Solving the Traveling Salesman Problem with Reinforcement Learning AI Models for Physics Exam Report

Filippo Monaco
Università degli Studi di Milano-Bicocca
Universtità degli Studi di Milano Statale
ID: 840089

***Abstract*—In this paper, I will showcase how to use Reinforcement Learning to solve the Traveling Salesman Problem. I will go over how I built a custom Gymnasium environment, the different reward functions I tested, and the difference in training algorithms. Finally, I will compare the results obtained to some classical heuristics and will show that the Reinforcement Learning approach here proposed does not yet reach the performance of standard algorithms.**

Link to the Python Notebook: https://drive.google.com/file/d/12C2SyOFBoIzuo01uUHUpXWtDwVAmoJ54/view?usp=sharing

1. Introduction

The Traveling Salesman Problem (TSP) is a classic optimization challenge that has intrigued mathematicians and computer scientists for decades. Given a set of nodes and the distance cost between them, the goal is to find the shortest possible route that visits each node exactly once. Despite its simple formulation, the TSP is known to be an NP-hard problem, meaning that as instance size grows it becomes increasingly difficult to find an optimal solution.
In recent years, advancements in machine learning, specifically reinforcement learning (RL), have shown great promise in tackling complex optimization problems, such as the TSP. RL provides a framework for an agent to learn from its interactions with an environment in order to maximize a policy by receiving a reward.
This project report aims to explore the application of reinforcement learning techniques to solve the TSP. By leveraging the capabilities of RL algorithms, I aim to develop an agent that can learn effective strategies for finding acceptable solutions to the TSP.
The report is structured as follows: first, I will provide a brief overview of the TSP and its significance in various fields and discuss the existing methods employed to solve the TSP. Next, I will delve into the fundamentals of reinforcement learning, explaining key concepts such as Markov Decision Processes, policies, and value functions.
After establishing the theoretical foundation, I will present my proposed approach for applying RL to the TSP. I will describe the formulation of the problem as an RL task, including the environment, action, and reward definitions. Furthermore, I will outline the architecture of the RL agent and the training process employed to learn effective policies for solving the TSP.
To evaluate the performance of my RL-based approach, I will compare the results obtained by our RL agent with those of other traditional algorithms. Finally, I will address the limitations of our approach and suggest possible directions for future research. In the end, this project aims to contribute to the growing RL-driven optimization field by providing a custom-built TSP environment.

2. The Traveling Salesman Problem

2.1. Brief introduction to the problem

The Traveling Salesman Problem (TSP) is a well-known combinatorial optimization problem that has been extensively studied in the fields of computer science, operations research, and mathematics. It poses a fundamental challenge in finding the shortest possible route that visits a given set of cities exactly once.
The TSP is defined by a set of cities and the distances or costs associated with traveling between them. Mathematically, it can be represented as an undirected weighted graph, where cities correspond to nodes and edges represent the distances or costs between them. The objective is to find a Hamiltonian cycle, a closed path that visits each city once and returns to the starting city, with the minimum total cost.
The significance of the TSP extends beyond its theoretical interest, as it has numerous practical applications in various domains. In logistics and transportation, finding optimal routes for delivery or tour planning is crucial for minimizing costs and maximizing efficiency. In circuit board manufacturing, the TSP helps optimize the order of machine operations, reducing production time and resource utilization.
The TSP is known to be an NP-hard problem, meaning that finding an optimal solution becomes increasingly challenging as the number of nodes increases. Exhaustive search methods, such as brute force or dynamic programming, become computationally infeasible for large-scale instances due to the exponentially growing search space. Consequently, researchers have focused on developing approximation algorithms and heuristics that provide near-optimal

solutions within a reasonable amount of time.
Despite the effectiveness of these classical approaches, the TSP remains a challenging problem, particularly when faced with large-scale instances. This has motivated researchers to explore alternative methods, such as machine learning and reinforcement learning, to tackle the TSP and potentially discover novel strategies for optimization.

