Reinforcement Learning For Continuous Probabilistic Search Tasks With Unknown Priors

Oleg Zilman
EID: oz572
ozilman@cs.utexas.edu

Christian Onougu
EID: cuo225
chrisonuogu77@gmail.com

Abstract

[Student's note: we use the abstract section to relay important information about our project]

1. Code re-use: We used the framework of the abstract classes provided to us in the programming assignments. Otherwise we wrote all our code by ourselves, including implementation of DynaQ, semi-gradient-Sarsa, and function approximation.
2. See reference [15] regarding the nearest neighbor algorithm for PTSP.
3. Meeting the stretch goals:
   We did meet the approximation stretch goal.
   We did not meet the non-stationary problem stretch goal.

1 Introduction

Combinatorial optimization problems with probabilistic elements is a class of problems that have been widely studied in the field of operational research. In this paper, we study a less-known variant of the Probabilistic Traveling Salesman Problem (PTSP), and present a solution using the Reinforcement Learning framework.

Consider an agent that is tasked with finding a collection of tangible items in a physical environment, where each class of items is distributed in the physical space according to some probability distribution. The distribution of each class of items is not known to the agent a-priori, and the distributions are not necessarily stationary over time. Also, the search is not a one-time task, but the agent is expected to perform item finding tours many times over the course of its life. The agent's objective, in addition to finding all the items, is to minimize the time, or the distance, it takes it to complete the search in expectation.

A practical motivation for studying this problem is life-long learning. For example, imagine a house-hold assistant robot that is frequently tasked to find certain objects in the house; a foraging robot looking for different type of resources; or a purchasing robot assigned to buy certain types of goods available in different markets.

As a more elaborate example of our problem, imagine a robot assigned to forage for mushrooms in a forest. The agent, performing the search multiple times and exploring the forest, gradually learns that Shiitake mushrooms are usually located in one part of the forest, that Chanterelle mushrooms thrive in sunny patches, and that White Button mushrooms tend to be next to riverbanks. After the agent has done a fair amount of exploration and learning of the forest, it is asked to pick 5 Chanterelle mushrooms, 10 White Buttoned mushrooms, and 5 Shiitake mushrooms. Based on its learned experience, it would construct a motion plan to collect all the mushrooms in minimal time.

2 Related Work

The problem that we study can be modeled as a variant of the well-known Probabilistic Traveling Salesman Problem (PTSP), first presented by Jaillet [2]. The PTSP is an extension of the Traveling Salesman Problem where, at any given instance, only a stochastically selected subset of graph vertices needs to be part of the tour. Specifically, each vertex i has a certain probability P_i of being included in the tour, and these probabilities are independent. The goal is to find an a-priori tour through all the vertices, that will minimize the expected cost of the tour.

If we formulate our problem as a graph where we discretize the physical space to be the set vertices, and only the vertices where the items are located need to be part of a sample tour, then it is akin to the PSTP. However, our problem differs from the original PTSP; in our case the probabilities of each vertex being included in the tour are dependent on one another, whereas in the PTSP they are independent.

There are numerous variations of probabilistic TSP problems studied in the field of operational research. For example, the Stochastic TSP is a variant where the tour vertices are deterministic but the edge weights are stochastic; the Traveling Purchaser Problem with Stochastic Prices involves an agent that needs to buy merchandise in certain markets and where the prices of the items in each market are stochastic (Kang & Ouyang [8]); There is also the variation of the Stochastic Vehicle Routing Problem (Laport [9]), in addition to several others.

Previous works present a wide array of polynomial-time heuristics to find sub-optimal solutions for this NP-hard problem. These works include decades old 2-opt and spatial curve solutions presented by Bertsimas & Howell [3], and

https://github.com/zilmano/RL-stochastic-search.git
https://youtu.be/JhH0YdRJQG8

Bertsimas et al. [4], and up to modern methods such as-genetic algorithms or particle swarm optimization (Wang et al. [7]).

We were not able to find prior works on the exact specific variant of the PTSP that we present in our paper, and it appears to be yet unexplored. Moreover, the problems mentioned in the previous paragraph all assume that the probability elements of the problem are independent, and we were not able to find any papers presenting solutions for variants where the probabilistic variables of the problem are dependent on each other. Nonetheless, we will refer to the heuristic solution methods of those problems for a baseline comparison to our RL algorithm's performance.

