Syracuse University

SURFACE

Dissertations - ALL SURFACE

August 2017

Risk-aware navigation for UAV digital data collection

Zhi Xing
Syracuse University

Follow this and additional works at: https://surface.syr.edu/etd

Part of the Engineering Commons

Recommended Citation
Xing, Zhi, "Risk-aware navigation for UAV digital data collection" (2017). *Dissertations - ALL*. 776.
https://surface.syr.edu/etd/776

This Dissertation is brought to you for free and open access by the SURFACE at SURFACE. It has been accepted for inclusion in Dissertations - ALL by an authorized administrator of SURFACE. For more information, please contact surface@syr.edu.

ABSTRACT

This thesis studies the navigation task for autonomous UAVs to collect digital data in a risky environment. Three problem formulations are proposed according to different real-world situations. First, we focus on uniform probabilistic risk and assume UAV has unlimited amount of energy. With these assumptions, we provide the graph-based Data-collecting Robot Problem (DRP) model, and propose heuristic planning solutions that consist of a clustering step and a tour building step. Experiments show our methods provide high-quality solutions with high expected reward. Second, we investigate non-uniform probabilistic risk and limited energy capacity of UAV. We present the Data-collection Problem (DCP) to model the task. DCP is a grid-based Markov decision process, and we utilize reinforcement learning with a deep Ensemble Navigation Network (ENN) to tackle the problem. Given four simple navigation algorithms and some additional heuristic information, ENN is able to find improved solutions. Finally, we consider the risk in the form of an opponent and limited energy capacity of UAV, for which we resort to the Data-collection Game (DCG) model. DCG is a grid-based two-player stochastic game where the opponent may have different strategies. We propose opponent modeling to improve data-collection efficiency, design four deep neural networks that model the opponent's behavior at different levels, and empirically prove that explicit opponent modeling with a dedicated network provides superior performance.

RISK-AWARE NAVIGATION FOR UAV DIGITAL DATA COLLECTION

by

Zhi Xing

B.S., Nankai University, 2010

M.S., Syracuse University, 2017

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer and Information Science and Engineering.

Syracuse University

August 2017

Copyright © Zhi Xing 2017

All Rights Reserved

To my wife, Yuchen.

ACKNOWLEDGMENTS

I'm extremely grateful to my advisor Professor Jae C. Oh, who changed my life by giving me the opportunity to study in the program. During my study, Professor Oh, as a mentor and a friend, provided me the most wonderful guidance beyond doing research. Looking back after all these years, one of the things I appreciate the most is the complete research freedom he granted me. Admittedly, it was challenging at the beginning. But as time goes by, I've learned how to explore in addition to how to exploit, how to give in addition to how to receive, and how to lead in addition to how to follow. I've grown so much as Professor Oh's advisee that I'm well prepared to face any future challenges.

I want to thank my family – my wife, Yuchen Deng, parents, Hongxia Xu and Jinshu Xing, and parents-in-law, Guifang Xu and Hong Deng – for their unconditional trust and support throughout the years. Their love gives me the strength and courage to do what I want to.

Last but not least, I want to thank my committee members Professor Young B. Moon, Professor Chilukuri Mohan, Professor Qinru Qiu, Professor Sucheta Soundarajan and Professor Jian Tang. Their insights and suggestions have greatly enriched my work.

TABLE OF CONTENTS

Page

ABSTRACT i

LIST OF TABLES ix

LIST OF FIGURES xi

1 Introduction 1

1.1 Motivation 1

1.2 Problem framework 2

1.3 Thesis overview 3

2 Background 6

2.1 Hierarchical Clustering 6

2.2 Rooted k Minimum Spanning Tree 8

2.3 Convolutional Neural Network 8

2.4 Markov Decision Process and Stochastic Game 11

2.5 Reinforcement Learning 12

2.5.1 Q-learning 13

2.5.2 Policy gradient 13

2.5.3 Deep Reinforcement Learning 14

2.6 Opponent modeling 16

3 Graph formulation with uniform risk 19

3.1 Chapter overview 19

3.2 Contributions 20

3.3 Problem formulation 20

3.4 Algorithm 22

3.4.1 Clustering for the number of robots 22

3.4.2 Heuristics for building tours 25

Page

3.4.3 The Progressive Gain-aware Clustering algorithm 29
3.5 Evaluation 32
3.5.1 Comparison of tour-building algorithms 34
3.5.2 Comparison of clustering algorithms 35
3.5.3 Effect of node reward variance 36
3.6 Related work 37
3.7 Conclusion 39
4 Grid formulation with non-uniform risk and energy constraint 45
4.1 Chapter overview 45
4.2 Contributions 46
4.3 Problem formulation 47
4.4 Algorithm 49
4.4.1 Maximizing reward without energy constraint 49
4.4.2 Navigation under energy constraint 52
4.4.3 Finding the balance 55
4.5 Evalution 60
4.5.1 Different risk distributions 63
4.5.2 Effect of reward density 65
4.5.3 Effect of energy capacity 67
4.5.4 Single-item data collection 68
4.5.5 Comparison to other learning methods 71
4.6 Related work 72
4.7 Conclusion 73
5 Grid formulation with opponent and energy constraint 75
5.1 Chapter overview 75
5.2 Contributions 76
5.3 Problem formulation 78
5.4 Algorithm 80
5.4.1 State representation 80

Page

5.4.2 No Opponent Modeling 80
5.4.3 Implicit Opponent Modeling 82
5.4.4 Explicit Opponent Modeling with same network 83
5.4.5 Explicit Opponent Modeling with separate network 85
5.4.6 Comparison of the networks 86
5.5 Evaluation 87
5.5.1 Environmental settings 88
5.5.2 Network structure and hyperparameters 89
5.5.3 Evaluation in the standard setting 89
5.5.4 Effect of energy capacity 92
5.5.5 Investigation on the behaviors 95
5.5.6 Five million steps of training 104
5.5.7 Change of opponent strategy 109
5.6 Related work 110
5.7 Conclusion 111
6 Summary and future work 113
LIST OF REFERENCES 115
VITA 118

LIST OF TABLES

Table Page

3.1 Overview of tour-building algorithms. **Criterion** summarizes the what is evaluated by each algorithm. **Insertion position** specifies where does an algorithm insert a new node. **Time complexity** shows the computational complexity of an algorithm. 25

4.1 Results of experiments on the tour-building heuristics from Chapter 3 adopted to DCP. The average is taken from 10,000 runs. For every run, there is one agent with infinite amount of energy, the total reward is 30, and the risk distribution is as Figure 4.1b. **Reward Avg.** is the average of total reward collected during a game. **Reward SD.** is the standard deviation. 50

4.2 Risk values of the three different risk distributions used in Section 4.5.1. Each distribution has four layers of risk values. From outer to inner, the risk values are $\rho^{(0)}$, $\rho^{(1)}$, $\rho^{(2)}$ and $\rho^{(3)}$. As an example, high risk is shown in Figure 4.1b. 61

4.3 Results of experiments on the high-risk distribution specified in Table 4.2. The average is taken from 10,000 runs. For every run, the total reward is 30, and the initial and maximal energy of agent is 8 and 16 respectively. **Reward** is the total reward collected during a game. **Energy** is the total energy consumed during a game. **Reward / Energy** is the reward collected per energy consumed during a game. **Avg.** means the average over 10,000 runs. **SD.** is the standard deviation. **Incr.%** is ENN's increment as a percentage of an algorithm's corresponding average. For example, Safe-Reward's reward incr.% is the difference between ENN's reward and Safe-Reward's reward as a percentage of Safe-Reward's. 62

4.4 Results of experiments on the medium-risk distribution specified in Table 4.2. Terms used in the table and other experimental settings are the same as Table 4.3. 62

4.5 Results of experiments on the low-risk distribution specified in Table 4.2. Terms used in the table and other experimental settings are the same as Table 4.3. 62

4.6 Results of experiments where the total reward is 15. Terms used in the table and other experimental settings are the same as Table 4.3. 66

Table Page

4.7 Results of experiments where the initial and maximum energy levels are set to 16 and 32 respectively. Terms used in the table and other experimental settings are the same as Table 4.3. 67

4.8 Results of experiments where there is only one item of reward one in the environment. Terms used in the table and other experimental settings are the same as Table 4.3. 68

4.9 Results of experiments for a simple linear learner. Terms used in the table and other experimental settings are the same as Table 4.3. 69

4.10 Results of experiments for the EQN shown in Figure 4.4. Terms used in the table and other experimental settings are the same as Table 4.3. . . 69

LIST OF FIGURES

Figure Page

2.1 An example dendrogram from Hierarchical Clustering of 8 elements. The black dots represent elements. The solid lines indicate merges or splits. The dotted lines show different positions for cuts. In top-down order, the cuts result in 2, 4 and 6 clusters. 7

2.2 An example of a convolutional layer with input and outputs. The input is on the left, with a dimension of $8 \times 8 \times 3$. The outputs are on the right, with varying dimensions. There are 6 filters. The 2 red (top) filters have filter size $4 \times 4 \times 3$ and stride size 1. The 4 blue (bottom) filters have filter size $3 \times 3 \times 3$ and stride size 2. The input is padded with zeros for the blue filters. 9

3.1 The performance of tour-building algorithm combined with NC. Node rewards range from 4 to 6. The x-axes show the robot value α and the y-axes show **(a)** the average of the expected reward from 100 runs, where the values are normalized against the maximum, and **(b)** the standard deviations of the expected reward from 100 runs. 41

3.2 The performance of tour-building algorithm combined with PGC. Node rewards range from 4 to 6. The meanings of the charts are the same as those in Figure 3.1. 42

3.3 The performance ratio of PGC to NC. Node rewards range from 4 to 6.The x-axis shows the robot value α and the y-axis shows the ratio of the averaged expected reward of PGC to that of NC. 43

3.4 Node rewards range from 1 to 9. The meanings of the charts are the same as those in Figure 3.1 and Figure 3.3. 44

3.5 Node rewards are all 5. The meanings of the charts are the same as those in Figure 3.1 and Figure 3.3. 44

4.1 An example of the DCP environment. **(a)** shows the locations of the base, agent and items. The robot represents the agent. The money bags represent items. The house represents the base. **(b)** shows a color-coded risk distribution, where the numbers are the risk values. 48

Figure Page

4.2 Extracing a complete undirected graph from a game state. **(a)** shows the original game state. The bottom grid contains *risk values*. **(b)** shows the safest paths calculated by Dijkstra's Algorithm for (from top down) the agent, the left item and the right item. The numbers are *success probabilities* instead of risk values. **(c)** shows the complete undirected graph created from the safest paths. The nodes represent the agent and items, and the edge weights are the success probabilities of safest paths. 50

4.3 Structure of ENN. ENN is composed of a Convolultional Neural Network (CNN) and a number of heuristics H_i, all of which take game state s as input. The heuristics output action vectors $H_i(s)$ and the CNN outputs one weight $w_i(s;\theta)$ for each action vector, a bias $b(s;\theta)$, and a state value estimation $V(s;\theta_v)$, where θ and θ_v represent network parameters. The final policy $\pi(s;\theta)$ is a softmax function σ of a linear combination of all the outputs. 56

4.4 Structure of EQN. EQN is composed of a Convolultional Neural Network (CNN) and a number of heuristics H_i, all of which take game state s as input. The heuristics output action vectors $H_i(s)$ and the CNN outputs one weight $w_i(s;\theta)$ for each action vector and a bias $b(s;\theta)$, where θ represents network parameters. The final output is an estimate of the state-action value function $Q(s,a;\theta)$ for every action a, which is a linear combination of all the heuristic outputs. 70

5.1 An example of the DCG environment. The robot represents the collector. The ghost represents the adversary. The money bags represent items. The house represents the base. 77

5.2 Structure of VNN in Section 5.4.2. VNN is a Convolultional Neural Network. It takes state s as input, and gives the policy $\pi(s,h;\theta)$ and the value estimation $V(s,h;\theta_v)$ as outputs, where θ and θ_v are network parameters. 81

5.3 Structure of IMNN in Section 5.4.3. IMNN is a Convolultional Neural Network. It takes state s and history h as inputs, and gives the policy $\pi(s,h;\theta)$ and the value estimation $V(s,h;\theta_v)$ as outputs, where θ and θ_v are network parameters. 82

5.4 Structure of EMNN in Section 5.4.4. EMNN is a Convolultional Neural Network. It takes state s and history h as inputs, and gives the policy $\pi(s,h;\theta)$, the value estimation $V(s,h;\theta_v)$, and opponent strategy $\pi^o(s,h;\theta_o)$ as outputs, where θ, θ_v and θ_o are network parameters. . . . 83

Figure Page

5.5 Structure of OMN and CoNN in Section 5.4.4. OMN and CoNN are Convolultional Neural Networks. OMN takes state s and history h as inputs, and gives the opponent strategy $\pi^o(s, h; \theta_o)$ as output. CoNN takes state s, history h, and opponent strategy π^o as inputs, and gives the policy $\pi(s, h, \pi^o; \theta)$ and the value estimation $V(s, h, \pi^o; \theta_v)$ as outputs. θ, θ_v and θ_o are network parameters. 85

5.6 Results on episode reward for networks trained in 1 million steps. The labeled bars in the chart show the averaged episode rewards over 10,000 episodes. The error bars are the standard deviations. 90

5.7 Results on reward per energy (RPE) for networks trained in 1 million steps. The labeled bars in the chart show the averaged RPEs over 10,000 episodes. The error bars are the standard deviations. 90

5.8 Results on episode reward for different maximum energy levels. The labeled bars in the chart show the averaged episode rewards over 10,000 episodes. The error bars show the standard deviations. 92

5.9 Results on reward per energy (RPE) for different maximum energy levels. The labeled bars in the chart show the averaged RPE over 10,000 episodes. The error bars show the standard deviations. 93

5.10 Reward areas of different sizes. The sizes are defined by the edge length in terms of the number of cells in the square. For a given reward area, one item of reward one is uniformly randomly placed in a cell within the square. 95

5.11 Results on episode reward for different sizes of reward area. The labeled bars in the chart show the averaged episode rewards over 10,000 episodes. The error bars show the standard deviations. In each episode, there is only 1 item of reward 1, and it is randomly placed in a square to the bottom-right of the grid. The size of reward area indicates the edge length of the square. 96

5.12 Results on reward per energy (RPE) for different sizes of reward area. The labeled bars in the chart show the averaged RPE over 10,000 episodes. The error bars show the standard deviations. In each episode, there is only 1 item of reward 1, and it is randomly placed in a square to the bottom-right of the grid. The size of reward area indicates the edge length of the square. 97

Figure Page

5.13 Gameplays of VNN and OMN + CoNN when the opponent uses the **patrol** strategy. The robot represents the collector, the ghost represents the opponent, and the money bag represents the item. The base and the energy level of the collector are not shown. The reward of the item is one. The numbers under the grids show the time steps, and the number on the top and left edges of the first grid are the coordinates of the grid cells. The three big dots between grids indicate omitted steps, while those at the end indicates infinite number of steps. 100

5.14 Gameplays of VNN and OMN + CoNN when the opponent uses the **restricted** strategy. The robot represents the collector, the ghost represents the opponent, and the money bag represents the item. The base and the energy level of the collector are not shown. The reward of the item is one. The numbers under the grids show the time steps, and the number on the top and left edges of the first grid are the coordinates of the grid cells. The three big dots between grids indicate omitted steps, while those at the end indicates infinite number of steps. 101

5.15 Gameplays of VNN and OMN + CoNN when the opponent uses the **with-fog** strategy. The robot represents the collector, the ghost represents the opponent, and the money bag represents the item. The base and the energy level of the collector are not shown. The reward of the item is one. The numbers under the grids show the time steps, and the number on the top and left edges of the first grid are the coordinates of the grid cells. The three big dots between grids indicate omitted steps. 102

5.16 Results on episode reward for networks trained in 5 million steps. The labeled bars in the chart show the averaged episode rewards over 10,000 episodes. The error bars are the standard deviations. 104

5.17 Results on reward per energy (RPE) for networks trained in 5 million steps. The labeled bars in the chart show the averaged RPEs over 10,000 episodes. The error bars are the standard deviations. 105

5.18 Results on episode reward for networks trained in 1 million steps. The labeled bars in the chart show the averaged episode rewards over 10,000 episodes. The error bars are the standard deviations. The **with-fog** strategy is replaced by the **double-blind with-fog** strategy where agents cannot see each other when the collector goes into special “foggy” cells. . . 107

5.19 Results on reward per energy (RPE) for networks trained in 1 million steps. The labeled bars in the chart show the averaged RPEs over 10,000 episodes. The error bars are the standard deviations. The **with-fog** strategy is replaced by the **double-blind with-fog** strategy where agents cannot see each other when the collector goes into special “foggy” cells. . . 108

1. INTRODUCTION

1.1 Motivation

Robots have been working among humans for many years, on simple and repetitive tasks such as cashier, phone operator and bank teller, or on dangerous tasks such as extraterrestrial explorations. With the advances in areas such as computer vision, machine perception, and control theory, robot's capabilities and potentials have been greatly increased. As a result, humans are gradually being replaced by robots in more complex tasks as driving, package delivery, and image/video surveillance.

Robots have many advantages over humans. They can vary in sizes to suit the task to be accomplished, they can perform repetitive tasks faster, they can survive in harsh environments, and they usually cost less than human employees. In addition, they are even more reliable when it comes to tasks that require precision, such as surgery, because they can eliminate human errors. In this thesis, we want to make use of these good traits in the domain of autonomous data collection.

Consider scientific exploration in an uninhabited environment like a rainforest or an island. Scientists need to collect data from different locations periodically. But the environment may be too dangerous for human expedition. It would be too costly and inefficient. Fortunately, the scientists have a lot of expandable UAVs that

are able to collect data and communicate with the satellite.[1] The autonomous UAVs are so affordable that even hiring humans to operate them would be more costly than using autonomous agents to control them instead. The scientists can set the locations of interests and let the UAVs collect data autonomously. However, there are threats in the environment that can destroy the UAVs, such as animals and bad weathers, but the autonomous UAVs can select routes efficiently thanks to their state-of-art navigation algorithms. When a UAV is navigated to a location of interest, it can collect data using its various sensors and immediately send the data back to the base station via satellite. Occasionally, UAVs may be destroyed and lost forever, but the algorithms anticipate that and can learn from it. Therefore the goal of the scientists is to collect as much data as possible with a given UAVs. Scenarios like this motivate the work in this thesis.

1.2 Problem framework

As a general framework for the problems studied in this thesis, we consider a data-collection task where autonomous robots are to collect *digital* data. We assume that the robots have communication capability, so they are able to send and receive information. When a robot collects a piece of data, it can immediately send the data to a receiving end, and receives a positive reward amounts to the value of the data. In addition, we assume there are internal (energy) and/or external (environment, animal, etc.) threats that can disable the robot, after which the robot can no longer move or collect data, and the value of the robot, if there is any, is lost.

[1] The UAVs are, hopefully, made of environment friendly materials.

We propose three different problem formulations that deal with different real-world scenarios. The problems have different formalism but they share the common goal, which is to maximize the total reward collected from the data with given robots.

1.3 Thesis overview

With the focus on studying the mission of data collection in risky environment, the rest of this thesis is organized as follows.

Chapter 2 provides some theoretical and technical backgrounds for the other chapters. Section 2.1 and 2.2 are applied in Chapter 3. Section 2.3, 2.4 and 2.5 are necessary for both Chapter 4 and Chapter 5. Section 2.6 is utilized by Chapter 5.

Chapter 3 formulates a planning problem on a weighted complete undirected graph. It assumes a known uniform distribution of risk, which is a probability of the robot being disabled, and a unspecified number of robots, and it does not consider energy constraints. This formulation is applicable in cases where finer-grained modeling of risk is impossible or too costly, and the robots have relatively high energy capacity. We propose heuristic algorithms to solve the problem. These heuristic algorithms are able to determine the number of the robots to be deployed and find routes with high expected reward for the robots.

Chapter 4 formulates a Markov Decision Problem (MDP) in a grid-based world. It allows for non-uniform risk, and considers the energy constraints. The risk is also a probability of the robot being disabled. There is only one robot in the MDP and

the robot is able to go back to the base for recharge. This formulation is application in a wider range of cases where the risk distribution is diverse and the energy capacity of the robot is limited. In such cases, a delicate balance of *safety and energy* (S&P) needs to be struck, therefore we propose a deep neural network named the *Ensemble Navigation Network* (ENN) to automatically find this balance.

Chapter 5 formulates a two-player Stochastic Game (SG) in a grid-based world. It also considers energy constraint. However, different from the other two formulations, the SG does not model probabilistic risk. Instead, the risk is in the form of an opponent. In real-world scenarios, this opponent can model a living entity such as an animal. The robot not only needs to avoid getting disabled by the opponent while collecting data, but also needs to worry about going back to the base for recharge. The focus of Chapter 5 is to use opponent modeling to improve data-collection performance. We propose four deep neural networks that model the opponent in different ways.

We move from graph-based formulation in Chapter 3 to grid formulations in Chapter 4 and 5 due to the following reasons: (1) Although graph is a more general data structure, grid, as a special type of graph, is usually able to model the real-world reasonably well, which is why it is often the data structure used in video games. (2) Grid can be implemented as array which has much better time and space complexities than graph. (3) Convolutional neural network (CNN), which gained its popularity from the field of computer vision, can be easily applied to grid world by treating each cell as a “pixel”. CNN is able to extract localized features and generalizes well in large state space, both of which are essential for our goal.

Since Chapter 3 belongs to a different field of research than Chapter 4 and Chapter 5, to be consistent with each research community's convention, we do *not* share symbols across chapters, i.e., the same symbol may have different meanings in different chapters. The meaning of symbols are clarified wherever it is necessary. In addition, we use the terms *robot* and *agent* interchangeably, and the terms *location of interest* and *item* interchangeably across the proposal. But these terms are used properly to the context of discussions.

Finally, Chapter 6 summarizes all the work presented in this thesis and discusses future directions.

2. BACKGROUND

In this chapter, we provide some of the theories and technologies used in the other chapters. Each section assumes the knowledge of all previous sections. Therefore, if a term is not defined in a certain section, it is defined in the previous ones.

2.1 Hierarchical Clustering

Hierarchical Clustering is a method used to build a hierarchy of clusters. There are two types of Hierarchical Clustering [30]:

- **Agglomerative Hierarchical Clustering** builds the hierarchy from bottom up. Starting from singletons that consists of a single element, it iteratively merges two clusters into one.
- **Divisive Hierarchical Clustering** builds the hierarchy from top down. Starting from one cluster containing all the elements, it iteratively splits one cluster into two.

In general, the merges and splits follow some greedy criteria and the resulting hierarchy of clusters is presented in a dendrogram. The dendrogram can be cut at different positions according to different optimization goals, which result in different number of clusters. See Figure 2.1 for an example. In the figure, the black dots represent elements, the solid lines indicate merges or splits, and the dotted lines

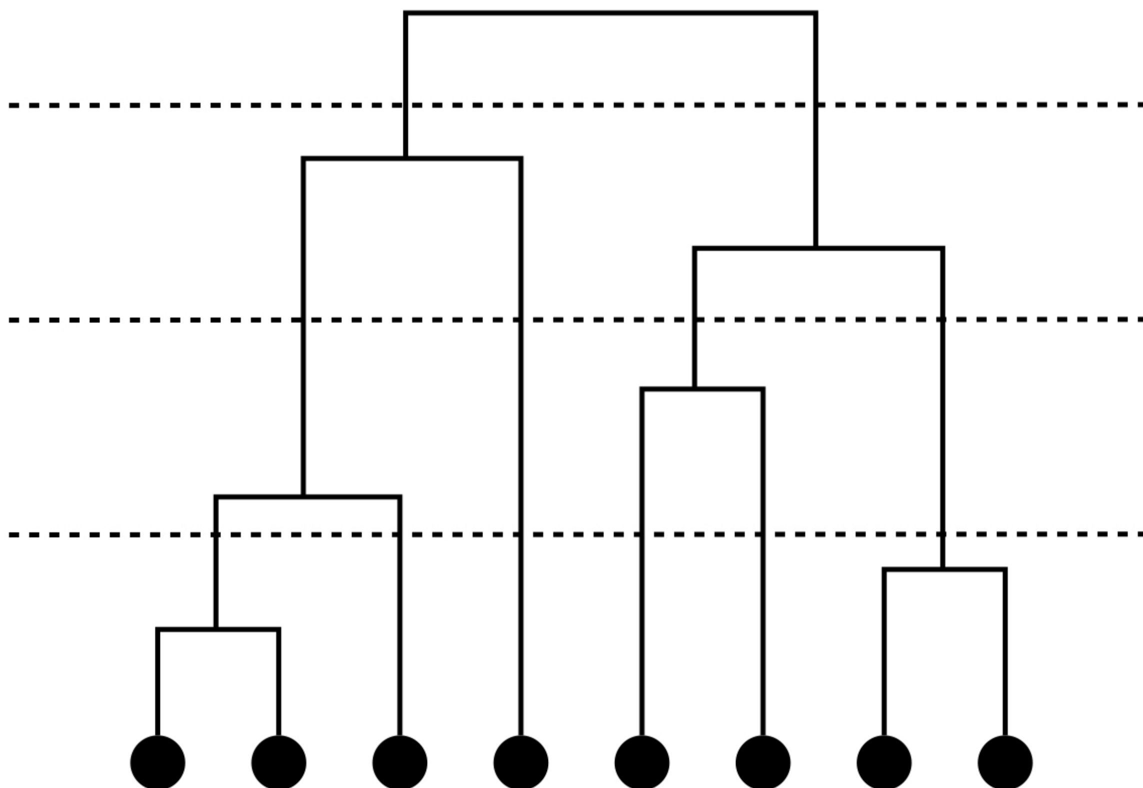


Fig. 2.1.: An example dendrogram from Hierarchical Clustering of 8 elements. The black dots represent elements. The solid lines indicate merges or splits. The dotted lines show different positions for cuts. In top-down order, the cuts result in 2, 4 and 6 clusters.

show different positions for cuts. In top-down order, the three cut positions result in 2, 4 and 6 clusters from the same 8 elements.

One of the key characteristics of Hierarchical Clustering is that, unlike more common clustering methods like k-means Clustering, it does not require a predefined number of clusters. This is useful when the number of clusters should be decided based on the clustering criteria, and this clustering method can automatically cut the dendrogram to produce the optimal number of clusters.

2.2 Rooted k Minimum Spanning Tree

Given a connected undirected graph with weighted edges, a Minimum Spanning Tree (MST) is a subset of the edges that connects all the vertices of the graph, without any cycles, and has the minimum total edge weights. The classical methods for finding the MST include Prim's algorithm [27] and Kruskal's algorithm [17], both of which have $O(m \log n)$ time complexity for a graph of m edges and n vertices.

A k-MST, however, asks for a tree of minimum total edge weights that connects exactly k vertices. When k is a fixed constant, the solution can be found in polynomial time by trying all the subsets of size k. When k is a variable, the problem is NP-hard [22, 29].

A *rooted tree* is a tree that consists of a selected vertex, called a *root*. A rooted k-MST is therefore a MST that has k vertices, one of which must be the root. Having a root makes Prim's algorithm easily applicable in finding the k-MST. Starting from the tree that contains only root, the new vertex with minimal-weight edge is added to the tree at each iteration. The process has $k - 1$ iterations, so it is $O(kn)$ for a graph of n vertices.

