk. More complex problems wont work with this, as returning to states would cause errors in how both algorithms would work with discounting. However, any problem where $\gamma = 1$ would yield the same results.

Chapter 3

Traveling Salesman Problem

The traveling salesman problem (TSP) is a very widely studied combinatorial problem. In a TSP an agent is tasked with finding the shortest path that would take the agent through every city exactly once, and then return it to its starting location. The only information the agent is given is the locations of the cities (and hence the distances between all the cities).

At first glance this problem can appear to be quite trivial, however, as more cities are added to the problem, the number of solutions grows rapidly. The number of possible solutions (or paths) through a TSP with n cities is given by $(n-1)!/2$. This means that if we tried to solve a TSP with a brute force algorithm (A search through every possible path) the algorithm would have execution time of $O(n!)$. With complexity of this level it becomes wildly impractical to brute force even small TSPs.

The level of complexity in TSP has lead to it being a benchmark for many different algorithms from a variety of families of algorithms, and is a problem that is still being worked on today [1, 4].

3.1 Applying Reinforcement Learning to TSP

Although TSPs are usually solved by meta-heuristic algorithms, the problem can be adapted to fit many different optimisation methods, including reinforcement learning [13]. To do this we model the TSP as a Markov decision process (MDP).

Each city in the TSP is considered as a state (s), with an additional termination state in the same location as our initial state. This forms our set of states (S). The end state is added to avoid complications in the implementation of the algorithm. Our set of actions (a) lets the agent move to any of the states. The reward function ($R_t(s,s')$) is

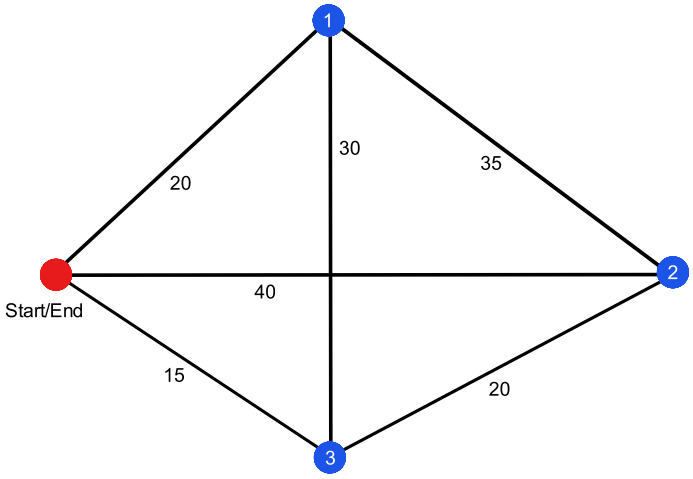

Figure 3.1: Example TSP

Figure 3.2: Example Traveling Salesman Problem (not to scale)

the euclidean distance between s and s' multiplied by -1.

For this MDP to work effectively with Q-learning, we need to make some adjustments to how our qTable works. Primarily, we need to prevent the agent from revisiting states, this is done by using a dynamic qTable. Once a state has been visited, the values for all actions that could lead to that state are set to -∞, preventing them from being selected by the algorithm. The actions that lead to the end state is initially set to -∞, but is set to its normal values when all other states have been visited.

The most important aspect of this MDP is how we control the reward signal. The way we have formed this problem causes the agent to only ever receive negative reward. This forces the RL agent to try and minimise its negative reward. Finding the shortest path through all the cities will result in the smallest negative reward.

During initial experimenting with Q-learning, we found a significant performance increase when initialising the qTable using the following heuristic taken from [7]. Q_0 is set to 1 / the average distance between all the cites multiplied by the number of cities.

3.2 Finding Hyperparameters for AS and Q-Learning

When using any algorithm in machine learning, finding optimal hyperparameters is key to optimising the algorithm's performance. It is also important when comparing an algorithm's performance that all algorithms have been given a suitable amount time

to find good values for their hyperparameters. This keeps our comparisons fair.

To find our hyperparameters we will conduct a grid of the hyperparameters on the Oliver30 benchmark TSP problem. The optimal hyperparameter values are different on every problem. However, it is unrealistic to conduct a hyperparameter search on all of our problems, so Oliver30 has been chosen to select our hyperparameters. Oliver30 uses the location of 30 USA cities for its cityscape. It has been frequently used as a benchmark in TSP research, and its optimum path is known (423.7) so we will be able to better judge how optimise our algorithms are.

It is also worth nothing that as the algorithms are non-deterministic, we have to conduct multiple trials with each hyperparameters pair and then we take an average of the results. For our grid searches we average results over 50 trials.

3.2.1 Q-learning

We are going to use Q-learning to try and solve the MDP we described in 3.1.

For our initial grid search in q-Learning we conducted trials with α (learning rate) $\in$ [1,0.1] with interval of 0.1 and γ (discount factor) $/in$ [0.1, 1] with interval of 0.1. ε (exploration rate) has been set to constant 0.05, and each trial ran for 500 episodes. The best path found in these episodes is the result of the trial, where shorter paths are better solutions. Figure 3.3 shows the results of the grid search.

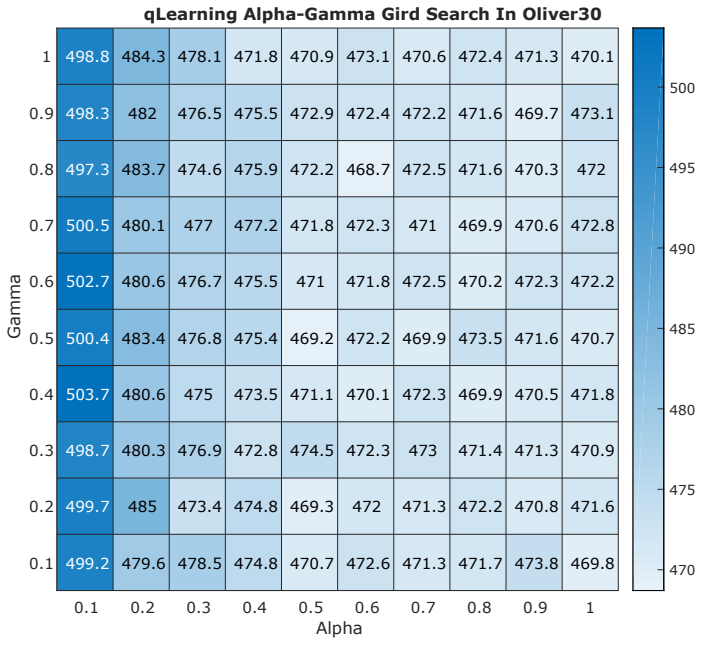

Figure 3.3: Heatmap of Alpha-Gamma search for Q-learning in Oliver30

The heatmap shows very little variation across the γ axis. This raises questions about Q-Learning ability to look far forward in this domain. In contrast, a pattern can be seen across the α axis. Smaller α values require more updates to impact the Q-table enough for the algorithm to choose different actions, this wastes a few iterations and gives worse performance. We can see that when α is high enough to effect action selection immediately there is little change in raising α further.

The low point on the heatmap is found at α 0.6 and γ 0.8, so these values will be used in further hyperparameter searches.

We then explored exploration rates, it is important to find a good balance between exploration and exploitation. We have introduced a new parameter here in λ. λ is the decay rate of of ε. After each episode ε is updated by $\varepsilon = \varepsilon.(1-\lambda)$ so at time step (t) $\varepsilon = (1-\lambda)^t$. ε is often needed to converge to optimum solutions in reinforcement learning problems, so we are curious to see if it will have an affect in this domain.