2.2. Mathematical formulation

As briefly mentioned in the previous paragraph, the TSP can be modeled as an undirected weighted complete graph, such that the cities correspond to the graph's nodes, and the paths between them are the graph's edges, with an associated cost. In this formulation, the problem can be seen as a minimization problem: starting from a specific node, what is the shortest path that visits all the other nodes exactly once? Note that in this formulation, the term "shortest" refers to the total sum of costs collected at each edge. This cost can be seen as the physical distance between the nodes, the time it takes to travel between them, or even more specific concepts such as fuel consumption.
In mathematical notation, this can be stated as:

$$min \sum_{i=0}^{N-1} \sum_{j=0, j\neq i}^{N-1} c_{ij} x_{ij}$$

where

$$x_{ij} = \begin{cases} 1 & \text{if the path goes from i to j} \\ 0 & \text{otherwise} \end{cases}$$

with the following constraints:

$$\sum_{i=0, i\neq j}^{N-1} x_{ij} = 1 \quad \text{for } j = 0, ..., N-1$$

and

$$\sum_{j=0, j\neq i}^{N-1} x_{ij} = 1 \quad \text{for } i = 1, ..., N-1$$

.
Where these ensure that each node is visited exactly once, since they require for each node to have only one incoming and one outgoing active connection.
A solution to the TSP is thus the shortest path between the nodes, which is usually expressed as a number, computed by the weighted sum of all edges crossed.

2.3. Traditional Solving Algorithms

Due to the NP-hardness nature of the problem, the TSP does not have an optimal solving algorithm. We can differentiate the traditional solving algorithms in to two main categories:

- Exact Algorithms, that always solve the problem to optimality. However, this only works for small problem sizes.
- Heuristic Algorithms, which trade solution optimality for speed.

In the first category we have exact solutions such as Brute Force, which computes all possible permutations of ordered nodes to see which one gives the best solution. This approach is obviously not scalable, since it has a time complexity of $O(N!)$, which means that even for a small problem with 100 nodes, it would need to permute over 10^{150} different solutions.
Among Heuristics we can have more complex algorithms, such as the Nearest Neighbor, a greedy algorithm that at each step chooses the the nearest unvisited node. This obviously does not guarantee an optimal solution, so it is usually employed along with pairwise exchange approaches, such as k-opt, which iteratively removes k edges from the current path and reconnects the fragments created into a new and shorter tour. Compared to the factorial time complexity of brute force approach, this heuristic finds a solution in quadratic time ($O(N^2)$). Other heuristic algorithms might be simulated annealing or local search.

Furthermore, in recent years improvements in the field of Quantum Computing have shown that an exact algorithm can be improved drastically by exploiting quantum advantage. The current best quantum exact algorithm for the TSP runs in time $O(1.728^N)$

2.4. TSP in this project

In this project, the TSP instance is implemented by a hollow symmetric matrix which contains the pairwise distances between all nodes. This means that the distance matrix defined on the points is symmetric w.r.t. its main diagonal, which contains only zeros, since the distance to each node to itself is zero. This means that the actual position of the nodes in the space is not relevant; however, for interpretation reasons, we can still create a plot and see the route between the nodes, as can be seen in figure 1. In figure 2 we can see the corresponding distance matrix, which is sufficient to define the graph. Note the zero-valued main diagonal and the matrix symmetry mentioned previously.

3. Reinforcement Learning

Now that we've seen a quick overview of the TSP formulation, we can dive into the details of Reinforcement Learning, and how it is applied for this specific task.

3.1. Introduction to Reinforcement Learning

Reinforcement Learning (RL) is an area of machine learning that differs significantly from the two other main fields of Supervised and Unsupervised Learning: the fundamental idea behind RL is the concept of Sequential Decision Making. This process involves an **Agent**, which represents

2. this footnote should also be treated as an exponent.

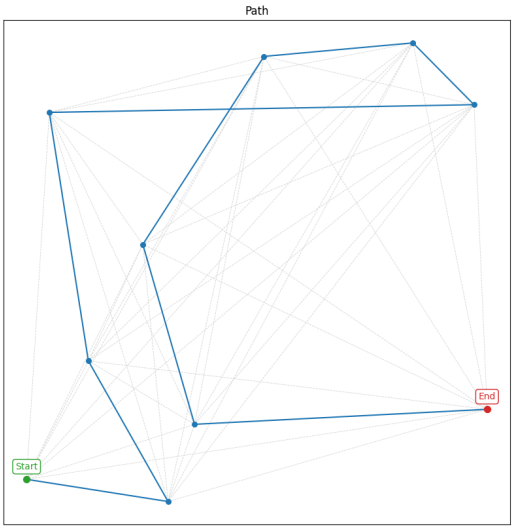

Figure 1: A simple TSP instance with 10 nodes, and a random route between them.