2.3 Convolutional Neural Network

Convolutional Neural Networks (CNN) is a bio-inspired feed-forward artificial neural network that has been the key to the recent breakthroughs in Machine Learning [21]. It has been successfully applied to fields including computer vision, speech recognition and natural language processing. Besides these traditional

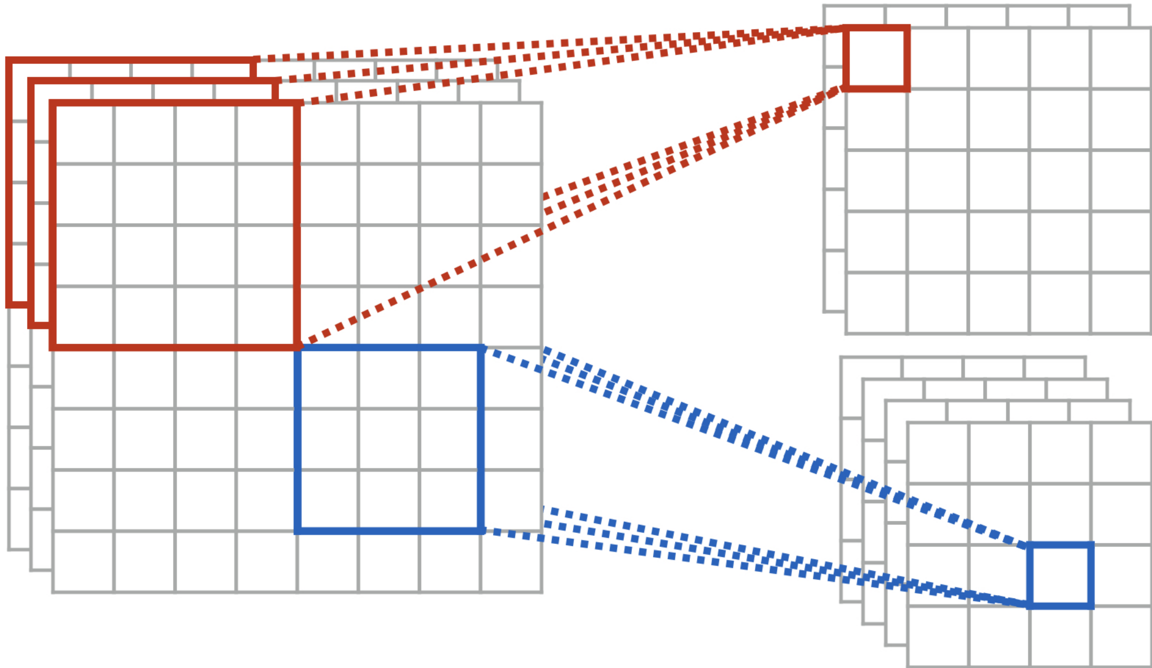


Fig. 2.2.: An example of a convolutional layer with input and outputs. The input is on the left, with a dimension of $8 \times 8 \times 3$. The outputs are on the right, with varying dimensions. There are 6 filters. The 2 red (top) filters have filter size $4 \times 4 \times 3$ and stride size 1. The 4 blue (bottom) filters have filter size $3 \times 3 \times 3$ and stride size 2. The input is padded with zeros for the blue filters.

supervised learning domains, it is also making aspiring progress in reinforcement learning domains [23, 24, 34]. The key characteristic of CNN is its reduced number of network parameters and its ability to extract different localized features from inputs.

CNN contains one or more convolutional layers, each convolutional layer contains a number of convolutional matrices of parameters, or *filter*s, of different sizes. The size of a filter is usually much smaller than that of the input. Each filter acts as a sliding window over the input matrix. At each step of the sliding, also referred to as a *stride*, the filter transforms the input submatrix within the window to a single value. This transformation, or *convolution*, is done at every stride during

the forward pass, and the input matrix is converted to an output matrix of equal or smaller size. Since the same filter is applied at every stride, the number of parameters is reduced and the network can be deep, which is proven to be crucial for its effectiveness [38].

Figure 2.2 shows an example convolutional layer with input and outputs. The input is on the left, with a dimension of $8 \times 8 \times 3$. The outputs are on the right, with varying dimensions. There are 6 filters. The 2 red (top) filters have filter size $4 \times 4 \times 3$ and stride size 1. The 4 blue (bottom) filters have filter size $3 \times 3 \times 3$ and stride size 2. The input is padded with zeros for the blue filters.

During gradient-descent training, filters can specialize towards different directions [16]. For example, in the field of computer vision, one filter may specialize in detecting horizontal edges, while another may specialize in detecting vertical edges; one filter may specialize in colors, while another may specialize in contrasts. In other words, different filters are able to extract different localized features from their input.

A convolutional layers is usually followed by a max-pooling layer, which works in a similar way but, instead of convolving with the input, a filter simply picks the highest value within the window. The output of the max-pooling layer is therefore smaller in size than the input, which further reduces the number of network parameters. At the end, before the output layer, a fully connected layer, as seen in regular neural networks, is normally used for high-level reasoning. This layer summarizes the localized features extracted by the convolutional and max-pooling layers. The summaries are then used for the final output.

2.4 Markov Decision Process and Stochastic Game

Markov Decision Process (MDP) is an extension of Markov Chain. It satisfies the Markov property and provides a framework for modeling decision making in scenarios where the outcome is partially random and partially in the control of the decision maker. It is commonly used to formulate problems that can be solved by dynamic programming or reinforcement learning.

In a standard MDP, an agent interacts with an environment over a number of discrete time steps. At each time step t, the agent receives a state s_t and selects an action a_t from a set of possible actions $\mathcal{A}$ according to a policy π, which is a mapping from states s_t to actions a_t. After the action is chosen, the environment transits from the state s_t to the next state s_{t+1} according to some transition probability function $\mathcal{T}(s_t, a, s_{t+1}) = Pr(s_{t+1}|s_t, a_t)$. In return, the agent receives a scalar reward r_t. This process repeats until the agent reaches a terminal state, after which the process restarts. The discounted accumulated return, or *return* for short, is defined as $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$, where the discounting factor $\gamma \in [0, 1]$ signifies the importance of immediate rewards. The goal of the agent is to maximize the return for every state s_t, in other words, to find the optimal policy π^*.

The generalization of MDP to multiagent case is the *Stochastic Game* (SG), in which a number of agent interacts with an environment over a number of discrete time steps. At each time step t, each agent z selects an action $a_t^{(z)}$ from its own set of possible actions $\mathcal{A}^{(z)}$ according to state s_t and its policy $\pi^{(z)}$. All the actions from all the agents form a joint action $\mathbf{a}$. When the joint action is executed, the

environment transit from s_t to s_{t+1} according to transition function $\mathcal{T}(s_t, \mathbf{a}, s_{t+1}) = Pr(s_{t+1}|s_t, \mathbf{a})$. In return, each agent z receives a scalar reward $r_t^{(z)}$. The return for agent z is defined as $R_t^{(z)} = \sum_{k=0}^{\infty} \gamma^k r_{t+k}^{(z)}$ and the goal of agent z is to maximize its own return for every state s_t by finding its optimal policy $\pi^{(z)*}$.

If there is no communication between the agents, SG is the same as MDP in the perspective of a particular agent z. In addition, if the other agents use fixed policy, or *strategy*, they can be considered as part of the environment and the SG can be reduced to MDP for agent z. But if the other agents use mixed strategy, i.e., their mappings from states to actions may change, the transition function $\mathcal{T}$ become unstable for agent z, and it is more difficult for z to find $\pi^{(z)*}$.

2.5 Reinforcement Learning

Reinforcement Learning (RL) is a common method for MDPs and SGs. In RL, the state-action value function $Q^\pi(s, a) = \mathbb{E}[R_t|s_t = s, a]$ is the expected return for selecting action a in state s and following policy π afterwards. The optimal value function $Q^*(s, a) = \max_\pi Q^\pi(s, a)$ gives the maximum value achievable for state s and action a by any policy. Similarly, the state-only value function $V^\pi(s) = \mathbb{E}[R_t|s_t = s]$ is the expected return of state s for following policy π. The advantage function $A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s)$ indicates the *advantage* of action a in state s.

2.5.1 Q-learning

Q-learning is a value-based RL method [35]. It works by learning the optimal value function $Q^*(s,a) = \max_\pi Q^\pi(s,a)$ using the update rule:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(r_t + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1}) - Q(s_t,a_t))$$

where α is the learning rate that decides the strength of each update, and γ is the discounting factor. The values of $Q(s,a)$ is traditionally stored in a table, but can also be in the format of an approximating function if the number of state-action pairs is intractable. When an approximation function is used, the function is denoted as $Q(s,a;\theta)$ with θ representing the parameters of the function. Q-learning is model-free and off-policy, which means it does not learn a model for the environment and it does not maintain a policy. The policy is implicitly obtained by always choosing action $a = \arg\max_a Q(s,a)$ ior every state s. In practice, an ϵ-greedy strategy, which follows the greedy strategy with probability $1-\epsilon$ and selects random action with probability ϵ, is often used to ensure adequate exploration of the state space.

2.5.2 Policy gradient

Policy gradient is another common method of RL. It is a policy-based method that optimize a policy approximation function $\pi(a|s;\theta)$ directly. In particular, REINFORCE [41] is a policy gradient method that updates function parameter θ in

the direction of $\nabla_\theta \log \pi(a_t|s_t;\theta)R_t$. A baseline $b_t(s_t)$ is usually subtracted from the return R_t to reduce the variance of gradient estimate while keeping the unbiasedness, giving a gradient direction $\nabla_\theta \log \pi(a_t|s_t;\theta)(R_t - b_t(s_t))$. In practice, an estimate of the state-only value function $V(s_t;\theta_v)$ is often used as the baseline, where θ_v represents function parameters. This gives us an approximation of the advantage function $A(s_t, a_t; \theta_v) = R_t - V(s_t;\theta_v)$ where R_t is an estimate of $Q(s_t, a_t)$. The gradient direction then becomes:

$$\nabla_\theta \log \pi(a_t|s_t;\theta)A(s_t, a_t; \theta_v)$$

Policy gradient is model-free and on-policy. In an actor-critic policy gradient method, the critic updates the value function parameters, and the actor updates policy parameters according to the gradient suggested by the critic. Compared to a value-based method like Q-learning, the advantages of policy gradient include (1) better convergence properties, (2) effective in high-dimensional or continuous action spaces and (3) can learn stochastic policies. But it typically converge to local rather than global optimum, and evaluating a policy is typically inefficient and has high variance.

2.5.3 Deep Reinforcement Learning

In Deep Reinforcement Learning (DRL), deep neural networks are used as the approximation functions. As a result, DRL brings the benefits of Deep Learning to RL. In particular, the key advantages of DRL methods over traditional RL methods

include (1) its ability to learn from untraceable state space efficiently and (2) its ability to automatically extract features from inputs.

Deep neural network can approximate the state-action value function $Q(s, a; \theta)$. In their seminal work [23, 24], Mnih et al. proposed the Deep Q Learning algorithm, which uses a CNN named the Deep Q Network (DQN) as $Q(s, a; \theta)$ and achieved professional-human performance on 49 Atari 2600 games. The Q Network learns by taking only raw pixels from the console images and game scores as input. It outputs $Q(s, a; \theta)$ for all the actions $a \in \mathcal{A}$ for a given state s. The same network structure works well for all the 49 games, showing a hint of general intelligence. During training, DQN uses a replay buffer to store past experience, which is sampled from for parameter updates. This helps smooth out learning and avoid oscillations or divergence of the parameters.

Many works have been done to improve the performance of DQN. Van Hasselt et al. [39] use Double DQN (D-DQN) to deal with the overestimate problem in Q-learning. Schaul et al. [33] proposed prioritized experience replay, which uses more important experience more often to improve learning efficiency. Wang et al. [40] employ a dueling network architecture to estimate state value function $V(s)$ and the associated advantage function $V(s, a)$, which are combined to estimate the state-action value function $Q(s, a)$. This architecture is proven to speed up convergence.

Deep neural network can also approximate the policy function $\pi(a|s; \theta)$ directly. In their following work [25], Mnih et al. proposed Asynchronous Advantage Actor-Critic (A3C) algorithm, which uses a CNN to learn both the policy $\pi(a|s; \theta)$

and the $V(s;\theta_v)$ function. This is an actor-critic method where the policy is the actor and an estimate of $A^{\pi}(s,a)$ using $V(s;\theta_v)$ is the critic. The CNN used in A3C learns from the same type of input and shares similar network structure as DQN. A3C uses asynchronous threads to explore state space in parallel to stabilizes training. It runs faster and performs better or equally good with DQN and all the extensions of DQN mentioned above. In addition, A3C is easy to implemented compared with the extensions of DQN, it does not need a replay buffer, and it is able to learn stochastic policies. Therefore, it is chosen as the learning method of our work.

2.6 Opponent modeling

Opponent modeling, or agent modeling in general, is a way to improve a controlled agent's performance by identifying and exploiting other agents' behaviors. In a multiagent environment such as a SG, the environment state s_t is affected by the joint action $\mathbf{a}$ of all the agents. Let a denote the action of agent z and o denote the actions of all the other agents, i.e., $\mathbf{a} = (a, o)$. From the perspective of z, the transition function $\mathcal{T}(s, a, o, s')$ and reward function $\mathcal{R}(s, a, o, s')$ are unstable, because z cannot predict o. As a result, RL methods normally takes a long time to converge, and the results may be suboptimal.

If the other agents use a fixed joint strategy, their behaviors can be considered as part of the environment. Mathematically, let π^o be the joint strategy of all the other agents, the transition function and reward function can be redefined as:

$$\begin{aligned} \mathcal{T}^o(s,a,s') &= \sum_o \pi^o(o|s,a)\mathcal{T}(s,a,o,s') \\ \mathcal{R}^o(s,a,s') &= \sum_o \pi^o(o|s,a)\mathcal{R}(s,a,o,s') \end{aligned} \tag{2.1}$$

Since π^o is fixed, $\mathcal{T}^o(s,a,s')$ and $\mathcal{R}^o(s,a,s')$ are determined. The multiagent problem is thus reduced to a single-agent one and agent modeling is not necessary [13].

But if the other agents use mixed joint strategy, Equation 2.1 becomes:

$$\begin{aligned} \mathcal{T}_t^o(s,a,s') &= \sum_o \pi_t^o(o|s,a)\mathcal{T}(s,a,o,s') \\ \mathcal{R}_t^o(s,a,s') &= \sum_o \pi_t^o(o|s,a)\mathcal{R}(s,a,o,s') \end{aligned} \tag{2.2}$$

the policy π_t^o can change over time and so can $\mathcal{T}_t^o(s,a,s')$ and $\mathcal{R}_t^o(s,a,s')$. As a result, the problem cannot be reduced to single-agent. But with an accurate agent model, $\pi_t^o(o|s,a)$ can be predicted and exploited. Therefore, the goal of agent modeling is to predict information, such as strategies or actions, of other agents in order to smooth learning and improve performance.

Agent modeling is most helpful in games with imperfect information, such as Poker, because the agent model provides a significant amount of extra information about the game state and can greatly stabilizes the learning process of the

controlled agent. In this work, however, the controlled agent has perfect information about the state, and the objective is to study the effectiveness of different approaches of agent modeling.

3. GRAPH FORMULATION WITH UNIFORM RISK

3.1 Chapter overview

In this chapter, we formulate the data-collection task as a planning problem on a complete undirected graph. This type of multiagent planning problems are generally considered as variants of the Vehicle Routing Problem (VRP). Although many variants of VRP are well studied [37], they usually consider agents to be humans or human operated, implicitly assuming that the assigned tasks are completed with certainty. In contrast, unexpected events may destroy robots and terminate assigned tasks. This practical aspect is generally not considered in the VRP research community.

Unlike existing formulations of the VRPs, in this chapter, the probabilities of robots breaking down and the value of the robots are explicitly modeled. The value of a robot can be the hardware cost of the robot or the strategic importance of the robot quantified by a real number. Therefore, the objective of the proposed algorithms is to generate a routing plan that maximizes the expected reward with the optimal number of robots. To our best knowledge, these two aspects, i.e., *risks* and *value of lost robots*, of our formulation are unique to any other existing VRP formulations.

3.2 Contributions

- Introducing the Data-collecting Robot Problem (DRP), which explicitly models the value of robots and the risk of losing robots,
- Introducing heuristics for clustering and tour-building steps for solving DRP,
- Showing that the Greedy Insertion (GI) and Total-Loss (TL) algorithms have the top performance among tour-building algorithms, and
- Showing that the Progressive Gain-aware Clustering (PGC) algorithm produces quality results with a better time complexity.

3.3 Problem formulation

The world is modeled as a complete undirected graph $G = (V, \mathcal{D}, \alpha, \beta, \psi)$. $V = \{0, 1, \ldots\}$ is the set of nodes, where 0 represents the base station and the others represent the locations of interest. $\mathcal{D} : V \times V \rightarrow \mathbb{R}^+$ is a symmetric distance function, i.e., $\mathcal{D}(u, v) = \mathcal{D}(v, u)$ is the distance between nodes u and v. $\mathcal{D}$ satisfies the triangle inequality. The value of a robot is α. Data collected from location $v \in V$ has a value of $\beta(v)$. The probability of a robot successfully traversing one unit distance is ψ, so if a robot traverses from node u to v for $u, v \in V$, the probability of success is $\psi^{\mathcal{D}(u,v)}$.

We assume that data at a location is collected only once, which means there is no extra gain by visiting the same node redundantly. Since G is a complete graph

that satisfies the triangle inequality, visiting an extra node before a target node always decreases the expected gain.

A tour $t = (v, \ldots, 0)$ for $v \in V \backslash \{0\}$ is a vector of distinct nodes. A robot starts a tour at node 0, then sequentially visit all the nodes in the vector, which leads it back to 0 eventually. The objective is to find a plan T consisting a set of tours, that maximizes the sum of the expected rewards of all the tours. Suppose $\mathcal{D}_t(u, v)$ is the distance between u and v along tour t, $P_t(u, v) = \psi^{\mathcal{D}_t(u,v)}$ is the probability of the robot successfully traveling from u to v along tour t. We use $P_t(v)$ to denote $P_t(0, v)$. Let $|T|$ be the cardinality of T, and let $t\backslash\{0\}$ be the *subtour* excluding the final returning edge. We assume that there are unlimited number of robots at disposal, and the nodes can be left unvisited. Therefore, in addition to generating the tours, the planning involves deciding the number of robots to deploy and which nodes to visit.

In addition, we assume that data collection and transmission are instantaneous. Therefore, the optimal strategy is always to upload the data right after collecting it. If the robot is at u, the expected marginal gain from v is $\psi^{\mathcal{D}(u,v)}\beta(v)$, regardless of which nodes are visited after v. Without loss of generality, let $\beta(0) = \alpha$ so that

$P_t(0)\alpha = P_t(0)\beta(0)$. Then the objective function can be expressed mathematically as:

$$\begin{aligned} &\max \sum_{t\in T}\left(\sum_{v\in t\setminus\{0\}} P_t(v)\beta(v) - (1-P_t(0))\alpha\right) \\ =&\max \sum_{t\in T}\left(\sum_{v\in t} P_t(v)\beta(v) - \alpha\right) \\ =&\max \left(\sum_{t\in T}\sum_{v\in t} P_t(v)\beta(v) - |T|\alpha\right) \end{aligned} \tag{3.1}$$

In this formulation, the inner summation term in Equation 3.1 is referred to as the (expected) *gain* (of rewards) from the visiting nodes in a tour t; the expected cost due to the risk of losing a robot on t is referred as the *cost* of the tour; and the difference between the gain from all the nodes in a tour and the cost of the tour is referred as the *reward* of the tour.

3.4 Algorithm

Our solution consists of two steps: (1) clustering for the number of robots and (2) tour building for a single robot. We introduce the Progressive Gain-aware Clustering (PGC) and compare it with a naive clustering approach. Six tour building heuristics are proposed and compared.

3.4.1 Clustering for the number of robots

For a thorough discussion of the effects of the number of robots, refer to [14]. In that work, one important observation is that robots are reluctant to visit a node

Algorithm 1 Clustering Algorithm *Cluster*

Input: G, $\mathcal{R}$ // G: the world model, $\mathcal{R}$: a cluster evaluation function
Output: S // a set of clusters

```
1: S ← ∅
2: for all v ∈ V \ {0} do
3:     S ← S ∪ {v}
4: end for
5: loop
6:     C_i^* ← ∅, C_j^* ← ∅
7:     Δ^* ← 0 // the highest difference in reward
8:     for all C_i ∈ S do
9:         for all C_j ∈ S, j > i do
10:            Δ ← R(C_i ∪ C_j) − R(C_i) − R(C_j)
11:            if Δ > Δ^* then
12:                Δ^* ← Δ, C_i^* ← C_i, C_j^* ← C_j
13:            end if
14:        end for
15:    end for
16:    if Δ^* = 0 then
17:        break
18:    end if
19:    S ← S \ {C_i^*, C_j^*} ∪ {C_i^* ∪ C_j^*}
20: end loop
21: return S
```

that is too faraway, because there is a higher chance of breaking down as they travel to the node and therefore the expected reward would be negligible or even negative. However, if there is a cluster of nodes that are equally faraway, a robot may visit all of them because the high cost due to the initial long edge to the cluster is effectively distributed among all the nodes within the cluster. Therefore, clustering methods should be used to find these clusters. After clustering, the problem is reduced to a single-robot problem. Each cluster is assigned to one robot and a tour covering *all* the nodes in each cluster is generated for one robot.

We use the Bottom-up Hierarchical Clustering, also known as Agglomerative Clustering, employed in [14] to determine the number of robots needed. Starting from single-node clusters, each iteration finds the best merge that gives the highest increase in total reward. Formally, if $\mathcal{R}^*(C_i)$ is the maximum reward a robot can get from cluster C_i, and $C_i \cup C_j$ is the merged cluster, then at each step we merge C_i and C_j that give the maximum positive value $\mathcal{R}^*(C_i \cup C_j) - \mathcal{R}^*(C_i) - \mathcal{R}^*(C_j)$, until there's no more positive merge values. The base node, 0, is excluded from this process. See Algorithm 1 for more details. The algorithm takes as input an evaluation function $\mathcal{R}$, which can be $\mathcal{R}^*$ or a function that estimates $\mathcal{R}^*$. Suppose the time complexity of $\mathcal{R}$ is $O(m)$, where m is a polynomial expression as we show later in this section, and there are n nodes in the graph. This procedure has $O(mn^3)$ operations. However, since the evaluation of one pair of clusters is completely independent of another, parallelism can be easily achieved, which, in the best case, is $O(m + n^3)$.

The only way to get $\mathcal{R}^*(C_i)$ is to find the optimal tour visiting all the nodes in cluster C_i for a single robot, which is a variant of the NP-hard MLP [4]. Therefore, instead of trying to find the optimal tour, we propose six tour-building heuristics, and use the tours built by the heuristics as estimations of the optimal tour. In addition, we also propose an efficient clustering heuristic that uses k Minimum Spanning Tree (k-MST) to estimate the expected reward of a cluster without explicitly building a tour.

Algorithm 2 Tour-Building Algorithm $BuildTour_{\mathcal{K}}$

Input: G, C // G: the world model, C: a cluster of nodes
Output: t // a tour contains all the nodes in C
1: $t \leftarrow \emptyset$
2: **while** $C \neq \emptyset$ **do**
3: $u, i \leftarrow \mathcal{K}(C, t)$ // subroutine $\mathcal{K}$ returns a node and an insertion position
4: $t \leftarrow t \oplus_i u$ // insert node u after position i of t
5: $C \leftarrow C \setminus \{u\}$
6: **end while**
7: **return** t

Algorithm	**Criterion**	**Insertion position**	**Time complexity**
NG	Marginal reward of next step	end	$O(n^2)$
OSA	Marginal reward of two steps	end	$O(n^3)$
TL	Total loss from all nodes	end	$O(n^3)$
GML	Marginal reward & min loss	end	$O(n^3)$
LPG	Marginal reward & total loss	end	$O(n^3)$
GI	Marginal reward of insertion	anywhere	$O(n^3)$

Table 3.1: Overview of tour-building algorithms. **Criterion** summarizes the what is evaluated by each algorithm. **Insertion position** specifies where does an algorithm insert a new node. **Time complexity** shows the computational complexity of an algorithm.

3.4.2 Heuristics for building tours

A *partial tour* is a vector of nodes $(0, \ldots, v)$ for $v \in V$ that defines an acyclic ($v \neq 0$) or a cyclic ($v = 0$) path starting from node 0. Given a partial tour t, $t \oplus_i u$ is a new partial tour extended by inserting node u *after* the i-th node of t, $\rho(t)$ is the reward of t, defined as the gain from all the nodes in t, and $|t|$ is the total number of nodes, including the starting 0, in t. The *marginal reward* of an extended partial tour is calculated as the difference between its reward and the original's.

The tour-building heuristics are *incremental* in the sense that they build a tour by assigning one node at a time. All these six heuristics share the same algorithmic

structure described in Algorithm 2. The difference is the subroutine for choosing a node for insertion and the insertion position given a partial tour and a set of unassigned nodes. The subroutine is denoted as $\mathcal{K}$ (subscript of $BuildTour_{\mathcal{K}}$) in Algorithm 2. Table 3.1 shows an overview of the algorithms.

The Naive Greedy (NG) algorithm ($BuildTour_{\mathcal{K}_{\mathrm{NG}}}$). This simple heuristic picks the next node solely based on the marginal reward. Formally, to assign the next node, the algorithm calculates the marginal reward of visiting u next

$$\pi_t^{\mathrm{NG}}(u) = \rho(t \oplus_{|t|} u) - \rho(t) \tag{3.2}$$

for all unassigned node u in the cluster, and picks the maximum u with $\pi_t^{\mathrm{NG}}(u)$. The insertion position is always $|t|$, which means it always appends a node at the end of a partial tour. This is an $O(n^2)$ operation for a cluster of n nodes.

The One-Step-Ahead (OSA) algorithm. This heuristic considers one more step than NG. Namely, it calculates:

$$\pi_t^{\mathrm{OSA}}(u) = \max_{v \neq u} \left(\rho(t \oplus_{|t|} u \oplus_{|t|+1} v) - \rho(t) \right) \tag{3.3}$$

for all unassigned u and v, and picks the u with highest $\pi_t^{\mathrm{OSA}}(u)$. The insertion position is always $|t|$. Notice that node v is used only for the evaluation of node u, there is no guarantee in that the next step actually picks node v. This is an $O(n^3)$ operation for a cluster of n nodes.

The Total-Loss (TL) algorithm. This heuristic calculates the sum of all the "losses" that visiting a node u incurs, and picks the u that minimizes this total loss. The loss from a node v incurred by visiting $u \neq v$ is defined as:

$$\delta_t(u,v) = \Big(\rho(t \oplus_{|t|} v) - \rho(t)\Big) - \Big(\rho(t \oplus_{|t|} u \oplus_{|t|+1} v) - \rho(t \oplus_{|t|} u)\Big) \tag{3.4}$$

where $\rho(t \oplus_{|t|} v) - \rho(t)$ is the marginal reward of appending v to t, while $\rho(t \oplus_{|t|} u \oplus_{|t|+1} v) - \rho(t \oplus_{|t|} u)$ is the marginal reward of appending v to $t \oplus_{|t|} u$. The difference of these two signifies the *minimum reduction* in the marginal reward of appending v to a partial tour caused by appending u first. For each assignment, TL calculates:

$$\pi_t^{\mathrm{TL}}(u) = \sum_{v \neq u} \delta_t(u,v) \tag{3.5}$$

for all the unassigned u and v, and picks the u with the minimum $\pi_t^{\mathrm{TL}}(u)$. The insertion position is always $|t|$. This takes $O(n^3)$ operations for a cluster of size n.