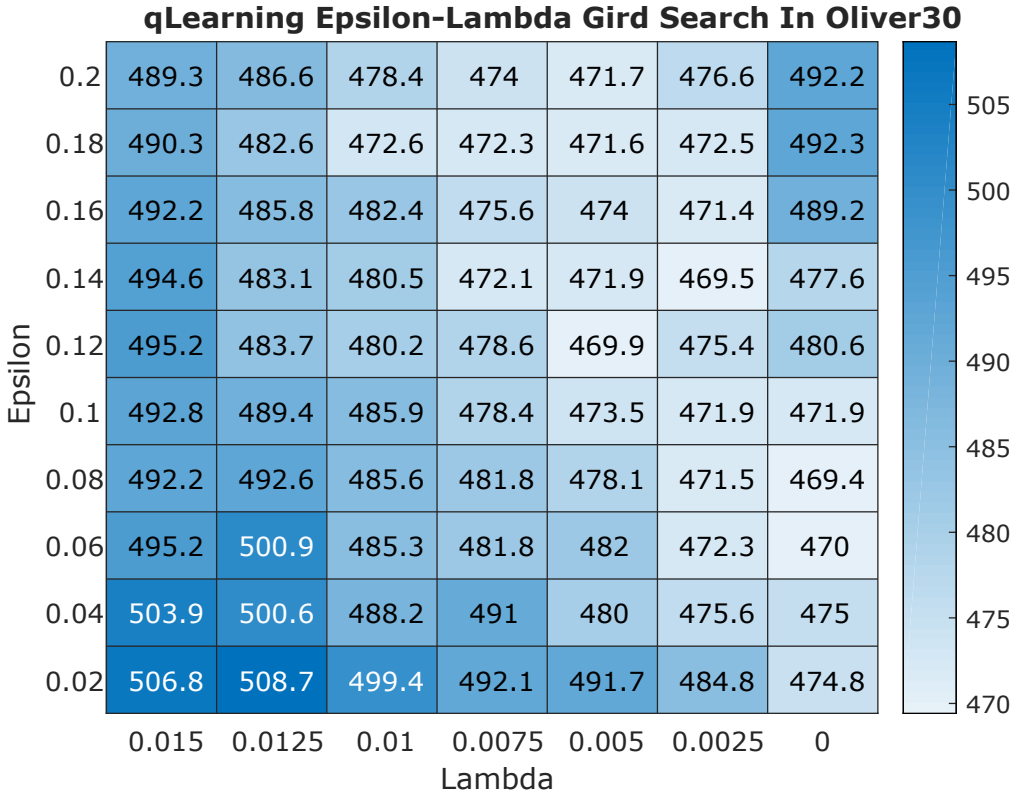

Figure 3.4: Heatmap of epsilon-lambda search for Q-learning in Oliver30

The optimal value of ε is around what we would expect to see. At 0.08 we would expect the agent to take 2.4 exploration steps in an episode on a TSP with 30 cities. This is low enough that many runs will not take any exploration steps (so will follow

the current expected best path), but there will also be many runs that will explore and learn more about the environment. It was also interesting to see that λ had the optimal value of 0 meaning ε will be constant through out the trial. This is likely due the fact Q-learning hasn't been able to find the optimal solution to this TSP. It is possible the ε decay would help speed up learning in smaller problems.

3.2.2 Ant System

We will be using the Ant System (AS) algorithm described in section 2.2.2 for our comparison to Q-learning. We will be using the 'best solution' update (equation 2.5) and $\Delta\tau_{sa}^{best} = \frac{1}{L}$, where L is the length of the best tour found by the ants on that iteration.

Comparing AS and Q-learning comes with a parameter that can be hard to decide on. That is the number of ants. As Q-Learning does not rely on a population, it could be argued that it should be compared to AS when it only uses 1 ant. We consider this an unfair advantage towards Q-Learning, as one of the key principles of AS is its use of a population. It is also considerably cheaper to run a single ant tour than it is a single Q-Learning tour. We decided that it will be fair to compare the two algorithms in a situation in which they have both conducted the same number of tours of the problem. As Q-Learning is run for 500 iterations in its searches, we will run AS for 100 iterations and give it a population of 5 ants. AS will only get to update its pheromone trail 100 times, but these updates will be stronger as only the best path found by the 5 ants on each iteration will be used in the update.

For our initial grid search in AS, we conducted trials with varying values for α (pheromone control) and β (heuristic control). Searching in these two parameters simultaneously is important as they control the ants exploitation vs exploration in their tour, as well as how decisive they will be in the pheromone exploitation. The grid search was conducted with $\rho = 0.95$.

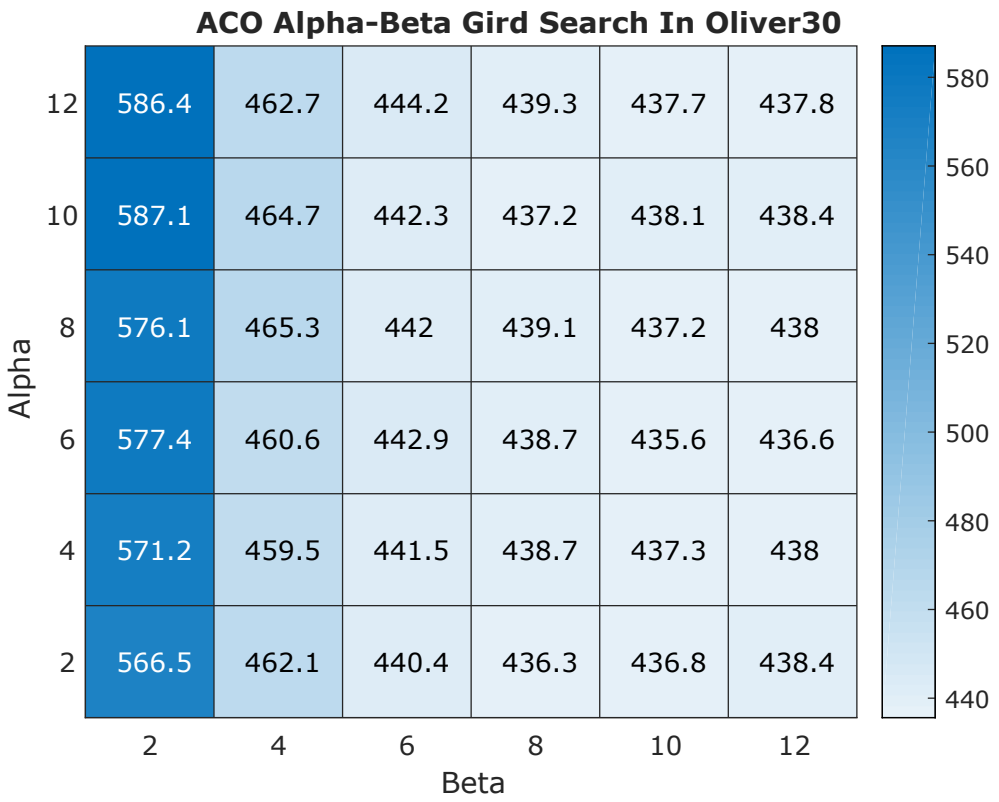

Figure 3.5: Heatmap of alpha-beta search for AS in Oliver30

We can see that far more variation is seen across the β axis. This is likely due to a numerical size difference between values in the η matrix being rather small, so they need to be greatly exaggerated before they have can have much of an impact on the decision process. α sees little variation across its axis, but it is possible that there is a small trend in the noise. We have decided on the optimal values of $\beta = 10$ and $\alpha = 6$.

With α and β decided on, we conducted a strip search using our optimal α β values to find our optimal value of ρ (pheromone decay rate).

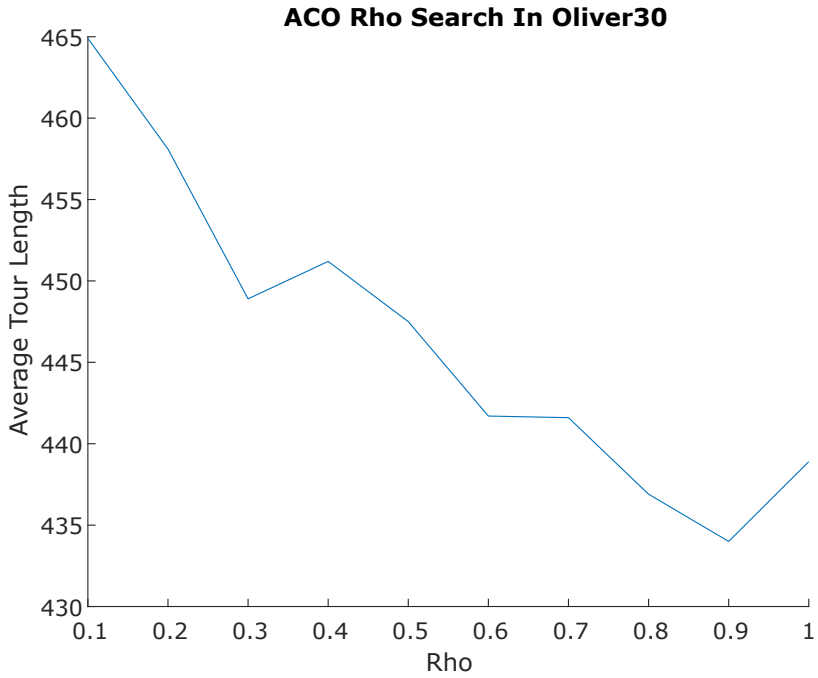

Figure 3.6: Line graph of rho for AS in Oliver30

The optimal ρ is at 0.9. This gives the algorithm quite a long memory. This could be due to ants often straying off the current best path as they explore the environment.

3.3 Benchmark Problems

In this section we will apply our algorithms to a set of TSPs with varying city counts.