An important distinction between the problem that we present and the papers discussed above is that they all assume the probability distributions are known a-priori, whereas in our case the probability distributions of the item locations (or the vertices selection, under the PTSP formulation) are not known to the agent. The agent would be learning the probability distributions and in parallel would be constructing the optimal search path.

In the Reinforcement Learning and Robotics fields, our problem is most closely related to a class of problems known as *Continuous Sweeping Tasks*. Ahmadi & Stone [5] presented a method to find an optimal sweeping policy for an area where items are being randomly generated according to an unknown distribution, and where the agent learns the optimal sweeping policy while traversing the space and sampling the distribution. However, our problem differs from *Continuous Sweeping Tasks* in that we are not trying to find a sweeping policy to cover all the points in the search space; on the contrary, the agent should visit as few points as possible on its way to find the items.

In Robotics, there are papers dealing with *Robot Foraging*. Some of those do not involve Reinforcement Learning, and focus planning (S. Liemhetcharat [10]). Other papers do involve reinforcement learning and specifically Q-learning (Gau & Meng [11]), however those papers pertain mostly to multi-robot interaction and optimization strategies**.** We did not find any papers about *Robot Foraging* that discussed path optimization for foraging a stochastically distributed resource with an unknown prior, like we do in this present paper.

Finally, there has been research on applying Reinforcement Learning to the original TSP (see Nazari et al [17] and Gambardella & Dorigo [16]). However, in these works there is no stochasticity at all, and Reinforcement Learning is explored as another method to try to find a polynomial-time solution heuristic to this NP-hard problem. They are not directly relevant to our work, but we thought it would be an interesting mention.

3 Description of The Problem and Our Approach

In this section, we formulate our problem in detail, describe our Reinforcement Learning (RL) approach, and present a simple heuristic solution for the same problem when the prior distribution is known, to serve as a baseline to evaluate the effectiveness of our solution.

3.1 Problem Formulation

We model our problem as a graph *G(V,E)*, where $V = \{v_1, \dots v_m\}$ is the set of discrete locations in the physical space, and $E = \{e_{ij}\}$ is the set of edges connecting each couple of locations *i,j*. Each edge has a non-negative cost d_{ij} which is the distance between locations i and j, and is deterministic and known. Let $I = \{i_1, \dots i_{\}k}\}$ be a list of items types. Each item type has an associated probability mass function $P_i(v), v \in V$, according to which the location of the item is determined. These probability mass functions are not known a-priori.

For each instance of the problem, that we call a hunt or an episode, a non-negative number of items for each item type is generated according to the probability mass functions. The agent is given a list items that it should find, and the agent's goal is to construct a path that will pass through vertices containing all the items in the list, such that the expected travel distance of the path will be minimized. The agent terminates the path as soon as it encounters the last item on the list, and then the total traveled distance for the hunt is calculated. Additionally, we assume that the items that the agent was tasked to find actually exists in the hunt.

The path does not necessarily need to be Hamiltonian, though this would be a desired property. As known from graph theory, the task of finding the shortest Hamiltonian path in a graph is NP-hard. Though we do not require the path to be Hamiltonian, the problem is still NP-hard, as we can equate retracting through previously visited vertices to reach a new vertex as having an additional edge in the graph.

After providing this general formulation, in this paper we will focus on a specific case, where the graph is a rectangular grid graph with holes (Acharya and Gill [12]). This coincides with the grid world example widely used in Reinforcement Learning (Sutton & Barto [1]). Each vertex in the grid graph corresponds with a cell in the grid world, each edge will have a cost of 1, and each barrier in the grid world will correspond to a hole in the graph For the purpose of this paper, we will use the terms grid graph and grid world interchangeably, and the terms cell and vertex interchangeably.

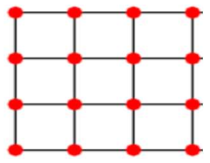


Fig 1. Rectangular Grid Graph

3.2 Baseline Combinatoric Optimization Algorithm

In order to see how well our reinforcement learning agent performs, we need a baseline for comparison. One such baseline may be a simple deterministic sweeping motion plan that passes through every cell in the grid. Such policy is not difficult to manually find in a rectangular grid. We will be using the "crisscross" plan presented in fig 2. The mean travel distance of such a policy would be dependent on nature of the probability distribution – how likely that the last

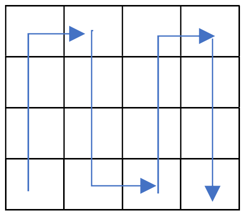


Fig 2. “crisscross” sweeping policy

item would be closer to the start or end point of the sweep. We do not derive a formula for the mean length of the “crisscross” here, and will estimate its average experimentally in section 4.