The Gain-Minus-Loss (GML) algorithm. This heuristic calculates the difference of gain (marginal reward) and loss; then it uses only the minimum loss instead of the total. For each assignment, the algorithm calculates:

$$\pi_t^{\mathrm{GML}}(u) = \pi_t^{\mathrm{NG}}(u) - \min_{v \neq u} \delta_t(u,v) \tag{3.6}$$

for all unassigned u and v, and picks the u with the maximum $\pi_t^{\mathrm{GML}}(u)$. The insertion position is always $|t|$. This is an $O(n^3)$ operations for a cluster size n.

The Loss-Per-Gain (LPG) algorithm. This heuristic considers both gain and *total* loss by taking the ratio of the total loss to the gain. For each assignment, the algorithm calculates:

$$\pi_t^{\mathrm{LPG}}(u) = \frac{\pi_t^{\mathrm{TL}}(u)}{\pi_t^{\mathrm{NG}}(u)} \tag{3.7}$$

for all the unassigned u, and picks the u with the minimum $\pi_t^{\mathrm{LPG}}(u)$. The insertion position is always $|t|$. This is an $O(n^3)$ operation for a cluster of n nodes.

The Greedy Insertion (GI) algorithm. This algorithm is a single-robot variation of the Sequential Greedy Algorithm (SGA) proposed in [8][1]. At each step, GI assigns the next node by trying out all the possible insertions of all the unassigned nodes, and picks the one with the highest marginal reward. However, for each partial tour, this algorithm calculates the reward of the corresponding cyclic partial tour, which appends node 0 at the end and sets $\beta(0) = \alpha$ (see Section 3.3). Formally, given the partial tour t, inserting node u *after* the i-th node of t gives a marginal reward:

$$\pi_t^{\mathrm{GI}}(u, i) = \rho(t \oplus_i u \oplus_{|t|} 0) - \rho(t \oplus_{|t|} 0) \tag{3.8}$$

At each assignment, the algorithm calculates $\pi_t^{\mathrm{GI}}(u, i)$ for all the unassigned u and all the integral $i \in [1, |t|]$, and chooses the u and i with the maximum $\pi_t^{\mathrm{GI}}(u, i)$. This is an $O(n^3)$ operation for a cluster of n nodes.

Other than GI, all these algorithms build a partial tour by inserting nodes at the end, and therefore can be adopted for online planning. Since GI needs to insert

[1]The reward function in our problem does not satisfy the Diminishing Marginal Gain property, so the performance guarantee of SGA doesn't hold.

node at any position of a partial tour and the robot cannot change the path already taken, it can only be used offline.

With a small modification (see Section 3.5), the above tour-building algorithms can be used as the input function $\mathcal{R}$ in Algorithm 1 to evaluate cluster merging. All of the algorithms except NG have $O(n^3)$ time. Fortunately, all the $O(n^3)$ algorithms can reduce the time complexity by at most a factor of n using parallelism, because the evaluation of one candidate node is independent of another.

3.4.3 The Progressive Gain-aware Clustering algorithm

The Progressive Gain-aware Clustering algorithm (PGC) has a better time complexity than the naive clustering. Based on the technique of using rooted k-MST to approximate the optimal solution of MLP [5, 7], PGC estimates the reward obtainable from a cluster without building a tour. Algorithm 3 shows the merging procedure. The complete algorithm is in Algorithm 4, which follows a similar structure as the clustering algorithm.

The data structures associated with an existing cluster C_i are: (1) an entry node e_i which is the closest node to the base node in the cluster; (2) an estimated gain g_i from all the nodes in the cluster; (3) an estimated cost c_i due to the risk of losing the robot; (4) an adjacency list $\mathcal{L}_i$ that keeps track of the Minimum Spanning Tree (MST) of the nodes in the cluster, where $\mathcal{L}_i(v)$ is the list of neighbors of node v in the MST; and (5) a total length l_i that is the sum of all the edge lengths in the

Algorithm 3 PGC Merging Algorithm *PGCMerge*

Input: G, $(C_i, e_i, l_i, g_i, c_i, \mathcal{L}_i)$, $(C_j, e_j, l_j, g_j, c_j, \mathcal{L}_j)$ // G: the world model, $(C_i, e_i, l_i, g_i, c_i, \mathcal{L}_i)$: data structure of C_i, see Section 3.4.3
Output: $(C_{ij}, e_{ij}, l_{ij}, g_{ij}, c_{ij}, \mathcal{L}_{ij})$ // data structure of C_{ij}

1: $C_{ij} \leftarrow C_i \cup C_j$
2: **if** $\mathcal{D}(0, e_i) > \mathcal{D}(0, e_j)$ **then**
3:    **swap**(i, j)
4: **end if**
5: $e_{ij} \leftarrow e_i,\ d \leftarrow +\infty$
6: **for all** $v_i \in C_i$ **do**
7:    **for all** $v_j \in C_j$ **do**
8:       **if** $\mathcal{D}(v_i, v_j) < d$ **then**
9:          $d \leftarrow \mathcal{D}(v_i, v_j),\ v_i^* \leftarrow v_i,\ v_j^* \leftarrow v_j$
10:       **end if**
11:    **end for**
12: **end for**
13: $\mathcal{L}_{ij} \leftarrow \mathcal{L}_i \cup \mathcal{L}_j \cup \{\{v_i^*, v_j^*\}\}$
14: $H \leftarrow \{(\mathcal{D}(v_i^*, v_j^*), v_j^*)\}$ // min-heap as BFS queue, sorted on the edge length
15: $M \leftarrow C_i$ // the visited nodes
16: $l_{ij} \leftarrow l_i,\ g_{ij} \leftarrow g_i$
17: **while** $H \neq \emptyset$ **do**
18:    $d, v \leftarrow H.$**pop** // the top of H
19:    $l_{ij} \leftarrow l_{ij} + d,\ g_{ij} \leftarrow g_{ij} + \psi^{\mathcal{D}(0,e_{ij})+l_{ij}}\beta(v),\ M \leftarrow M \cup \{v\}$
20:    **for all** $u \in \mathcal{L}_j(v), u \notin M$ **do**
21:       $H \leftarrow H \cup \{(\mathcal{D}(u, v), u)\}$
22:    **end for**
23: **end while**
24: $c_{ij} \leftarrow (1 - \psi^{\mathcal{D}(0,e_i)+\mathcal{D}(0,e_j)+l_{ij}})\alpha$
25: **return** $(C_{ij}, e_{ij}, l_{ij}, g_{ij}, c_{ij}, \mathcal{L}_{ij})$

MST. For the rest of this section, subscript ij is used to denote a variable associated with cluster $C_i \cup C_j$. For example, C_{ij} is $C_i \cup C_j$ and e_{ij} is the entry node of C_{ij}.

Given C_i and C_j, the estimated reward before merging is computed trivially as $g_i - c_i + g_j - c_j$. But the computation of the estimated reward after merging is more complicate. The first step is to choose the entry node $e_{ij} \in \{e_i, e_j\}$ of C_{ij} to be the one that is closer to the base node. Without loss of generality, assume $e_{ij} = e_i$. Then the algorithm obtains $\mathcal{L}_{ij}$ by merging $\mathcal{L}_i$ and $\mathcal{L}_j$ and adding the shortest edge

Algorithm 4 PGC Algorithm *PGC*

Input: G // the world model
Output: S // a set of clusters

1: $S \leftarrow \emptyset$
2: **for all** $v \in V \setminus \{0\}$ **do**
3: $C_v \leftarrow \{v\},\ e_v \leftarrow v,\ l_v \leftarrow 0,\ g_v \leftarrow \psi^{\mathcal{D}(0,v)}\beta(v),\ c_v \leftarrow (1-\psi^{2\mathcal{D}(0,v)})\alpha,\ \mathcal{L}_v \leftarrow \emptyset$
4: $S \leftarrow S \cup C_v$
5: **end for**
6: **loop**
7: $C_i^* \leftarrow \emptyset,\ C_j^* \leftarrow \emptyset,\ C_k^* \leftarrow \emptyset,\ \Delta^* \leftarrow 0$
8: **for all** $C_i \in S$ **do**
9: **for all** $C_j \in S, j > i$ **do**
10: $(C_k, \ldots, g_k, c_k, \ldots) \leftarrow \mathit{PGCMerge}((C_i, \ldots, g_i, c_i, \ldots), (C_j, \ldots, g_j, c_j, \ldots))$
11: $\Delta \leftarrow (g_k - c_k) - (g_i - c_i) - (g_j - c_j)$
12: **if** $\Delta > \Delta^*$ **then**
13: $\Delta^* \leftarrow \Delta,\ C_i^* \leftarrow C_i,\ C_j^* \leftarrow C_j,\ C_k^* \leftarrow C_k$
14: **end if**
15: **end for**
16: **end for**
17: **if** $\Delta^* = 0$ **then**
18: **break**
19: **end if**
20: $S \leftarrow S \setminus \{C_i^*, C_j^*\} \cup \{C_k^*\}$
21: **end loop**
22: **return** S

that connects nodes $v_i \in C_i$ and $v_j \in C_j$. To compute g_{ij}, we perform a breadth-first search (BFS) starting from node v_i on the MST of C_{ij}, with all the nodes in C_i marked as visited, l_{ij} set to l_i and g_{ij} set to g_i. At each iteration, the shortest edge that connects the visited subtree to an unvisited node is picked. Its length is added to l_{ij}, and the unvisited node on the edge, say v, is marked as visited. Assume at this point there are n visited nodes, then l_{ij} is the total edge length of the n-MST of C_{ij} rooted at node e_{ij}, which can be used as a lower-bound of the n-th node's latency in the optimal minimum-latency tour of C_{ij} starting from node e_{ij}. After adding this edge, g_{ij} is increased by $\psi^{\mathcal{D}(0,e_{ij})+l_{ij}}\beta(v)$, where ψ is the probability of

Algorithm 5 *BuildTour*$_{\mathcal{K}}$ + *Cluster*

Input: G // the world model
Output: T // a plan of tours

1: $T \leftarrow \emptyset$
2: $S \leftarrow \mathit{Cluster}(G, \mathit{BuildTour}_{\mathcal{K}}')$ // $\mathit{BuildTour}_{\mathcal{K}}'$ returns the reward of the built tour
3: **for all** $C \in S$ **do**
4: $\quad t \leftarrow \mathit{BuildTour}_{\mathcal{K}}(G, C),\ T \leftarrow T \cup \{t\}$
5: **end for**
6: **return** T

the robot successfully traveling one unit distance, and $\beta(v)$ is the reward of node v. At last, the estimated cost is calculated as $c_{ij} = (1 - \psi^{\mathcal{D}(0,e_i)+\mathcal{D}(0,e_j)+l_{ij}})\alpha$, where α is the value of the robot, and the estimated reward is $g_{ij} - c_{ij}$.

Finding the closest nodes v_i and v_j is $O(n^2)$ for two clusters of size n each; BFS is $O(n)$ as the graph is a tree. Other operations are constant. Therefore, merging takes $O(n^2)$.

For the whole algorithm, assume the graph contains N nodes. During one merge iteration, where all the clusters are pair-wise evaluated and the best pair is merged, regardless of how many clusters are there: each *node* is paired with each of the $O(N)$ nodes outside its own cluster exactly once for finding v_is and v_js; each node is visited $O(N)$ times in all the BFS's; and there are $O(N)$ constant operations. Therefore, one iteration is $O(N^2)$ and the whole algorithm is $O(N^3)$. Note that, given an $O(n^2)$ evaluation function $\mathcal{R}$, a similar argument, which considers nodes instead of clusters, can be made for the clustering algorithm in Algorithm 1. It makes the overall time complexity of Algorithm 1 $O(N^4)$ instead of $O(N^5)$.

3.5 Evaluation

Algorithm 6 $BuildTour_{\mathcal{K}}$ + PGC

Input: G // the world model
Output: T // a plan of tours
1: $T \leftarrow \emptyset,\ S \leftarrow PGC(G)$
2: **for all** $C \in S$ **do**
3: $\quad t \leftarrow BuildTour_{\mathcal{K}}(G, C),\ T \leftarrow T \cup \{t\}$
4: **end for**
5: **return** T

We run experiments with two clustering algorithms, the naive clustering algorithm (NC) in Algorithm 1 and PGC in Algorithm 4, combined with each of six tour-building heuristics (NG, OSA, TL, GML, LPG and GI) in Algorithm 2. Therefore, there are 12 different combinations of clustering and tour-building algorithms (2×6). The tour-building-NC combinations are detailed in Algorithm 5, where $BuildTour_{\mathcal{K}}'$ is the variation of $BuildTour_{\mathcal{K}}$ that still uses $\mathcal{K}$ to build tours but returns the reward of the built tour instead of the tour itself. The tour-building-PGC combinations are in Algorithm 6.

For each combination, we run experiments with different robot values and different variances in node rewards, which is also referred to as *node variance*. For each combination of clustering and tour building, robot value, and node variance, we experiment on 100 uniformly random graphs, each of which contains 100 nodes within a world of size 100×100. The success rate ψ is fixed to 0.99. The results are averaged from the 100 runs. Since the problem is NP-hard, computing the optimal solution is infeasible. Therefore we compare the solutions given by our algorithms.

Section 3.5.1 compares the performance (*quality of solution*) of the tour-building heuristics when they are combined with NC with node reward drawn from a integral

uniform distribution from 4 to 6. Section 3.5.2 shows the results of the tour-building-PGC combinations to compare PGC's performance against NC, when node reward drawn from a integral uniform distribution from 4 to 6. Finally, Section 3.5.3 discusses the interesting effects of changing the node variance.

3.5.1 Comparison of tour-building algorithms

To evaluate the performance of the six tour-building heuristics, we use NC for clustering and draw the node rewards from a integral uniform distribution from 4 to 6, then plot the normalized mean of expected reward against different robot values (α). The normalized mean of expected reward is the ratio of the mean in an experimental setting to the maximum, which is obtained in the setting where the robot value is 0, i.e. there is no cost in losing robot, and therefore the best plan is to send one robot to each node. The value of standard deviation is normalized by the same factor in order to be meaningful. The combined algorithms are described in Algorithm 5, and the plot is in Figure 3.1.

As shown in Figure 3.1, the six combinations have different performances. The order from high to low is GI, TL, LPG, OSA, GML and NG, with GI, TL and LPG being nearly identical, and GML and NG being nearly identical. Although, as shown in Section 3.5.3, the difference between GI, TL and LPG becomes more obvious when the data variance becomes larger. The standard deviation is *relatively* large for larger robot values for all the algorithms. This indicates that when the robot value is high, the algorithms are less stable. One possible reason is that, due

to the high cost of losing robot, even a small difference in distance can change the decision made by an algorithm.

GI being the best is not surprising, because insertion-based construction methods generally perform better than nearest-neighbor heuristics [31]. All the other five construction methods are only able to look ahead by appending new nodes to the end of a partial tour. Therefore, they are intuitively less powerful than any nearest-neighbor heuristics that adjust tours in various places. Consequently, it is surprising that TL, which takes only losses into account, performs almost as well as GI. Since NG and OSA consider the gain from the immediate next steps, and LPG and GML consider both gain and loss, this result indicates that the immediate loss, which is caused by robot traveling extra distance that discounts future gains, has more impact on the overall performance the immediate gain.

In addition, one advantage of a "look-ahead" heuristic like TL is that it can be easily adopted for online planning, in cases where nodes of interest appears dynamically, or edge length is not known until the robot reaches a node. And as shown by the results, TL performs almost as well as GI that requires all information before making decisions.

3.5.2 Comparison of clustering algorithms

To evaluate the performance of the PGC algorithm, we run the same experiments as in Section 3.5.1, with PGC instead of NC. The algorithms are

described in Algorithm 6, and the results are shown in Figure 3.2; the performance ranking is consistent with the ranking in using NC in Section 3.5.1.

To give a clearer comparison of PGC and NC, we plot the ratios of tour-building-PGC's expected reward to those of tour-building-NC, against different robot values (α) in Figure 3.3. PGC gives good results when combined with better tour building algorithms. Specifically, when combined with GI, TL or LPG, the ratio is above 0.8 when the robot value is below 30. However, when the robot value becomes relatively high, the performance ratio is fairly low. For example, when the robot value is around 60, the ratio is only about 0.4. It should be noted that, when the robot value is very high, above 60 in our experiments, the cost becomes too high that the algorithms rarely send out any robot at all. In such cases, the measures have a large variance and are therefore less indicative. They are included in the results only for completeness. PGC's performance downgrades when low-performance tour-building algorithms are used, which indicates that a better performing tour-building heuristic has more stable performance and is less sensitive to the choice of clustering algorithm.

3.5.3 Effect of node reward variance

We also evaluate the effects of node variance on the performance of the proposed algorithms. The same experiments as those in Section 3.5.1 and 3.5.2 are done with the node rewards randomly sampled from integral uniform distributions from 1 to 9, shown in Figure 3.4, and all set to 5, shown in Figure 3.5.

Interestingly, when the difference in performance of the tour-building algorithms varies in accordance with the node variance. Namely, as seen in Figure 3.4, when node variance becomes larger, the difference in performance becomes larger, and vice versa in Figure 3.5.

With respect to the performance of PGC, as shown in Figure 3.4(b), 3.4(c), 3.5(b) and 3.5(c), the result is similar to that of in Section 3.5.2. Namely, PGC performs well for low robot values (< 30) but bad for high robot values (30 to 60). In addition, comparing across node variances, PGC performs better for larger node variance.

3.6 Related work

A large volume of literature exists on route planning problems such as the Traveling Salesman Problems (TSPs) and the Vehicle Routing Problems (VRPs). In [11], several variations of the TSPs with profits are defined. The objective function may be the maximization of the collected total profit (Orienteering Problem), the minimization of the total traveling cost (Prize-Collecting TSP) or the optimization of a combination of both (Profitable Tour Problem). Traditionally, except for the Orienteering Problem, these problems assume a single-tour solution for only one vehicle [2]. Archetti et al. [2] use the term *VRPs with profits* to refer to the class of problems involving multiple vehicles.

The above problems are *company centric* because they maximize the payoff for the party that executes the plan. Some problems are *customer centric*, which means

that the average satisfaction rate of all customers is the most important requirement. The Minimum Latency Problem (MLP) is a general formulation for such a goal with variations such as: the traveling repairman's problem, the delivery man problem, the cumulative TSP, the cumulative capacitated vehicle problem, TSP with cumulative costs, and the school bus driver problem. [26] gives a comprehensive taxonomy of MLP problems in many different parameters such as the characteristics of nodes, arcs, the depot, vehicles, etc.

Most of the problems studied in the VRP literature do not consider the value of the vehicles or the risk of losing them. As mentioned above, the value of a vehicle can be the monetary cost of the vehicle or strategic importance of the vehicle, quantified by a real number. An interesting exception is the Cash-in-Transit VRP [36], which takes into account the values carried by the vehicle and the risk of the vehicles being robbed. However, since the risk is modeled as an integer constraint, it is not optimized as an objective and there's no uncertainty involved in executing plans. Also in CIT, all the customers must be visited, however, in our problem, as long as the reward is maximized, not all customers are to be visited.

In DRP, robots are deployed to collect items available at each location in a large area. Because we consider these robots to be UAVs, robots can travel from one location to another through the straight-line route between them. This assumption makes the problem environment a fully connected graph. The data can be photograph, video, temperature, etc. The robots are set off from a base station, where they need to return.

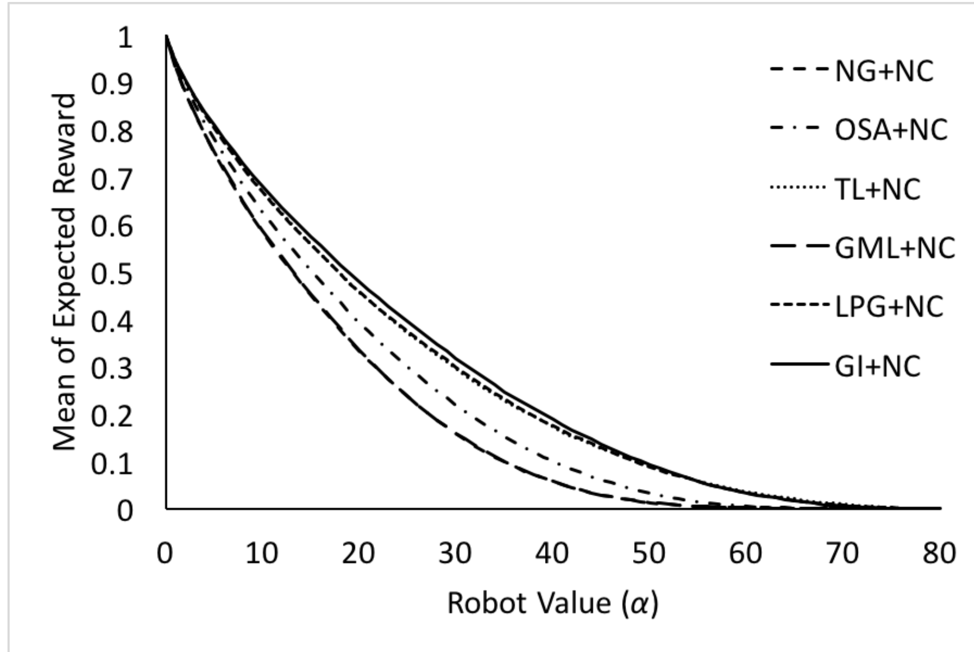
The closest work to DRP is the discounted-reward TSP problem studied in [5], where the reward a robot can get from a location is discounted by the distance it has to travel to reach there. However, in their work, the robot does not have to return to the base and its value is not considered, which makes the reward always positive. In DRP, the expected reward from a location is discounted due to the uncertainty of robot's breakdown along the route, and due to the value of the robot, the reward can be negative. The Multiple Agents Maximum Collection Problem [10] is similar to the discounted-reward TSP in the sense that the rewards decreases over time. However, the reward function is linear and it doesn't consider the agent's value either.

3.7 Conclusion

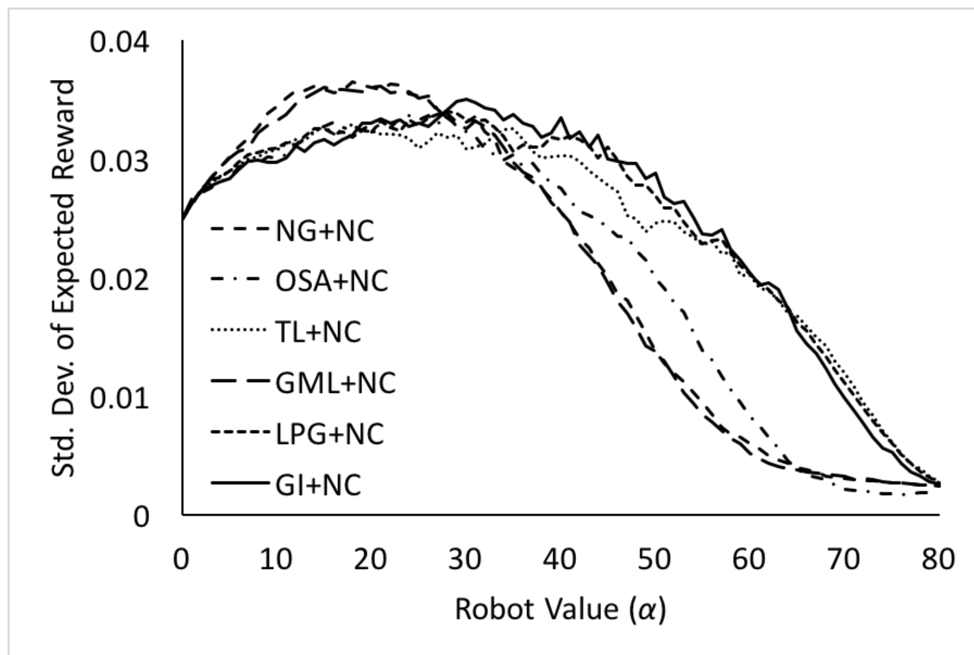
In this chapter, the Data-collecting Robot Problem is formulated as a planning problem on a complete graph. The objective is to maximize the expected reward of a plan (a set of tours). The expected reward is determined by both the gain from visiting nodes and loss of robots.

Six tour-building heuristics are proposed and compared. Among them, GI performs the best, followed closely by TL. However, since GI modifies a partial tour at any point, it cannot be used for online planning. TL, on the other hand, builds a tour by appending nodes at the end of a partial tour, so it can be adopted for online planning and we empirically prove that it performs almost as good as GI. We also propose PGC heuristic for clustering. With a better time complexity, this algorithm

approximates the naive clustering algorithm well for low robot values, and it does better when node variance becomes larger. In addition, we discover that the most important factor in maximizing the reward is the immediate loss. Another interesting observation is that the difference of performance among tour-building algorithms increases as the data variance increases.

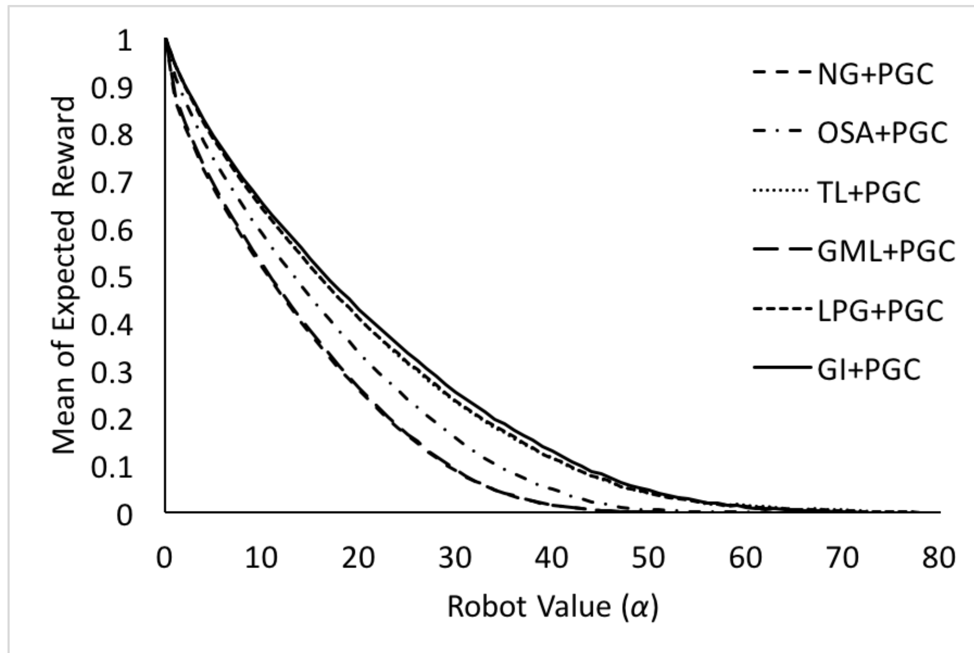


(a) Normalized mean of expected reward.