3.3.1 Benchmark Algorithm

When comparing results it is important to have a benchmark to compare them to. This shows us that algorithms are working effectively. For our benchmark we will use a greedy algorithm that will always opt to travel to the closest unvisited city to its current location. We get the algorithm to generate this path from all starting locations and record the best path it finds. Such an algorithm can actually produce surprisingly good results considering how computationally cheap it is. It also can give us some insight on the downsides of a pure exploitation approach.

Algorithm 3 Greedy Algorithm

```
for city i in cities do
    Start Path at city i
    while Path Length < City Count do
        Travel to closest unvisited city
        Add new city to path
    end while
end for
return Shortest path found
```

3.3.2 Results

Q-learning used $\alpha = 0.6$, $\gamma = 0.8$, $\varepsilon = 0.08$, $\lambda = 0$, and ran for 500 iterations for all problems. AS used $\rho = 0.9$, $\alpha = 8$, $\beta = 10$, 5 ants, and ran for 100 iterations. We ran both algorithms on each problem 50 times for the minimum, mean, and standard deviation to be recorded.

Name	City Num	Greedy	ACO			Q-learning			Mean % Difference
			Min	Mean	STD	Min	Mean	STD	
UK	12	2716	1872.8	**1872.8**	0	1872.8	**1872.8**	0	0%
LAU	15	526.7	284.4	**284.4**	0	284.4	**284.4**	0	0%
wg	22	1241	798.0	**818.1**	20.7	828.1	849.0	5.7	3.8%
Oliver	30	762.8	423.7	**436.6**	7.3	454.4	466.6	4.4	6.9%
att	48	79913	37807	**38465**	313	39868	42223	1148	11.7%

Table 3.1: Results of Q-learning and AS on a set of TSPs

3.3.3 Discussion

Both the algorithms have achieved perfect results on the first two smaller problems, which does confirm some potential for Q-Learning in this area. However, as the problems grow in size we see the performance gap between the two algorithms increasing. We now ask ourselves why this might be?

An important aspect of a good optimisation algorithm is the avoidance of re-sampling, calculating the same solution many times is wasteful and prevents exploration. Examining the data used to produced table 3.1 we can see how many unique paths each

algorithm took on an average trial on each problem. The more unique paths, the less re-sampling done by the algorithm, and so in turn, the more exploration of the environment.

	Average Unique Paths		Percent
	AS	qLeaning	Difference
UK12	112.8	**145.5**	29.0%
wg22	**359.1**	348.0	3.2%
Oliver30	**472.1**	433.6	8.9%

Table 3.2: Number of unique paths walked by AS and TSP in results from 3.1

Table 3.2 does show us that AS has resampled less than Q-learning in the harder problems, meaning it has explored more. From this you could conclude that simply exploring more routes gives better performance. But, we know from figure 3.3 that if we increased ε we won't get an improvement in performance. So, why is this? We believe the issue, in part, lies in quality of Q-learnings exploration compared to AS's. Exploration in both algorithms is inherently random, but, AS selects randomly from a weighted policy, whereas Q-learning moves randomly in exploration steps.

Assume an algorithm thinks two actions could both have some potential for good results, for example the value of action $a_1 = 1$ and $a_2 = 9$. AS will pick a_1 90% of the time ($\frac{a_1}{a_1+a_2}.100$) this can let it explore but it is more likely to exploit. Q-learning function somewhat similarly, it can also see that both a_1 and a_2 have value, but, its greedy policy will always cause it to pick a_2 so it only has a chance of picking a_1 when an ε step is taken. If it turns out a_2 leads to a local minima and a_1 leads to the global minima then both of these algorithms are presented with the issue of preferring an option that will negatively impact their performance.

A problem like the simple one stated above is a basic deception. The more valuable option leads our algorithm away from the best solution. We expect the deception in the TSP problems has also played a role in the performance differences we have seen, and we will explore it further in the next section.

3.4 Deception

An important strength of any problem solving algorithm is its ability to deal with deception. Deception in a problem will lead an algorithm into local minima in the so-

lution. This prevents the algorithm from finding the global minima. Therefore, algorithms that can better avoid deception can have a greater chance of finding the global minima.

To study deception in the TSP domain we have devised a simple toy problem. The TSP is comprised of 4 cities (A,B,C,D) with distances $|AB| = |BC| = |CD| = |DA| = 1$, $|AC| = a$ and $|BD| = 2\sqrt{1-\left(\frac{a}{2}\right)^2}$. This creates a TSP with 2 different tour distances: $ABCDA$ with constant length $L_0 = 4$ and $|ACBDA|$ with the length $L_1 = 2+a+2\sqrt{1-\left(\frac{a}{2}\right)^2}$. In this problem the agent always starts and ends at A. They will also not be given the option to travel from A to D. This gives both the possible paths the same prior probability.

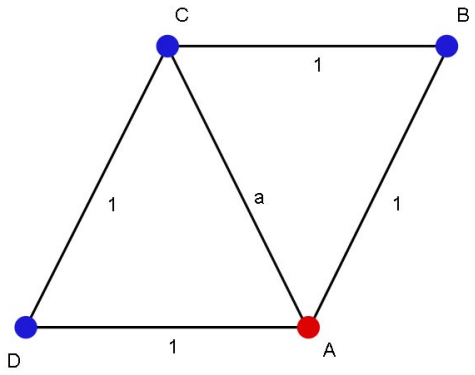

Figure 3.7: Toy Deception Problem (not to scale)

We will consider a in the range [1,0] which will cause L_1 to always be larger than L_2 but as a decreases to 0 the immediate reward of picking $|AC|$ (exploiting) will increase as $|AC|$ becomes shorter and shorter than $|AB|$.

3.4.1 Algorithm Examination

To beat this problem, first an agent needs to take the shorter path and then it needs to stick to it over the longer path. We can learn a lot about the how the algorithms will react to the problem by looking at the updates they conduct. We will examine how both algorithms update the first step over episodes.

3.4.1.1 In Ant Systems

$$p(\overrightarrow{AB}) = \frac{\tau_{AB}^{\alpha}\eta_{AB}^{\beta}}{\tau_{AB}^{\alpha}\eta_{AB}^{\beta} + \tau_{AC}^{\alpha}\eta_{AC}^{\beta}} \tag{3.1}$$

$$p(\overrightarrow{AC}) = \frac{\tau_{AC}^{\alpha}\eta_{AC}^{\beta}}{\tau_{AB}^{\alpha}\eta_{AB}^{\beta} + \tau_{AC}^{\alpha}\eta_{AC}^{\beta}} \tag{3.2}$$

Where:

$$\eta_{XY} = \frac{1}{|XY|} \tag{3.3}$$

We will now examine the first decision made in each episode of AS and see how it changes as the pheromone trail is updated. We will use the same relative hyperparameter setting that we used in section 3.3. For AS $\beta = 10$, $\rho = 0.9$, and we will set a to a low deception of 0.9 ($|AC| = 0.9$). First we will examine a case in which AS falls for the deception on every iteration (takes action $\overrightarrow{AC}$).

Iteration	Action	τ^{α}	η^{β}	Probability (1 Ant)	Probability (5 Ants)
0	$\overrightarrow{AB}$	1^6	1^{10}	25.85%	77.6%
	$\overrightarrow{AC}$	1^6	$\frac{1}{0.9}^{10}$	74.15%	99.8%
1	$\overrightarrow{AB}$	0.9^6	1^{10}	9.09%	37.9%
	$\overrightarrow{AC}$	1.11^6	$\frac{1}{0.9}^{10}$	90.90%	99.99%
2	$\overrightarrow{AB}$	0.81^6	1^{10}	2.97%	14.0%
	$\overrightarrow{AC}$	1.22	$\frac{1}{0.9}^{10}$	97.03%	99.99%

Table 3.3: Probability of overcoming deception decreasing as wrong path followed in AS

We can see that in AS the more the deception is fell for, the more likely it becomes that it will be fallen for again. This is some what offset, however, by having multiple tours before having to update its table. The probability is low of taking $\overrightarrow{AB}$, but the chances of it being taken by just one of the ants in an iteration are considerably higher (as seen in probability (5 ants)). As only one ant needs to take the best path for it to apply to the update, we can see the strength of a population to overcome deception. In turn, we will also look at a case in which AS does not fall for the deception (takes action $\overrightarrow{AB}$).

Iteration	Action	τ^{α}	η^{β}	Probability (1 Ant)	Probability (5 Ants)
0	$\overrightarrow{AB}$	1^6	1^{10}	25.85%	77.6%
	$\overrightarrow{AC}$	1^6	$\frac{1}{0.9}^{10}$	74.15%	99.8%
1	$\overrightarrow{AB}$	1.15^6	1^{10}	60.29%	99.01%
	$\overrightarrow{AC}$	0.9^6	$\frac{1}{0.9}^{10}$	39.71%	92.03%
2	$\overrightarrow{AB}$	1.29^6	1^{10}	84.76%	99.99%
	$\overrightarrow{AC}$	0.81^6	$\frac{1}{0.9}^{10}$	15.24%	56.25%

Table 3.4: Probability of overcoming deception increasing as wrong path followed in AS

We can see how dramatically unlikely it becomes for the deception to be fallen for as correct moves are taken. From both these examples we can expect AS to beat this small deception in most runs. This also gives us an intuitive understanding of why using iteration best to update the pheromone trail can be more powerful than letting every ant update the trail like in a standard ant system.

