APPROXIMATION ALGORITHMS FOR THE TRAVELING SALESMAN PROBLEM

Andy Gnias
Temple University
Philadelphia, PA
andy.gnias@temple.edu

ABSTRACT

The Traveling Salesman Problem (TSP) is one of the most studied NP-complete problems. Approximation methods, such as Ant Colony Optimization, have become state of the art in developing near-optimal solutions in a reasonable amount of time. Other methods, such as Reinforcement Learning, have been attempted with some success but not fully explored. In this paper, Ant Colony Optimization and Reinforcement Learning algorithms were implemented and tested against problems of various shape and size. In general, Max-Min Ant System was the best performing algorithm except in the case of large asymmetric instances, in which Double Q-Learning showed better performance.

***K*eywords** algorithms · traveling salesman · ant colony optimization · reinforcement learning

1 Introduction

The Traveling Salesman Problem (TSP) is a combinatorial optimization problem. Modeling the problem as a graph, the objective is to generate an optimal tour of n nodes, starting and ending at the same node and visiting each node only once. All nodes are connected by weighted edges, and optimal is defined as the path with the smallest possible sum of weights, referred to as the cost. The problem is named after the concept of a salesperson who has to travel to a number of cities and then return to their home city.

The Traveling Salesman Problem has no known solution in polynomial time. It is an NP-complete problem that is most commonly reduced to the the Hamiltonian Cycle Problem, which determines if a cycle exists in a graph that visits each vertex exactly once[1]. To solve the problem exactly, there is a brute-force solution which solves the problem in $O(n!)$. There is also a dynamic programming solution which solves the problem in $O(n^2 2^n)$, popularized by Held and Karp in [2]. However, these runtimes are unacceptable for even medium-sized problems, i.e. 17+ nodes[3]. Because of this,

approximation methods which return a near-optimal solution are often used. While many approximations and heuristics exist, this paper focuses on Ant Colony Optimization and Reinforcement Learning.

Ant Colony Optimization (ACO) algorithms mimic the real-world behavior of ants. When ants travel, they release pheromones. Other ants follow the scent of these pheromones, as the presence of other ants indicates that something positive, like a food source, is at the end of the pheromone trail. The more a pheromone builds up on a trail, the more likely an ant is to follow that trail. When applied to the Traveling Salesman Problem, ants act as agents that construct tours across all nodes in the problem space, starting and ending at the same node. After ants complete a tour, they simulate releasing pheromone by updating a pheromone matrix with a positive value for all entries corresponding to their tour. In subsequent runs of the algorithm, ants are influenced by this pheromone matrix, where they will account for the pheromone strength of traveling to a particular node. In addition to the pheromone matrix, a heuristic based on neighboring distance is also considered. These two factors are combined to generate the probability in which an ant will travel to a specific node, allowing tours to be generated based on distance and past successful behavior[4].

Reinforcement Learning implements an agent which explores an environment. During exploration, an agent performs certain actions to move between different states. Each decision reaps a certain reward. Repeated exploration allows an agent to better understand which sequence of decisions will give the highest reward, and thus exploit its knowledge to make optimal decisions. When applying Reinforcement Learning to the Traveling Salesman Problem, the problem can be modeled similar to [5], where

- Episode - Generation of a complete tour, starting and ending at the same node
- Agent - Salesperson traveling from node to node
- Environment - Nodes that have not yet been visited
- State - Current node of the salesperson
- Action - Next node to visit
- Reward - Prize or penalty for moving to the next state with the chosen action

In each episode of Reinforcement Learning, the agent travels from node to node, learning which reward it gets for each action it takes. In subsequent episodes, the agent can use this knowledge to determine which action is likely the best action to take, converging on an optimal route.

The work in this paper focuses on using a subset of Reinforcement Learning called Q-Learning. In Q-Learning, a Q-Table is maintained which indicates the reward, or penalty, of making a certain decision. In the Traveling Salesman Problem, $Q(s, a)$ indicates the reward from traveling from node s to node a, from the current state to the chosen action. The Q-Table is used as the agent traverses the environment to determine which node it should travel to next and to record reward values for taking certain actions in certain states. Over repeated episodes, the agent uses the Q-Table to learn an optimal route. This method can also be modified to implement Deep Q-Learning, in which the Q-Table is replaced with a feedforward neural network.

Ant Colony Optimization and Q-Learning were chosen as the focus for this paper because of their efficacy in solving the Traveling Salesman Problem and for their ability to be expanded and improved upon. Ant Colony Optimization has been heavily studied for the TSP, particularly by Dorigo and Stützle[4]. However, research on Q-Learning for the TSP is limited, with Wang et al. [5] being a primary resource on this subject. Both methods lend themselves to further exploration, especially with special problem types.

Using implementations of these algorithms, different variations of the TSP were evaluated. Standard problems from TSPLIB were used as a baseline for testing each algorithm. Asymmetric problems, problems in which the distance from node $i \to j \neq j \to i$ were looked at as a special problem type. Special problem shapes, including star-shaped problems, spiral problems, clustered problems, and tetrahedral problems were also studied. Algorithm efficacy in terms of cost was analyzed on each of these problem types.

2 Related Work

Solutions to the Traveling Salesman Problem have been well studied, with discussion first entering the field of mathematics in the 1930s[6]. In 1954, Dantzig, Fulkerson, and Johnson in [7] published how a 49-city problem could be solved. Since then, solutions for problems of up to 1.9 million cities have been developed[8].

Dorigo is one of the originators of Ant Colony Optimization algorithms, starting with Ant System described in [9]. Ant System was the building block of several more performant ACO algorithms, such as Stützle's Max-Min Ant System in [10]. Dorigo and Stützle's work is best summarized in [4], where Chapter 3 is dedicated to applying ACO algorithms to the Traveling Salesman Problem.

Reinforcement Learning Methods for the Traveling Salesman Problem were explored by Wang et al. in [5], where Q-Learning, SARSA, and Double Q-Learning were used to solve the TSP. Dorigo also attempted to apply the principles of Q-Learning to ACO algorithms with Ant-Q in [11]. However, the method is not commonly used, as his Ant Colony System algorithm is seen as an improved and less complex version of Ant-Q[4]. Deep Q-Learning is rarely used in combinatorial optimization problems. In [12], which introduced Deep Q-Learning, the model was used to play Atari games. However, its use for the TSP is discussed in [13], albeit without implementation details. The implementation of Deep Q-Learning for this project was adapted from the PyTorch tutorial at [14], in which a DQN was implemented to solve the CartPole problem.