However, we would prefer a better baseline for comparison than a sweeping policy that does not take any probabilistic knowledge into account. We compare our RL algorithm to a competitor that knows the a-priori distributions and uses one of the heuristic methods for solving the PSTP described in section 2 to calculate the optimal motion plan through the grid. For simplicity, we are going to use a *Nearest Neighbor* greedy algorithm (Yadidsion et al [15], see the references section). The algorithm would work in the following way at each step:
From all neighboring vertices that have not been visited, select the vertex that has the highest aggregate probability (the sum of the probabilities for each item to be in that vertex). If all neighbors have been already visited, go to the next depth of neighbors (neighbors of the neighbors) and do the same, and so on.
[Students' note: As this is a course project, for simplicity we are using a nearest neighbor technique. Ideally we would have tried to adapt and modify one of the modern PTSP heuristics, such as the PSO, to work for our variant of the problem.]

Nearest Neighbor baseline algorithm

```
K := size of list of items
curr_V := arbitrary $v \in V$
visited_vertices := {curr_V,}
item_count := 0
if <item found in curr_V>:
    item_count += 1
loop until item_count = K or visited_vertices = |V|:
    unvisited_neighbors := {}
    depth = 0
    while unvisited_neighbors ={}:
        neighbors := BFS(curr_V,depth)
        foreach neighbor in neighbors:
            if neighbor not in visited_vertices:
                append neighbor to unvisited_neighbors
        depth += 1
        neighbors := BFS(curr_V,depth)

    next_V:=<vertex with max aggregate probability in
                neighbors>
    if <item found in next_V>:
        item_count += 1
    append next_V to visited_vertices
    curr_V:=next_V
```

Here, the *aggregate probability* of vertex *v* is a sum of the probabilities of each item on the list to be in vertex v, and thus can be greater than one. That is acceptable, as the *aggregate probability* is not a true probability; we simply use it as an heuristic metric of “how a vertex is worth visiting”**.**

3.3 Reinforcement Learning Approach

3.3.1 Algorithm

We decided to use the DynaQ algorithm, presented by Sutton [13], for our problem. As the problem we preset involves a machine moving through physical space, each step will be fairly time consuming in comparison to a simulation step; the machine moving pace would be slower than the calculation frequency of a modern computer. We would like to use the “dead” time while the machine moves between different cells to perform additional simulations and speed up our learning process. This is why we selected a method that integrates learning and planning, such as DynaQ.

The DynaQ algorithm is quite similar to regular Q-learning (Watkins [18]) as it too uses real environment experience to directly update its value function, but it adds a key component of a model. It gradually develops the model by using the aforementioned experience, and it can then use the model to simulate experiences, which are used to update the value function. This hybridization of learning and planning is done to make use of the benefits of both direct and indirect learning to speed up the learning process.
Neither [13] nor [1], that we refer to regarding DynaQ, do not specify how a model should be created. To create the model, we use Monte-Carlo to estimate the probability of next state and reward combination given an action and a state. For each state and action S and A, we store the amount of times the transition to next state and reward, S’ and R, has occurred. Let us denote it as $|S, A \to S', R|$.We also store the total number of times each state and action combination occurred. Let us done it as $|S, A|$. Then, to estimate the probability of the transition S,A->S`,R, we do $|S, A \to S', R|/|S, A|$.

3.3.2 Formulation as a Reinforcement Learning Problem.

Here we module our problem for the RL framework and define the state space, the action space, and the reward scheme we are going to use. Our modeling is as follows:

- Action Space:
  There are 4 available actions - “north”, “west”, “east”, and “south”. Each of these actions will take the agent to the next cell in the corresponding direction.

- State space:

Let G be the set of all grid cells, and $|G|$ the size of the grid. Let I be the set of items for a given instance of the problem, and K be the number of the items. Let S be the set of all states, and $|S| = |G| x (2^{|k|} - 1) x 16 + 1$. each $s \in S$ will represent the cell the agent is currently at, the combination of items found, and the combination of the visit status of the neighbor cells of the current location (visit status is a Boolean saying if the cell has been previously visited or not; there are 4 neighbor cells and therefore 16 possible combinations). The "combination of items found" is of size $2^{|k|} - 1$, because the last combination where all items are found would be a terminal state $s^+ \in S$ (regardless of which grid cell the agent is at, or the neighbors' visit status).