3.4.1.2 In Q-Learning

Examining Q-learning's first action in each iteration is a bit different. As Q-learning uses ε-greedy action selection, as given in equation 2.2.3. We will examine the Q-Values in first state over a few iterations. As the qTable is initialised to all equal the same value (by equation **??**) we will first examine a case in which it falls for the deception in t = 0. $\alpha = 0.6$ and $\gamma = 0.8$ as set in section 3.3. In this case we will assume $\varepsilon = 0$.

t	0	1	2	3	4	5	6
Q(A,B)	0.4487	**0.4487**	-0.2051	**-0.2051**	-0.7805	-0.7805	-0.7805
Q(A,C)	0.4487	-0.1452	**-0.1452**	-0.3827	**-0.3827**	**-0.4777**	**-0.5157**

Table 3.5: Q-learning's Q-values from first action of toy deception problem.

We can see how the Q-learning agent oscillates between taking and not taking the deceptive route. On one hand this is good as it ensures that the shorter route is taken at least once no matter the deception level. However, we can also see that over time the agent will take the deceptive route more and more often as the negative reward from taking $\overrightarrow{AC}$ is less than than that of $\overrightarrow{AB}$. This will likely cause issues in larger deceptive problems as it can only be corrected by ε actions.

3.4.2 Empirical Examination

We will now examine experimental results from Q-Learning and AS's performance in the toy problem as the deception increases in the problem. We will be running the algorithms for 50 trials as the angle *DAB* increases from 120° to 180° with an interval of 1° (this is the same as distance $a \in [0, 1]$). Both algorithms are given 10 iterations each trial, we will record the best path of each trial, as well as the average path length of each trial. These results will be averaged across the 50 trials to produce the plot. We used the same hyperparameters as used in section 3.3.

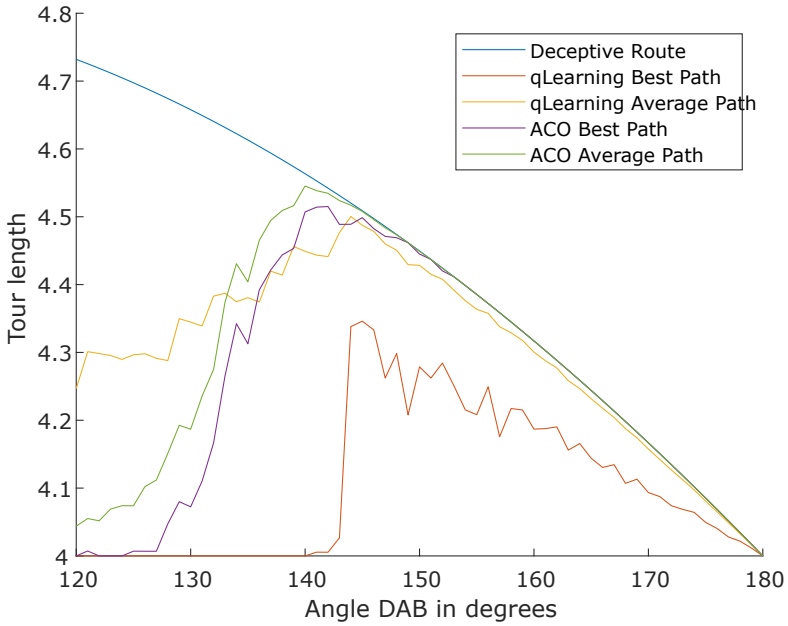

Figure 3.8: Results on toy problem as deception increases

As expected, Q-learning consistently finds the best path until the deception grows too large. At this point ε action selection and random chance on the initial step taken affect whether the short path is taken, which gets averaged over the trials. We can see that AS is more consistent in its paths, it gradually falls for the deception more and more often as the length of a decreases.

These results appear to show that Q-learning is more capable in problems with high deception. However, we should keep in mind that this is a very simple problem. Luckily, it can be easily expanded to have as many cities as we see fit. We can 'stack' more and more city pairs on top of the toy problem, presenting the agent with the problem at angle *DAB* multiple times.

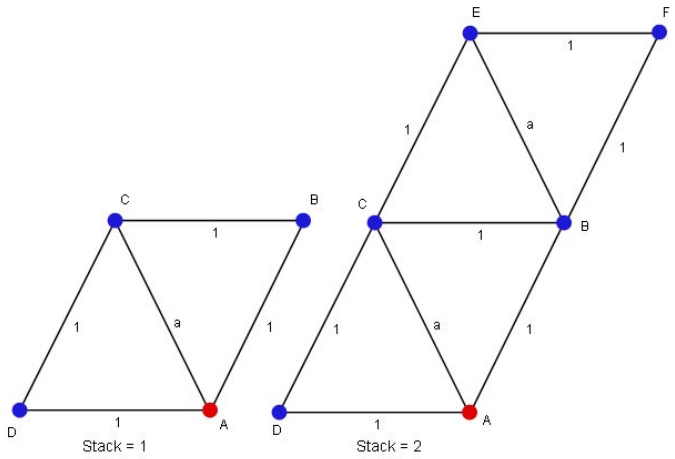

Figure 3.9: Stacked Toy Deception Problem

We can see that in the stack angle *DAB* and *CBF* will be the same as *a* changes in length. We also have a new 'most deceptive route' (not to be confused with worse possible route) depending on the stack size. The shortest route in the problem will still be the the number of cities in the problem ((stack + 1)*2). As the stack increases we also need to the give the algorithms more iterations to solve the TSP, so, the iteration count for solving a stack for AS is given by stack*10, and Q-learning is given the AS iteration count multiplied by the ant count (in this case 5).

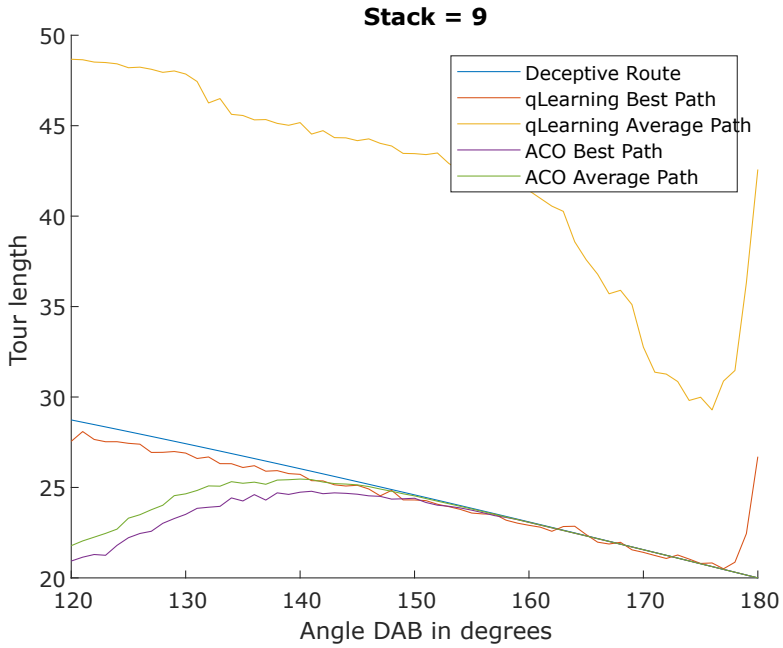

Figure 3.10: Results on toy problem with stack of 9 as deception increases

We can see that Q-Learning is let down by its inability to effectively solve TSPs. From our experimenting, we have concluded that this is due to the way Q-Learning has to process the reward signal. The information available a single step in the future is not enough to make good long term decisions. In RL this is normally countered by the temporal difference learning controlled by γ (discount rate), however, as we saw in figure 3.3, adjusting the value of γ makes little difference to the performance of the algorithm. So, without the ability to make decisions based on long term gain and only short term gain the algorithm often finds itself in local minima as it exploits only ever traveling to close cities.

AS makes all its updates based on the end result of an iteration. Effectively, this gives equal value to every decision that was made in making that path. This is useful in practice, but, it is unlikely that all decisions being made on a path are as good as each other (Unless it is the global best path). Ideally, we would give more value to decisions that lead to more value. We will explore this idea more in section 3.5.

3.4.3 Examination of AS hyperparameters in deception

It is worth noting that the hyperparameters used here are not exactly a fair representation of these algorithms ability to deal with deception. We can easily argue that because these are the best hyperparameter values we found for solving TSPs, then we should use them on these problems. However, if you knew that the problems you were applying the algorithms to were highly deceptive you could adjusts your hyperparameters to combat the deception.