In [15], Price studied Ant Colony Optimization and Reinforcement Learning for a Masters Thesis with some focus on the Traveling Salesman Problem. However, this work was more of an overview and did not go into specific problem types. Hougardy and Zhong studied hard to solve instances of the TSP in [16], but their work mostly focused on identifying these problems and using the established Concorde solver to solve them.

3 Methodology

3.1 Ant Colony Optimization Algorithm

The general procedure for an ACO algorithm is defined by Dorigo and Stützle in [4] as

Algorithm 1 ACO Metaheuristic

```
procedure ACOMETAHEURISTIC
    Initialize Pheromone Trails
    while i < MAX_ITERATIONS do
        Construct Solution for Each Ant
        Update Pheromone Matrix
    end while
    return Best Ant Tour from Final Iteration
end procedure
```

Initialize Pheromone Trails The pheromone matrix is defined by τ, where $\tau_{i,j}$ indicates the pheromone strength of traveling from node i to node j. Each entry in τ is initialized in [4] to

$$\frac{n}{\text{NNSC}} \tag{1}$$

where n is the number of nodes and NNSC is the cost found from constructing a tour using Nearest Neighbor Search. While not an optimal TSP solution, NNS gives a rough estimate of the search cost and is effective in setting starting values for the pheromone matrix.

Construct Solution for Each Ant In each iteration of an ACO algorithm, m ants will construct m solutions, each starting at a different node. When constructing a tour, each ant decides which node to visit next using the probabilistic action choice function defined in [4] as

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in \mathcal{N}_i^k} [\tau_{il}]^\alpha [\eta_{il}]^\beta} \tag{2}$$

In this equation,

- i is the starting node
- j is the node the ant is considering traveling to
- k represents 1 of m ants constructing a tour
- p is the probability of ant k traveling from i to j
- τ is the pheromone matrix
- α is a hyperparameter that determines the influence of τ
- η is a heuristic matrix that represents the distance of traveling from i to j

- β is a hyperparameter that determines the influence of η

Each entry in the heuristic matrix is calculated in [4] as

$$\eta_{ij} = \frac{1}{d_{ij}} \tag{3}$$

where d_{ij} is the distance from i to j. The denominator of Equation 2 is similar to the numerator, but includes the cost of traveling to all unvisited nodes, $\mathcal{N}$, including j[4]. As the probabilistic action function is only used for comparison between nodes in the same state, this denominator can be discarded in the algorithm implementation, as it will be the same for each $j \in \mathcal{N}$.

When considering which node to travel to next, the ant calculates p_{ij} for all $j \in \mathcal{N}$. The node with the highest value for p is chosen as the next node to travel to. This process repeats until a complete tour is generated for all ants.

Update Pheromone Matrix Once all ants complete a tour, the pheromone matrix is updated. First, each entry in τ undergoes evaporation, where all entries are reduced by a constant. This is defined in [4] by the equation below

$$\tau_{ij} = (1 - \rho)\tau_{ij} \tag{4}$$

where ρ must be between 0 and 1. A higher value will discard previous pheromone information more quickly, and a lower value will retain information from the past longer. This process allows values from bad tours to lose influence. After this evaporation, each entry is updated with the following equation from [4]

$$\tau_{ij} = \tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^{k} \tag{5}$$

$\Delta\tau_{ij}^{k}$ evaluates to 0 if ant k did not go from i to j in the tour it constructed, and $\frac{1}{C^k}$ if it did go from i to j, where C^k is the overall cost of the tour for ant k. This equation encourages ants to visit arcs that were part of successful tours, as the update is proportional to the overall tour cost [4].

Choosing the Optimal Tour By the final iteration, ants converge to an optimal tour. The best ant from the final tour will be used to return the best cost and tour found from the ACO solution.

3.1.1 Max-Min Ant System

The procedure described in 3.1 relates to how a specific algorithm, Ant System, is implemented. Other ACO algorithms, such as Max-Min Ant System (MMAS), use Ant System as a base and improve upon the process. MMAS adds the following improvements to Ant System which were first presented in [10].

Allow only 1 ant to update the pheromone matrix After each iteration where all ants have completed a tour, only 1 ant as opposed to every ant is allowed to update the pheromone matrix. Either the iteration-best ant, or the best ant out of all iterations, is allowed to update the matrix. This prevents ants that have taken sub-optimal tours from having an effect. Throughout the algorithm, use of the iteration-best and best-so-far ant is swapped, as the iteration-best ant can encourage exploration without negatively impacting the pheromone matrix[4].

Set a max and min pheromone value Setting a range of pheromone values can prevent search stagnation[4]. The max value of a pheromone update is set in [4] to

$$\tau_{max} = \frac{1}{\rho C_{best}} \tag{6}$$

Where ρ is a hyperparameter and C_{best} is the best-so-far cost found by the algorithm. When the algorithm is initialized, τ_{max} is used to initialize the pheromone matrix, with $C_{best} = NNSC$.

τ_{min} is set in [4] to

$$\tau_{min} = \tau_{max} \cdot \frac{1-\rho}{(\frac{n}{2}-1) \cdot \sqrt[n]{0.05}} \tag{7}$$

where n is the number of nodes in the problem.

Prevent stagnation via reinitialization If a certain number of iterations pass without a new best-so-far ant being found, the pheromone matrix is reinitialized with τ_{max} via Equation 6. This prevents the algorithm from stagnating on a bad route and allows for new, potentially better routes to be explored[4].

3.2 Q-Learning Algorithm

The general procedure for a Q-Learning algorithm is

Algorithm 2 Q-Learning for TSP

```
procedure QLEARNING
    for episode in EPISODE_COUNT do
        while there are unvisited nodes do
            state = current_node
            environment = remaining_nodes
            action = NextAction(state, environment)
            reward = Reward(state, action)
            updated_environment = environment - action
            next_action = NextAction(action, updated_environment)
            UpdateQTable(state, action, reward, next_action)
        end while
    end for
    return Q-Table used to find optimal route
end procedure
```

Define Action The action to take is determined by the starting node (the state) and the set of available nodes to travel to (the environment). Q-Learning utilizes an ϵ-greedy action choice. Epsilon is set to a value between 0 and 1, usually by a function that decays the value as the algorithm runs. A random number is generated between 0 and 1. If the number is below ϵ, a random action is taken, referred to as exploration. If the random number is at or above ϵ, the Q-Table is used to decide on the best action to take, known as knowledge exploitation[17]. Exploitation is done via

$$A = max_{a \in \mathcal{N}} Q(s, a) \tag{8}$$

Get Reward Once an action is taken, a reward value is returned. This can either be positive, where a high value indicates a good action was taken, or negative, where a low or negative value indicates a bad action was taken.