- Reward scheme:

The agent receives a reward of +1 when it finds an item by moving on to the grid cell containing that item; a reward of -10 for trying to go off the grid (bumping into the wall); a reward of -5 for moving into a cell that it has already been to in that episode; a reward of -1 for any other action. We were trying to incentivize the agent to complete the task as quickly as possible (hence the -1 reward) and also avoid taking unconstructive actions such as bumping into walls or searching in a location it has already been to (as there can no longer be any item in that cell). We do allow the agent to go through previously visited cells, though not encourage it, as there may be cases were the agent needs to backtrack in order to finish the episode.

To illustrate the state space, here is an example. An agent in grid cell (4,4) with no items found that has only previously been to its north neighbor will be in one state, however an agent in the same cell that has found only the second item will be in a different state. Similarly, an agent located in that same cell that has found no items, but has visited its north and south neighbors will be in a third, distinct, state. This results in the agent learning to perform different actions from the same cell depending on which items on the board it has found and which cells it has already looked in.

4 Experiment Results

4.1 Tabular

We were trying to see if a Reinforcement Learning approach such as DynaQ would be a good fit for this environment, so we compared its performance to (a) the crisscross sweep policy and (b) the nearest neighbor algorithm that knows the probabilities and moves into the cell with the highest aggregate probability in each step. The experiments were done an 8x8 grid with two items, each with their own mean and variance, and with Gaussian probability distributions for their locations. In each experiment, the following applied:

- The mean for each item was randomly selected among all the grid cells, and the variance was randomly set to be 1 or 2. These parameters remain constant throughout the experiment.
- The DynaQ algorithm was trained for 1000 epsidos. At the start of each episode, a new location was selected for both items according to the probability distributions.
- After the DynaQ agent was trained, we ran 100 episodes to evaluate the algorithms. At the start of each episode, a new location was selected for both items according to the probability distributions, same as before. We recorded the average number of steps per episode for each of the methods, over the 100 episodes (for the nearest neighbor we calculated the expected number of steps rather than recording an empirical average).

Our results are presented in figure 3.

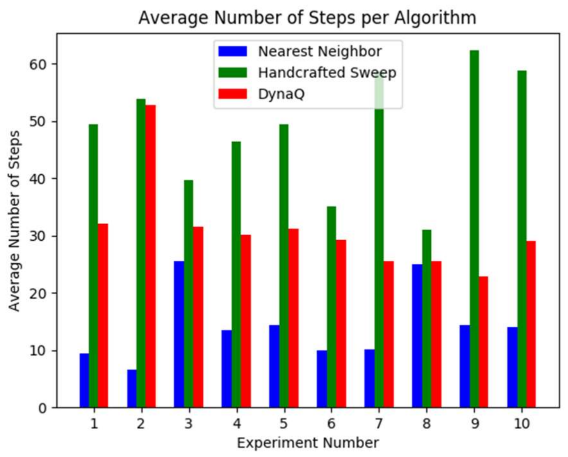


Fig 3. Comparison of the performance of the RL agent, the handcrafted sweep, and the nearest neighbor algorithm

From figure 3. we can see that the DynaQ algorithm was routinely able to beat the handcrafted sweep policy, but was itself outperformed by the nearest neighbor algorithm. Our results seem to suggest that a reinforcement learning approach such as DynaQ can achieve good for this environment even with the stochasticity. We think that the algorithm would likely perform even better than the average on larger environments as the effect of the variance on the item locations would be lessened.

[Students' note: We actually made that experiment, did not have time to include the results in the formal paper. We made a test for a 15x15 grid, and the RL agents was actually very close to the nearest neighbor there, and in many cases beating the nearest neighbor. You could modify the grid parameter in our code to run that test if you wish, the instructions are in the readme file.]

In fact, we found it a little bit surprising that the agent was able to achieve these good results. Technically, with the state representation we presented, the agent does not have enough information to make well informed decisions to avoid getting into loops. Once an agent reaches a cell that all its neighbors are already visited, it would not know how is best to proceed, and in what direction the next unexplored cell is to be found. And in fact, the deterministic policy that the DynaQ algorithm produces does include loops. We do not quite understand the mechanism, but somehow the combination of what the agent was able to learn from its limited observations, and the stochasticity we introduce by keeping the policy ε – greedy, produce good results.