The parameter that causes the most deception in ACO is β (exploitation control). This value affects how decisive the ant is in it's exploitation of the η huristic. Larger β values exaggerate the size differences on η matrix leading to far greater probabilities of selecting exploitative actions presented by the eta heuristic. We know that large values of β are useful in standard problems, but we expect to see different results in this more deceptive problem.

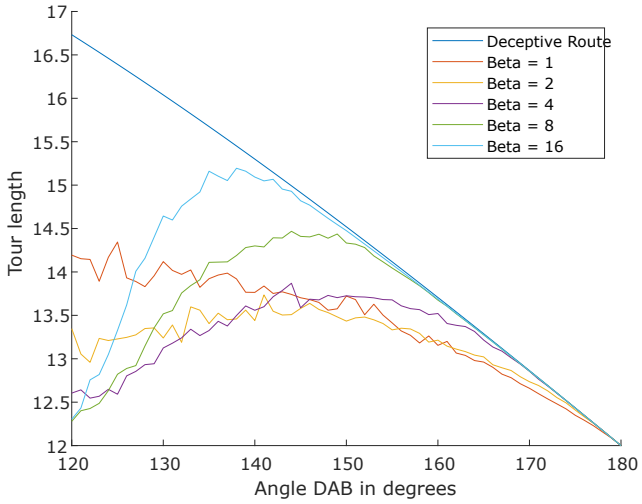

Figure 3.11: Average tour lengths on stacked toy problem at different beta values

Interestingly, the highest β values have the best performance on the smallest deception, but quickly reduce in performance after about 130° (a = 0.845). It would be reasonable to assume that smaller deceptions are more prevalent in the real world, which could be why our hyperparameter searches give high β values.

Another issue we would like to address is the ρ parameter. As we know, the pheromone decay rate can let the algorithm forget about poor decisions that the algorithm has made in the past by reducing the pheromone across the entire pheromone matrix at every iteration. This gives the affect of only letting often walked trails survive. We are curious to find out what effect values of this parameter will have on our deception problem. On the one hand, having a very short memory will be useful for forgetting about times the algorithm has been deceived, however, the opposite could also be true for situation in which the best path isn't often being walked. If our optimum path decayed away to nothing it would not be walked on again.

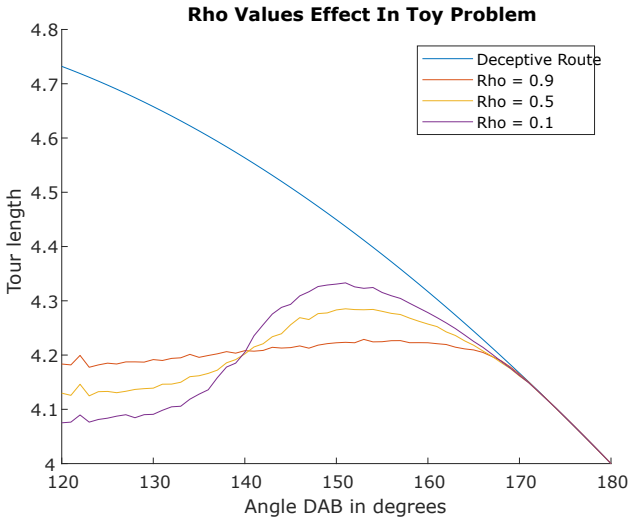

Figure 3.12: Average tour lengths on toy problem at different rho values

From figure 3.12 we can see that the value of ρ has an interesting affect in this problem. Large ρ produces consistent results, it is likely that once's a path is chosen it is stuck to throughout the trial. However, the lower ρ values can explore more, finding the shorter path more often in the lower deception cases. In the higher deception cases, lower ρ values are less effective because as the more deceptive path is travelled more often, the better path is unable to persist through the pheromone decay.

3.5 Ant-Q in TSP

At the end of section 3.4.2 we discussed how Q-learning and AS have to apply their reward signal. Q-learning updated based on imitated reward, as well as a value obtained in the next state on each timestep. ACO updated states equally at the end of an iteration, with reward based on the performance of the iteration. Ant-Q can be seen as a combination of these two approaches. Ant-Q [7] functions similarly to ant colony systems (section 2.2.3), in fact the only change is the value we use for $\Delta\tau_{sa}$ in the local update. In ANT-Q it is set it $\gamma \cdot \max_a \tau(s,a)$. This is very similar to update function in Q-learning, the difference being Ant-Q doesn't know the reward value for its local step yet. The addition of reward is delayed until the global update, where $\Delta\tau^d_{sa}elta$ has the same value we gave it section 3.2.2 $(1/L)$.

3.5.1 Comparison of AS and Ant-Q

From testing we have found that Ant-Q requires more resources to yield good results. Running it with the same restrictions as our experiments in section 3.4.2 we found the ant system algorithm to yield better results, showing AS is still a good choice in situations with limited resources. To better examine Ant-Q's potential, we decided to compare AS and Ant-Q in a situation with slightly more available computational resources. We increased the iteration count to 500 and let the ant population count be half the number of cities in the TSP on both algorithms.

Name	City Num	Greedy	AS			Ant-Q			Mean % Difference
			Min	Mean	STD	Min	Mean	STD	
Oliver	30	2716	423.7	435.8	0	423.7	**430.6**	8.99	1.2%
dj	38	526.7	6672.9	6672.9	0.12	6659.4	**6663.9**	1.91	0.13%
att	48	1241	37906	38222	133.5	37656	**37812**	393.3	0.25%
KN	57	762.8	13836	13995	83	13567	**13767**	167.1	1.7%

Table 3.6: AS and Ant-Q results on TSPs. Averaged over 50 trials.

We can see a small but consistent improvement in Ant-Q, confirming the ability of the delayed reinforcement. We believe the increased performance comes from exploration in Ant-Q. The local update in Ant-Q adds a TD learning element to the algorithm and also encourages the ant population to construct different solutions in each iteration because the τ matrix is updated as ants construct solutions. This has the side effect of making the algorithm somewhat unstable, as can been seen in the STD in the results. Also, this local update does mean that the ant tours need to be conducted in order. This prevents the possibility of running Ant-Q with parallel computing. Although it is outside of the scope of this project, we would expect AS to be able to produce better results than Ant-Q in the same time period if it had access to parallel computing.

To back up our claim that the ants in Ant-Q explore more than in AS we have run both algorithms with 5 ants on the stacked toy deception problem.

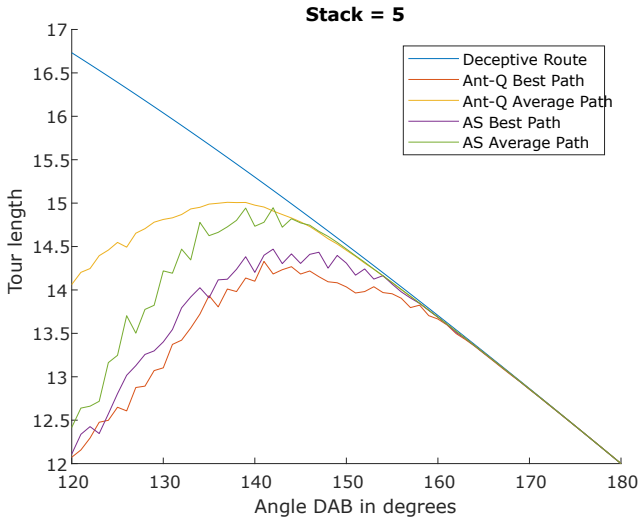

Figure 3.13: Minimum and Mean performance of AS and Ant-Q on the stacked toy deception problem.

We can see that Ant-Q has found on average slightly better solutions than AS because of its better exploration. We can also see, however, that the mean performance of Ant-Q is worse. Each iteration the local updates encourage the ants to explore areas that have not yet been visited in that iteration. This strategy is somewhat risky, if the first few ants that construct their solution on the pheromone trail (before it is updated by local updates) fail at reproducing the previous best, and other ants also do not find a good solution, the algorithm risks undoing its progress to a good solution. We believe this is the reason for the higher mean path in the deception problem, and for higher standard deviations in table 3.6.

It is worth mentioning that we have not achieved as good results as in the paper introducing Ant-Q [7] this could be because we ran our experiments for more trails. Ant-Q is a somewhat unstable (see the standard deviation in table 3.6) algorithm that occasionally produces poor results, so, running it for only 15 trails is less like to include this poor results as running it for 50 trails like we did.

Chapter 4

Gridworlds And Dynamic Problems

4.1 Cliff Problem

The cliff problem is used in [12] to compare off-policy (SARSA) and on-policy (Q-learning) methods. The problem tasks an agent with travelling from the start to the end point of the gridworld, while receiving -1 reward at each step to motivate to take the shortest route. However, there is also a cliff between the start and end states that, if transitioned to, will set the agent back to the start and gives it -100 reward.