Determine Next Likely Action One an action is taken, it is removed from the updated environment, and the next best action to take from this new state is chosen. This will be used to update the Q-Table.

Update Q-Table This is the crux of Q-Learning. The Q-Table indicates which actions are likely to give which rewards. The equation from [5] is used

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha[r(s, a) + \gamma Q(s\prime, a\prime) - Q_t(s, a)] \tag{9}$$

to update the Q-Table entry for choosing an action at a particular state. In this equation

- $Q_t(s, a)$ represents the current value in the Q-Table for traveling from the current node (the state) to the destination node (the action)
- α represents the learning rate
- $r(s, a)$ represents the reward from traveling from the current state to the selected action
- γ is the discount factor, a value from 0 to 1 that influences how much impact future actions have on decisions made using the Q-Table
- $Q(s\prime, a\prime)$ represents the value in the Q-Table for traveling from the selected action (the state of the next iteration) to the best possible destination from this state (the action in the next iteration)

After a set number of episodes run, the Q-table is used to construct an optimal tour. Constructing an optimal tour follows the procedure in Algorithm 2, but it only runs 1 episode, and all actions are taken using exploitation vs. exploration.

3.2.1 Double Q-Learning

Equations 8 and 9 can be expanded to implement Double Q-Learning. Double Q-Learning adheres to the same principles as Q-Learning, but implements two separate Q-Tables. Q-Learning can suffer from the problem of overestimating the value of a certain action. Double Q-Learning can help regulate action values and thus improve performance[18]. The differences in implementation come in choosing an action via exploitation and updating the Q-Table.

Choosing an action via exploitation When choosing an action, the average value of both Q-Tables are used to find the maximum value that can be obtained by choosing an available action, defined as

$$A = max_{a \in \mathcal{N}}(\frac{Q^A(s, a) + Q^B(s, a)}{2}) \tag{10}$$

Update Q-Table In the Q-Table update, both tables are updated via the equation described by Wang et al. in [5] as

$$\begin{aligned} Q^A_{t+1}(s, a) &= Q^A_t(s, a) + \alpha[r(s, a) + \gamma Q^B(s\prime, a\prime) - Q^A_t(s, a)] \\ Q^B_{t+1}(s, a) &= Q^B_t(s, a) + \alpha[r(s, a) + \gamma Q^A(s\prime, a\prime) - Q^B_t(s, a)] \end{aligned} \tag{11}$$

3.2.2 Deep Q-Learning

Deep Q-Learning modifies Q-Learning to utilize a feedforward neural network in place of a Q-Table to determine the next optimal action. As episodes of Q-learning run, the state, action, reward, and next state are recorded as an entry in a batch. Once enough entries in this batch are saved, in this case 128, the neural network is trained on the batch. Two neural networks are utilized in this process, a policy network and a target network. The policy network predicts the given rewards for a

state-action-reward-next action pair. The target network gets updated every 1000 pairs and is used to compute an expected value in calculating loss on the policy network. Smooth L1 Loss is used to calculate the loss, and the Adam optimizer is used to optimize the neural network. After 10,000 episodes, the policy network is used to generate an optimal tour.

3.3 Project Implementation

An object-oriented approach was taken in the software design. A UML of the inheritance model is shown in Figure 1.

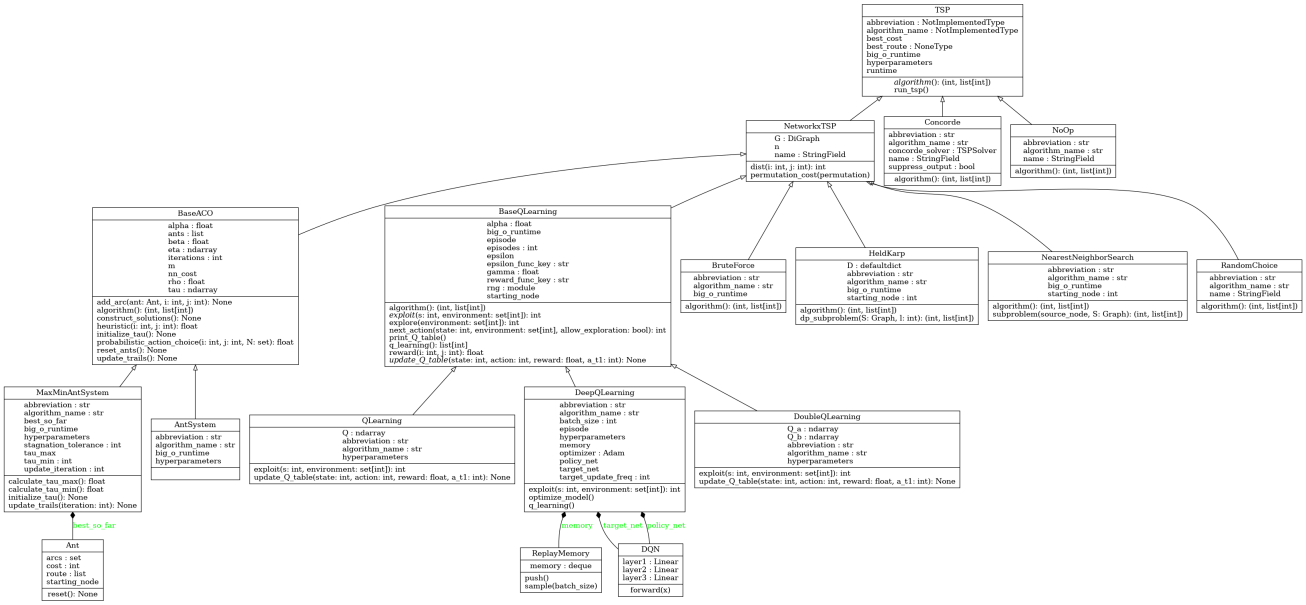

Figure 1: UML generated with pyreverse[19]

All TSP algorithms inherit from the TSP class, defining the functionality that every algorithm is expected to have. Each subclass implements its own $algorithm$ method, which returns the best cost and route found. The Concorde algorithm uses a 3rd party software to provide a solution, so its implementation is minimal. The NoOp class simply reports the best known solution to a problem, used in situations where either Concorde cannot be installed or Concorde cannot solve the problem.