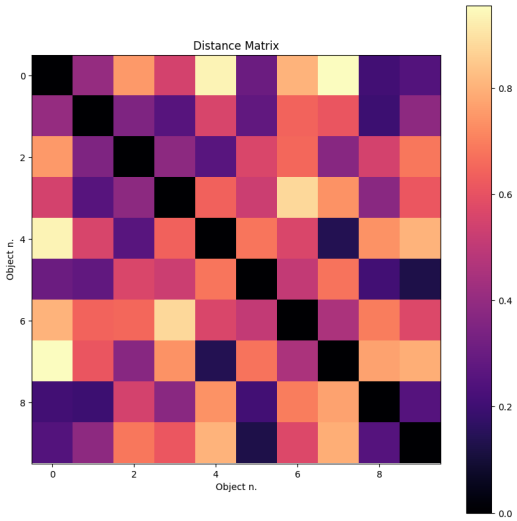

Figure 2: Distance Matrix of the graph shown in 1.

a process that takes decisions in order to reach a specific goal, and an **Environment**, which includes all the variables the agent can interact with.

The main idea behind the RL process is thus the following: based on the current state of the system, the Agent performs an Action according to a Policy that it wants to optimize. The Environment then rewards the agents based on the action taken, and it updates the state of he system. Refer to flowchart 3 for a visual representation of the main loop.

3.2. Mathematical Formulation

If we were to consider the entire history of the RL process at each iteration, the algorithm would quickly explode in terms of complexity. Because of this, we are only interested in states that depend exclusively on the immediate previous one: a Markov State.

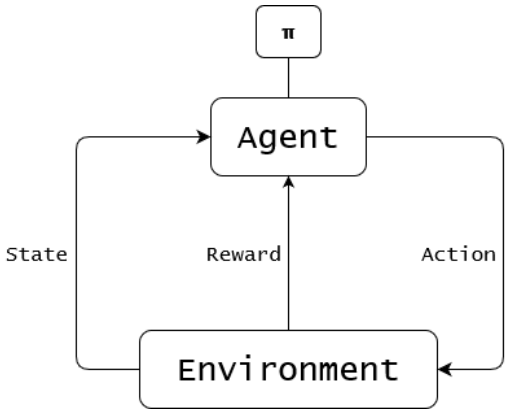

Figure 3: Reinforcement Learning main loop.

We can consider a state to be Markov if it follows the following condition:

$$p(s_{t+1}|s_t) = p(s_{t+1}|s_1...s_t)$$

which can be read as: a state s_{t+1} is a Markov state if the probability of the system of being in said state only depends on the previous state s_t.

From this we can then define what is known as a Markov Decision Process, which is a tuple (S, A, R, P, γ), where:

- S: finite set of states
- A: finite set of actions
- P: state transition probability function, which gives the probability of going from state s' to state s by performing action a

$$P^a_{s's} = p(s_{t+1} = s'|s_t = s, a_t = a)$$

- R: reward function, which returns the reward given by the environment to the agent after performing action a while in state s

$$R^a_s = \mathbb{E}[R_{t+1}|s_t = s, a_t = a]$$

- γ: discount factor $\in [0, 1]$, which tunes the importance of future rewards.

The solution to the problem can thus be seen as an optimal state s_{goal} which must be reached via a sequence of actions, which will grant a reward and will change the current state. Such a chain of states and actions is called an episode of the Reinforcement Learning algorithm, and it ends whenever certain conditions are met (goal state has been found, or after a maximum number of iterations).

The core concept of a Markov Decision Process is the Reward function, which evaluates a state and the action preformed when the Agent is in it. This value, however, cannot only depend on the very next state, since it would give us no information w.r.t. the goal state. For this reason, the reward function needs to consider all possible future

steps. Thus, any RL algorithm aims to accumulate the maximum amount of reward at the end of an episode: the cumulative reward, also called the return function G_t.