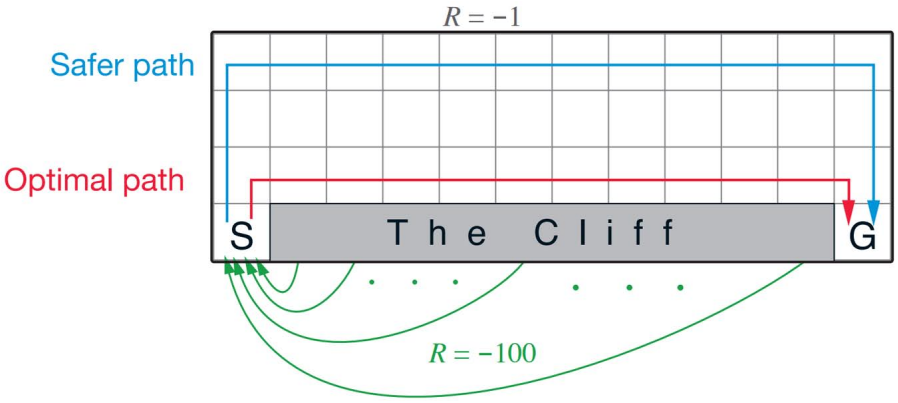

Figure 4.1: The Cliff gridworld problem from [12]

When applying SARSA and Q-learning to this problem, it is found that Q-learning will learn the optimal path, while SARSA will learn the safer path. Q-learning is able to learn the optimal path because its TD difference calculation only takes into account the best possible next step ($\max_A Q(s',A)$). It doesn't consider the possibility of falling off the cliff, so it can't learn to avoid it. SARSA, however, uses its policy in the TD difference calculation ($Q(s',a')$), letting it take into account the possibility of falling

off the cliff, and so it can learn the safer policy. Interestingly, it turns out that the safer path does yield better overall reward due to the high negative reward given for falling off the cliff.

We want to apply our ACO algorithms to this problem to see how they will behave. To do this we will need to make some changes to how the problem functions. Our ACO algorithms don't facilitate negative reward in the same way the RL algorithms can, so we will remove the -100 punishment for falling off the cliff. Our algorithms will still want to avoid the cliff, because the cliff still sends the agent back to the start and RL agents are still given -1 reward for each transition. To motivate our ACO algorithms to take the shortest route $\Delta\tau_{sa}^{best}$ = (1 / the best path length). This will also make them want to avoid the cliff, as longer paths will give lower reward.

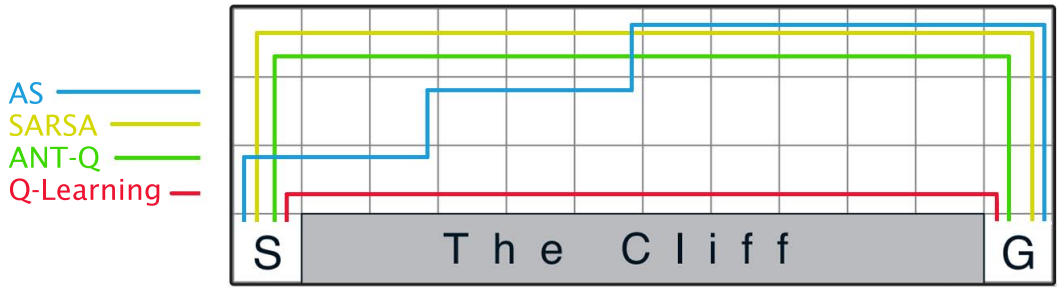

Figure 4.2: Greedy routes from AS, SARSA, ANT-Q, and Q-Learning after running in a modified Cliff problem for 500 iterations

Unsurprisingly, Q-learning and SARSA have learnt the optimal path and safer path respectively. SARSA still accumulated less negative reward (over the 500 iterations we ran the algorithms for) showing safer paths are still better in this problem. More interesting results are seen in the paths the ACO algorithms have learnt. AS steps further and further from the cliff as it moves towards the goal. A more efficient way of moving away from the cliff is seen in SARSA's path, it is the same length as AS's path but immediately moves the agent away from danger. AS's moves like this because of how it handles reward. Staying close to the cliff gives its the best reward update across the path (1/PathLength), but the longer it stays by the cliff, the more likely it is to fall off. Occasionally an ant will take the optimal path and lay strong pheromone next to cliff, however, this path can't be reliably followed without falling off the cliff. So, after a couple of steps, the ants that have moved away from the cliff become more likely to make it to the goal and thus get their pheromone used in the τ update. This shows a rare situation in which a larger population doesn't necessarily improve the end

solution. We found safer paths were taken when using a single ant then using many due to the fact a large population is more likely to get the optimal path for the best path update.

We were surprised to see the path ANT-Q used. Given how it uses the Q-learning update function, we expected it to copy the path Q-learning uses. We expect the reason it stays away from the cliff is how its reward signal is handled. Like AS, it will avoid the cliff edge but the TD updates to τ cause it to exit the start state and move away from the cliff immediately.

4.2 Shortcuts In Gridworlds

The cliff problem is an example of a static environment, at no point does the environment change. In contrast, dynamic problems can change their environments during an episode or trial. Due to the nature of dynamic environments they are often harder for algorithms to learn, as they often require more exploration or prolonged memory. We are going to test these aspects of our algorithms with a dynamic gridworld problem. The agent will conduct 250 episodes in the gridworld with the goal of getting from the initial state to the termination state. At episode 51 a shortcut will open, reducing the distance to termination state dramatically.

Q-Learning will be given -1 reward at each transition, and AS will have $\Delta\tau_{sa}^{best}$ = (1 / the best path length). These both motivate the algorithms to find the shortest routes.

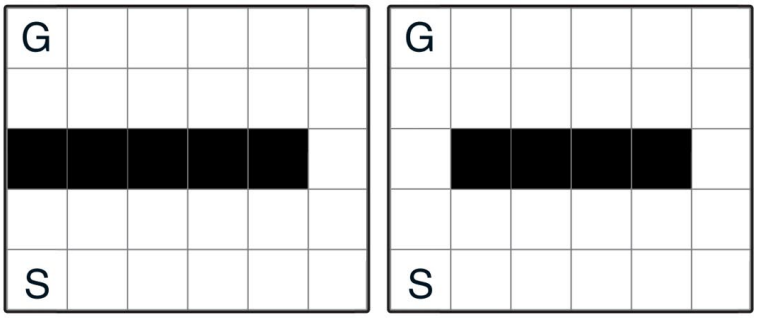

Figure 4.3: Dynamic shortcut girdworld. Left: shortcut close. Right: shortcut open.

This problem is split into 3 phases:

- Before Shortcut (Phase 1): Iteration 1 to 50. The algorithm needs to learn the initial route.

- Shortcut Open (Phase 2): Iteration 51 to 200. The algorithm needs to discover and exploit the shortcut.
- Shortcut Closed (Phase 3): Iteration 201 to 250. The algorithm needs to relearn or revert to the original route.

We should expect algorithms with better exploration to do well in the second phase, but this exploration might negatively effect their performance in the other two phases. We have produced a heatmap of a hyperparameter search in the problem to help investigate this.