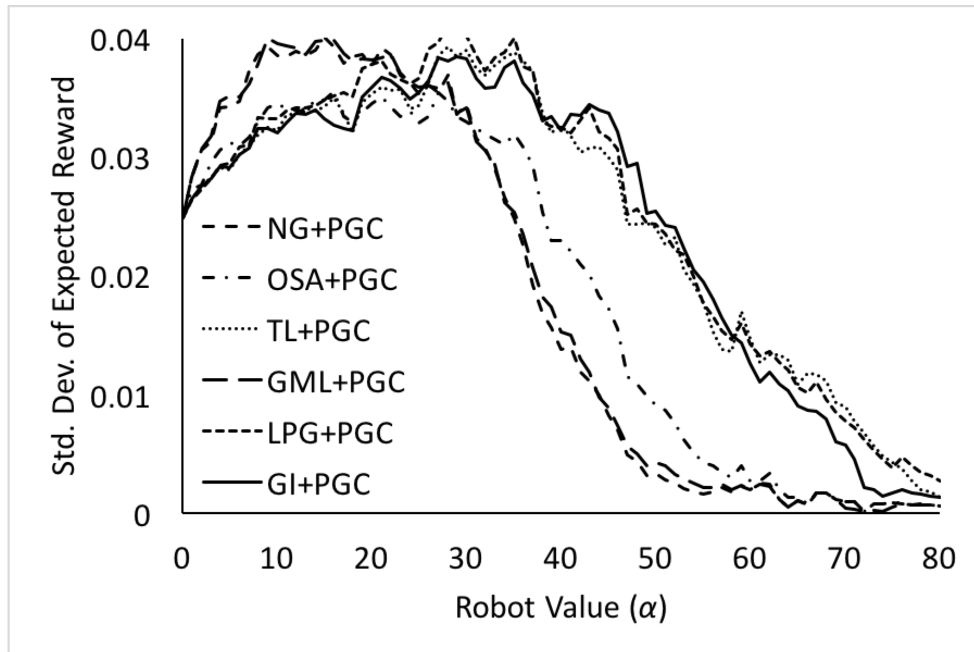


(b) Normalized standard deviation of expected reward.

Fig. 3.1.: The performance of tour-building algorithm combined with NC. Node rewards range from 4 to 6. The x-axes show the robot value α and the y-axes show **(a)** the average of the expected reward from 100 runs, where the values are normalized against the maximum, and **(b)** the standard deviations of the expected reward from 100 runs.



(a) Normalized mean of expected reward.



(b) Normalized standard deviation of expected reward.

Fig. 3.2.: The performance of tour-building algorithm combined with PGC. Node rewards range from 4 to 6. The meanings of the charts are the same as those in Figure 3.1.

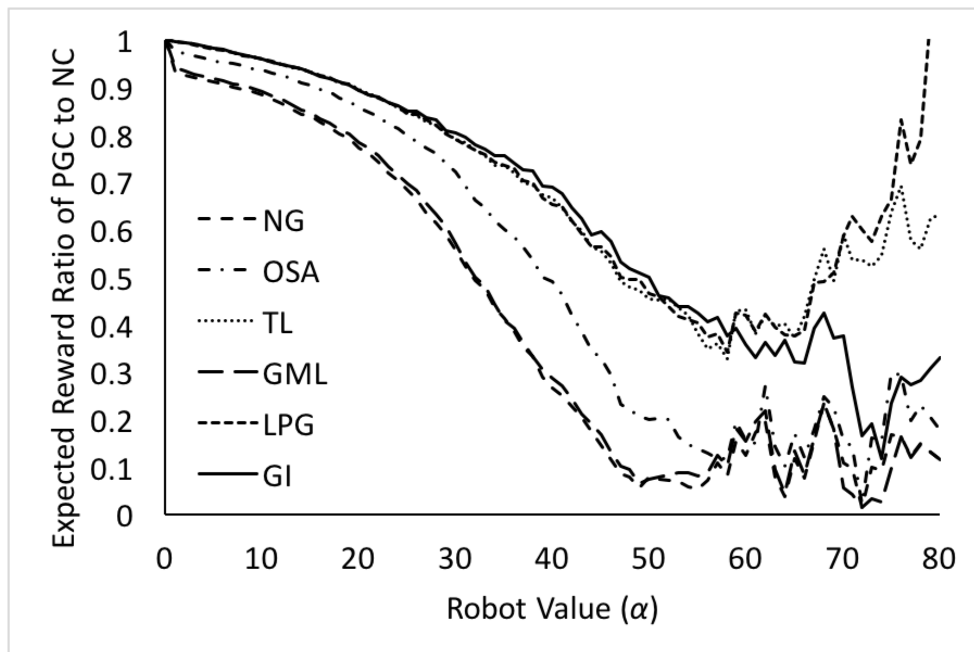
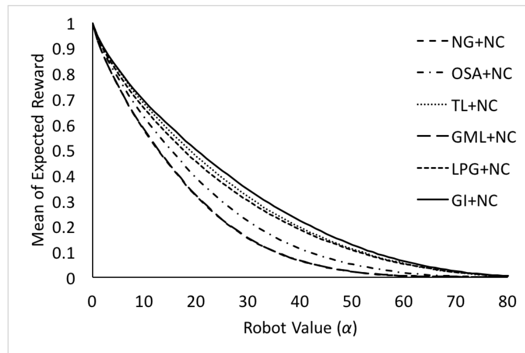
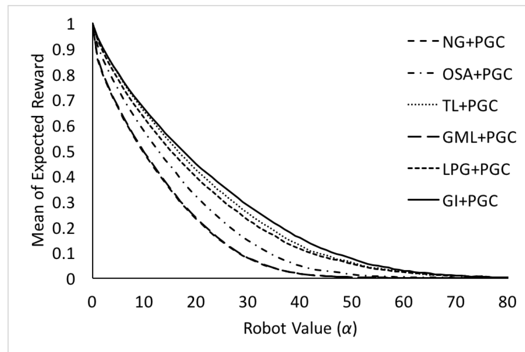


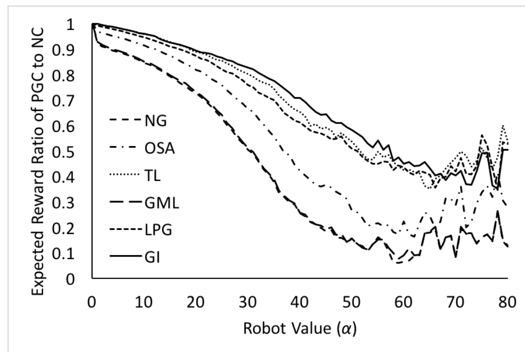
Fig. 3.3.: The performance ratio of PGC to NC. Node rewards range from 4 to 6.The x-axis shows the robot value α and the y-axis shows the ratio of the averaged expected reward of PGC to that of NC.



(a) Performance of tour-building-NC.

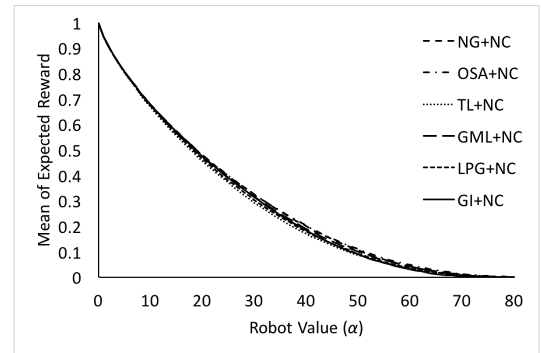


(b) Performance of tour-building-PGC.

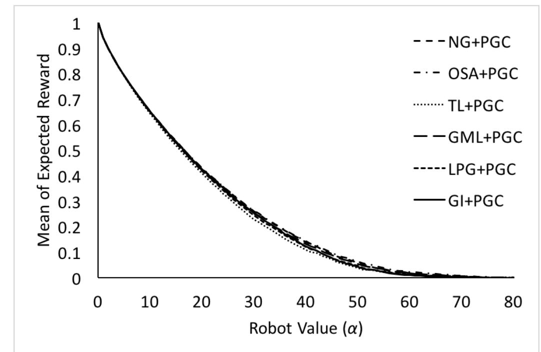


(c) Performance ratio of PGC to NC.

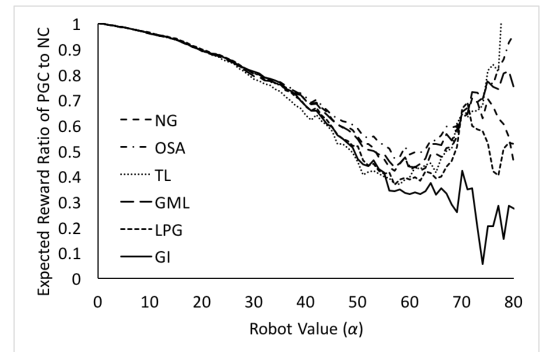
Fig. 3.4.: Node rewards range from 1 to 9. The meanings of the charts are the same as those in Figure 3.1 and Figure 3.3.



(a) Performance of tour-building-NC.



(b) Performance of tour-building-PGC.



(c) Performance ratio of PGC to NC.

Fig. 3.5.: Node rewards are all 5. The meanings of the charts are the same as those in Figure 3.1 and Figure 3.3.

4. GRID FORMULATION WITH NON-UNIFORM RISK AND ENERGY CONSTRAINT

4.1 Chapter overview

In Chapter 3, we assume a uniform distribution of risk and unlimited energy of robots. With these assumptions, the problem is formulated as a planning problem on a complete graph, the objective is to decide which locations of interests should be visited, in what order, and by how many robots, at the same time. Heuristic algorithms that consist of clustering step and tour-building step are proposed. However, in more realistic situations, the distribution of risk is not uniform, and robot's energy may run out and needs to be recharged. In such cases, more elaborate formulation is necessary. Inspired by [19, 20], where graph-based algorithms are employed to solve grid-based Markov Decision Process, we formulate DCP in a grid world, which enables us to model non-uniform risks. In addition, our formulation considers energy consumption of robots; a robot is destroyed if it runs out of energy during data collection. Therefore the solution needs to consider recharging at base station.

The energy constraint greatly increases the difficulty of the problem, especially in an environment of non-uniform risk. When there is a short path with high risk and a long path with low risk, decisions need to be made based on various factors,

such as the risk, current energy level, distance to the recharging station, positions of other locations of interests, etc. A good solution will find an effective balance between *safety and energy* (S&E). Due to this difficulty, the objective in this formulation is to collect as much data as possible using given UAVs, i.e., the solution does not consider whether a location should be visited or how many robots should be used. In this chapter, we first propose heuristic algorithms for navigating the UAVs, then we use deep neural network to learn a linear combination of the heuristics, which is proved to be more effective and efficient.

4.2 Contributions

- Proposing the *Data-collection Problem* (DCP), a Markov Decision Process that models autonomous data-collection in risky environment under energy constraints. Good solution of DCP needs to find good balance between *safety and energy* (S&E).

- Designing four navigation algorithms that have different priorities between S&E during navigation.

- Developing the *Ensemble Navigation Network* (ENN) that learns new heuristic from the four navigation algorithms and other heuristic information.

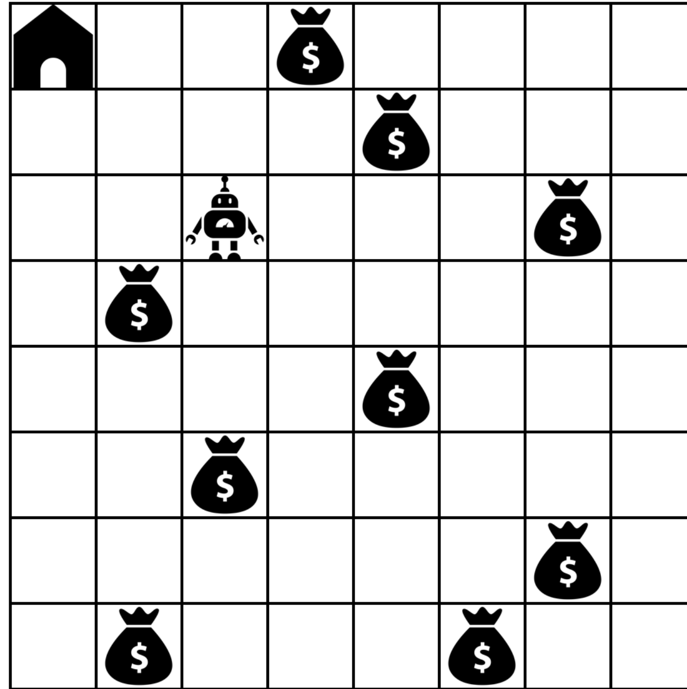
- Showing that ENN is able to find better balance between S&E from the given heuristics.

4.3 Problem formulation

DCP is a Markov Decision Process (MDP) on a $N \times N$ grid of cells. Each cell (x, y) has a risk value $\rho_{x,y}$, which is visible to the agent and indicates the probability of the agent being disabled when visiting the cell. Cell $(0, 0)$ is the base station, whose risk $\rho_{0,0} = 0$. Each item or the agent in the world occupies one cell. See Figure 4.1 for an example. In Figure 4.1a, the robot represents the agent, the money bags represent items, and the house represents the base. In Figure 4.1b, the numbers are the risk values. In addition to the information shown in the figure, there is a current energy level and a maximum energy level of the agent.

The MDP proceeds in discrete time steps. The initial state s_0 is randomly generated, therefore the initial locations of the agent and items are random. At each step t, the agent receives the state of the environment s_t, and selects an action a_t from the action set $\mathcal{A} = \{$*stay, up, down, left, right, up-left, up-right, down-left, down-right*$\}$ according to some policy π, which is a mapping from states s_t to actions a_t. In return, the agent receives the next state s_{t+1} and a scalar reward r_t. This process repeats until the agent reaches a terminal state, after which the process restarts.

The transition from s_t to s_{t+1} is as follows. The agent first moves to a adjacent cell according to a_t. If a_t makes the agent go out of a grid boundary, the agent stays along the axis perpendicular to that boundary. The agent then collects item if there is any in the new cell, after which the agent may be disabled with a probability equals to the risk of the cell. At the end, the energy level of the agent is reduced by



(a) The base, agent and items.

0.00	0.01	0.01	0.01	0.01	0.01	0.01	0.01
0.01	0.04	0.04	0.04	0.04	0.04	0.04	0.01
0.01	0.04	0.16	0.16	0.16	0.16	0.04	0.01
0.01	0.04	0.16	0.64	0.64	0.16	0.04	0.01
0.01	0.04	0.16	0.64	0.64	0.16	0.04	0.01
0.01	0.04	0.16	0.16	0.16	0.16	0.04	0.01
0.01	0.04	0.04	0.04	0.04	0.04	0.04	0.01
0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01

(b) The risk distribution.

Fig. 4.1.: An example of the DCP environment. **(a)** shows the locations of the base, agent and items. The robot represents the agent. The money bags represent items. The house represents the base. **(b)** shows a color-coded risk distribution, where the numbers are the risk values.

one, regardless of what action was taken, unless it is in the base. If the agent is in the base, its energy level is restored to the maximum. Otherwise, if the energy level drops to 0, the agent is disabled.

Items contain positive rewards, whose values are non-uniform and visible to the agents. The agent obtains a reward immediately after it collects an item, which is returned to the agent as r_t. There is *no* negative reward when the agent is disabled. The terminal state is reached when the agent is disabled or all the items are collected. The objective is to maximize the total reward collected in an *episode*, i.e., from initial state to terminal state, of the process.

4.4 Algorithm

In this section, we first choose a planning algorithm from previous work for navigation without energy constraint, which then is used in designing four navigation algorithms that have different priorities between S&E during navigation with energy constraint. With these four navigation algorithms as heuristic inputs, a deep neural network is eventually developed for finding the balance between S&E in DCP.

4.4.1 Maximizing reward without energy constraint

As a starting point, we ignore the energy constraint for the time being and focus on finding the path that maximizes the expected reward from collecting items. This reduces the problem to the one similar to the the one in Chapter 3 [42], where six

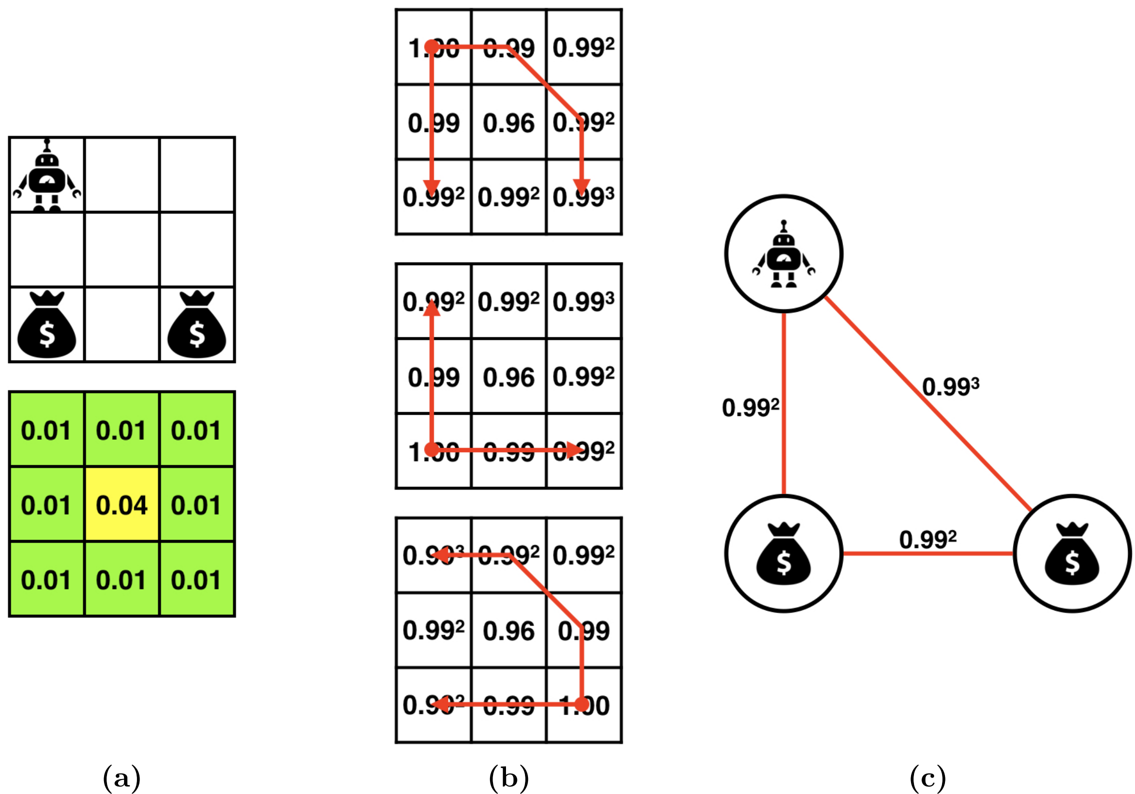


Fig. 4.2.: Extracing a complete undirected graph from a game state. **(a)** shows the original game state. The bottom grid contains *risk values*. **(b)** shows the safest paths calculated by Dijkstra's Algorithm for (from top down) the agent, the left item and the right item. The numbers are *success probabilities* instead of risk values. **(c)** shows the complete undirected graph created from the safest paths. The nodes represent the agent and items, and the edge weights are the success probabilities of safest paths.

Algorithm	**Reward Avg.**	**Reward SD.**
Naive-Greedy (NG)	15.64	8.51
One-Step-Ahead (OSA)	19.18	8.80
Total-Loss (TL)	20.05	8.58
Gain-Minus-Loss (GML)	18.09	9.02
Loss-Per-Gain (LPG)	20.25	8.40

Table 4.1: Results of experiments on the tour-building heuristics from Chapter 3 adopted to DCP. The average is taken from 10,000 runs. For every run, there is one agent with infinite amount of energy, the total reward is 30, and the risk distribution is as Figure 4.1b. **Reward Avg.** is the average of total reward collected during a game. **Reward SD.** is the standard deviation.

Algorithm 7 Loss-Per-Gain (LPG)

Input: Current state s, action set $\mathcal{A}$, the safest paths $\mathcal{P}_{\cdot,\cdot}$ calculated by Dijkstra's Algorithm, see Section 4.4.1 for more details
Output: Agent's action a and the target item i^*
  Initialize minimum loss-per-gain $v^* \leftarrow +\infty$
  Get agent location (x_z, y_z) from s
  for all item i in current state s **do**
    Initialize loss-per-gain $v \leftarrow +\infty$
    Get location (x_i, y_i) and reward r_i of item i from s
    for all item $j \neq i$ in current state s **do**
      Get location (x_j, y_j) and reward r_j of item j from s
      Compute the difference in success probabilities
        $\Delta = \mathcal{P}_{x_z,y_z}(x_j, y_j) - \mathcal{P}_{x_z,y_z}(x_i, y_i) \times \mathcal{P}_{x_i,y_i}(x_j, y_j)$
      $v \leftarrow v + \Delta \times r_j$
    end for
    $v \leftarrow v/(\mathcal{P}_{x_z,y_z}(x_i, y_i) \times r_i)$
    if $v < v^*$ **then**
      $v^* \leftarrow v$, $i^* \leftarrow i$
    end if
  end for
  Select the $a \in \mathcal{A}$ that follows the safest path from (x_z, y_z) to (x_{i^*}, y_{i^*}) according to $\mathcal{P}_{x_z,y_z}$
  return a, i^*

heuristics were proposed to maximize expected reward from a graph-based world. Out of the six, five heuristics are shown to solve the DCP without incurring high time complexity.

To adopt the heuristics, Dijkstra's algorithm [9] can be used to compute the *safest path* from any cell to any other cell on the grid. Given agent's and items' locations, a complete undirected graph can be built from the grid, where a node represents a *key cell* in the grid that contains the agent or an item, and the edge weight represents *success probability* of traveling from one key cell to another alone the safest path. See Figure 4.2 for an example, which shows how a 3×3 grid of the agent and two items is converted to a graph. Figure 4.2a is the state of the

environment. Figure 4.2b shows the success probabilities of following the safest path from any key cell to any other cell. In top-down order, the three grids are for the agent cell, the left item cell and the right item cell. The red arrows indicate the safest paths from one key cell to another. Finally, Figure 4.2c is the constructed graph, where the edge weights are from the cells containing the arrow heads in Figure 4.2b.

The tour-building heuristics from Chapter 3 can then be applied to find paths that have high expected reward. To test which heuristic works best in DCP without energy constraint, we run simulations on 10,000 randomly generated environments.[1] In all the environments, the risk distribution in Figure 4.1b is used, and the total reward is 30. The allocation of rewards is a multinomial distribution over the 8×8 options with 30 repeats, so the number of items is random. Table 4.1 shows the averaged results. *Reward* means the total reward collected during one episode. *Energy* means the total energy consumed during an episode. Loss-Per-Gain (LPG) algorithm gives the highest reward on average. Algorithm 7 presents the LPG algorithm that is adopted to DCP. In the algorithm, $\mathcal{P}_{x,y}(x', y')$ stands for the success probability of following the safest path from cell (x, y) to cell (x', y').

4.4.2 Navigation under energy constraint

With the energy constraint, a *navigation algorithm* needs to consider both collecting items and going back to the base station for recharging. There are two

[1] Although expected reward can be calculated, simulation results are used to make it consistent with the other results of this chapter.

Algorithm 8 Closest-First (CF)

Input: Current state s, action set $\mathcal{A}$
Output: Agent's action a and the target item i^*
  Initialize minimum distance $d^* \leftarrow +\infty$
  Get agent location (x_z, y_z) from s
  for all item i in current state s **do**
    Get location (x_i, y_i) of item i from s
    $d \leftarrow \mathbf{max}(|x_z - x_i|, |y_z - y_i|)$
    if $d < d^*$ **then**
      $d^* \leftarrow d$, $i^* \leftarrow i$
    end if
  end for
  Select the $a \in \mathcal{A}$ that minimizes $\mathbf{max}(|x_z - x_{i^*}|, |y_z - y_{i^*}|)$
  return a, i^*

extreme routes to take for navigating to the base station. One extreme is to follow the shortest path by going directly towards the base station, which uses the least amount of energy but can be risky. The other extreme is to follow the safest path calculated by Dijkstra's algorithm, which is the least risky but may be longer and therefore cost more energy.

Similarly, there are two extremes for collecting items, but these are intractable to compute, so we use the LPG algorithm in Algorithm 7 to approximate the safety-conservative extreme and the Closest-First (CF) algorithm, which always navigates an agent towards the item that has the minimal Manhattan distance regardless of the reward and risk, as an approximate to the energy-conservative extreme. Algorithm 8 presents the CF algorithm.

From the above-mentioned four methods, called *planning algorithms* hereafter, four navigation algorithms are designed:

Algorithm 9 Structure of the navigation algorithms discussed in Section 4.4.2

Input: Current state s, planning algorithm $\mathcal{N}_0$ for navigation to base, planning algorithm $\mathcal{N}_1$ for navigation to items, assume $\mathcal{N}_0$ and $\mathcal{N}_1$ take a location (x, y) as input and return an action that follows a path from (x, y) to a target, the target (an item or the base), and the length of the path
Output: Agent's action a

```
Get energy level e of agent from s
Get location (x_z, y_z) of agent from s
a_0, _, _ ← N_0(x_z, y_z)  // "_" means the value is not useful
a_1, i*, l_1 ← N_1(x_z, y_z)
Get location (x_{i*}, y_{i*}) of item i* from s
_, _, l'_0 ← N_0(x_{i*}, y_{i*})
if e < l_1 + l'_0 then
  a ← a_0  // Go to the base
else
  a ← a_1  // Go to the item
end if
return a
```

- **Safe-Reward** uses LPG to navigate to items, and follows the shortest path back to the base.
- **Safe-Recharge** uses CF to navigate to items, and follows the safest path back to the base.
- **Safe-Both** uses LPG to navigate to items, and follows the safest path back to the base.
- **Safe-Neither** uses CF to navigate to items, and follows the shortest path back to the base.

Algorithm 9 shows the details of these algorithms, which follow the same structure but use different planning algorithms. The navigation algorithms only consider if there is enough energy to reach the next item. With pre-calculated safest

path information, their time complexities are of the same order as those of the planning algorithms. This makes them ideal to be used in the deep learning method discussed in the following. More effective algorithms can be crafted by considering every step during the navigation, however, that would increase the time complexity by a factor linear to the number of steps from the current location to the target.

4.4.3 Finding the balance

Finding a good balance between S&E *manually* requires creating elaborate rules, which easily becomes intractable for large and diverse environments. Therefore, we proposed an *Ensemble Navigation Network* (ENN) to learn good balance using a linear combination of the above-mentioned navigation algorithms and some other information discussed in the following. Figure 4.3 shows the structure of ENN.