All other algorithms were created as part of this project and inherited from the NetworkxTSP class. Problems in the TSPLIB format were parsed using the Python tsplib95[20] library, which provides each problem as a NetworkX Graph[21]. This allowed a common graph object, G, and distance function to be used for each inheriting algorithm. Brute Force, Held-Karp (Dynamic Programming), Nearest Neighbor, and Random Choice algorithms inherited directly from this class. The main purpose of the Brute Force and Held-Karp algorithms was to gain an understanding of the runtime of exact algorithms. The Nearest Neighbor algorithm is used as part of the Max-Min Ant System Algorithm, as shown in Equation 1. Nearest Neighbor Search, as well as Random Choice, also act as baseline algorithms in testing.

The BaseACO Class implements the Ant System algorithm, which is the basis for Ant Colony Optimization Algorithms. The AntSystem class simply inherits everything from this base class with its own hyperparameters. MaxMinAntSystem builds upon BaseACO with the updates described in Section 3.1.1. The BaseQLearning class implements the core of what is needed for Q-Learning. The QLearning class implements exploitation as per Equation 8 and the Q-Table update as per Equation 9. Similarly, the DoubleQLearning class implements exploitation as per Equation 10 and the Q-Table update as per Equation 11. Deep Q-Learning inherits from the Base Q-Learning class and requires its own dependencies. ReplayMemory is used to store state-action-reward-next action pairs, and DQN is the class for the feedforward neural network.

These algorithms can be called by the main class via the run_tsp function, where the best cost and route for a problem are returned along with the algorithm's runtime.

3.4 Data

Several different problem types were established to test each algorithm

- Symmetric - Standard TSP problem with no particular problem shape or type. The distance from node $i \rightarrow j = j \rightarrow i$. Problems for this data type were taken from TSPLIB[22].
- Asymmetric - Problem where the distance from node $i \rightarrow j \neq j \rightarrow i$. Problems for this data type were taken from TSPLIB[22].
- Star - Problems assume a star or asterisk shape as seen in Figure 2. Problems for this type were custom generated using the Concorde Windows GUI tool[23].
- Spiral - Problems assume a spiral shape as seen in Figure 3. Problems for this type were custom generated using the Concorde Windows GUI tool[23].
- Cluster - Problems contain several clusters of nodes as seen in Figure 4. Problems for this type were custom generated using the Concorde Windows GUI tool[23].
- Tetrahedral - Problems took on the shape of a 2D tetrahedron as seen in Figure 5. Problems for this type were taken from the data provided in [24].

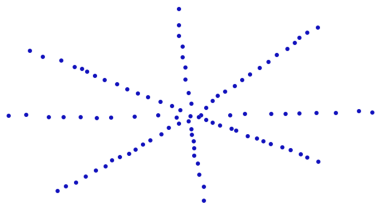

Figure 2: Asterisk (star) shaped problem

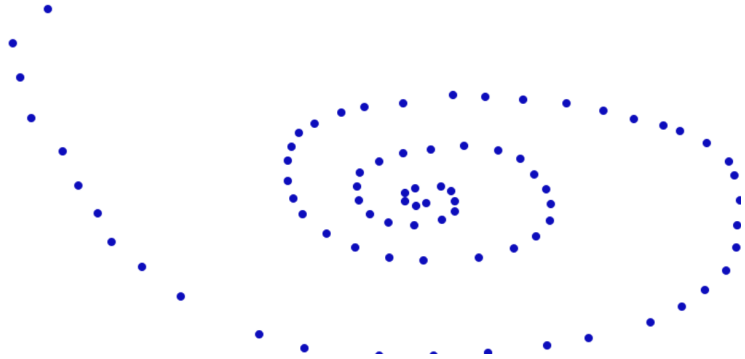

Figure 3: Spiral shaped problem

Figure 4: Clustered problem

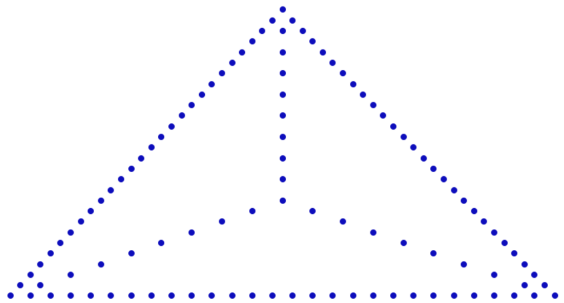

Figure 5: Tetrahedral problem. tnm_106 from [24]

3.5 Algorithms

The specific algorithms used for evaluation purposes were

- Random Choice - Generates 10,000 random permutations of possible solutions and chooses the best one. Acts as a baseline for the minimum barrier of algorithm performance.
- Concorde - 3rd party TSP solver[23]. Used to represent the optimal solution.
  - The Concorde Solver cannot solve asymmetric instances. For problem $br17$, the problem was small enough that the Held-Karp Dynamic Programming algorithm

referenced in 3.3 was able to be used. For all others, the given optimum from TSPLIB's documentation at [25] was used.

- For some instances of the Tetrahedral problems, using the Concorde solver resulted in a Segmentation Fault. In this case, the given optimum from [24] was used.

- Nearest Neighbor Search - Greedy algorithm that always chooses the shortest available path, returning to the starting node when no available nodes remain. Acts as a higher quality baseline for performance compared to Random Choice.
- Max-Min Ant System - ACO algorithm as described in Sections 3.1 and 3.1.1
- Double Q-Learning - Reinforcement Learning algorithm as described in Sections 3.2 and 3.2.1
- Deep Q-Learning - Deep Learning algorithm as described in Sections 3.2 and 3.2.2

Ant System and Q-Learning (i.e. not Double or Deep Q-Learning) were left out of the analysis as they are simpler and generally less performant versions of the other algorithms being used.

3.5.1 Hyperparameters

Research, particularly [4] and [5] provided helpful starting points for setting hyperparameters. Hyperparameters were further optimized using Optuna[26], a library for optimizing model hyperparameters. Using the tool, a range of acceptable hyperparameters was established. Optuna ran the algorithms on several problems, iterating over several variations of possible hyperparameters and recording the cost with the hyperparameters used.

For the project results, one set of hyperparameters per algorithm was used across all problems. Hyperparameters were chosen based on the values that worked well across several problems during hyperparameter tuning. Using one set vs. refining for each problem prevented results from being skewed towards algorithms that responded well to changes to hyperparameters. Although these values can be adjusted, for novel problem instances, there is not the ability to fine tune the hyperparameters. A more effective strategy is to find a set that consistently does well, and apply those hyperparameters across the board.