We think the agent could have performed better if the state space would have included the visit status of all the cells in the grid, and it would have the information on what cells have not been visited. This, however, would result in an astronomical state space ,$O(2^{|G|})$, but trying to approximate it using some function approximation method could be a potential approach. In the next paragraph we present a linear approximation scheme to try and capture that information.

Alternatively, the use of options (Sutton et al [19]) could be explored to improve performance. The agent would have a table of the visit status of all the grid cells at its disposal. There is no exponentiality in |G| in this case, as the agent only needs to know which cells are yet to be visited, and not a combination of thereof. We extend the action space such that, in addition to the fundamental actions "West", "East", "South" and "North", the agent would have options to move to any previously unvisited cells. An option could for example be a sequence of the fundamental actions that represent the shortest path to the selected unvisited cell (here the agent will use planning to determine the shortest path). Therefore in each step, the number of options would be equal to the number of unvisited cells. Then these options could be treated just as regular actions in the Q-learning done by the agent. Exploring this approach is a potential direction for future work.

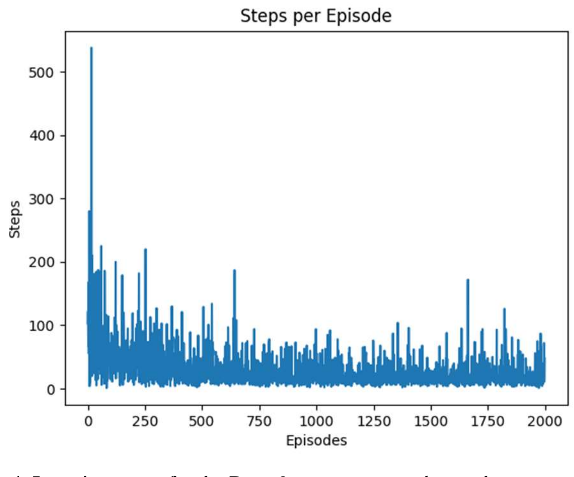


Fig 4. Learning curve for the DynaQ agent, averaged over the ten experiments.

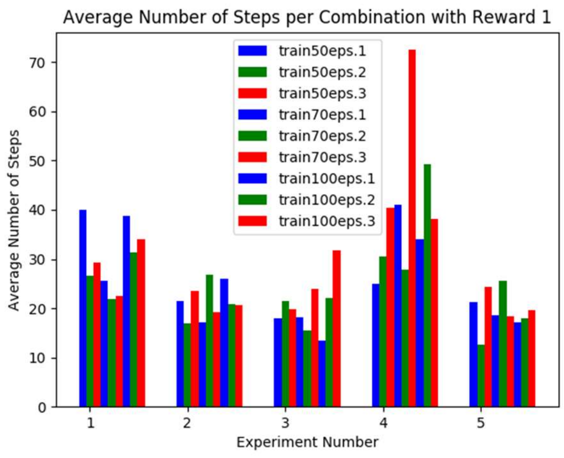


Fig 5. Parameter sweep results - inconclusive

When choosing the parameters we used for our experiments, we tried to strike an effective balance between performance and computation time. We chose 1000 episodes of training because 500-750 episodes seemed to be about where our Dyna-Q algorithm reached its best performance (see fig 4). We chose 50 model training steps as we didn't see a significant increase in performance when increasing it past this point, and having it higher results in an increase in computation time. We chose a reward of 1 for finding an item as we tested many different reward numbers from -1 to 20 and didn't really see any significant difference, so we just set it to 1. We had similar results with epsilon determining how greedy our learned policy should be (tested 0.1, 0.2, and 0.3), so we set it a 0.3 in hope that more exploration would help with faster convergence. Figure 5. illustrates a parameter sweep we did in an attempt to determine which parameters setting are preferred. We tried different variations of n and epsilon, and as it can be seen, the results were inconclusive.