Structure of ENN

The ENN is composed of a Convolutional Neural Network (CNN) and one or more heuristics discussed above. The heuristics, H_i for $i \in [1, n]$ and $n \in \mathbb{N}^+$, take state s as input and generate an action vector $H_i(s)$ which gives each action $a \in \mathcal{A}$ a *score*. The CNN takes state s as input and generates one weight $w_i(s; \theta)$ vector for each action vector of the heuristics, a bias $b(s; \theta)$ and a state value estimation

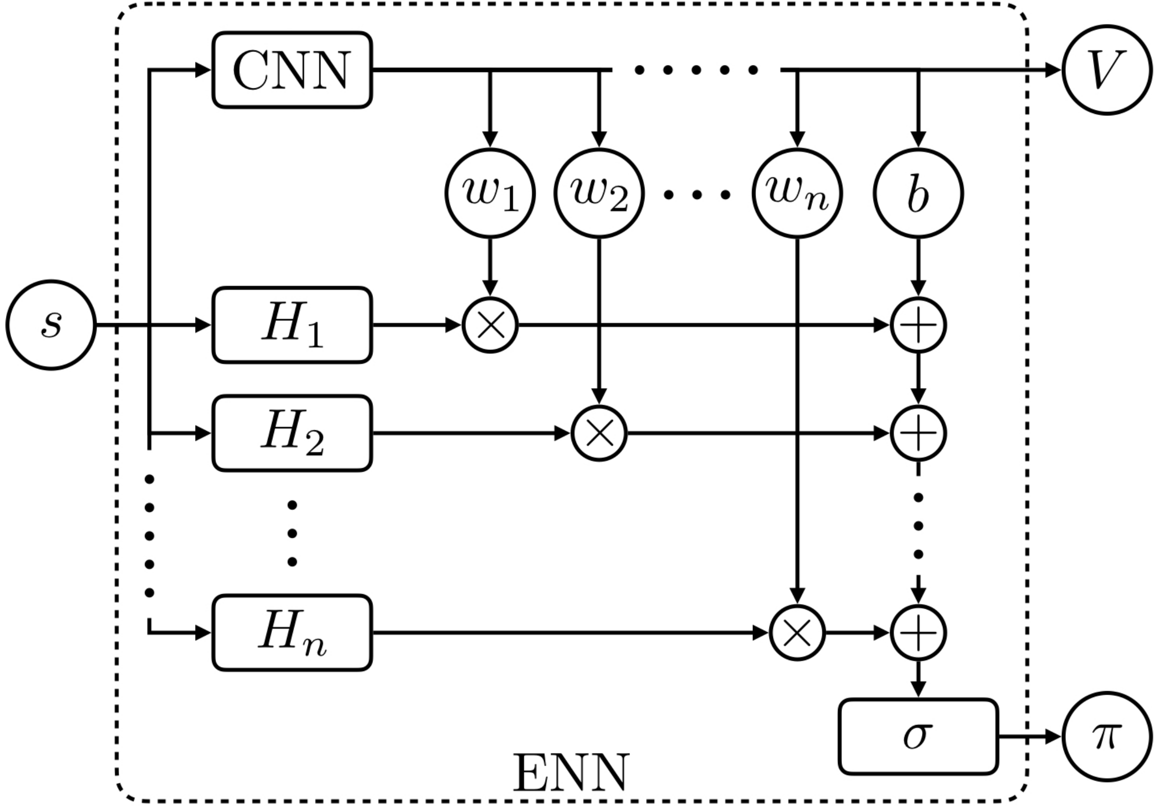


Fig. 4.3.: Structure of ENN. ENN is composed of a Convolultional Neural Network (CNN) and a number of heuristics H_i, all of which take game state s as input. The heuristics output action vectors $H_i(s)$ and the CNN outputs one weight $w_i(s;\theta)$ for each action vector, a bias $b(s;\theta)$, and a state value estimation $V(s;\theta_v)$, where θ and θ_v represent network parameters. The final policy $\pi(s;\theta)$ is a softmax function σ of a linear combination of all the outputs.

$V(s;\theta_v)$, where θ and θ_v represent the network parameters. The final policy $\pi(s;\theta)$ is a softmax function σ of a linear combination of all the outputs:

$$\pi(s;\theta) = \sigma(\sum_{i=1}^{n} w_i(s;\theta) \times H_i(s) + b(s;\theta))$$

where $\times$ denotes *element-wise* multiplication, also known as the *Hadamard product.*

The CNN in ENN

The core of proposed ENN is CNN. As CNN's input, a state s is represented by two $N \times N$ matrices, M^{item} and M^{agent}. M^{item} represents the locations of items, the value at the x-th row y-th column, m^{item}_{xy}, is the reward of the item at coordinate (x, y). If there is no item at (x, y), $m^{\text{item}}_{xy} = 0$. M^{agent} encodes the agent's current location and energy level. $m^{\text{agent}}_{xy} = e$ where e is the current energy level if agent is at coordinate (x, y), $m^{\text{agent}}_{xy} = 0$ otherwise.

The CNN takes the two matrices M^{item} and M^{agent} as input. Following the input layer is a number of convolutional layers, after which is a fully connected layer that summarizes features from the last convolutional layer. The final layer is the output layer, which gives the aforementioned weights $w_i(s; \theta)$, bias $b(s; \theta)$ and state value estimation $V(s; \theta_v)$ linear outputs. $V(s; \theta_v)$ is the critic in the actor-critic learning system[2]. Although the network parameters θ and θ_v are shown differently for generality, they share all the parameters except for those in the output layer in our experiments.

The Heuristics in ENN

There are ten heuristics in ENN, each of which gives an action vector that consists of a score for each action $a \in \mathcal{A}$. The four navigation algorithms proposed in Section 4.4.2 are used as heuristics. Their action vectors are one-hot vectors that indicate the chosen actions. Other than these four, the followings are also included:

[2]The actor is the final policy generated by ENN.

- **Loss-Per-Gain** gives a one-hot vector for the action chosen by LPG algorithm.
- **Closest-First** gives a one-hot vector for the action chosen by CF algorithm.
- **Safe-Trip-Home** gives a one-hot vector for the action that follows the safest path to base.
- **Fast-Trip-Home** gives a one-hot vector for the action that follows the shortest path to base.
- **Distance-To-Home** gives a score for each action that is the shortest-distance-to-base after the actions is taken.
- **Risk** gives a score for each action that is the risk of taking the action.

Training

The pseudocode for the training procedure of ENN is presented in Algorithm 10. This procedure is adopted from the the Asynchronous Advantage Actor-Critic (A3C) algorithm in [25], which requires multiple threads to run in parallel to counteract the dependencies among consecutive steps and hence stabilizes training. All the threads share the same network, but each thread has its own environment and performs optimization updates asynchronously. Adam optimization algorithm [15] with a linearly decaying learning rate is used for training.

In order to compute a single update, a thread needs to use the global network to proceed for $t_{\max}$ steps or until the terminal state is reached. For each state s_t, the

Algorithm 10 Training procedure of a thread from [25]

Input: Shared iteration counter T and maximum $T_{\max}$, shared step maximum $t_{\max}$, shard network parameter θ and θ_v

```
while T < T_max do
    T ← T + 1
    if s_t is not defined or is terminal then
        Initialize a new environment
    end if
    Initialize step counter t ← 0
    repeat
        Store current parameters θ' ← θ, θ'_v ← θ_v
        Get current state s_t
        Select action a_t according to policy π(a_t|s_t; θ')
        Execute action a_t and receive reward r_t and state s_{t+1}
        t ← t + 1
    until s_t is terminal or t = t_max
    Initialize R_t = { 0,             if s_t is terminal
                     { V(s_t; θ'_v),  otherwise
    for i ← t − 1 to 0 do
        R_i ← r_i + γR_{i+1}
    end for
    Update θ and θ_v using data batch from steps 0, ..., t − 1
end while
```

global network generates a policy $\pi(a_t|s_t;\theta)$, which is used to sample an action a_t from the action set $\mathcal{A}$, and a value estimate $V(s_t;\theta_v)$ for state s_t. The thread receives the reward r_t and the next state s_{t+1} from the environment after the action is taken. With a maximum of $t_{\max}$ number of (s_t, a_t, r_t, s_{t+1}) tuples, the thread performs batch gradient descent to minimize the loss function. This process repeats until a total number of $T_{\max}$ updates are performed by all the threads.

The loss function mainly consists of two terms. Let θ' and θ'_v be the parameters before a parameter update, s_t, a_t and r_t be the state, action and reward at time step

t, and γ be the discounting factor that signifies immediate rewards. The first term specifies the loss from the policy:

$$L_\pi(\theta) = -\log \pi(a_t|s_t;\theta)A(s_t, a_t; \theta'_v)$$

where $A(s_t, a_t; \theta'_v) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta'_v) - V(s_t; \theta'_v)$ is the estimate of the advantage of taking action a_t in state s_t, and k is the number of time steps in a parameter update.

The second term specifies the loss from the estimate of value function, a Huber loss is used instead of simple squared error loss:

$$L_v(\theta_v) = \begin{cases} 0.5(R_t - V(s_t;\theta_v))^2, & \text{if } |R_t - V(s_t;\theta_v)| \le 1 \\ |R_t - V(s_t;\theta_v)| - 0.5, & \text{otherwise} \end{cases}$$

where $R_t = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(s_{t+k}; \theta'_v)$ for k time steps.

4.5 Evalution

This section compares ENN with the navigation algorithms proposed in Section 4.4.2 in different environmental settings. In all the following experiments, ENN has two convolutional layers, all of which have filters of size 3×3 and strides of size 1 with same paddings[3]. The filter counts are 32 and 16 in that order. The

[3]Input is padded with 0s so that the input and output are of the same size.

Risk distribution	$\boldsymbol{\rho^{(0)}}$	$\boldsymbol{\rho^{(1)}}$	$\boldsymbol{\rho^{(2)}}$	$\boldsymbol{\rho^{(3)}}$
Low Risk	0.01	0.02	0.04	0.08
Medium Risk	0.01	0.03	0.09	0.27
High Risk	0.01	0.04	0.16	0.64

Table 4.2: Risk values of the three different risk distributions used in Section 4.5.1. Each distribution has four layers of risk values. From outer to inner, the risk values are $\rho^{(0)}$, $\rho^{(1)}$, $\rho^{(2)}$ and $\rho^{(3)}$. As an example, high risk is shown in Figure 4.1b.

fully connected layer has 256 hidden units. All the hidden layers are followed by rectifier nonlinearity. The network is implemented using TensorFlow [1].

All the trainings are done in 1 million steps by 16 threads, i.e. $T_{\max} = 1{,}000{,}000$. The maximum batch size $t_{\max} = 32$. We use a reward discount factor $\gamma = 0.99$. For the Adam optimizer, the initial learning rate is set to 1×10^{-4} and is linearly annealed to 0 over the course of training. The β_1 and β_2 parameters of Adam optimizer are set to 0.9 and 0.999 respectively.

The output of ENN is used as a stochastic policy in all the experiments, i.e., instead of always choosing the action with the highest probability, the agent randomly samples an action from the probability mass function defined by the output. This helps break symmetries during navigation when the agent is trapped into infinite loops.

We also observed the behavior of ENN for about 100 episodes, so the following discussions are based on both the experimental results and our observations.

Algorithm	Reward			Energy			Reward / Energy		
	Avg.	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**
Safe-Reward	11.55	6.38	**13.99**	29.65	19.75	-19.49	0.495	0.301	**38.18**
Safe-Recharge	7.39	5.33	**78.26**	12.03	9.42	98.46	0.691	0.339	**-0.95**
Safe-Both	8.51	5.16	**54.73**	24.35	18.27	-1.99	0.429	0.241	**59.52**
Safe-Neither	8.05	6.08	**63.45**	11.05	9.85	116.07	0.854	0.341	**-19.87**
ENN	13.16	7.17	—	23.87	16.88	—	0.685	0.314	—

Table 4.3: Results of experiments on the high-risk distribution specified in Table 4.2. The average is taken from 10,000 runs. For every run, the total reward is 30, and the initial and maximal energy of agent is 8 and 16 respectively. **Reward** is the total reward collected during a game. **Energy** is the total energy consumed during a game. **Reward / Energy** is the reward collected per energy consumed during a game. **Avg.** means the average over 10,000 runs. **SD.** is the standard deviation. **Incr.%** is ENN's increment as a percentage of an algorithm's corresponding average. For example, Safe-Reward's reward incr.% is the difference between ENN's reward and Safe-Reward's reward as a percentage of Safe-Reward's.

Algorithm	Reward			Energy			Reward / Energy		
	Avg.	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**
Safe-Reward	13.43	6.62	**9.66**	33.14	19.88	-25.94	0.495	0.266	**44.31**
Safe-Recharge	9.51	6.95	**54.84**	17.56	15.36	39.77	0.625	0.311	**14.33**
Safe-Both	9.08	5.45	**62.21**	25.05	18.58	-2.03	0.439	0.241	**62.80**
Safe-Neither	12.04	8.17	**22.34**	19.16	16.80	28.09	0.771	0.312	**-7.34**
ENN	14.73	7.51	—	24.54	16.46	—	0.715	0.302	—

Table 4.4: Results of experiments on the medium-risk distribution specified in Table 4.2. Terms used in the table and other experimental settings are the same as Table 4.3.

Algorithm	Reward			Energy			Reward / Energy		
	Avg.	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**
Safe-Reward	18.08	8.37	**7.10**	39.55	23.23	-14.09	0.546	0.238	**17.37**
Safe-Recharge	13.41	8.42	**44.34**	27.58	21.54	23.21	0.578	0.269	**11.00**
Safe-Both	11.78	6.82	**64.38**	29.90	21.64	13.64	0.478	0.245	**34.07**
Safe-Neither	17.88	9.60	**8.29**	34.30	24.83	-0.93	0.655	0.265	**-2.08**
ENN	19.36	8.87	—	33.98	18.68	—	0.641	0.250	—

Table 4.5: Results of experiments on the low-risk distribution specified in Table 4.2. Terms used in the table and other experimental settings are the same as Table 4.3.

4.5.1 Different risk distributions

We first consider environments of 8×8 cells. The initial state of an environment contains a random number of items with a total reward of 30. The initialization of items is a random process that repeatedly puts a reward of 1 to a random cell in the environment, which follows a multinomial distribution of 30 repeats over 64 choices. The initial energy level of agent is 8 and the maximum energy level is 16, which is large enough for a round trip from the base station to any other cell with some leeway. With this setting, the number of states is of the order of $64^{30} + 64^{29} + \ldots + 64^{0}$ due to the arrangement of rewards.

Three different risk distributions are considered in this section. Each distribution has four risk values $\rho^{(0)}$, $\rho^{(1)}$, $\rho^{(2)}$ and $\rho^{(3)}$ that form four layers in the environment. In Figure 4.1b, the distribution, denoted as *high risk*, has $\rho^{(0)} = 0.01$, $\rho^{(1)} = 0.04$, $\rho^{(2)} = 0.16$ and $\rho^{(3)} = 0.64$. Table 4.2 shows the risk values for all three distributions: *high risk*, *medium risk* and *low risk*. The terms high, medium and low describe both the risk values and the differences between risk values, or the *risk gradient*.

The results of experiments on the high-risks, medium-risk and low-risk environments are shown in Table 4.3, Table 4.4 and Table 4.5 respectively. ENN constantly performs the best in all the risk distributions, followed by the Safe-Reward navigation algorithm. The others' performance vary with the risk distribution. In the tables, the difference in measurement between ENN and a

navigation algorithm is shown as a percentage of the navigation algorithm's measurement, highlighted in bold.

In high-risk environments, as shown in Table 4.3, Safe-Reward collects 11.55 rewards and consumes 29.65 energy per episode on average. This gives it a 0.495 *reward per energy* (RPE)[4]. In comparison, ENN can collect 13.16 rewards, which is about 65% of the LPG result in Section 4.4.1 where agent has infinite energy, and consumes 23.87 energy per episode on average. With 0.685 RPE, ENN achieves a 13.99% increase in rewards and a 38.18% increase in RPE over Safe-Reward.

As the risk distribution varies from high to low, ENN's advantage in reward collection decreases, this is possibly because as the variance in risk values decreases, the difference between expected rewards of different routes become smaller. The only exception is the Safe-Both navigation algorithm, for which ENN's advantage increases. This is because Safe-Both follows the safest paths both towards the items and towards the base, this makes the route too long to be finished and the agent stays in base most of the time. Therefore for this algorithm, energy is a much more limiting factor than risk, and reducing the risk does not improve its reward collection as much as for the others.

The situation for RPE is more interesting. As the risk distribution goes from high to low, ENN's advantage goes up then down. There are two ways for ENN to improve upon the navigation algorithms, **(a)** is via saving energy by taking a shorter path of higher risk, and **(b)** is via improving safety by following a safer yet longer path and thus consuming more energy. **(a)** increases RPE and **(b)** decreases

[4]Reward per energy is also an average over 10,000 runs.

it. In a medium-risk environment, **(a)** has better effect compared to that in a high-risk environment, therefore the improvement in RPE is better. In a low-risk environment, **(a)** should have even better effect. However, since the risk is too low, the agent can collect most of the items, and as the items are collected, the reward density decreases and the distances between items increase regardless of what route is taken. This undermines the effect of **(a)** and decreases ENN's advantage in RPE. But there is any exception in this RPE trend for the Safe-Neither algorithm, for which ENN's advantage is negative and monotonically increases. This is because Safe-Neither completely disregards the risk by always following the shortest path, which provides high RPE. But as the risk decreases, the agent survives for longer, so the density of reward decreases and so does Safe-Neither's RPE.

Another interesting observation is that, Safe-Both performs better compared to Safe-Recharge and Safe-Neither in terms of reward collected in high-risk environments, but becomes worse in medium-risk and low-risk environments, because shortest paths are more beneficial in safer environments.

4.5.2 Effect of reward density

To study the effect of reward density, we run additional experiments with 15 total rewards instead of 30. The risk distribution is fixed to high risk shown in Figure 4.1b. And the other experimental settings are the same as those in Section 4.5.1. The same network trained in Section 4.5.1 is used here because the network has experienced states with 15 or less reward during training.

Algorithm	Reward			Energy			Reward / Energy		
	Avg.	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**
Safe-Reward	6.42	3.65	**17.57**	24.31	15.80	-23.23	0.327	0.232	**49.19**
Safe-Recharge	4.11	3.12	**83.81**	11.05	8.71	68.90	0.448	0.314	**9.10**
Safe-Both	4.60	2.96	**64.00**	19.60	13.69	-4.79	0.265	0.176	**83.96**
Safe-Neither	4.78	3.62	**57.90**	10.10	9.49	84.84	0.594	0.311	**-17.80**
ENN	7.55	3.76	—	18.66	12.39	—	0.488	0.235	—

Table 4.6: Results of experiments where the total reward is 15. Terms used in the table and other experimental settings are the same as Table 4.3.

Algorithm	Reward			Energy			Reward / Energy		
	Avg.	SD.	Incr.%	Avg.	SD.	Incr.%	Avg.	SD.	Incr.%
Safe-Reward	17.66	9.10	**12.01**	34.18	21.14	14.30	0.598	0.235	**-5.44**
Safe-Recharge	10.46	7.71	**89.04**	12.66	10.61	208.71	0.948	0.312	**-40.35**
Safe-Both	19.43	8.71	**1.82**	41.10	21.87	-4.92	0.536	0.218	**5.49**
Safe-Neither	9.73	7.17	**103.34**	10.87	9.31	259.36	1.001	0.298	**-43.50**
ENN	19.78	8.54	—	39.07	20.29	—	0.566	0.229	—

Table 4.7: Results of experiments where the initial and maximum energy levels are set to 16 and 32 respectively. Terms used in the table and other experimental settings are the same as Table 4.3.

As shown in Table 4.6, compared to the numbers in Table 4.3, ENN has better advantage over all the navigation algorithms. This indicates that ENN is more effective in navigating towards items than the any navigation algorithm alone.

Notice the low reward density does not decrease ENN's advantage in RPE as it does in the low-risk environment in Section 4.5.1. This is simply because ENN collects much more reward than the navigation algorithms, which outweighs the influence of low reward density on RPE.

4.5.3 Effect of energy capacity

To study the effect of energy capacity, we run additional experiments with initial and maximum energy set to 16 and 32 respectively. The risk distribution is fixed to high risk shown in Figure 4.1b. And the other experimental settings are the same as those in Section 4.5.1. A new ENN is trained with the new energy capacity because the network used in Section 4.5.1 has never experienced states with energy level higher than 16 during training.

Algorithm	Reward			Energy			Reward / Energy		
	Avg.	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**
Safe-Reward	0.70	0.46	**18.78**	5.80	4.60	15.46	0.232	0.308	**16.98**
Safe-Recharge	0.43	0.50	**93.82**	5.32	3.33	25.99	0.164	0.273	**65.67**
Safe-Both	0.45	0.50	**84.57**	5.83	3.33	14.85	0.156	0.269	**74.10**
Safe-Neither	0.55	0.50	**52.09**	4.78	3.68	40.22	0.228	0.317	**18.93**
ENN	0.83	0.37	—	6.70	9.32	—	0.272	0.294	—

Table 4.8: Results of experiments where there is only one item of reward one in the environment. Terms used in the table and other experimental settings are the same as Table 4.3.

As shown in Table 4.7, with more energy at disposal, the algorithms that take advantage of safest paths really shine. Specifically, Safe-Both and Safe-Reward become dominantly better than Safe-Recharge and Safe-Neither, with Safe-Both being the best. ENN's advantage over Safe-Both is trivial because the need to change Safe-Both's choices, and save energy, too small to make a difference in reward collection.

Another fact worth noting is that with 16 initial and 32 maximum energy, ENN and Safe-Both can collect over 19 reward per episode, which is above 95% of the performance of LPG algorithm with infinite energy shown in Table 4.1.

4.5.4 Single-item data collection

To show ENN's advantage in data-collection efficiency in a different perspective. We run additional experiments with the same environmental settings as those in Table 4.3 except the number of item and the total reward are both set to 1, i.e., there is always one single item of reward one in any initial state. The same network trained in Section 4.5.1 is used in these experiments.

Algorithm	Reward			Energy			Reward / Energy		
	Avg.	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**
Safe-Reward	11.74	6.36	**-37.10**	30.17	19.69	-30.09	0.491	0.300	**-13.85**
Safe-Recharge	7.41	5.42	**-0.31**	12.17	9.64	73.28	0.683	0.336	**-38.07**
Safe-Both	8.56	5.20	**-13.75**	24.52	18.33	-13.99	0.428	0.248	**-1.11**
Safe-Neither	8.16	6.12	**-9.49**	11.19	9.86	88.52	0.852	0.344	**-50.38**
Linear Learner	7.39	4.55	—	21.09	17.18	—	0.423	0.241	—

Table 4.9: Results of experiments for a simple linear learner. Terms used in the table and other experimental settings are the same as Table 4.3.

Algorithm	Reward			Energy			Reward / Energy		
	Avg.	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**	**Avg.**	**SD.**	**Incr.%**
Safe-Reward	11.74	6.45	**-22.99**	30.13	19.70	-66.54	0.484	0.276	**107.99**
Safe-Recharge	7.45	5.39	**21.30**	12.24	9.69	-17.61	0.688	0.343	**46.33**
Safe-Both	8.53	5.14	**6.01**	24.46	18.18	-58.77	0.425	0.241	**137.12**
Safe-Neither	8.13	6.16	**11.18**	11.21	10.01	-10.05	0.851	0.346	**18.34**
EQN	9.04	4.42	—	10.08	9.21	—	1.007	0.352	—

Table 4.10: Results of experiments for the EQN shown in Figure 4.4. Terms used in the table and other experimental settings are the same as Table 4.3.

From Table 4.8, we can see that the advantages of ENN over the navigation algorithms are all much higher, compared to those shown in Table 4.3. The minimum increment percentage of average episode reward is now 18.78% for the Safe-Reward algorithm. And noticeably, the advantages of RPE are now all positive. ENN is shown to be more energy-efficient than the Safe-Neither algorithm which always follows the shortest path, because it is able to successfully collect the item more often due to its choice of safer routes.

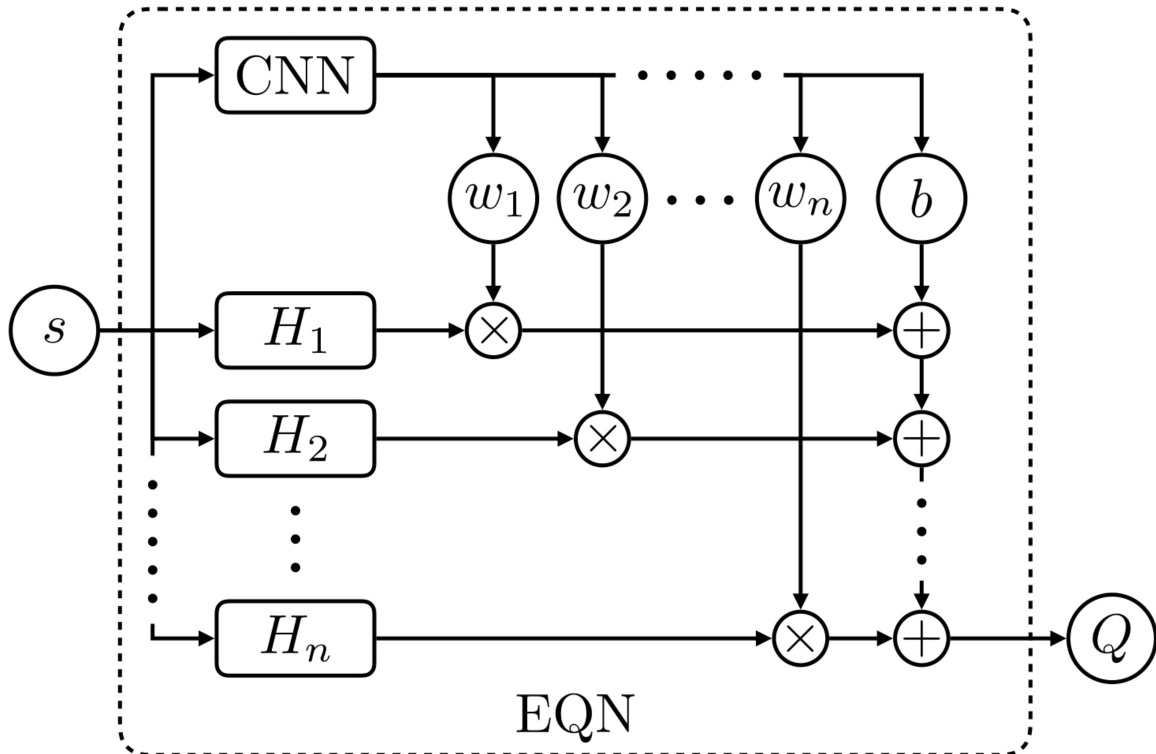


Fig. 4.4.: Structure of EQN. EQN is composed of a Convolultional Neural Network (CNN) and a number of heuristics H_i, all of which take game state s as input. The heuristics output action vectors $H_i(s)$ and the CNN outputs one weight $w_i(s;\theta)$ for each action vector and a bias $b(s;\theta)$, where θ represents network parameters. The final output is an estimate of the state-action value function $Q(s, a; \theta)$ for every action a, which is a linear combination of all the heuristic outputs.

4.5.5 Comparison to other learning methods

In this section, we compare ENN with alternative learning methods. Additional agents are trained in the same environmental settings as those in Table 4.3, and are evaluated with experiments.

The first method is a simple linear learner. This learner uses a linear function to approximate the policy $\pi(a|s;\theta)$, and another linear function to approximate the state-only value function $V(s;\theta_v)$. The training procedure is the same as that of ENN (Algorithm 10). The results for this linear learner is shown in Table 4.9. We can see that this simple learner performs worse than the worst navigation algorithm, Safe-Recharge. This is mainly due to two reasons: (1) the linear learner does not have the help from the heuristics, some of which are designed to maximize expected reward in a probabilistic environment, and (2) the linear learner's simple structure is not able to deal with the large state space and therefore does not generalize well.

The second method is an Ensemble Q Network (EQN) that is inspired by the Deep Q Network from [23] and [24]. The structure of EQN is shown in Figure 4.4, which is the same as that of ENN except for the final output. The final output is an approximation of the state-action value function $Q(s,a;\theta)$ as a linear combination of all the heuristic outputs. The same set of heuristics presented in Section 4.4.3 are used. The training procedure is the Asynchronous n-step Q-learning algorithm found in [25]. For evaluation, the ϵ-greedy policy is employed where $\epsilon = 0.05$. The results are shown in Table 4.10, which indicate that EQN does not perform as well

as our ENN in terms of average episode reward, but it does perform significantly better in terms of average RPE.