Ant Colony Optimization Algorithm Hyperparameters The hyperparameters used were

Algorithm	alpha	beta	rho	Iterations	Stagnation Tolerance
Max-Min Ant System	1	2	0.02	1000	100

Table 1: Hyperparameters of Max-Min Ant System Algorithm

For ACO algorithms, each hyperparameter controls the following

- alpha - Effect pheromone matrix has on next node traveled to
- beta - Effect distance has on next node traveled to

- rho (Ant System) - How quickly information from past trails evaporates in the pheromone matrix. Also used to calculate τ_{max} and τ_{min}
- Iterations - How many times ants generate trails
- Stagnation Tolerance - Number of runs without a new best-so-far ant before resetting the pheromone matrix.

Q-Learning Hyperparameters

Algorithm	episodes	alpha	gamma	epsilon	reward
Double Q-Learning	10,000	0.02	0.12	1 - (0.1 * (i / episodes / 10)))	-dist
Deep Q-Learning	10,000	0.01	0.6	1 - (0.1 * (i / episodes / 10)))	-dist

Table 2: Hyperparameters of Q-Learning Algorithms

For Q-learning algorithms, each hyperparameter controls the following.

- episodes - How many times to run Q-Learning
- alpha - Learning rate
- gamma - Discount factor. Lower values take less consideration towards future rewards, whereas higher values place more importance on future rewards
- epsilon - Determines whether the model should explore or exploit. Higher values encourage more exploration, so typically starts high and decays. In the implementation chosen from [5], epsilon is a step function that decreases by 0.1 every 1000 episodes.
- reward - Determines the reward / penalty for making a certain decision. In the implementation chosen from [5], the negative distance from traveling from $i \rightarrow j$ is used, so better paths will receive less of a penalty.

3.6 Test Plan

1-3 problems of each data type from Section 3.4 were run on the algorithms established in Section 3.5. The best cost, best route, and runtime were recorded, and results were saved in MLflow[27]. Success for each problem was measured by the algorithm with the lowest overall cost.

4 Results

Problem size is given in parenthesis with the problem name. Note that for all figures except for Asymmetric Problems, the Random Choice algorithm is excluded. In most cases, the cost for this algorithm was so much higher that it skewed the visualization of all other results.

4.1 Symmetrical TSP

Algorithm	fri26 (26)			berlin52 (52)			a280 (280)		
	Cost	Error	Time (s)	Cost	Error	Time (s)	Cost	Error	Time (s)
Concorde	937	0	0.018	7542	0	0.03	2579	0	0.761
MMAS	955	1.9	6.38	8092	7.3	55.8	2988	15.9	6035
NNS	1112	18.7	0.0003	8980	19.1	0.003	3157	22.4	0.041
Double Q	1050	12.1	26.2	9179	21.7	77.8	3554	37.8	2740
Deep Q	1027	9.6	1073	10922	44.8	3981	7817	203	67784
Random	1853	97.8	0.273	23522	212	0.4	29909	1059	2.7

Table 3: Tabular Results of the Symmetric Problems

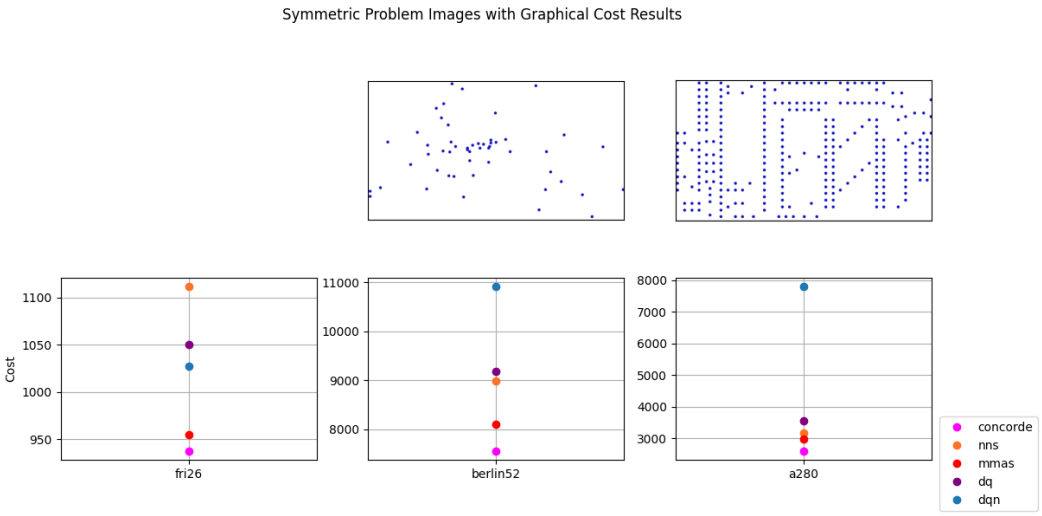

Figure 6: Symmetric Problem Images with Graphical Cost Results. Note that the fri26 problem image could not be loaded by the Concorde GUI.

4.2 Star Shaped Problems

Algorithm	Star (85)			Asterisk (100)		
	Cost	Error	Time (s)	Cost	Error	Time (s)
Concorde	275	0	0.047	254	0	0.094
MMAS	295	7.3	250	270	6.3	319
NNS	358	30.2	0.004	345	35.8	0.005
Double Q	390	41.8	224	373	46.9	382
Deep Q	464	68.7	9488	553	118	12526
Random	1817	561	0.261	1582	523	0.305

Table 4: Tabular Results of the Star Problems

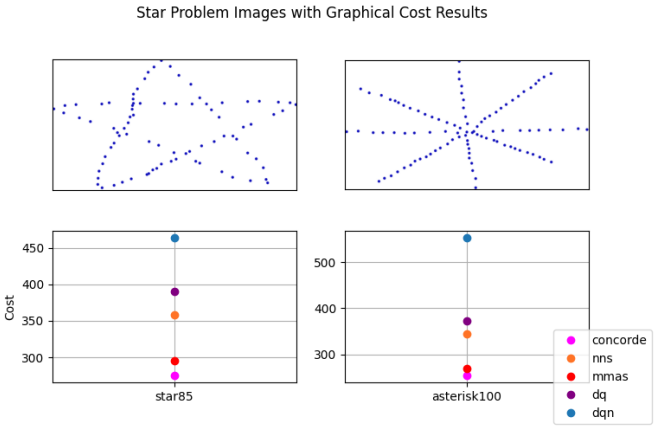

Figure 7: Star Problem Images with Graphical Cost Results

4.3 Spiral Problem

Algorithm	Spiral (79)		
	Cost	Error	Time (s)
Concorde	286	0	0.879
MMAS	292	2.1	185
NNS	298	4.2	0.004
Double Q	301	5.25	168
Deep Q	306	6.9	8413
Random	1370	379	0.241