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

If the RL algorithm's main goal is that of maximizing the total amount of reward, then it follows that it is up to the Agent to try and perform the best possible action at any given state. In order to choose what action to take, the Agent follows what is known as a Policy, which is an evaluation distribution over the possible actions at a given state:

$$\pi(a|s) = p(A_t = a|s_t = s)$$

The final goal thus is that of finding an optimal policy π^* that maximizes the return function G_t:

$$\pi^* = argmax(\mathbb{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|\pi])$$

Once such probability is defined, we can introduce the following:

- State Value function, which evaluates how good being in state s at time t under policy π is.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s]$$

- Action Value function (or q-function), which evaluates how good performing action a while in state s at time t under policy π is.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|s_t = s, a_t = a]$$

The link between the Value function and the q-function is given by Bellman equation, which, in turn, gives a recursive definition of the q-function itself:

$$q(s, a) = R_s^a + \gamma \sum_{s'} \max_{a'} q(s', a')$$

This is a fundamental equation, since it allows the RL algorithm to build the q-table, needed to train the agent. In the Q-learning algorithm, the q-table is constructed so that each cell q_{s_i,a_i} contains the action value for performing action a_i while in state s_i:

	a_0	$\cdots$	a_m
s_0	$q(s_0, a_0)$	$\cdots$	$q(s_0, a_m)$
s_1	$q(s_1, a_0)$	$\cdots$	$q(s_1, a_m)$
$\vdots$	$\vdots$	$\ddots$	$\vdots$
s_n	$q(s_n, a_0)$	$\cdots$	$q(s_n, a_m)$

4. Implementation

Now that we have a more robust theoretical basis, we can see into more details how the implementation of the Reinforcement Learning algorithm is used to try and solve the TSP.
To properly actualize the entire system, we need to implement the following:

1) Define the Environment in which the Agent operates
2) Define the Action space of possible Actions the Agent can take
3) Define the Reward function which compensates the Agent after taking an Action
4) Initialize the Agent
5) Repeat:
    a) the Agent selects an Action to perform according to policy π
    b) the Agent performs the Action, receives a Reward, and the state is updated
    c) the policy π is updated

4.1. Environment

As previously stated, in a Reinforcement Learning model the Environment is one of the two main components that needs to be properly implemented.
In order to do so, we are using the Gymnasium Python library [1], which provides an API for all single agent reinforcement learning environments, and, what interests us the most, a way to define and build custom ones.
The actual implementation can be viewed in the provided Jupyter Notebook , while here we simply highlight the design choices made to build the custom environment.
If we recall the TSP formulation seen in section 2, we can see that using a graph-based environment is the best choice for this kind of problem. Thus, the Environment can be seen as the TSP instance graph, and the Agent as the traveling salesman themselves. The RL episode can therefore be summarized as:

1) The Agent starts from the Start node
2) The Agent chooses what node to visit next
3) The Agent moves to the next node, accumulating a reward based on the node chosen
4) Repeat steps 2-3 until all nodes have been visited once, or after a certain number of steps

Formally, this means defining the Observation Space (i.e. what the agent can observe from the system in order to perform an action), and the Action Space (what the agent can actually do).
The former is defined by three variables:

- Agent Position, that represents the node where the agent is currently at and is encoded as the index of the node in the graph (arbitrarily ordered)
- Visited Nodes, that consists in the list of nodes in the graph, each marked with the number of times they have been visited (updated at each iteration)
- Neighbors, which is the list of outgoing connections from the current node

While the Action Space is simply the entire list of nodes, since instance is defined as a completed graph, and the Agent can move from any node to any other. Note that this means that the Agent is not prohibited from moving to a

node they already visited, and nodes can thus be visited multiple times: since the TSP requires a maximum of one visit per node, we will see later in section 4.2 that this is handled by the reward function.
Another two important variables to keep track of are the current path through the graph, encoded as a dynamic list of node indices, and the total cost accumulated, exclusively in terms of weight of edges traveled (so it does not correspond to the reward function).