Since most of the heuristics provide information on actions, ENN is able to leverage them better than EQN because ENN generates policy (probability of each action) directly while EQN generates the state-action value function (evaluation of state and action). EQN needs to learn a function of higher dimension and therefore its convergence is slower. On the other hand, EQN is inherently better at evaluating how good a state is, which is directly related to the current energy level. Therefore EQN is better at staying in high-energy states and has better RPEs than ENN.

4.6 Related work

Markov Decision Processes (MDPs) with mean payoff objectives (total reward collected) have been heavily studied since the 60s. See [28] for a comprehensive survey. And MDPs with energy constraints are also studied quite extensively [18]. However, the combination of the two has just started to attract attention. The most recent and related work is [6], where the Energy Markov Decision Process (EMDP) is proposed. Given an EMDP and its initial state, the task is to compute a safe strategy that maximizes the expcted mean payoff, where safe means the energy never drops to 0. The focus of their work is to construct approximations of optimal strategies, using linear programming methods, with different assumptions of the problem structure. In contrast, our work focuses on providing a practical end-to-end solution using deep reinforcement learning.

Lane and Kaelbling model a robotic package delivery problem as MDP [19, 20]. To deal with the intractable number of states, they create action *macro*s that treats an entire policy of the MDP as an action, then the original problem is reduced to subproblems with much smaller state spaces and a problem of selecting the order in which the packages should be delivered. The grid-based MDP is partially transformed to a graph-based optimization problem, and off-the-shelf combinatorial optimization routines for the Traveling Salesman Problem is employed to achieve a exponential speedup. Their work considers the stochasticity of robot movement, but it does not have any kind of energy constraint.

4.7 Conclusion

In this chapter, *Data-collection Problem* (DCP) is proposed. The problem models a situation, in which a robotic agent collecting digital data in a risky environment under energy constraint. A good solution finds a good balance between *safety and energy* (S&E).

Four navigation algorithms that have different priorities during the mission are designed. They represent different optimization goals, e.g. safety-first or energy-first. A *Ensemble Navigation Network* (ENN), which consists of a Convolutional Neural Network and a number of heuristics including the four navigation algorithms, is developed to automatically find a good balance between S&E.

ENN is trained using deep reinforcement learning and has superior performance in all the experiments conducted. From the experiments, we learn that ENN has

better advantage over the four navigation algorithms when the risk in the environment is high, when the density of reward is low, and when the amount of energy is limited.

5. GRID FORMULATION WITH OPPONENT AND ENERGY CONSTRAINT

5.1 Chapter overview

The previous chapters consider the risk as a probability of robot being disabled, which is an abstract way of modeling risk. The solution are concerned with how to move the robot in the static environment so that it can collect more data before disabled. We first use expected reward as an optimization goal to create plans in Chapter 3, then use reinforcement learning to make the robot learns through trial-and-error in Chapter 4.

In the real world, the risk can come from internal factors such as hardware issues, or from external factors from the environment. For the external factors, however, they can be static, like tree branches, or they can be dynamic, like thunderstorms or animals. While modeling all these different types of risk in a unified probabilistic way is reasonable, there are patterns in the dynamic type that we can exploit to further improve data-collection efficiency.

In this chapter, we model the dynamic risk as a deterministic one in the form of an opponent agent. The data-collecting agent is disabled when comes close to the opponent. The opponent has different types of behaviors, which exhibit different action patterns. These behaviors correspond to the behaviors of the thunderstorms

or animals in our motivating example. For example, a thunderstorm usually moves towards the same direction in a short period of time; an animal, such as a bird or a monkey, may move from point A to point B repeated for daily routines, or it may chase a robot because it perceives the robot as a threat. If the robot can recognize these behaviors, it can improve its own safety greatly and collect more data. Therefore, our goal is to recognize the opponent's behaviors from the exhibited patterns, and effectively avoid the opponent while collecting data.

5.2 Contributions

- Proposing the *Data-collection Game* (DCG), a Stochastic Game that models autonomous data-collection in an environment with opponent under energy constraints. The opponent may employ different strategies, which can be exploited to improve data-collection efficiency.
- Designing four deep neural networks for navigation the data-collecting agent in DCG. Three of the networks model the opponent, implicitly or explicitly.
- Empirically showing that opponent modeling significantly improve data-collection performance by helping the agent make more effective movements, and explicit opponent modeling help the networks converge faster during training.

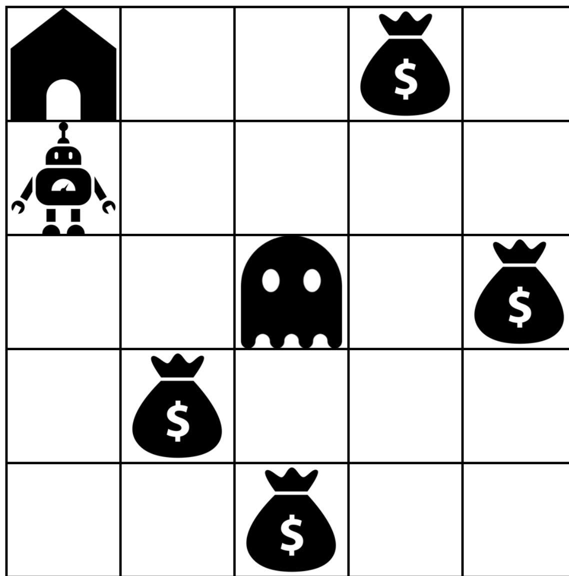


Fig. 5.1.: An example of the DCG environment. The robot represents the collector. The ghost represents the adversary. The money bags represent items. The house represents the base.

5.3 Problem formulation

DCG is a two-player Stochastic Game (SG) on a $N \times N$ grid of cells. The controlled player is the data-collecting agent, referred to as the *collector* hereafter, and the other player is the opponent agent, referred to as the *opponent* hereafter. The opponent has a set of strategies Π, from which it uniformly randomly chooses one for each *episode* of the game. Cell $(0, 0)$ is the base. Each agent or item occupies one cell. See Figure 5.1 for an example. In the figure, the robot represents the collector, the ghost represents the opponent, the money bags represent items, and the house represents the base.

The SG proceeds in discrete time steps. In the initial state s_0, the collector is located in the base, but the locations of the opponent and items are randomly selected. At each step t, the collector receives the state of the environment s_t, and selects an action a_t from the action set $\mathcal{A} = \{$*stay, up, down, left, right, up-left, up-right, down-left, down-right*$\}$ according to some policy π, which is a mapping from states s_t to actions a_t. The opponent automatically selects an action o_t according to its strategy/policy $\pi^o \in \Pi$. The actions selected by both agents form a joint action, which is then executed upon the environment. In return, the collector receives the next state s_{t+1} and a scalar reward r_t. This process repeats until the terminal state is reached, after which it restarts.

The transition from state s_t to state s_{t+1} is defined as follows. The opponent first makes a move according to its action o_t. If this movement leads the opponent to the same cell as the collector, the collector is disabled. Otherwise, the collects

makes a move according to its action a_t. If this movements leads the collector to a cell that contains an item, the item is collected. If the collector ends up in the same cell as the opponent, the collector is disabled. Any movement that would lead an agent out of the grid boundary makes the agent stays along the axis perpendicular to that boundary. At the end, the energy level of the collector is reduced by one, regardless of what action was taken, unless it is in the base. If the collector is in the base, its energy level is restored to the maximum. Otherwise, if the energy level drops to 0, the collector is disabled.

Items contain positive rewards, whose values are non-uniform and visible to the agents. The collector obtains a reward immediately after it collects an item, which is returned to the collector as the scalar reward r_t for time step t. There is *no* negative reward when the collector is disabled. The terminal state is reached when either the collector is disabled or all the items are collected. The objective is to maximize the total reward collected by the collector in an *episode*, i.e., from initial state to terminal state, of the game.

The game is similar to the Pacman game, however, there are two major differences: (1) the collector needs to consider its energy level so it has to make precise moves and go back for recharge periodically, and (2) the opponent can employ different strategies and therefore exhibit different behaviors.

5.4 Algorithm

To study the effect of OM, we use four different deep neural networks to control the collector in DCG. These networks have different ways to model opponents. In this section, θ, θ_v and θ_o represent network parameters. They are reused across subsections for succinctness, but no parameters are shared between different networks unless explicit stated.

5.4.1 State representation

For all of the networks, a state s_t of the environment is represented by two $N \times N$ matrices, M_t^{item} and M_t^{agent}. M_t^{item} represents the locations of items at time t, the value at the x-th row y-th column, $m_{t,x,y}^{\text{item}}$, is the reward of the item at coordinate (x, y) at time t. $m_{t,x,y}^{\text{item}} = 0$ if there is no item at (x, y) at t. M_t^{agent} encodes the agents' locations and the energy level of the collector at time t. $m_{t,x,y}^{\text{agent}} = e$ where e is the current energy level if the collector is at coordinate (x, y) at t, $m_{t,x,y}^{\text{agent}} = -1$ if the opponent is at coordinate (x, y) at t, $m_{t,x,y}^{\text{agent}} = 0$ otherwise.

5.4.2 No Opponent Modeling

The *Vanilla Navigation Network* (VNN) is a Convolutional Neural Network that takes only the state s as input. Following the input layer is a number of convolutional layers, after which is a fully connected layer that summarizes features from the last convolutional layer. The final layer is the output layer, which consists of a policy $\pi(s; \theta)$ as a softmax output that is a probability mass function over the

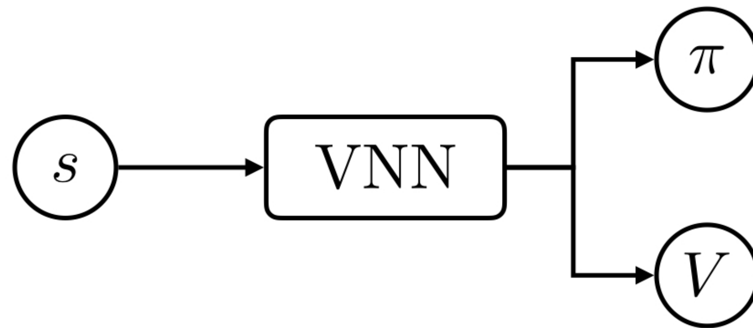


Fig. 5.2.: Structure of VNN in Section 5.4.2. VNN is a Convolultional Neural Network. It takes state s as input, and gives the policy $\pi(s, h; \theta)$ and the value estimation $V(s, h; \theta_v)$ as outputs, where θ and θ_v are network parameters.

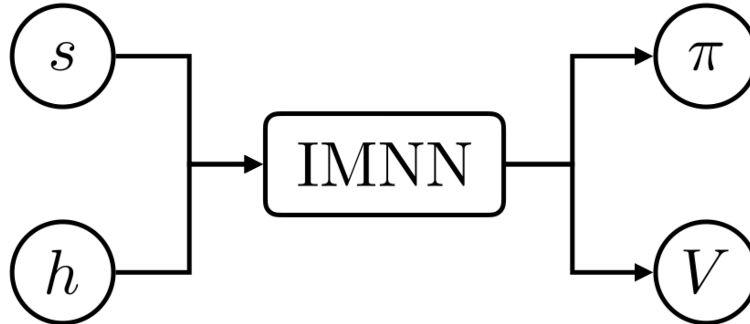


Fig. 5.3.: Structure of IMNN in Section 5.4.3. IMNN is a Convolultional Neural Network. It takes state s and history h as inputs, and gives the policy $\pi(s, h; \theta)$ and the value estimation $V(s, h; \theta_v)$ as outputs, where θ and θ_v are network parameters.

action set $\mathcal{A}$, and an estimate of the state value function $V(s; \theta_v)$ as a linear output. Although the network parameters θ and θ_v are shown differently for generality, they share all the parameters except for those in the output layer in our experiments. The structure of VNN is shown in Figure 5.2.

Because VNN only have the current state as input, it does not have enough information to model the opponent.

Similar as in Section 4.4.3, the loss function consists of two terms, one for the loss from policy $\pi(s; \theta)$, the other for the loss from value estimation $V(s; \theta_v)$. And the training procedure is adopted from the the Asynchronous Advantage Actor-Critic (A3C) algorithm in [25], shown in Algorithm 10.

5.4.3 Implicit Opponent Modeling

The *Implicit Modeling Navigation Network* (IMNN) has the same network structure as the VNN in Section 5.4.2, expect for the input layer. Instead of only taking the current state s_t as input, it also takes the history h of agents' locations.

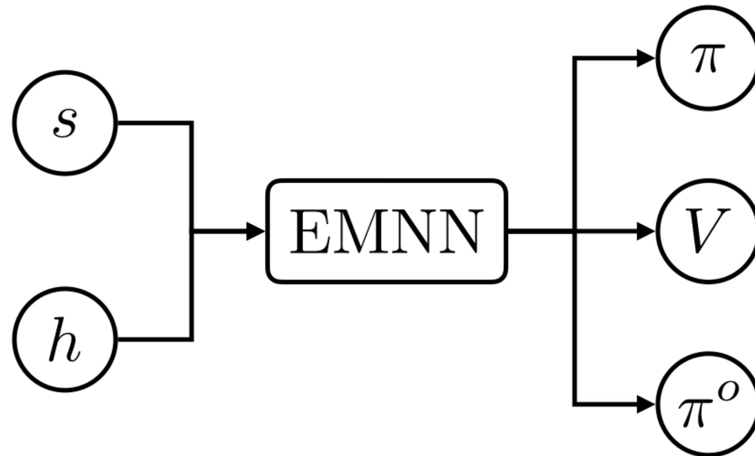


Fig. 5.4.: Structure of EMNN in Section 5.4.4. EMNN is a Convolultional Neural Network. It takes state s and history h as inputs, and gives the policy $\pi(s, h; \theta)$, the value estimation $V(s, h; \theta_v)$, and opponent strategy $\pi^o(s, h; \theta_o)$ as outputs, where θ, θ_v and θ_o are network parameters.

This history has a fixed length k, so the input consists of matrices $M_t^{\text{item}}, M_t^{\text{agent}}, M_{t-1}^{\text{agent}}, \ldots, M_{t-k}^{\text{agent}}$ for time t. It has the same types of outputs: a policy $\pi(s, h; \theta)$ and value estimation $V(s, h; \theta_v)$. The structure of IMNN is shown in Figure 5.3.

Because IMNN have the history of the opponent's behavior, it is able to predict its action and take advantage of the prediction *implicitly*. See Section 5.5 for more discussion on the effect of this implicit modeling.

The loss function and training procedure of IMNN are the same as those of VNN in Section 5.4.2, with a slight modification of how the input is obtained.

5.4.4 Explicit Opponent Modeling with same network

The *Explicit Modeling Navigation Network* (EMNN) has the same network structure as the IMNN in Section 5.4.3, expect for the output layer. In addition to

Algorithm 11 Training procedure of a thread for EMNN in Section 5.4.4

Input: Shared iteration counter T and maximum $T_{\max}$, shared step maximum $t_{\max}$, shard network parameter θ, θ_v and θ_o
  while $T < T_{\max}$ **do**
    $T \leftarrow T + 1$
    if s_t is **not** defined **or** is terminal **then**
      Initialize a new environment
    end if
    Initialize step counter $t \leftarrow 0$
    repeat
      Store current parameters $\theta' \leftarrow \theta$, $\theta'_v \leftarrow \theta_v$, $\theta'_o \leftarrow \theta_o$
      Get current state s_t and history h_t
      Select action a_t according to policy $\pi(a_t|s_t, h_t; \theta')$
      Execute action a_t and receive reward r_t, state s_{t+1}, and opponent action o_t
      $t \leftarrow t + 1$
    until s_t is terminal **or** $t = t_{\max}$
    Initialize $R_t = \begin{cases} 0, & \text{if } s_t \text{ is terminal} \\ V(s_t, h_t; \theta'_v), & \text{otherwise} \end{cases}$
    for $i \leftarrow t - 1$ **to** 0 **do**
      $R_i \leftarrow r_i + \gamma R_{i+1}$
    end for
    Update θ, θ_v, and θ_o using data batch from steps $0, \ldots, t - 1$
  end while

the policy $\pi(s, h; \theta)$ and value estimation $V(s, h; \theta_v)$, it also gives a prediction of the opponent strategy $\pi^o(s, h; \theta_o)$. In experiments, θ, θ_v and θ_o share the same parameters except for those in the output layer. The structure of EMNN is shown in Figure 5.4.

In addition to the policy loss and the value estimation loss, EMNN's loss function has an extra cross entropy term:

$$L_o(\theta_o) = \alpha \sum_{a \in \mathcal{A}} -\mathcal{I}(a) \log \pi^o(a|s, h; \theta_o) \tag{5.1}$$

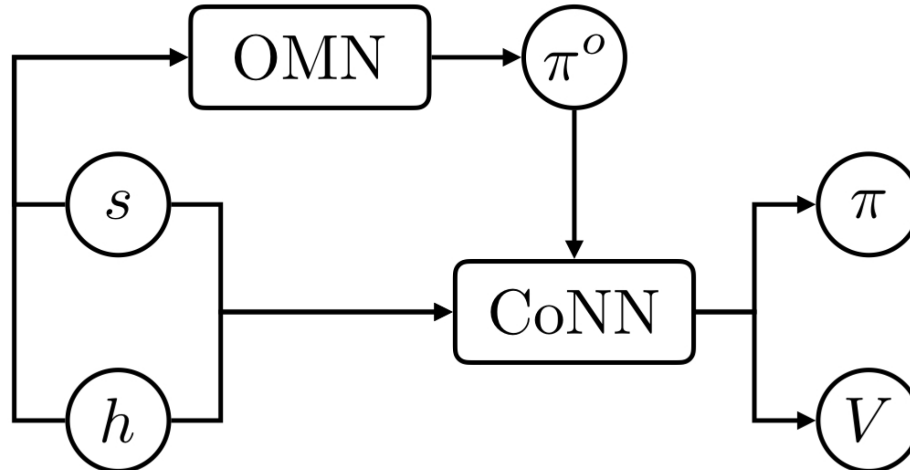


Fig. 5.5.: Structure of OMN and CoNN in Section 5.4.4. OMN and CoNN are Convolultional Neural Networks. OMN takes state s and history h as inputs, and gives the opponent strategy $\pi^o(s, h; \theta_o)$ as output. CoNN takes state s, history h, and opponent strategy π^o as inputs, and gives the policy $\pi(s, h, \pi^o; \theta)$ and the value estimation $V(s, h, \pi^o; \theta_v)$ as outputs. θ, θ_v and θ_o are network parameters.

where α specifies the strength of the term, $\mathcal{I}$ is an indicator function that returns 1 if a is the action taken by the opponent and 0 otherwise, and $\pi^o(a|s, h; \theta_o)$ is the probability assigned to action a by the network. $L_o(\theta_o)$ measures the accuracy of $\pi^o(s, h; \theta_o)$ against the actual action taken by the opponent. The training procedure is detailed in Algorithm 11.

5.4.5 Explicit Opponent Modeling with separate network

A designated network named *Opponent Modeling Network* (OMN) is used for explicit opponent modeling. The OMN has the same hidden layers as VNN. It takes s and h as input and produces $\pi^o(s, h; \theta_o)$ as the output. Combined with s and h, this output is used as input to another network called *Cooperating Navigation Network* (CoNN), which also has the same hidden layers but maintains the policy

Algorithm 12 Training procedure of a thread for OMN and CoNN in Section 5.4.5

Input: Shared iteration counter T and maximum $T_{\max}$, shared step maximum $t_{\max}$, shard network parameter θ, θ_v and θ_o

```
while T < T_max do
    T ← T + 1
    if s_t is not defined or is terminal then
        Initialize a new environment
    end if
    Initialize step counter t ← 0
    repeat
        Store current parameters θ' ← θ, θ'_v ← θ_v, θ'_o ← θ_o
        Get current state s_t and history h_t
        Get prediction of opponent strategy π^o(s_t, h_t; θ_o)
        Select action a_t according to policy π(a_t|s_t, h_t, π^o(s_t, h_t; θ_o); θ')
        Execute action a_t and receive reward r_t, state s_{t+1}, and opponent action o_t
        t ← t + 1
    until s_t is terminal or t = t_max
    Initialize R_t = { 0,                                  if s_t is terminal
                     { V(s_t, h_t, π^o(s_t, h_t; θ_o); θ'_v),  otherwise
    for i ← t − 1 to 0 do
        R_i ← r_i + γR_{i+1}
    end for
    Update θ, θ_v, and θ_o using data batch from steps 0, ..., t − 1
end while
```

$\pi(s, h, \pi^o; \theta)$ and value estimation $V(s, h, \pi^o; \theta_v)$. The structures of the OMN and CoNN are shown in Figure 5.5.

The loss function of CoNN is the same as that of IMNN. The loss function of OMN only contains the cross entropy loss defined by Equation 5.1. The training procedure is presented in Algorithm 12.

5.4.6 Comparison of the networks

VNN can only see the current state of the environment, so it does not have any way to model the opponent. On the contrary, all the other networks have access to

the location history of the environment, which can be used to infer the strategy of the opponent.

IMNN is a simple extension of VNN that takes the history as additional input, and, as VNN, its optimization goal is to solely maximize the total reward collected during an episode of the game, i.e., it does not try to predict the behavior of the opponent explicitly.

EMNN is an extension of IMNN. With the same inputs, EMNN tries to maximize the episode reward and correctly predict the action of the opponent *at the same time*. Although the prediction of the opponent action is not used in controlling the collector in any way, it helps in shaping the network parameters during optimization.

OMN is an completely isolated network whose solo goal is to predict the action of the opponent. This makes the opponent modeling task more straightforward for the learning procedure, therefore the accuracy of opponent modeling is potentially better than that of EMNN. CoNN then uses the prediction from OMN, the state and history to maximize episode reward.

5.5 Evaluation

In this section, we compare the performance of the networks proposed in Section 5.4. We first talk about the environmental settings, the network structure and hyperparameters used in all the experiments, then we experiment on the networks in various perspectives.

5.5.1 Environmental settings

As the standard setting, the environment has a size of 5×5 with 10 total rewards. The allocation of items is a random process that repeatedly puts a reward of 1 to a random cell in the environment, which follows a multinomial distribution of 10 repeats over 25 choices. This makes the number of states in the order of $25^{10} + 25^9 + \ldots + 25^0$ due to the arrangement of rewards. The maximum energy level of the collector is 15. There is one collector and one opponent. The collector's policy is stochastic, i.e., the output of network is considered as a probability mass function over the action space. For each game state, an action is randomly drawn from the policy according to the probability distribution. The opponent has 3 strategies in its strategy set Π:

- **patrol** the opponent chooses a random location other than the base, and moves back and forth between the initial location and the chosen one.
- **restricted** the opponent always moves towards the collector, but the opponent can only choose actions from $\mathcal{A}^- = \{\textit{stay, up, down, left, right}\}$, i.e., it cannot move diagonally.
- **with-fog** if the collector is on the top or left edge, the opponent cannot see the collector and therefore it stays, otherwise the opponent moves towards the collector with any action from $\mathcal{A}$.

The opponent uniformly randomly chooses a strategy from Π for each episode of the game and follows it for the entire episode.

5.5.2 Network structure and hyperparameters

In our experiments, all the networks have the same types of hidden layers: following the input layer are two convolutional layers with 3×3 filters, stride size 1 and same paddings[1], and the filter counts are 32 and 16; afterwards there is a fully connected layer with 256 neurons, which is followed by the output layer. For the networks with opponent modeling, the length of history $k = 10$. All the networks are implemented using TensorFlow [1].

Unless otherwise stated, training is done by 16 threads in 1 million steps, i.e., $T_{\max} = 1{,}000{,}000$. The maximum batch size $t_{\max} = 32$. The reward discount factor $\gamma = 0.99$. We use Adam optimizer [15] for optimization. The initial learning rate is set to 1×10^{-4} and is linearly annealed to 0 over the course of training. The β_1 and β_2 parameters of Adam are set to 0.9 and 0.999 respectively.

5.5.3 Evaluation in the standard setting

We run experiments in 10,000 randomly generated environments. The results on reward collected per episode, or *episode reward*, are shown in Figure 5.6. The results on reward collected per energy spent in an episode, or *reward per energy* (RPE), are shown in Figure 5.7.

As shown in Figure 5.6, the effect of opponent modeling on episode reward is significant. With implicit modeling, IMNN collectors collect about 9% more reward on average compared to VNN collectors. With explicit modeling, EMNN collectors

[1] Input is padded with 0s so that the input and output are of the same size.

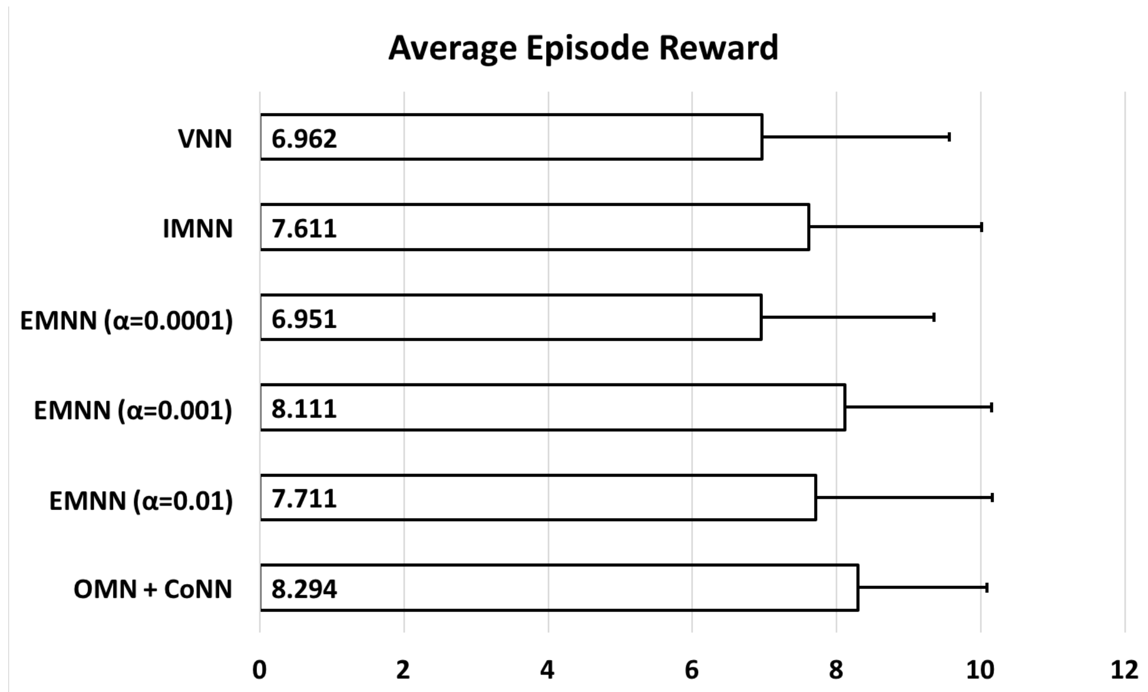


Fig. 5.6.: Results on episode reward for networks trained in 1 million steps. The labeled bars in the chart show the averaged episode rewards over 10,000 episodes. The error bars are the standard deviations.