Table 5: Tabular Results of the Spiral Problem

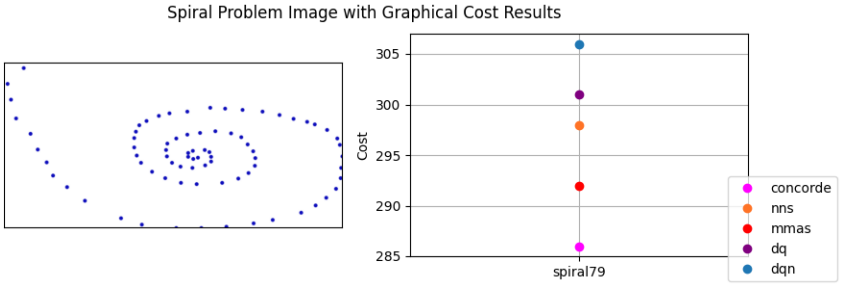

Figure 8: Spiral Problem Image with Graphical Cost Results

4.4 Cluster Problem

Algorithm	5Cluster (43)		
	Cost	Error	Time (s)
Concorde	100	0	0.119
NNS	111	11	0.002
MMAS	112	12	32.49
Double Q	118	18	35.2
Deep Q	141	41	3317
Random	471	371	0.135

Table 6: Tabular Results of the Cluster Problem

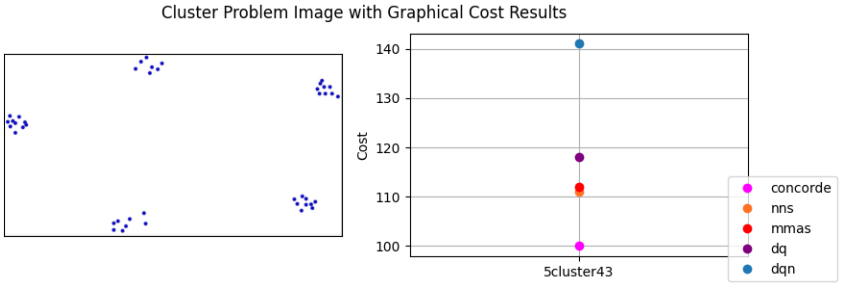

Figure 9: Cluster Problem Image with Graphical Cost Results

4.5 Tetrahedral Problems

Tnm160 could not be solved by Concorde without a Segmentation Fault, so the known optimum value from [24] was used in its place.

Algorithm	Tnm52			Tnm70			Tnm160		
	Cost	Error	Time (s)	Cost	Error	Time (s)	Cost	Error	Time (s)
Concorde	551609	0	12.85	881036	0	54.65	2.5e6	N/A	N/A
MMAS	607236	10.1	37.57	994978	12.9	125.4	2.6e6	6.5	1293
NNS	632300	14.6	0.002	1.0e6	14.1	0.005	2.7e6	9.3	0.014
Double Q	672125	21.8	74.25	1.0e6	17.14	135.6	2.8e6	12.3	627
Deep Q	1.8e6	231	4308	3.9e6	344.4	6968	3.3e7	1257	17549
Random	1.9e6	260	0.160	4.5e6	415	0.215	3.1e7	1141	0.518

Table 7: Tabular Results of the Tetrahedral Problems

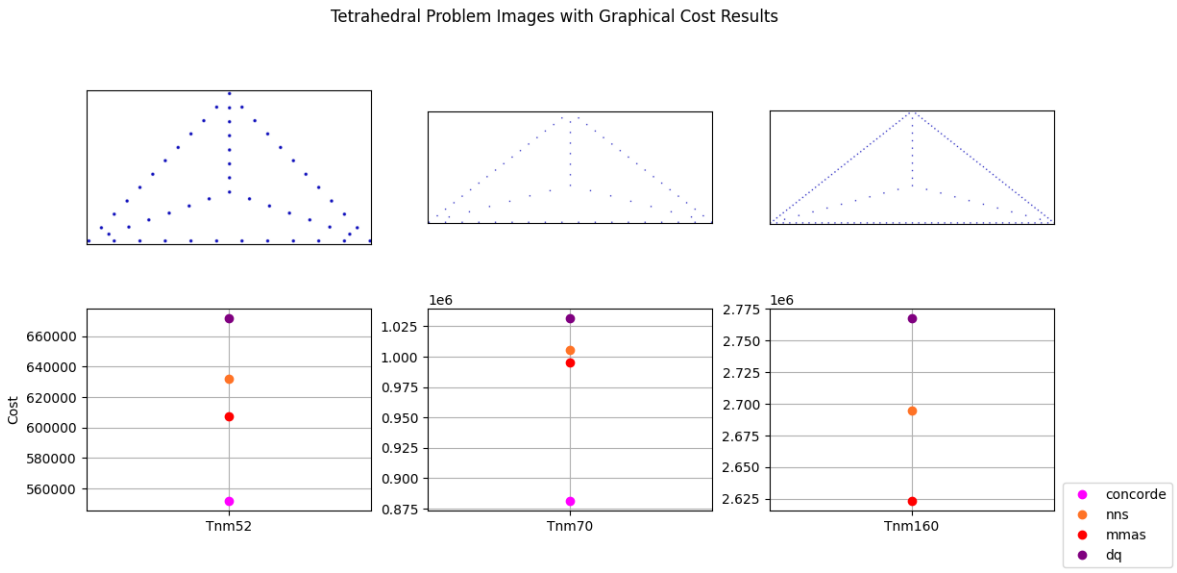

Figure 10: Tetrahedral Problem Images with Graphical Cost Results. Note that Deep Q-Learning results are omitted, as they were exponentially higher than all other solutions and skewed the chart.

4.6 Asymmetrical Problems

4.6.1 Small-Medium Size

Note that the Held-Karp algorithm was used to solve br17, as Concorde cannot solve asymmetric instances, and the problem was small enough that the dynamic programming solution could be used. ftv47 was too large, so the known optimum from [25] was used.