At the start of each episode the Agent is moved to the Start node of the graph, the nodes are all set as not visited, the current path is initialized just to the starting one, and the total cost is set to 0.
From then on, each time the Agent performs an Action based on an observation of the system, all the variables mentioned are updated, the state of the Environment is changed, and the Agent receives a Reward. In the next section, we will see how exactly the Reward function is calculated.

4.2. Reward function

The main difficulty in building a good Reinforcement Learning model is in defining the Reward function correctly. There is no global guideline on how to properly implement it, so a lot of trial and error is needed.
For this specific problem I wanted a reward function which could be explained intuitively by looking at the TSP formulation. Because of this, I decided to implement it so that the Environment rewards the Agent in the following way:

- Punish the agent for taking connections with higher cost
$$R_{ss'} = -a \cdot c_{ss'}$$
where $c_{ss'}$ is the cost of edge (s, s'), and a is a tuning hyperparameter.
- Reward the agent if they visit a new node
$$R_{ss'} = +b \cdot 1$$
where b is a tuning hyperparameter.
- Punish the agent if they visit an already visited node
$$R_{ss'} = -c \cdot v(s')$$
where $v(s')$ counts how many times s' has been visited, and c is a tuning hyperparameter.
- Greatly reward the agent if all nodes have been visited at least once (which corresponds to the final goal)
$$R_{ss'} = +d \cdot 1$$
where d is a tuning hyperparameter.

Note that many different versions of the Reward function were tested, both by tuning the weight hypyerparameters, and by considering sparser rewards (such as only rewarding the agent when completing a graph).

4.3. Rendering

A good implementation of a Reinforcement Learning model also includes a visual rendering of each action the agent takes and how they effect the environment. For this project, I decided to implement a simple console-based rendering: an episode is represented by the current path through the graph and the associated total cost of edges traveled.
An example of rendering of a certain episode might be the following:

$$(0- > 7- > 6- > 1- > 3- > 5) : 8.20$$

4.4. Agent Training

The main difficulty Reinforcement Learning is getting the Agent to actually learn from its actions. Building an RL training algorithm from scratch is not trivial, and goes beyond the scope of this project. Thus, we are using Stable Baseline [2], a Python library that offers a comprehensive set of algorithms that cover a wide range of RL techniques, including value-based methods like Deep Q-Network (DQN), policy-based methods like Proximal Policy Optimization (PPO), and actor-critic methods like Advantage Actor-Critic (A2C) or Soft Actor-Critic (SAC). These are all different training methods, each with advantages and disadvantages. For instance, SAC can only work in a continuous action space, which is not applicable in this case; on the other hand, PPO might take a long time to converge to a solution, and we might not see any improvement for numerous iterations. Moreover, DQN is very sensitive to hyperparameter settings, requiring careful tuning, and works best in tasks with sparse rewards.
For this project, I decided to test PPO and A2C, since they both work for discrete action spaces and do not require extensive hyperparameter optimization, since training took a non-insignificant amount of time. Therefore, the hyperparameter tested with both models are the following:

- Learning Rate: 0.0003,
- Gamma: 0.99,
- Policy Neural Network: two-layered network with a number of hidden nodes equal to the number of nodes of the problem instance.

5. Results

To start off, we can compare the results obtained by the Reinforcement Learning model by testing it on a simple 10-node graph, after being trained by using the previously-mentioned algorithm.
In figure 4 we can see a good solution to the problem: the agent visits every node in the graph with an associated cost of 3.34. The reward for this solution is 1.66. For problems this small, we can solve the problem and find its true solution by using an exact algorithm, as mentioned in section 2.3. This is shown in figure 5, which shows the

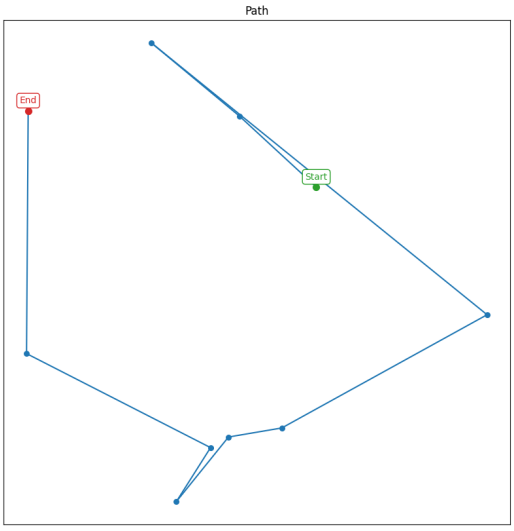

Figure 4: RL solution to the TSP for a 10 node instance (3.34).