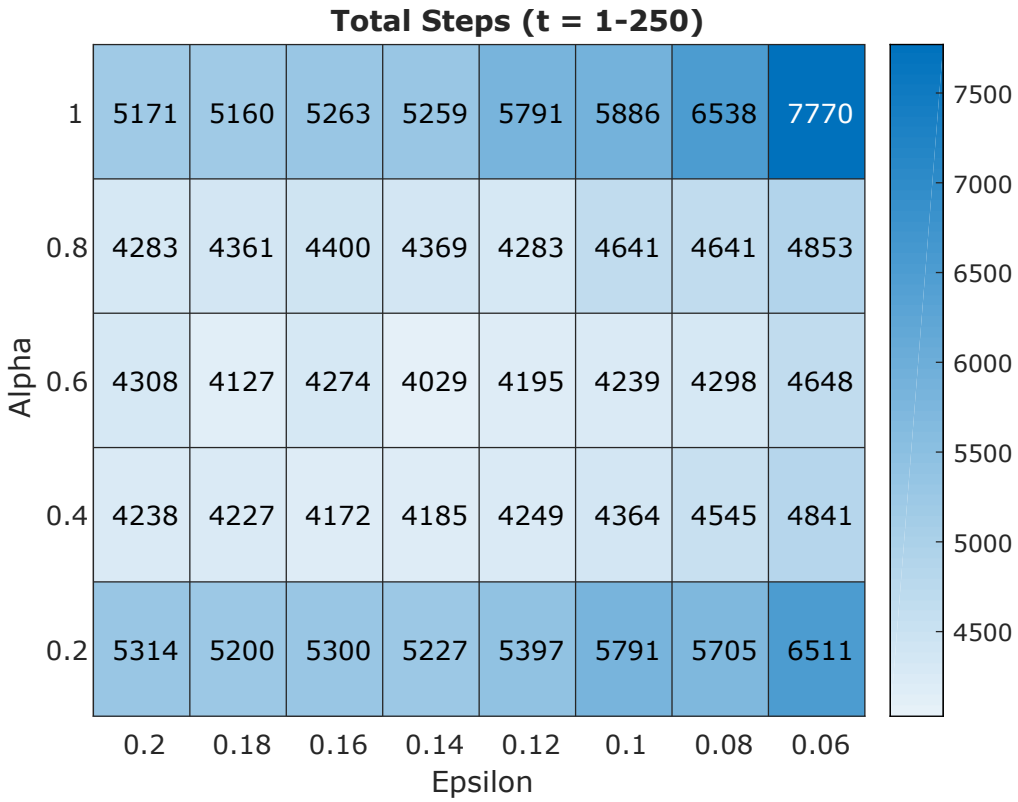

Figure 4.4: Heatmap of Alpha-Epsilon search with Q-Learning in shortcut gridworld problem (gamma = 0.7)

We can see a clear global optima at $\varepsilon = 0.14$ and $\alpha = 0.6$. However, when we examined the results of of the phases individually, we saw some good performance at phase 2 with $\varepsilon = 0.14$ and $\alpha = 0.6$. We decided to investigate this.

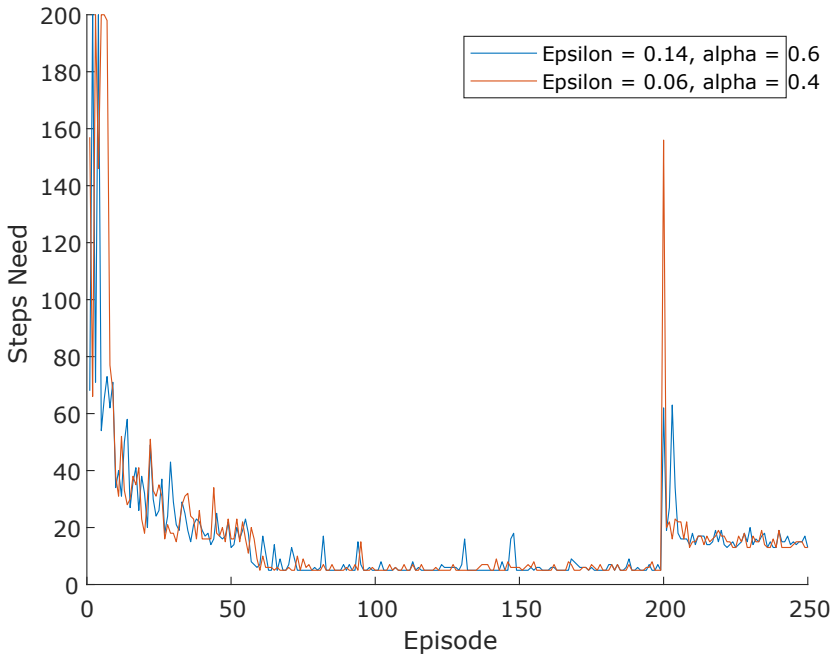

Figure 4.5: Plot of the Q-learning on shortcut problem with different hyperparameter settings

We can see that the higher learning and exploration rates are useful when initially learning the longer path, and again when the path needs to be re-learnt. However, having a high exploration rate is detrimental when the algorithm needs to exploit the shorter path. We can see the lower ε value has considerably better performance in phase 2, as it deviates from the short path far less. Its lower learning rate appears to also stop it taking the original, longer, path when exploration causes it to deviate from the shorter path.

We can see how this dynamic problem raises issues for our algorithms. The hyperparameter settings that are optimal for phase one and three of the problem are not optimal for the middle phase. A possible way around this issue could lie in a population. We have seen that lower exploration rates give better performance in the middle stage. We would be able to use lower exploration rates if we have a population to search with. As we have already seen in deception in TSP, a population can find a good solution even when the individual chance of finding it is low. To test this we will now compare results on this problem between Q-Learning and the Ant-Q algorithm.

Q-Learning hyperparameters were set as $\alpha = 0.4$, $\gamma = 0.7$, and $\varepsilon = 0.14$. Setting hyperparameters for Ant-Q proved difficult and mainly relied on trial and error and intuition. In the end we settled on $\alpha = 0$, $q_0 = 0.9$, $\gamma = 0.3$, $\rho = 0.3$, and $\phi = 0.3$. We

also used a population of 5 ants.

	Episodes 1:50	Episodes 51:200	Episodes 201:250	Episodes 1:250
Q-learning	2170.2	1014.6	853.1	4007.3
Ant-Q	1106.9	1150.9	1028.6	3546.9

Table 4.1: Q-learning and Ant-Q results on shortcut problem

We can see a large performance difference between Q-learning and Ant-Q. Interestingly, the performance increase was not seen in the middle phase as we expected. This is likely due to the q_0 value being larger than expected for the algorithm to remain stable. However, the population is able to exploit the optimal path in first stage far better than Q-Learning. Comparing Q-learning and Ant-Q like this has the same issue we faced in the TSP comparisons. Ant-Q's population lets it update more often than Q-learning. In TSP we addressed this by letting Q-learning run for more iterations, but this isn't an option in the shortcut problem. Our solution to this is to use Dyna-Q.

Dyna-Q uses the same update equation as Q-learning but it also uses 'model-learning' to let it learn about its environment between interactions. It does this by keeping a model of state transitions it has seen and the reward for these transitions. Then, between iterations, it can run 'planning' iterations that updates the Q-table with simulated steps in the environment using the model to know the outcome of these steps.

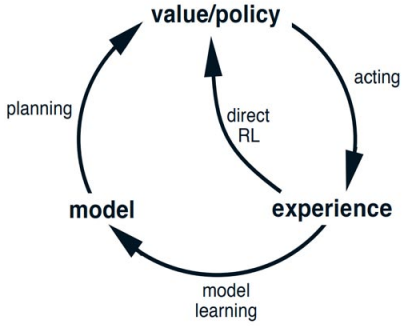

Figure 4.6: Model-learning illustration from [12]

We conducted a hyperparameter search in Dyna-Q to see how it will compare to what we saw in Q-learning. For the search we set γ to 0.7 and used 30 planning steps.

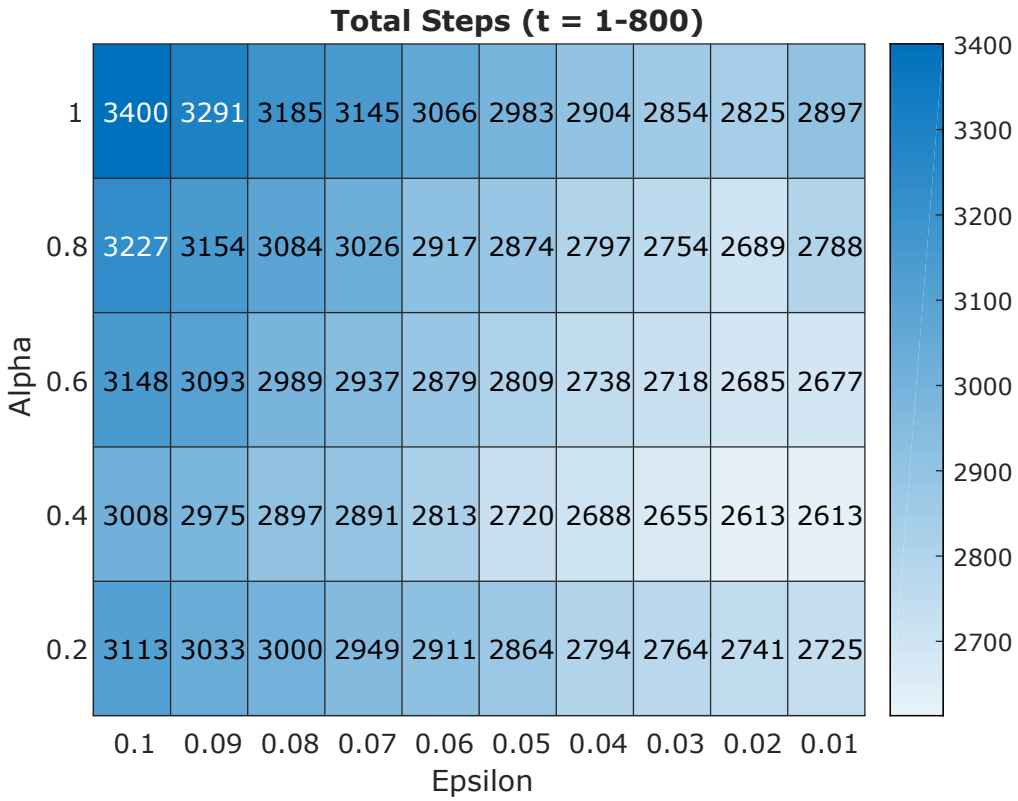

Figure 4.7: Heatmap of Alpha-Epsilon search for Dyna-Q in shortcut problem

We can see that, with the addition of planning steps, Dyna-Q is able to use a far lower exploration rate than Q-learning. Q-learning needed a high exploration rate to quickly learn the optimal path in phase 1 and 3. This caused poor performance in phase 2. Dyna-Q can use planning steps to quickly learn the optimal paths in phase 1 and 3 and so it can use a far lower epsilon that will give far better results in phase 2.