Algorithm	**br17**			**ftv47**		
	Cost	**Error**	**Time (s)**	**Cost**	**Error**	**Time (s)**
Held-Karp / Known Optimum	39	0	30.9	1776	0	N/A
MMAS	82	110	1.115	2173	22.4	31.8
NNS	92	135	0.001	2374	33.6	0.001
Double Q	92	135	12.37	2374	33.6	67.8
Deep Q	92	135	930	3046	71.5	2720
Random	70	79.5	0.061	5225	194.2	0.152

Table 8: Tabular Results of the Small-Medium Asymmetric Problems

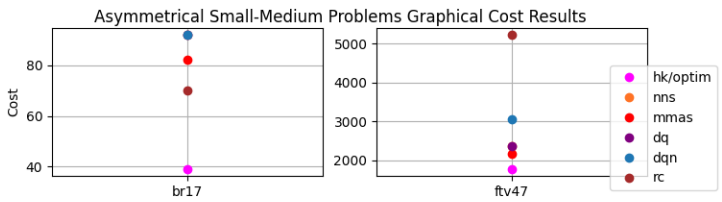

Figure 11: Asymmetrical Small-Medium Problems with Graphical Cost Results. Note that Some of the results are exactly the same and will overlap. See the Table 4.6.1 for exact costs. Additionally, note that the problem images could not be loaded. Images are generated with the Concorde GUI, which cannot load ATSP problems.

4.6.2 Large Size

Algorithm	**rbg323**			**rbg358**			**rbg403**		
	Cost	**Error**	**Time (s)**	**Cost**	**Error**	**Time (s)**	**Cost**	**Error**	**Time (s)**
Known Optimum	1326	0	N/A	1163	0	N/A	2465	0	N/A
NNS	1734	30.8	0.126	1812	55.8	0.230	3535	43.4	0.102
Double Q	2245	69.3	2322	1817	56.2	4075	3457	40.2	5136
MMAS	3488	163	15201	4180	259	20139	5685	130	16508
Random	5695	329	1.31	6378	448	1.64	7187	192	1.90
Deep Q	6119	361	11553	6711	477	90122	7575	207	23279

Table 9: Tabular Results of the Large Asymmetric Problems

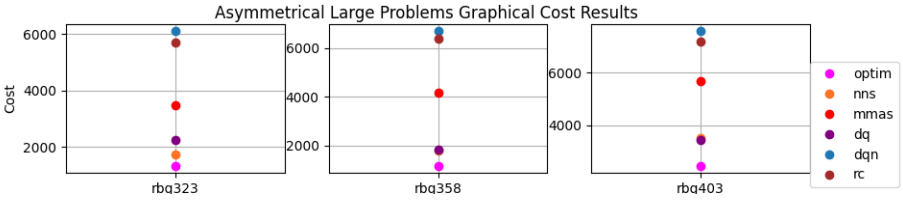

Figure 12: Asymmetrical Small-Medium Problems with Graphical Cost Results. Note that the problem images could not be loaded. Images are generated with the Concorde GUI, which cannot load ATSP problems.

5 Discussion

Among all the problem types tested, there were generally three categories that emerged: problems where Max-Min Ant System performed the best, problems where Nearest Neighbor Search performed the best, and problems where Double Q-Learning performed the best. The Concorde solver outperformed all algorithms that were implemented for this project, but that was expected, and simply used as the baseline for best possible performance.

5.1 Symmetrical, Star-shaped, Spiral, and Tetrahedral Problems

For the Symmetrical, Star-shaped, Spiral, and Tetrahedral problems, the general order of performance was

- Max-Min Ant system (MMAS)
- Nearest Neighbor Search (NNS)
- Double Q-Learning (DQ)
- Deep Q-Learning (DQN)
- Random Choice (RC)

Among these problem types, the error rate of MMAS ranged between 1.9 and 15.9%. Additionally, there were no instances among these types where Nearest Neighbor Search outperformed MMAS. In the case of Tnm160, MMAS actually showed advantages over Concorde. Concorde was unable to solve the problem without a Segmentation Fault error on the project PC, whereas MMAS solved the problem in 1293 seconds with a 6.5% error rate.

There are some interesting exceptions to this order of performance. In the fri26 problem, Deep Q-Learning was the second-best performer next to MMAS. The DQN algorithm tended to perform relatively well on smaller problems. However, as the problem size grew, DQN's performance grew exponentially worse. This is particularly prominent in the Tetrahedral Problem Tnm160, where the Random Choice algorithm actually performed better than DQN.

NNS was the worst performer in the fri26 problem, but performed better than DQ and DQN in the remaining problems outlined in this section. This seems to be more of an indictment on Q-Learning's suboptimal performance on larger instances of the TSP, but it also shows that in many cases, a simple algorithm such as NNS can give a good enough solution depending on the use case.

5.2 Cluster Problem

For the Cluster problem, the order of performance was

- Nearest Neighbor Search (NNS)
- Max-Min Ant system (MMAS)
- Double Q-Learning (DQ)
- Deep Q-Learning (DQN)
- Random Choice (RC)

In this case, the simplistic greedy NNS algorithm outperformed all other algorithms implemented for this project. One cause of this is likely the problems' simple shape. The cluster problem is always forced to traverse all 5 clusters. Any optimizations would happen within the clusters, which would not make a significant difference to the overall solution.

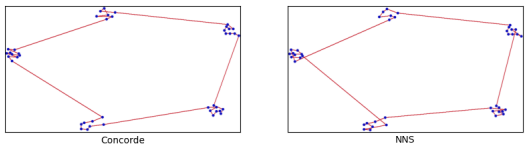

Figure 13: Concorde vs. Nearest Neighbor Search solutions to the cluster problem

This instance shows that if the general shape of the problem is known in advance and it has a simplistic path through all nodes, the NNS can give a near-optimal solution in a fraction of the time with a better solution than more complex but generally better performing algorithms such as MMAS.

5.3 Asymmetric Problems

Many of the algorithms struggle with asymmetric problems. This is most notable in br17, where Random Choice was the second-best performing algorithm, only behind the exact Held-Karp solution. As the problem size grew to 47, the same pattern identified in Section 5.1 held with MMAS performing the best. Also similar to Section 5.1, Deep Q-Learning performed exponentially worse as problem sizes increased. In all three large asymmetric instances of size 323, 358, and 403, it performed worse than Random Choice.