4.2 Approximation

As mentioned earlier, the state representation we used is fairly limited in the amount of information it provides the agent with. In order to have all the information it needed to solve the problem effectively, the agent should know exactly which vertices have already been visited. That way it would be able to direct itself to unexplored vertices from each state. Having this information represented in the state space would not be feasible, and for a 15x15 grid the state space would be of the order of 2^{225}. We would like to propose an linear approximation scheme, as suggested in Sutton & Barto [1], where the visit status of each grid cell is a feature of the state. Our approximation scheme would be as follows:

- We define a feature vector, $X(s) = X_c \cap X_i \cap X_v$ where:
  $X_c \epsilon \{x_1^c, \ldots x_{|V|}^c\}$, each x_j^c is one when the agent is in grid cell j and zero otherwise (this is basically a tabular representation of the states).

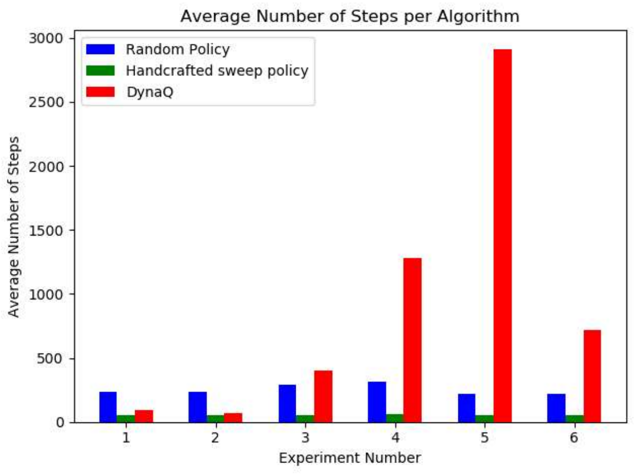
$X_i \epsilon \{x_1^i, \ldots x_K^i\}$,each x_j^i is one when item j is found and zero otherwise, and K is the total number of items in the hunt.

$X_v \epsilon \{x_1^v, \ldots x_{|V|}^v\}$, each x_j^v is one when cell j has already been visited, and zero otherwise.

* We use the semi-gradient Sarsa algorithm, as described by Sutton & Barto [1], with the feature vector *X(s)* to find the optimal policy (we keep it e-greedy).

We ran the algorithm on a 8x8 grid, with two items, generated according to the same distribution as in the tabular case. We did not implement an approximation version of Dyna-Q here as we wanted to see first if our method worked; rather, we used a regular semi-gradient Sarsa as it was simple to implement. We ran it on 40000 episodes, which would result in same amount of TD updates as 800 Dyna-Q episode (we divide by n, which is 50). That way we make sure both algorithms trained roughly the same time. We used $\varepsilon = 0.4$.

Our results here are not as good as in the tabular case. The learned policy was better than a completely random policy in some cases, which suggest that some learning did occur, but it was not able to beat the "crisscross" sweep policy in any scenario. In other cases, the learned policy completely diverged and was way worse than even a random competitor. (see figure 6). Our conclusion is that linear approximation is simply not strong enough to handle the high complexity of the state space we want to represent, and more powerful approximation technique, such as deep neural networks, is prudent.



5 Conclusions and Future Research

We have shown, that using a decades old RL algorithm (Dyna-Q) without any enhancements, using a simple uniform sampling for the planning phase, with only limited observations of the state, and with zero knowledge about combinatorial optimization techniques, we were be able to reach good results that favorably compare to our baseline heuristics for solving TSP-like problems. That shows that there is potential to using Reinforcement Learning methods for solving combinatorial optimization problems with unknown priors.

Though the grid sizes that we used for our experiments were small (8x8 and 15x15), and though movement grids for real-world robots or vehicles are likely to be much larger, our results can be of practical value as they are. Consider that we formulated the problem as a general graph, and though we made our experiments on a grid graph, the approach we presented will apply to any undirected graph. A graph of 225 vertices, such as the 15x15 grid, can represent realistic problems. For instance, if we say that each vertex in the graph represents a room in a building, and the edges are corridors between the rooms, we can represent a fairly large building. Alternatively, the edges could be routes between pick-up and drop-off locations for a delivery vehicle (Vehicle Routing Problem).

Possible directions for future research include introducing the use of neural networks to and solve the problem for larger grids or to achieve better results. Our linear function approximation did not handle well the high complexity of the problem, but we think it would be interesting to use deep neural networks to try to approximate the full-state representation. Another direction for future research is be to try RL on canonical combinatorial optimization problems, like PTSP, Stochastic Vehicle Routing Problem etc, when the priors are unknown. Additionally, we think it would be interesting to see to see how an RL approach would handle non-stationarity, and how we can introduce "forgetfulness" mechanisms to help it deal with it better. Finally, the assessment would not be complete without comparing the RL approach to methods that use combinatorial optimization algorithms and learn the probability distributions with experience, to see if RL can offer an advantage over Bayesian learning in this case.