	Episodes 1:50	Episodes 51:200	Episodes 201:250	Episodes 1:250
Dyna-Q	1285.1	926.5	810.1	3021.7

Table 4.2: Results of Dyna-Q with 5 planning steps on shortcut problem

Comparing Dyna-Q's results using 5 planning steps to table 4.1 shows us a dramatic increase in performance to regular Q-learning, as well as an improvement over Ant-Q. However, both Dyna-Q and Ant-Q can be scaled up very easily by simply increasing the number of planning steps and the population size in both algorithms respectively.

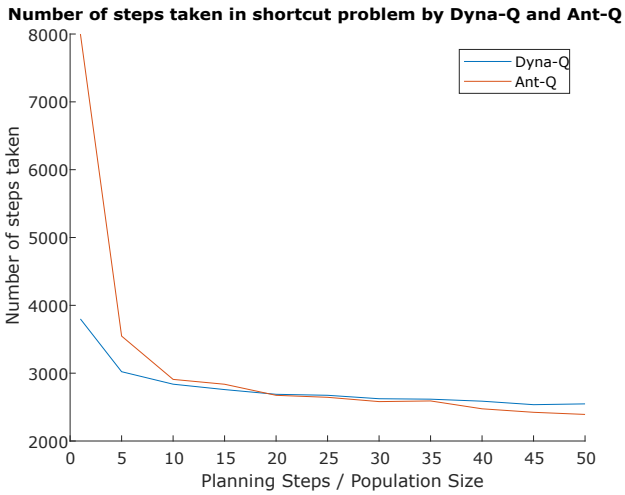

Figure 4.8: Dyna-Q and Ant-Q performance on shortcut problem as they are scaled up

Dyna-Q begins to plateau before Ant-Q, which keeps seeing an increase in performance as more ants are added. We would expect Ant-Q to also begin to plateau at some point, but, as we run our experiments for multiple trials, continuing past 50 proved too time intensive.

To help us try and understand why Ant-Q obtains better performance as ants are added over Dyna-Q getting more planning steps and how each algorithm behaves in the problem, we have produced a plot of the steps taken in each algorithm at each each phase.

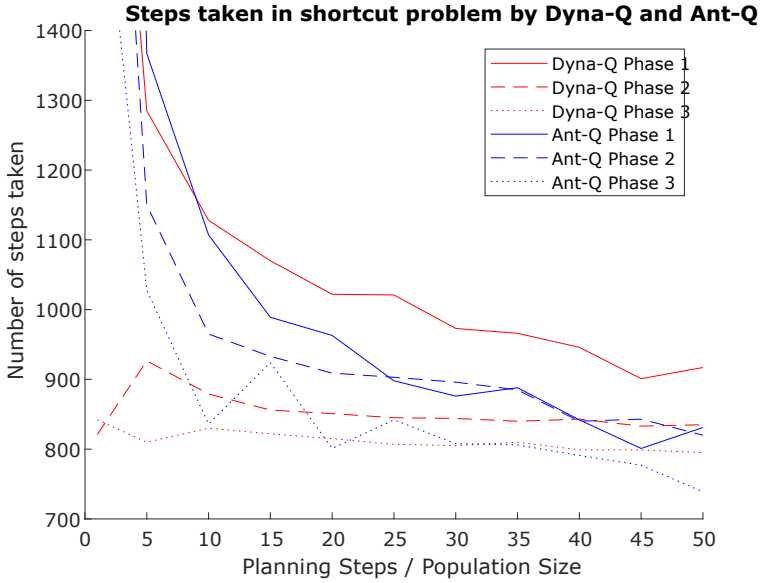

Figure 4.9: Plot of Dyna-Q and Ant-Q performance on each phase of the shortcut problem as they are scaled up

Ant-Q seems to have the most difficulty in phase 2 (shortcut opens) this is likely caused by how long it takes to shift to the shorter path. Once the shortcut is discovered, the algorithm needs to start exploiting it as soon as possible. Dyna-Q doesn't have an issue with this because its planning steps can very effectively reinforce the new shorter path. This is also seen in its consistent performance in phase 3, it can quickly relearn the old route.

The largest difference between the two is the performance in the first phase. Typically Q-Learning can use decaying ε to speed up learning at the start of the problem, but as this is a dynamic problem it would be unwise to decay the exploration rate. We can see that the planning steps do a good job of speeding this up, but the population of Ant-Q proves to be better. A large portion of the steps taken in the first phase are made in the first few episodes as the agent learns the initial route. Using a population to do this gives better results because all the bad solutions are filtered out by the best solution. If Dyna-Q struggles to find the goal state in one of the early episodes it has a large impact on the end results.

There is an argument for Dyna-Q performing better on a larger problem, e.g. if the gridworld was larger in size. In a small problem the chances of constructing the perfect solution randomly are higher, and so having multiple opportunities to do this increases the chances of randomly finding the solution. In our problem the perfect route in phase 1 is 14 steps, with 4 actions available at each step. This means it is a 1 in 38416 to take the perfect route randomly. We believe that this makes the problem large enough to conclude that Ant-Q will also be able to perform well on larger problems.

Chapter 5

Conclusions

To end this paper we give a conclusion of our findings and some thoughts on future work.

We feel that we have given good insight into how ACO algorithms work and how their hyperparameters will affect how they will function in a problem. We also think that we have shown our understanding of reinforcement learning, and how their parameters impact their performance. We have shown why ACO is such a popular algorithm in TSP, and that TSP is an environment that is not well suited for basic reinforcement learning algorithms. Although there is some potential there that might be seen with more advanced algorithms, we were unable to uncover it with Q-learning and eligibility traces that we ended up not discussing in this paper.

We believe that we have shown ACO's ability to be used in MDPs. The shortcut problem is advanced enough that the good performance of Ant-Q could justify further experiments of ACO and other metaheuristic algorithms in even more advanced MDPs. We did try to get Ant-Q working in a pole balancing control problem, however, due to instability of the algorithm the results were poor and any success was hard to replicate, but we do believe it would be possible.

Bibliography

[1] Haider Abdulkarim and Ibrahim Fadhil Alshammari. Comparison of algorithms for solving traveling salesman problem. *International Journal of Engineering and Advanced Technology*, ISSN:2249 8958, 08 2015.

[2] Bilal Abed-alguni. Action-selection method for reinforcement learning based on cuckoo search algorithm. *Arabian Journal for Science and Engineering*, 43, 10 2017.

[3] Bilal Abed-alguni and Bilal Abedalguni. Bat q-learning algorithm. *Jordanian Journal of Computers and Information Technology*, 3:51, 04 2017.

[4] Marek Antosiewicz, Grzegorz Koloch, and Bettina M. Kaminski. Choice of best possible metaheuristic algorithm for the travelling salesman problem with limited computational time: quality, uncertainty and speed. 2013.

[5] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[6] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, Feb 1996.

[7] Luca M. Gambardella and Marco Dorigo. Ant-q: A reinforcement learning approach to the traveling salesman problem. pages 252–260. Morgan Kaufmann, 1995.

[8] S Goss, Serge Aron, Jean-Louis Deneubourg, and Jacques Pasteels. Self-organized shortcuts in the argentine ant. naturwissenschaften 76: 579-581. *Naturwissenschaften*, 76:579–581, 12 1989.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[10] Ndedi Monekosso and Paolo Remagnino. Phe-q : A pheromone based q-learning. In Markus Stumptner, Dan Corbett, and Mike Brooks, editors, *AI 2001: Advances in Artificial Intelligence*, pages 345–355, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[11] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[12] Richard S Sutton and Andrew G Barto. *Reinforcement learning : an introduction*. Cambridge, Massachusetts : The MIT Press, [2018], 2018.

[13] Gabriela erban and Camelia Pintea. Heuristics and learning approaches for solving the traveling salesman problem. *Studia Universitatis Babe-Bolyai. Informatica*, 49, 01 2004.