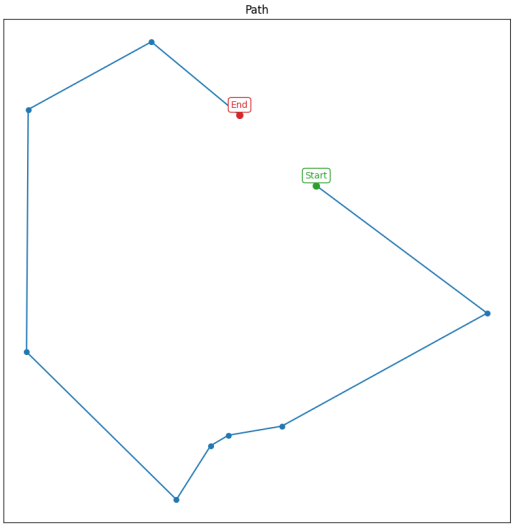

Figure 5: Exact solution to the TSP for a 10 node instance (2.42).

actual solution to the TSP for this case: the optimal path has an associated total cost of 2.42. However, this is just a specific solution to the problem found by the RL agent. In order to see if it can consistently find good solutions, a more systematic testing approach is needed.
Consider figure 6, which shows the distribution of solutions (in terms of total cost) to the previous problems, but with a non-deterministic agent over 10000 runs. We can obviously see how, on average, the RL agent finds a much worse solution compared to the exact one. Note that the solutions found have no actual guarantee to be acceptable, since the agent often visits each node more than once, even if it is being punished for doing so. If we calculate how many runs out of the 10000 would actually be accepted, the

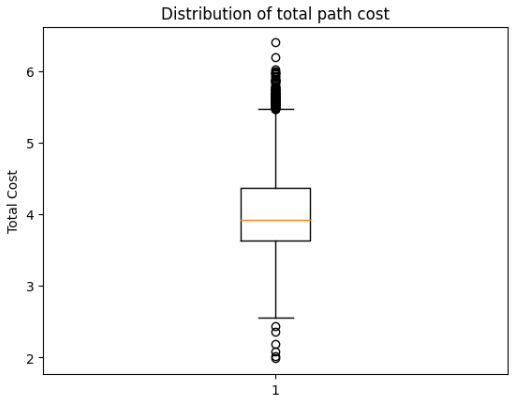

Figure 6: Distribution of solutions found by the RL agent.

result would be around 3.5% (one of which is reported in figure 4). We can also consider the cumulative reward in function of number of episodes, as shown in figure 7: we can see how the agent learns as the number of iterations increases. However, further testing is needed (by increasing the number of iterations) in order to make any conclusive statement.

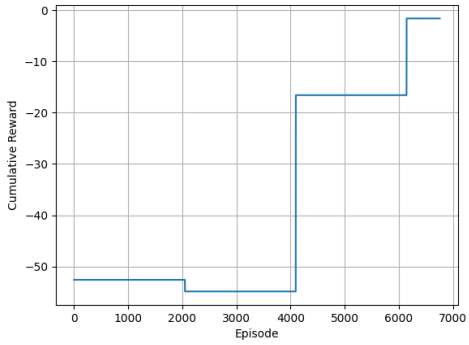

Figure 7: Cumulative reward in function of number of episode.

If we now consider bigger instance sizes, such as a graph with 50 nodes, we can see how the RL agent struggles to find a reasonable path, as can be seen in figure 8. It does not visit each node, and it often comes back to nodes it has already seen.