Most notable for these large asymmetric problems was Double Q-Learning outperforming MMAS. This was a notable exception to what had been seen thus far, as nothing specific was done to make the DQ algorithm perform better on asymmetric instances. The results were a concrete example of different algorithms being better suited towards different tasks.

While an encouraging result, the DQ algorithm performed similarly to the simple NNS algorithm on these algorithms. It performed worse on asymmetric problems of size 323 and 358, and only slightly better on a problem of size 403. It is possible that as the asymmetric problem size increases, the difference between DQ and NNS will be more prominent, but TSPLIB is limited in the number of asymmetric problem instances it has available. Additionally, all of these problems had the prefix rbg. It's possible they are all versions of a unique problem shape that DQ handles particularly well, and that these results will not translate to all problems. Nevertheless, its performance on these problems was impressive and showed clear advantages over MMAS.

5.4 Summarization of Results

Based on the problems tested, the Max-Min Ant System algorithm has the best general performance among the approximation algorithms implemented. It performed above all other algorithms in the problem types described in Section 5.1, and performed close to the others in all other problem types. If nothing is known in advance about a TSP problem, MMAS would be an ideal algorithm choice.

Nearest Neighbor Search performs well on problems with relatively simple shapes. If the problem shape is known and resembles something like several clusters, NNS will give a near-optimal solution

in an extremely short amount of time. Additionally, NNS gave reasonable results for all other problem types. In situations where a solution just needs to be good enough, and solution time is important, NNS may be the preferred algorithm to use when solving the TSP.

Double Q-Learning performs well on asymmetric problems. If the problem is known to be asymmetric, and especially if it is a larger problem, Double Q-Learning would be the preferred algorithm to use. As noted in Section 5.3, there may be some caveats to this, and more study would be needed to confidently say DQ always performs better on large asymmetric TSP instances. However, the results are promising enough to recommend its use for this problem type.

Deep Q-Learning showed little to no added value in solving the TSP. Although it gave results close to, and in the case of problem fri26 better than, DQ, its performance became exponentially worse as the problem size increased, in some instances being worse than random choice. This is somewhat counterintuitive to how neural networks perform in other applications. Typically they excel at handling more complex problems that heuristic or iterative algorithms are limited in. This could be indicative of two things. Either DQN is only suited for mediocre performance on small TSP instances, or the neural network used needs to be further optimized and expanded to truly be useful. Based on the limited research of using neural networks to solve the TSP and similar optimization problems, its likely that DQN is simply ill-suited for the TSP, but further research into this topic could at least bring upon improvements to the algorithm's use in this field.

6 Code

All code used for this project can be found at https://github.com/AGnias47/tsp-research. Any references used in the development of the code are referenced within the files they are used in.

7 PC Specs

All algorithms were run on a Linux PC running Ubuntu 24.04 with 32 GB of RAM. For the Deep Q-Learning algorithm, a GeForce RTX 3060 Ti 8GB GPU was utilized.

8 Acknowledgments

I would like to thank Dr. Alex Pang for his support as my advisor for this project. His encouragement and guidance were essential in allowing me to take this project in several exciting directions.

References

[1] J. Kleinberg and E. Tardos, *Algorithm Design*. Pearson, 1 ed., 2006.

[2] M. Held and R. M. Karp, "A dynamic programming approach to sequencing problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 1, pp. 196–210, 1962.

[3] R. Bellman, "Dynamic programming treatment of the travelling salesman problem," *RAND Corporation*, 1961.

[4] M. Dorigo and T. Stützle, *Ant Colony Optimization*. MIT Press, 2004.

[5] J. Wang, C. Xiao, S. Wang, and Y. Ruan, "Reinforcement learning for the traveling salesman problem: Performance comparison of three algorithms," *J. Eng.*, p. e12303, 2023.

[6] D. L. Applegate, R. E. Bixby, V. Chvatál, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[7] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem," *Journal of the Operations Research Society of America*, vol. 2, no. 4, pp. 393–410, 1954.

[8] W. Cook, "World tsp." https://www.math.uwaterloo.ca/tsp/world/, Feb 2021.

[9] 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)*, vol. 26, no. 1, pp. 29–41, 1996.

[10] T. Stützle and H. Hoos, "Max-min ant system and local search for the traveling salesman problem," in *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC '97)*, pp. 309–314, 1997.

[11] L. M. Gambardella and M. Dorigo, "Ant-q: A reinforcement learning approach to the traveling salesman problem.," in *Machine Learning, Proceedings of the Twelfth International Conference on Machine Learning*, pp. 252–260, 01 1995.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[13] J. Clark, "The skynet salesman." https://multithreaded.stitchfix.com/blog/2016/07/21/skynet-salesman/, 2016.

[14] PyTorch, "Reinforcement learning (dqn) tutorial." https://pytorch.org/tutorials/intermediate/reinforcement_q_l
2016.

[15] A. Price, "Ant colony optimisation and reinforcement learning," *University of Edinburgh School of Informatics*, 2019.

[16] S. Hougardy and X. Zhong, "Hard to solve instances of the euclidean traveling salesman problem," *Mathematical Programming Computation 13 (2021), 51-74*, vol. 13, no. 1, pp. 51–74, 2021.

[17] J. D. McCaffrey, "The epsilon-greedy algorithm." https://jamesmccaffrey.wordpress.com/2017/11/30/the-epsilon-greedy-algorithm/, 2020.

[18] H. Hasselt, "Double q-learning," in *Advances in Neural Information Processing Systems* (J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, eds.), vol. 23, Curran Associates, Inc., 2010.

[19] Logilab and P. contributors, "Pyreverse." https://pylint.readthedocs.io/en/stable/pyreverse.html, 2025.

[20] R. Grant, “Welcome to tsplib 95’s documentation!.” https://tsplib95.readthedocs.io/en/stable/index.html, 2018.

[21] N. developers, “Networkx: Network analysis in python.” https://networkx.org/, 2024.

[22] G. Reinelt, “Tsplib.” http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/, 2019.

[23] W. Cook, “Concorde tsp solver.” https://www.math.uwaterloo.ca/tsp/concorde.html, 2020.

[24] S. Hougardy and X. Zhong, “Hard to solve instances of the euclidean traveling salesman problem.” https://www.or.uni-bonn.de/ hougardy/HardTSPInstances.html, 2018.

[25] G. Reinelt, “Best known solutions for asymmetric tsps.” http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ATSP.html, 1997.

[26] Optuna, “Optuna.” https://optuna.org/, 2025.

[27] M. Project, “Mlflow.” https://mlflow.org/, 2025.