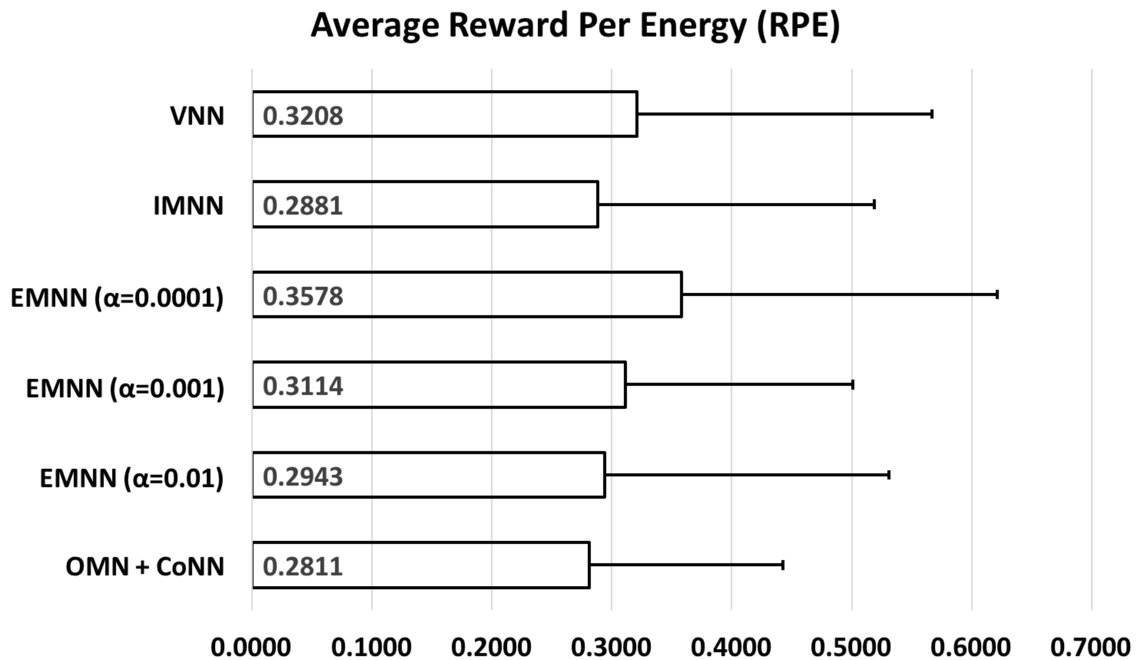


Fig. 5.7.: Results on reward per energy (RPE) for networks trained in 1 million steps. The labeled bars in the chart show the averaged RPEs over 10,000 episodes. The error bars are the standard deviations.

with $\alpha = 0.001$ can collect about 17% more and OMN + CoNN collectors can collect above 19% more rewards. OMN + CoNN achieve highest reward with the smallest standard deviation. These improvements are significant because collecting items becomes more difficult as items are collected.

We can see that explicit modeling works better than implicit modeling, and using a separate network for the modeling gives only slight improvement in performance over using the same network. However, our experiments show EMNN's accuracy in predicting opponent actions is only around 75%, but OMN's accuracy can achieve over 98%. In DCG, the impact of opponent action on the game state is limited: it can only move to the 8 adjacent cells. So the benefit of predicting the action is relatively small. Therefore, OMN's high accuracy in opponent modeling may have better impact on performance in other domains where the opponent action has higher impact on game states, such as Poker.

The RPE results in Figure 5.7 indicates EMNN with $\alpha = 0.0001$ is the best in RPE compared with the others, followed by VNN. To understand this result, we need to realize that RPE is affected by the density of the rewards, i.e., the average distance between any two items. Namely, the higher the density the higher the RPE, regardless of the network. As the items are collected, the density decreases, and so does the RPE. With that in mind, EMNN with $\alpha = 0.0001$ and VNN having better RPE than the others is no longer surprising. However, EMNN with $\alpha = 0.001$ also obtains a good RPE even though it can collect high reward. This shows, when trained in 1 million steps, EMNN with $\alpha = 0.001$ uses energy more efficiently than OMN + CoNN.

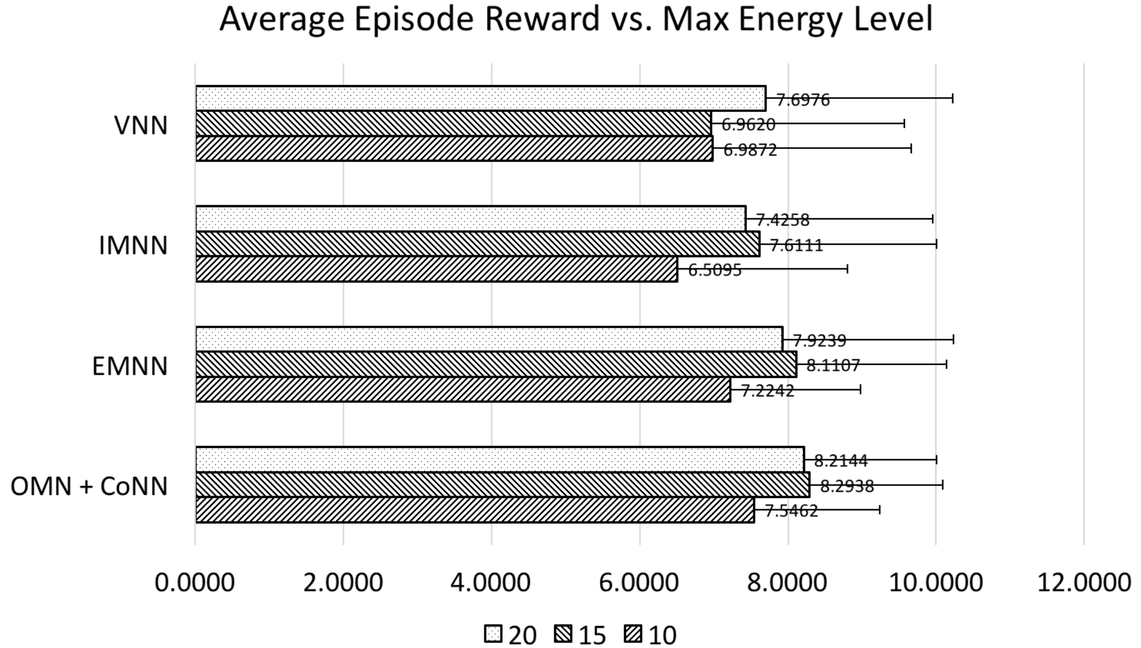


Fig. 5.8.: Results on episode reward for different maximum energy levels. The labeled bars in the chart show the averaged episode rewards over 10,000 episodes. The error bars show the standard deviations.

Since we show that $\alpha = 0.001$ gives the best results of EMNN, we only evaluate EMNN with $\alpha = 0.001$ and use "EMNN" to refer "EMNN with $\alpha = 0.001$" for the rest of this section.

5.5.4 Effect of energy capacity

To study the effect of energy capacity, we train additional networks with maximum energy level set to 10 and 20. All the other settings remain the same for these additional networks. Then, for each energy capacity, 10,000 experiments are conducted for each type of the networks. The results are shown in Figure 5.8 and Figure 5.9.

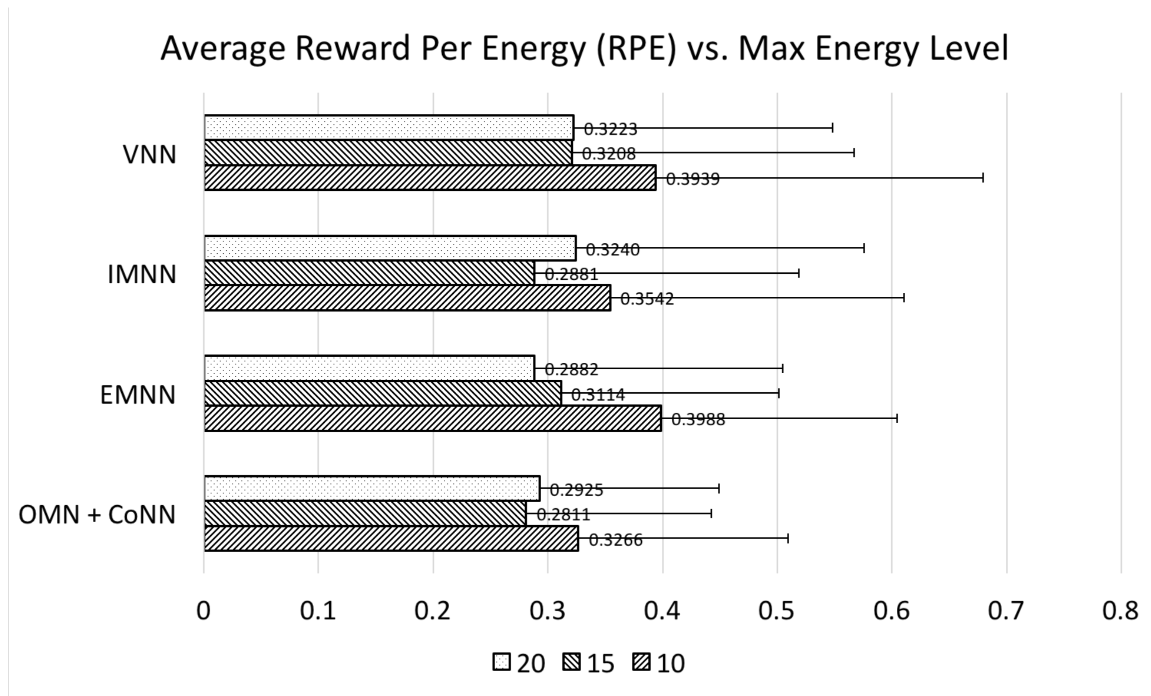


Fig. 5.9.: Results on reward per energy (RPE) for different maximum energy levels. The labeled bars in the chart show the averaged RPE over 10,000 episodes. The error bars show the standard deviations.

Figure 5.8 shows the average episode reward for different maximum energy levels. Apart from VNN, all the networks have reduced episode reward when the maximum energy level is reduced to 10, and all of them have similar but smaller episode reward when the maximum energy level is increased to 20. When the energy capacity is reduced, the collector's ability to evade the opponent is reduced, and hence the episode reward drops. However, when the energy capacity is improved, the extra energy does not help because 15 energy is enough to make most of the strategic movements necessary to avoid the opponent. Furthermore, it even affects the episode reward slightly. This is probably due to the extra steps the collector is now able to take, which may lead it to terminal situations.

As for VNN, it has similar episode reward when the energy capacity is increased, while has larger episode reward when the energy capacity is decreased. This shows that VNN is not able to use energy as efficiently as the networks with opponent modeling. Compared with the 10-to-15 change for the others, it is only when the max energy level is changed from 15 to 20, can VNN make better movements and collect more rewards.

Figure 5.9 shows the average RPE for different maximum energy levels. The results indicate that, when the energy capacity is increased to 20, the RPE of all the networks does not change much. This backs up the previous statement that 15 energy is enough for the collector to make most of the strategic movements, which are the main reason for energy waste. On the other hand, when the energy capacity is reduced to 10, it is not enough to make the strategic movements. Therefore the

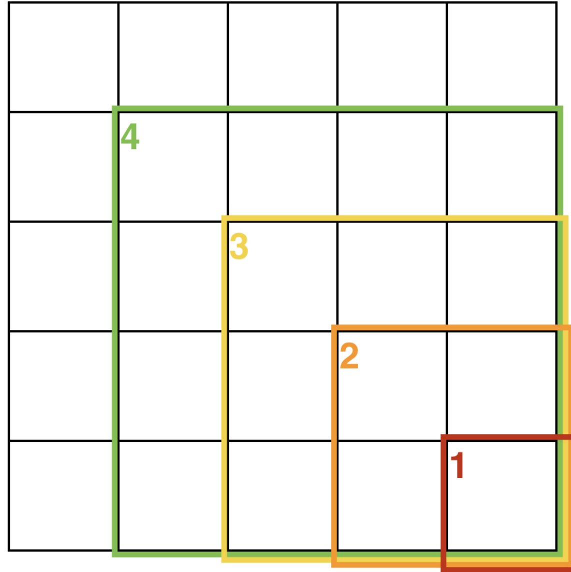


Fig. 5.10.: Reward areas of different sizes. The sizes are defined by the edge length in terms of the number of cells in the square. For a given reward area, one item of reward one is uniformly randomly placed in a cell within the square.

collector frequently gives up on items that are considered too "hard" to collect, and the RPE increases.

5.5.5 Investigation on the behaviors

To investigate more on the behaviors of the networks, we run additional experiments using the networks trained in Section 5.5.3. In these experiments, instead of using a random number of items with 10 total rewards, we place only 1 item with 1 reward in the environment. The location of the item is uniformly randomly chosen in a given square of varying size. This greatly reduces the size of state space. However, because the networks never encounter any state with only 1 item of reward 1 and no history during training, these limited number of states are

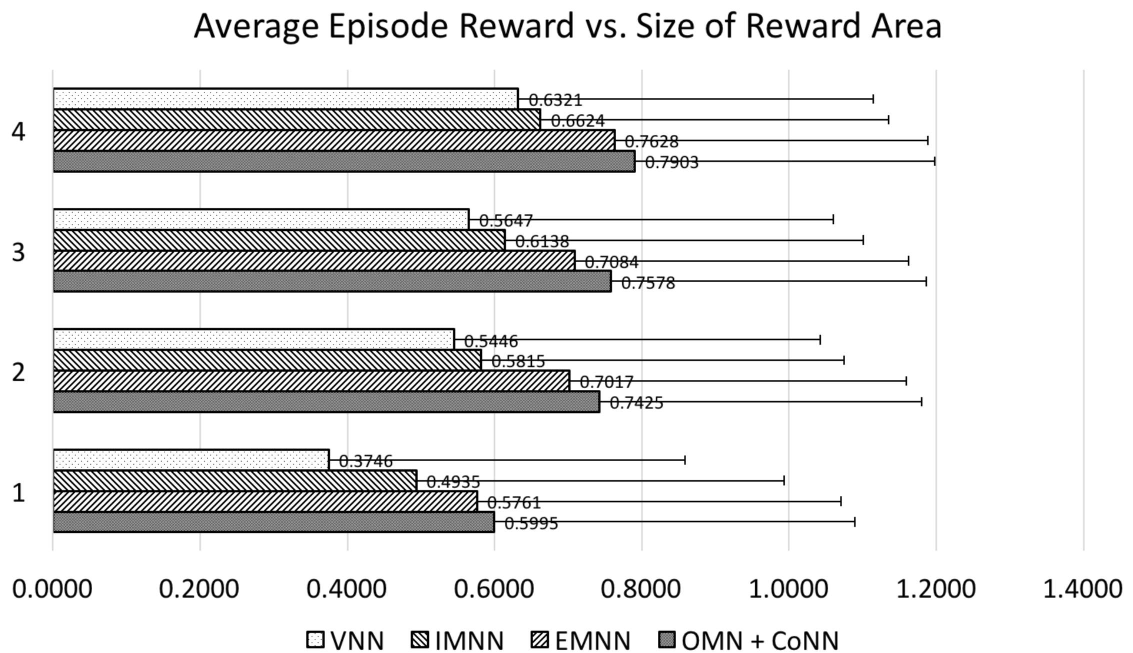


Fig. 5.11.: Results on episode reward for different sizes of reward area. The labeled bars in the chart show the averaged episode rewards over 10,000 episodes. The error bars show the standard deviations. In each episode, there is only 1 item of reward 1, and it is randomly placed in a square to the bottom-right of the grid. The size of reward area indicates the edge length of the square.

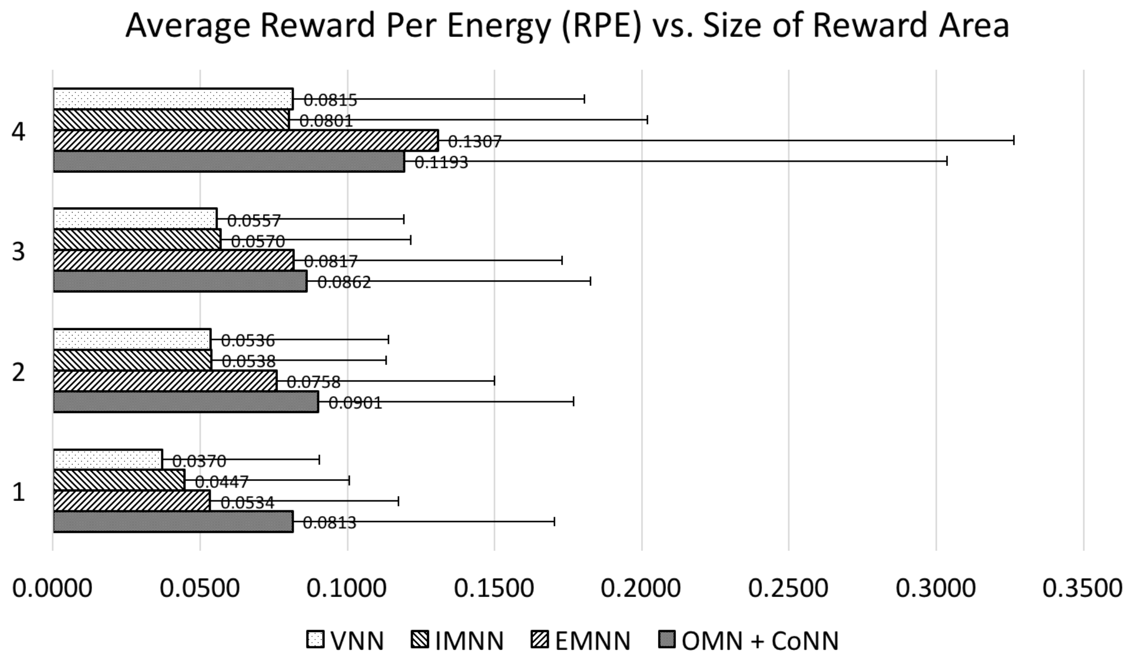


Fig. 5.12.: Results on reward per energy (RPE) for different sizes of reward area. The labeled bars in the chart show the averaged RPE over 10,000 episodes. The error bars show the standard deviations. In each episode, there is only 1 item of reward 1, and it is randomly placed in a square to the bottom-right of the grid. The size of reward area indicates the edge length of the square.

actually from similar but new state spaces. Therefore, these results also show the generalization capabilities of the networks.

See Figure 5.10 for the different squares used to select the location. The square is referred to as the *reward area*, and the edge length of the square is referred to as the *size of reward area* hereafter. We use 1 to 4 as the size of reward area. Size 5 is not used because the opponent strategy **with-fog** treats the cells on the top and left edges differently. The smaller the reward area is, the more difficult it is for the collector to collect the item. Because it is more likely for the collector to be intercepted by the opponent, the collector needs to make precise and effective moves to be able to collect the item. The experimental results on episode reward are shown in Figure 5.11, and those on RPE are shown in Figure 5.12.

Figure 5.11 shows, as expected, as the size of the reward area decreases, the average episode reward for all the networks decreases. However, networks with opponent modeling always perform better than the network without opponent modeling (VNN), especially in the extreme case where the reward area size is 1. This reaffirms that opponent modeling helps improving data-collection performance. Between the opponent modeling networks, OMN + CoNN is able to collect the item most frequently, followed closely by EMNN. IMNN's performance is suboptimal, indicating explicit opponent modeling is more effective than implicit modeling.

As for RPE, Figure 5.12 shows, similarly as for the episode reward, the energy efficiency decreases as the size of the reward area goes down. More importantly, the results also indicate that explicit modeling has better energy efficiency than implicit or no modeling. In addition, as the size of the reward area becomes smaller,

OMN + CoNN becomes dominantly superior than the other networks. This strongly suggests that OMN + CoNN is able to make very precise movements and hence save energy during data collection, probably thanks to its accurate prediction of the opponent action.

The above-mentioned results show the networks' behaviors in a high-level statistically. The conclusion is that opponent modeling, especially explicit modeling using separate network (OMN + CoNN), helps the collector make precise and effective movements. To demonstrate this conclusion more clearly, we present three typical gameplays of VNN and OMN + CoNN, one for each opponent strategy specified in Section 5.5.1 in Figure 5.13, 5.14, and 5.15. In the figures, the robot represents the collector, the ghost represents the opponent, and the money bag represents the item. The base and the energy level of the collector are not shown. The reward of the item is one. The numbers under the grids show the time steps, and the number on the top and left edges of the first grid are the coordinates of the grid cells. The three big dots between grids indicate omitted steps, while those at the end indicates infinite number of steps, i.e., the episode never reaches a terminal state. In these examples, all the games have the same initial state, and the collector uses deterministic policy, i.e., the collector always selects the action with the highest probability.

In Figure 5.13, the opponent uses the **patrol** strategy. The VNN collector keeps trying to find an opening for approaching the item, but it cannot because the opponent is moving back and forth and the collector cannot recognize the pattern. VNN collector is hence stuck and the episode never reaches a terminal state. On the

(a) VNN

(b) OMN + CoNN

Fig. 5.13.: Gameplays of VNN and OMN + CoNN when the opponent uses the **patrol** strategy. The robot represents the collector, the ghost represents the opponent, and the money bag represents the item. The base and the energy level of the collector are not shown. The reward of the item is one. The numbers under the grids show the time steps, and the number on the top and left edges of the first grid are the coordinates of the grid cells. The three big dots between grids indicate omitted steps, while those at the end indicates infinite number of steps.

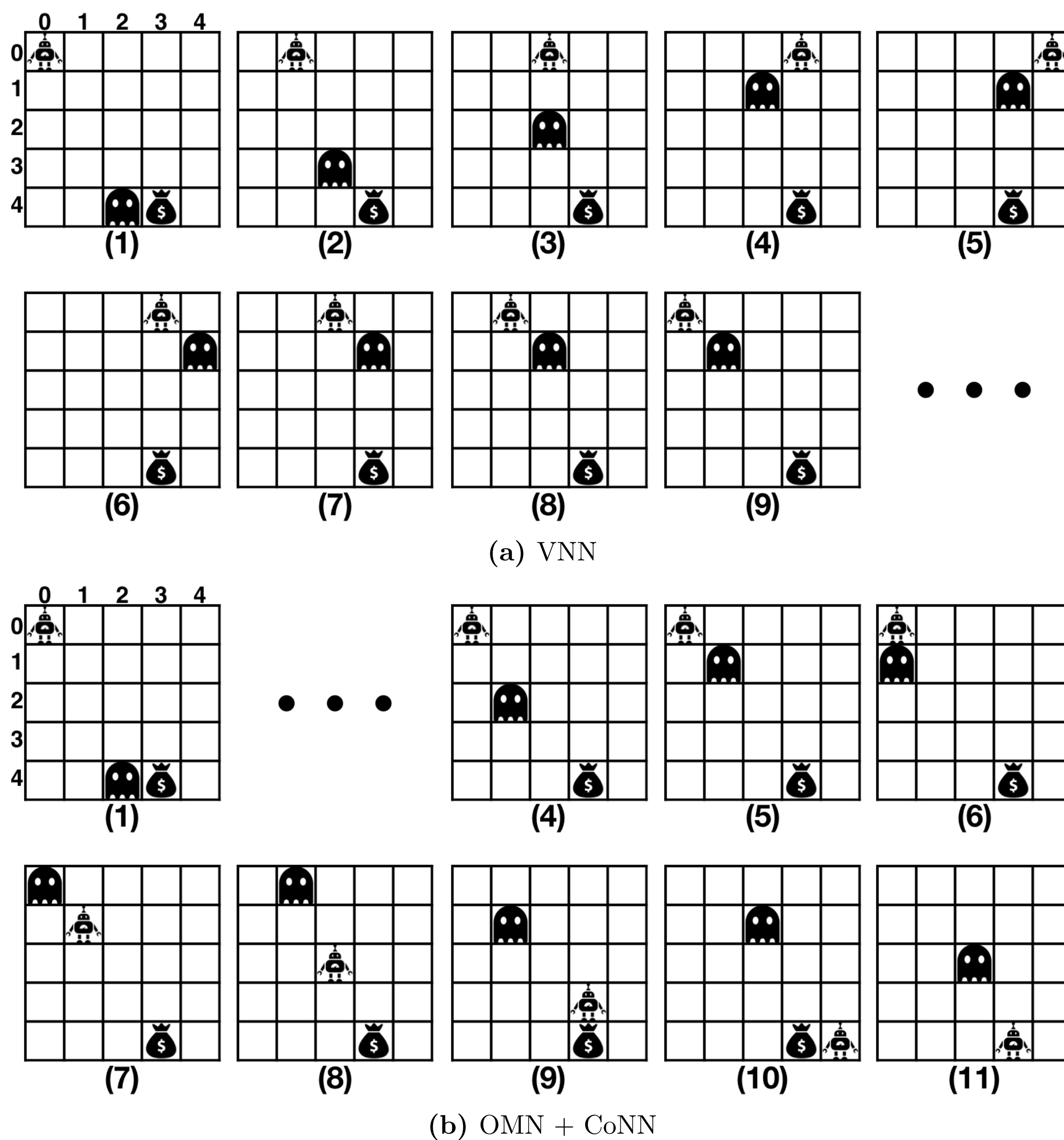
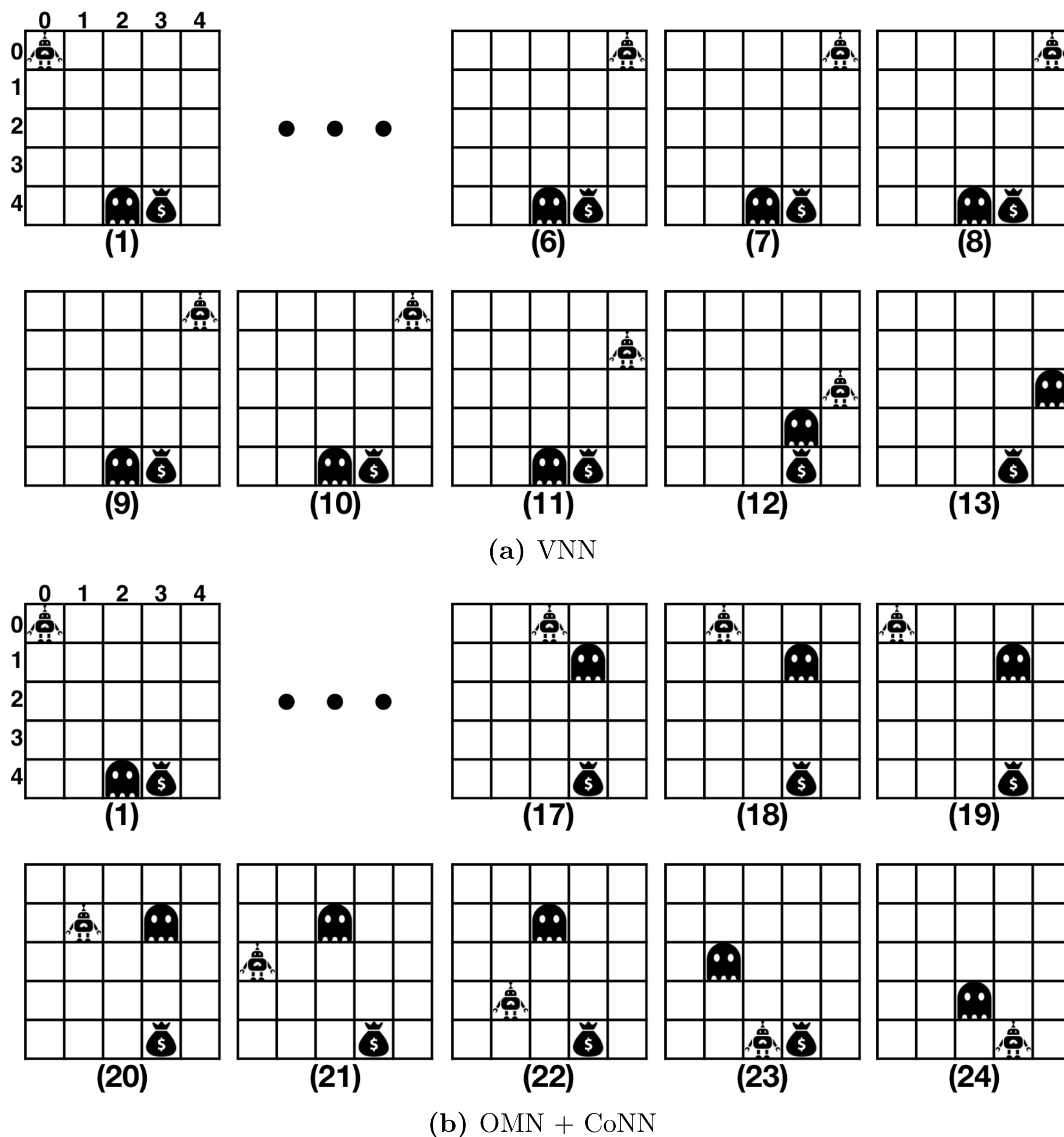


Fig. 5.14.: Gameplays of VNN and OMN + CoNN when the opponent uses the **restricted** strategy. The robot represents the collector, the ghost represents the opponent, and the money bag represents the item. The base and the energy level of the collector are not shown. The reward of the item is one. The numbers under the grids show the time steps, and the number on the top and left edges of the first grid are the coordinates of the grid cells. The three big dots between grids indicate omitted steps, while those at the end indicates infinite number of steps.