We can also compare the different approaches when increasing the problem size. For example, as shown in figure 9, we can see how the different algorithms behave when scaled in terms of execution time (note that the Reinforcement Learning model also takes into account the training phase). As we can clearly see, the difference between classical heuristics and the Reinforcement Learning model is not significant, while exact methods (such as brute-force) explode in terms of time complexity (only the

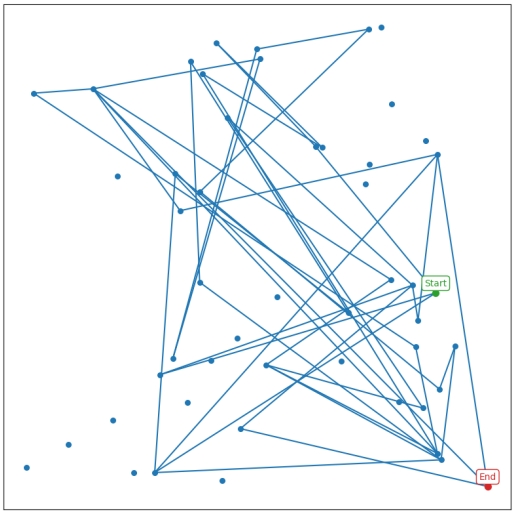

Figure 8: RL solution to the TSP for a 50 node instance.

first few runs have been shown for obvious reasons).
If we look at figure 10, we can see how the classical heuristic almost always finds a better solution compared to the RL agent. One might be prone to say that sometimes the machine learning approach reaches better solution, but, as discussed previously, there is no guarantee that the solution found is acceptable in terms of TSP constraints.

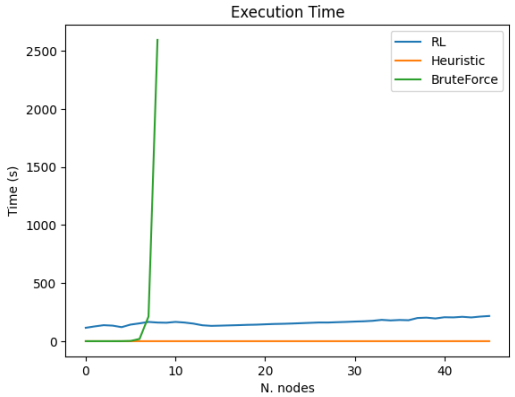

Figure 9: Execution times of different algorithms on increasing problem sizes.

6. Conclusion

From the results obtained, I can say that the RL model I built is not a good enough replacement for classical heuristics when solving the TSP. The aim of this project, however, was that of building a custom RL environment, and I believe I did so successfully.
Overall, I am very happy with my work, and I hope that it could prove useful for further testing.

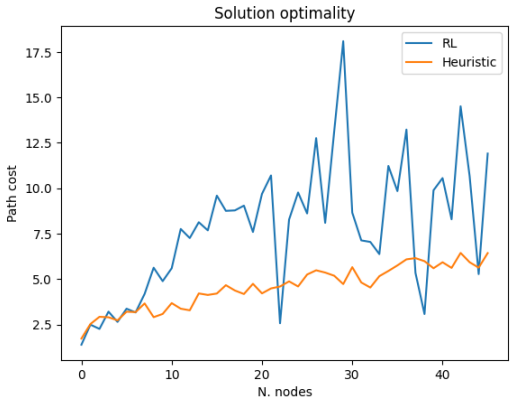

Figure 10: Solution optimality of different algorithms on increasing problem sizes.

7. Future Direction

This project provides a robust environment for testing different Reinforcement Learning models for the TSP. In the future, I would like to see some good results in terms of solution optimality, especially when solving problems of substantial size. To reach this goal, I would need to deploy a more systematic approach for tuning the model hyperparameter and testing different reward functions. Additionally, as mentioned multiple times in this report, the TSP constraints are not actually imposed on the RL agent, so there is no guarantee that the solutions found are actually acceptable; because of this, I would also like to experiment with giving the model some hard-coded constraints in terms of the moves the agent can take, removing the ability to move to already-visited nodes.
Furthermore, I would like to create a visual rendering of the agent and its interaction with the environment by plotting the graph at each step of an episode.

References

[1] **Gymnasium**: standard API for reinforcement learning [https://gymnasium.farama.org/]

[2] **Stable-baseline3**: set of reliable implementations of reinforcement learning algorithms in PyTorch [https://stable-baselines3.readthedocs.io/en/master/]

[3] **python-tsp**: python library for solving typical Traveling Salesperson Problems [https://github.com/fillipe-gsm/python-tsp]