Acknowledgments

The authors would like to thank Harel Yadidsion with helping to coming up with this project idea, for his suggestions for the baseline comparison, and general advice.

References

[1] Sutton, R. and Barto, *A. Reinforcement learning: An introduction*. Cambridge, MA, MIT Press (1998)

[2] P. Jaillet, *Probabilistic Travelling Salesman Problems,* Ph.D. thesis, MIT. (1985)

[3] D. J. Bertsimas and L. Howell. Further results on the probabilistic traveling salesman problem. *European Journal of Operational Research*, 65:68-95. (1993)

[4] D. J. Bertsimas, P. Chervi, and M. Peterson. Computational approaches to stochastic vehicle routing Problems. *Transportation Sciences*, 29(4):342-352. (1995)

[5] Mazda Ahmadi and Peter Stone. Continuous Area Sweeping: A Task Definition and Initial Approach. In *The 12th International Conference on Advanced Robotics* (2005).

[6] S. Al Dabooni and D. Wunsch**.** Heuristic dynamic programming for mobile robot path planning based on Dyna approach**.** In *2016 International Joint Conference on Neural Networks (IJCNN),* Vancouver, BC, , pp. 3723-3730. (2016)

[7] K.P. Wang,L. Huang, C.G. Zhou,W. Pang, Particle swarm optimizasion for traveling salesman problem, *International Conference on Machine Learning and Cybernetics* 1583–1585. (2003)

[8] Kang, Seungmo & Ouyang, Yanfeng. (2011). The traveling purchaser problem with stochastic prices: Exact and approximate algorithms. *European Journal of Operational Research.* 209. 265-272. (2010)

[9] Gilbert Laporte. The vehicle routing problem: An overview of exact and approximate algorithms. European journal of operational research, 59(3):345–358, (1992)

[10] S. Liemhetcharat, R. Yan, K. P. Tee, and M. Lee.Multi-robot item delivery and foraging: Two sides of a coin.*Robotics*, 4(3):365–397, (2015)

[11] H. Guo and Y. Meng, Distributed reinforcement learning for coordinate multi-robot foraging, *J. Intell. Robot. Sys.*, vol. 60, nos. 3–4,pp. 531–551. (2010)

[12] Acharya, B. D. and Gill, M. K. On the Index of Gracefulness of a Graph and the Gracefulness of Two-Dimensional Square Lattice Graphs. *Indian J. Math.* 23, 81-94. (1981)

[13] R. S. Sutton**.** Reinforcement Learning Architecture For Animates. In J.-*A. Meyer & S. Wilson (Eds.), From animals to animats: Proceedings of the First International Conference on Simulation of Adaptive Behavior*. Cambridge, MA: MIT Press. (1990)

[14] R. S. Sutton (1996). Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In *Touretzky, D., Mozer, M., & Hasselmo, M. (Eds.), Neural Information Processing Systems 8*. (1995)

[15] This not an actual paper (so far), but a project that a team of FRI students is working on under Harel's guidance. They develop combinatorial optimization heuristics (non-RL) to solve a problem similar to what we present here (on smaller scale graphs, and involving actual robots). We got the idea for the nearest neighbor on aggregate probabilities algorithm from them, and they are using it as well in their project. We wrote all our code for the algorithm by ourselves, but we wanted to mention that we were exposed to the idea of using the aggregate probability as a metric in our mutual discussions, and make sure we give proper credit.

[16] L.M. Gambardella and M. Dorigo. Ant-Q: A reinforcement learning approach to the traveling salesman problem in *Proc. Twelfth International Conference on Machine Learning (ML-95)*, A. Prieditis and S. Russell, Eds., Morgan Kaufmann Publishers, pp. 252–260, (1995)

[17] M. Nazari, A. Oroojlooy, L. V. Snyder, and M. Takáˇc. Deep reinforcement learning for solving the vehicle routing problem.*arXiv preprint arXiv*:1802.04240, (2018)

[18] Watkins, C. J. C. H. (1989).*Learning from Delayed Rewards*. PhD thesis, King's College, Oxford.

[19] Sutton,R.S.,Precup,D., Singh,S. Between MDPs and semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence* 112:181-211, (1999).