(a) VNN

(b) OMN + CoNN

Fig. 5.15.: Gameplays of VNN and OMN + CoNN when the opponent uses the **with-fog** strategy. The robot represents the collector, the ghost represents the opponent, and the money bag represents the item. The base and the energy level of the collector are not shown. The reward of the item is one. The numbers under the grids show the time steps, and the number on the top and left edges of the first grid are the coordinates of the grid cells. The three big dots between grids indicate omitted steps.

other hand, the OMN + CoNN collector is able to find the right opening, at step 9, and cross the opponent's patrol route, successfully collecting the item.

In Figure 5.14, the opponent uses the **restricted** strategy. The VNN collector moves back and forth because the opponent keeps chasing it and there is no space for the collect to go pass the opponent. As a result, the collector is once again stuck and the episode never reaches a terminal state. On the contrary, the OMN + CoNN collector is able to recognize the movement pattern of the opponent, and to exploit that pattern by moving diagonally. Thus, it is able to quickly obtain the item. However, the actions of the OMN + CoNN collector is not perfect, as shown in step 10 of Figure 5.14b.

In Figure 5.15, the opponent uses the **with-fog** strategy. In this case, the VNN collector gives up after wasting some energy, because the opponent stays beside the item. However, the OMN + CoNN collector is able to lure the opponent to one side of the environment, then go to the other side within the "fog", and finally collect the item.

These examples are deliberately selected to show, more concretely and in more details, the advantages of opponent modeling. In these cases, OMN + CoNN has much better performance than VNN, because VNN is not able to collect the item at all. It is worth noting that, in other cases, VNN does sometime outperform OMN + CoNN in terms of energy efficiency, i.e., VNN uses less steps than OMN + CoNN, which probably thanks to its simpler network structure. However, we have not observed a case where VNN is able to collect the item but OMN + CoNN is not.

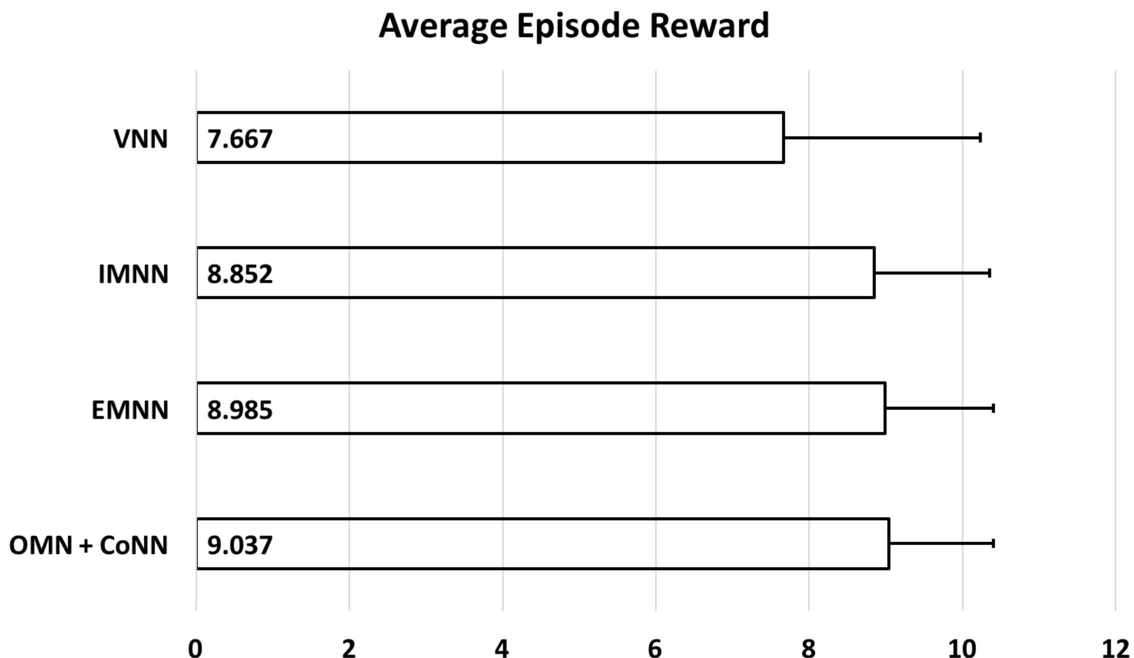


Fig. 5.16.: Results on episode reward for networks trained in 5 million steps. The labeled bars in the chart show the averaged episode rewards over 10,000 episodes. The error bars are the standard deviations.

5.5.6 Five million steps of training

To understand the full potential of all the networks, we train additional networks for 5 million steps, i.e., $T_{\max} = 5{,}000{,}000$, and run experiments in 10,000 randomly generated environments. This number of steps is chosen because all the networks completely converge after training, i.e., there is no more performance improvement after 5 million steps for any network. The results on episode reward are shown in Figure 5.16. The results on RPE are shown in Figure 5.17.

As shown in Figure 5.16, the episode rewards of all the networks are improved compared with those in Figure 5.6. IMNN, EMNN and OMN + CoNN have similar episode rewards, which are more than 15% improvement over VNN and have smaller

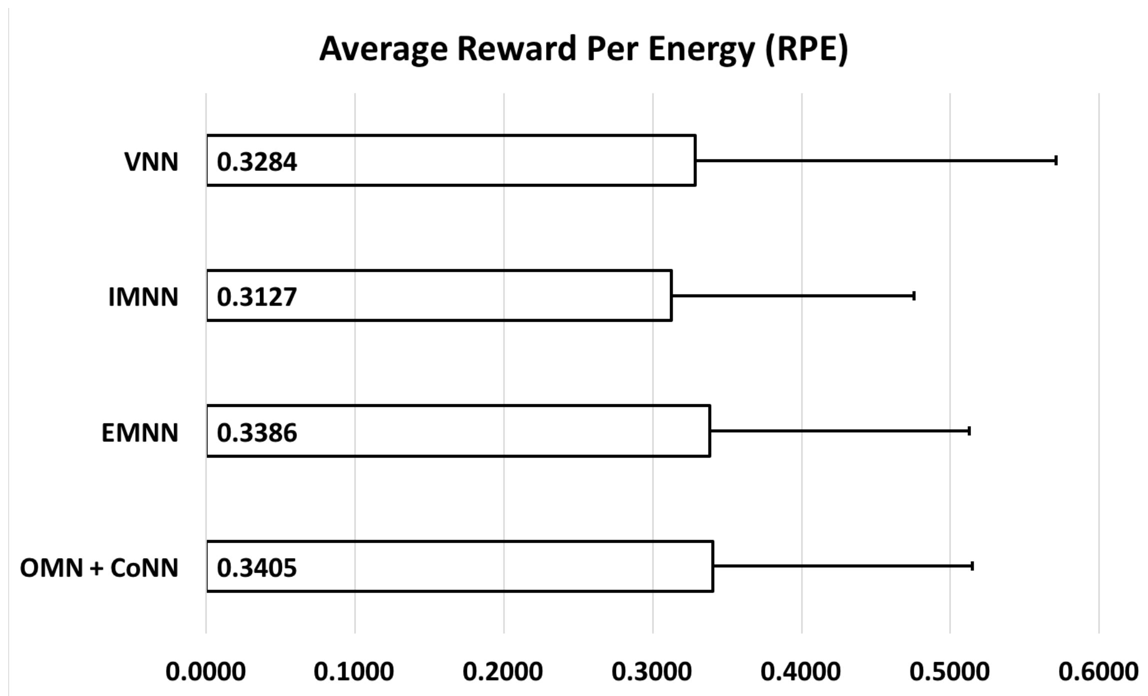


Fig. 5.17.: Results on reward per energy (RPE) for networks trained in 5 million steps. The labeled bars in the chart show the averaged RPEs over 10,000 episodes. The error bars are the standard deviations.

standard deviations. OMN + CoNN is the best and achieving over 9 out of 10 rewards and has the smallest standard deviation. These results further confirm that opponent modeling, either implicit or explicit, have a significant positive impact on performance.

Comparing the episode rewards from Figure 5.6 and Figure 5.16, we also see that EMNN and OMN + CoNN perform better than IMNN when trained in 1 million steps but they all perform similarly when trained in 5 million steps, this indicates that networks with explicit opponent modeling converge faster than that with implicit opponent modeling. In addition, the performance difference between EMNN and OMN + CoNN is also smaller when trained in 5 million steps, indicating the separate OMN with high accuracy provides the fastest convergence.

Figure 5.17 shows that, compared with those in Figure 5.7, the RPEs of all the networks are improved after 5 million steps of training, which means the agents learn to use energy more efficiently during training. OMN + CoNN achieves the best RPE, indicating that OMN's high accuracy in predicting the opponent action makes the collector collect rewards more efficiently. This is probably because the actions chosen by CoNN are more precise and thus the collector wastes less energy. VNN has higher RPE than IMNN, because it collects less reward and its reward density is generally higher than that of IMNN. This conclusion is backed up by high standard deviation of VNN.

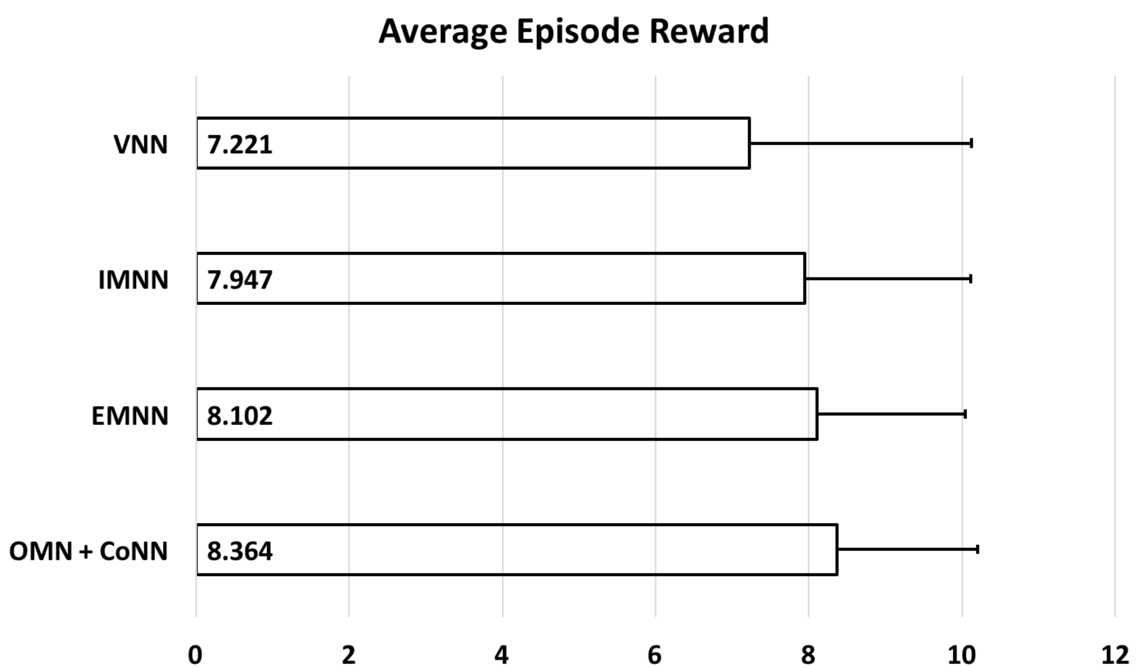


Fig. 5.18.: Results on episode reward for networks trained in 1 million steps. The labeled bars in the chart show the averaged episode rewards over 10,000 episodes. The error bars are the standard deviations. The **with-fog** strategy is replaced by the **double-blind with-fog** strategy where agents cannot see each other when the collector goes into special “foggy” cells.

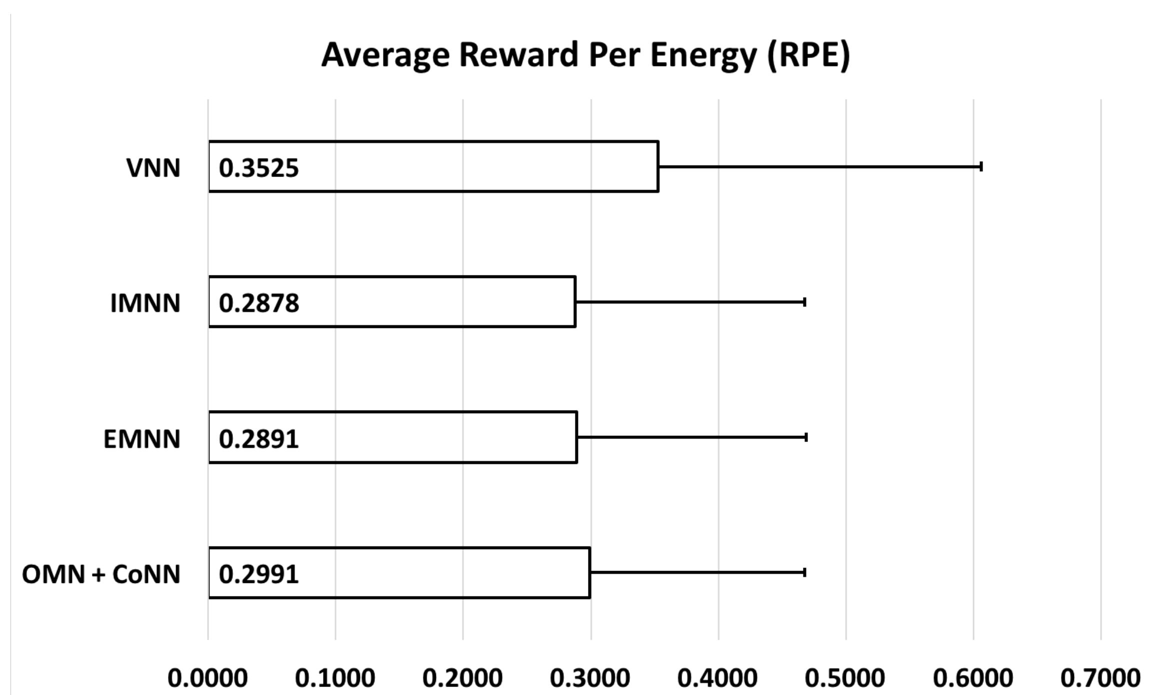


Fig. 5.19.: Results on reward per energy (RPE) for networks trained in 1 million steps. The labeled bars in the chart show the averaged RPEs over 10,000 episodes. The error bars are the standard deviations. The **with-fog** strategy is replaced by the **double-blind with-fog** strategy where agents cannot see each other when the collector goes into special "foggy" cells.

5.5.7 Change of opponent strategy

In this section, we replace the **with-fog** policy of the opponent with a more symmetric **double-blind with-fog** policy where both agents cannot see each other when the collector goes into the special "foggy" cells. The other policies stay the same. All the networks are retrained for 1 million steps and the results are presented in Figure 5.18 and 5.19.

From Figure 5.18, we can see that the performance of EMNN and OMN + CoNN stay almost the same with the change of opponent policy, but VNN and IMNN perform better, compared to the results in Figure 5.6. Gameplays of all the agents show that the behaviors of all the opponent-modeling networks do not change much with the different opponent strategy set. This is expected because these networks are able to see the history of the states. Even though they cannot see the opponent in the current state, they can learn to deduct this information from the history during training. On the other hand, this change does affect the VNN since it does not have access to the history, and the gameplays show that VNN agent simply stays in the base when the opponent chooses the new **double-blind with-fog** strategy.

The improvements of VNN and IMNN are likely due to more focused training. Without the explicit goal of predicting the opponent behavior, VNN and IMNN learn to associate rewards to the actions taken in the current state. Because the current state will very often not contain any information about the opponent when **double-blind with-fog** is chosen, there is not as much to learn in such situations. As a result, the optimization of VNN and IMNN during training is focused more on

the other opponent strategies, and the convergence for those strategies is thus faster. Since VNN and IMNN do not do well for the original **with-fog** strategy anyway, the overall performance of them are improved.

The RPE results in Figure 5.19 is again not indicative and is only shown for completeness, because the average distance between items increases as items are collected and the RPE will decrease as a result. But they are consistent with the results in Figure 5.7.

5.6 Related work

Most opponent modeling works are done in the domain of Poker games. Ganzfried and Sandholm [12] combine game theoretic reasoning and pure opponent modeling to a hybrid method that can exploit suboptimal opponents in Limit Texas Hold'em. Their method first computes an approximate equilibrium of the game. Then, as it plays by following the equilibrium strategy, it uses a Bayesian model to record the opponent's deviation from the equilibrium. After a number of games, it starts to exploit the opponent using a best response strategy based on the opponent model. In [32] and [3], different strategies are computed offline based on domain knowledge or past experience in Texas Hold'em. These "experts" are then selected online using multi-arm bandit algorithm, which models the opponents implicitly. The above-mentioned methods require either domain knowledge or special property of the game being played. As a consequence, their applicability is limited to specific domains.

Recently, He et. al. [13] apply Deep Reinforcement Learning to opponent modeling. In their work, two types of networks are designed. The first one simply concatenates the network parameters learned from the opponent behaviors to those learned from game states. The second one uses an expert network to capture different aspects of the game state, and uses a gating network to learn how to interpret the expert network's output according to the opponent behavior. Their networks are trained using Deep Q Learning [23, 24] with either implicit or explicit opponent modeling (with additional signal from the opponent actions). The methods are verified in a simulated soccer game and the quiz bowl trivia game. Similar to our methods, their approaches are neural based and do not require domain knowledge, so they can be adopted to other domains.

5.7 Conclusion

In this chapter, *Data-collection Game* (DCG) is proposed. It is a two-player Stochastic Game that models a robotic agent collecting digital data, under energy constraint, in an environment that contains an opponent agent. The opponent agent may employ different strategies. Therefore being able to identify the opponent strategy can potentially benefit the data-collecting agent (collector).

Four deep neural networks are designed to learn the game. The *Vanilla Navigation Network* (VNN) learns to play based only on the current state. The *Implicit Modeling Navigation Network* (IMNN) learns to play based on the current state and the location history of the agents. The *Explicit Modeling Navigation*

Network (EMNN) learns to play as well as to predict the opponent action, based on the current state and the location history of the agents. The *Opponent Modeling Network* (OMN) paired with the *Cooperating Navigation Network* (CoNN) have the same goals as EMNN but use a separate network, the OMN, to focus on modeling the opponent.

All the networks are trained using deep reinforcement learning methods. From the experiments, we learn that opponent modeling, either implicit or explicit, greatly improves the data-collecting performance of the collector. However, explicit modeling speeds up convergence during training, especially when the opponent model is accurate. In addition, an accurate opponent model also helps the collector collect data more efficiently, achieving high *reward per energy* (RPE).

6. SUMMARY AND FUTURE WORK

In this thesis, we first formulate the task of data collection as a planning problem on a complete graph, named the Data-collecting Robot Problem (DRP), where the risk is modeled as a probability of robot being disabled or destroyed. The objective is to maximize the expected reward that is determined by both the gain from visiting nodes and loss of robots. This formulation assumes a uniform risk distribution and unlimited energy of robot, among others. Heuristic planning algorithms that consist of a clustering step and a tour-building step are proposed to solve the problem.

Secondly, we relax the assumption to non-uniform probabilistic risk and limited energy, and formulate the task as a Markov Decision Process called the Data-collection Problem (DCP). The goal is to collect maximal reward with given robot. The key of good solution is to find good balance between safety and energy (S&E). We propose four navigation algorithms that have different priorities in S&E during data collection, and design an Ensemble Navigation Network (ENN) for automatically finding improved solutions from heuristic inputs. ENN is trained using reinforcement learning and is empirically proved to provide better performance.

Finally, we formulate the data-collection task as a Stochastic Game named the Data-collection Game (DCG), where the risk is in the form of an opponent instead of a probability distribution, and the robot still has energy constraints. The goal is

still to collect maximal reward with given robot. However, since the opponent can take different strategies to interfere with data collection, opponent modeling is utilized to improve performance. Three deep neural networks are designed to model the opponent in different ways, which are trained using reinforcement learning. We show good opponent modeling gives superior data-collection results in experiments.

As future directions, more constraints, such as time and number of robots, can be added to make DRP more realistic and general. For DCP, heuristics with more elaborate handcrafted rules can be designed to improve ENN, and it would be interesting to generalize the problem to multiagent and/or multiple bases. With respect to DCG, we are interested in generalizing the game to one with multiple data-collecting agents, so that the agents can cooperate to collect data more efficiently under the influence of the opponent. In addition, a combination of the risk models in DCP and DCG can produce fascinating problems where the risk exists as moving distributions, which may require a hierarchical structure for reinforcement learning. Last but not least, currently the energy level in DCP and DCG is a simple part of state representation, and therefore changing the maximum energy level often means a fresh network needs to be trained from scratch. In other words, the networks does not really understand what energy level means. Therefore, it is an intriguing task to investigate on better network generalization with respect to energy level, where a trained neural network can better understand what the value of energy level means and works well for all possible maximum energy levels.

LIST OF REFERENCES

LIST OF REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] C. Archetti, D. Feillet, A. Hertz, and M. Grazia Speranza. The capacitated team orienteering and profitable tour problems. *Journal of the Operational Research Society*, 60:831–842, 2009.

[3] N. Bard, M. Johanson, N. Burch, and M. Bowling. Online implicit agent modelling. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 255–262. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[4] A. Blum, P. Chalasani, D. Coppersmith, B. Pulleyblank, P. Raghavan, and M. Sudan. The minimum latency problem. *Proc. of the 26th Symp. Theory of Computing, STOC*, page 9, 1994.

[5] A. Blum, S. Chawla, D. R. Karger, T. Lane, A. Meyerson, and M. Minkoff. Approximation algorithms for orienteering and discounted-reward tsp. *SIAM Journal on Computing*, 37(2):653–670, 2007.

[6] T. Brázdil, A. Kučera, and P. Novotnỳ. Optimizing the expected mean payoff in energy markov decision processes. In *International Symposium on Automated Technology for Verification and Analysis*, pages 32–49. Springer, 2016.

[7] K. Chaudhuri, B. Godfrey, S. Rao, and K. Talwar. Paths, trees, and minimum latency tours. In *Proceedings - 44th Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 36–45. IEEE, 2003.

[8] H. L. Choi, L. Brunet, and J. P. How. Consensus-based decentralized auctions for robust task allocation. *IEEE Transactions on Robotics*, 25(4):912–926, 2009.

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[10] A. Ekici and A. Retharekar. Multiple agents maximum collection problem with time dependent rewards. *Computers and Industrial Engineering*, 64(4): 1009–1018, 2013.

[11] D. Feillet, P. Dejax, and M. Gendreau. Traveling salesman problems with profits: An overview. *Transportation Science*, 39:188–205, 2001.

[12] S. Ganzfried and T. Sandholm. Game theory-based opponent modeling in large imperfect-information games. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 533–540.

International Foundation for Autonomous Agents and Multiagent Systems, 2011.

[13] H. He, J. Boyd-Graber, K. Kwok, and H. Daumé III. Opponent modeling in deep reinforcement learning. In *International Conference on Machine Learning*, pages 1804–1813, 2016.

[14] J. Hudack and J. Oh. Multi-agent sensor data collection with attrition risk. In *Proceedings - The 26th International Conference on Automated Planning and Scheduling, ICAPS*, 2016.

[15] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[17] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1): 48–50, 1956.

[18] A. Kučera. Playing games with counter automata. *Reachability Problems*, pages 29–41, 2012.

[19] T. Lane and L. P. Kaelbling. Approaches to macro decompositions of large markov decision process planning problems. In *Intelligent Systems and Advanced Manufacturing*, pages 104–113. International Society for Optics and Photonics, 2002.

[20] T. Lane and L. P. Kaelbling. Nearly deterministic abstractions of markov decision processes. In *AAAI/IAAI*, pages 260–266, 2002.

[21] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521(7553): 436–444, 2015.

[22] D. Lozovanu and A. Zelikovsky. Minimal and bounded tree problems. *Tezele Congresului XVIII al Academiei Romano-Americane*, pages 25–26, 1993.

[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[24] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[25] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, 2016.

[26] M. Moshref-Javadi and S. Lee. A taxonomy to the class of minimum latency problems. In *Proceedings - IIE Annual Conference.*, page 3896. Institute of Industrial Engineers-Publisher, 2013.

[27] R. C. Prim. Shortest connection networks and some generalizations. *Bell Labs Technical Journal*, 36(6):1389–1401, 1957.

[28] M. L. Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[29] R. Ravi, R. Sundaram, M. V. Marathe, D. J. Rosenkrantz, and S. S. Ravi. Spanning treesshort or small. *SIAM Journal on Discrete Mathematics*, 9(2): 178–200, 1996.

[30] L. Rokach and O. Maimon. Clustering methods. *Data mining and knowledge discovery handbook*, pages 321–352, 2005.

[31] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. An analysis of several heuristics for the traveling salesman problem. In *Fundamental Problems in Computing: Essays in Honor of Professor Daniel J. Rosenkrantz*, pages 45–69. Springer Science & Business Media, 2009.

[32] J. Rubin and I. Watson. On combining decisions from multiple expert imitators for performance. *IJCAI-11*, 2011.

[33] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[34] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[35] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[36] L. Talarico, K. Sörensen, and J. Springael. Metaheuristics for the risk-constrained cash-in-transit vehicle routing problem. *European Journal of Operational Research*, 244(2):457–470, 2015.

[37] P. Toth and D. Vigo. *Vehicle routing: problems, methods, and applications*, volume 18. Society for Industrial and Applied Mathematics, 2014.

[38] G. Urban, K. J. Geras, S. E. Kahou, O. Aslan, S. Wang, R. Caruana, A. Mohamed, M. Philipose, and M. Richardson. Do deep convolutional nets really need to be deep and convolutional? *arXiv preprint arXiv:1603.05691*, 2016.

[39] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[40] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[41] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

[42] Z. Xing and J. C. Oh. Heuristics on the data-collecting robot problem with immediate rewards. In *International Conference on Principles and Practice of Multi-Agent Systems*, pages 131–148. Springer, 2016.

VITA

VITA

Zhi Xing was born in Tianjin, China. He received his Bachelor of Science degree in Bioscience at Nankai University (Tianjin, China) in June 2010. He received his Master of Science degree in Computer Science at Syracuse University (Syracuse, New York, USA) in June 2017. He received his Doctor of Philosophy degree in Computer and Information Science and Engineering from Syracuse University (Syracuse, New York, USA) in August 2017.