Robust shortest path with local information revelation

Edwin Meriaux

Center for Intelligent Machines
Electrical and Computer Engineering
McGill University
Montreal, Canada

September 2024

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Science.

Abstract

In this thesis, a robust path planning method is developed to find optimal path traversal policies in an uncertain environment. This method restricts itself to uncertainty in discrete sets rather than distributions for the uncertain costs. Additionally, the uncertainty of each edge does not remain independent, as a set of possible true worlds is known. This formulation allows for the creation of feasible sets of possible true worlds. As information is revealed about the true state of the world, the elements inside the feasible set are reduced until only one possible true world remains. This state and information revelation is similar to a Partially Observable Markov Decision Process (POMDP) structure.

The solution developed in this thesis shows that Value Iteration (VI) in infinite horizons can have guarantees to converge to the optimal solution. The complication of this method is that as the number of possible states grows in relation to the size of the environment and the uncertainty in the world, the computational requirements grow exponentially. To allow for a Robust Shortest Path (RSP) to be found in these cases, we show how a Monte Carlo Tree Search (MCTS) method can be used. This method builds on previous examples in the literature, which use MCTS in games such as Chess and Go. Furthermore, in larger worlds, certain nodes and edges can be pruned if they cannot be used along an optimal path. This pruning can reduce the computational requirements in both the VI algorithm and the MCTS method.

Résumé

Dans cette thèse, une méthode robuste de planification des chemins est développée pour trouver des politiques optimales de traversée des chemins dans un environnement incertain. Cette méthode se limite à l'incertitude dans des ensembles discrets plutôt qu'à des distributions pour les coûts incertains. De plus, l'incertitude de chaque arête ne reste pas indépendante, car un ensemble de mondes réels possibles est connu. Cette formulation permet de créer des ensembles réalisables de mondes réels possibles. Au fur et à mesure que des informations sont révélées sur l'état réel du monde, les éléments à l'intérieur de l'ensemble réalisable sont réduits jusqu'à ce qu'il ne reste plus qu'un seul monde réel possible. Cette révélation de l'état et des informations est similaire à la structure d'un Partially Observable Markov Decision Process (POMDP).

La solution développée dans cette thèse montre qu'une Value Iteration (VI) dans des horizons infinis peut garantir la convergence vers la solution optimale. La complication de cette méthode est que lorsque le nombre d'états possibles augmente en fonction de la taille de l'environnement et de l'incertitude du monde, les exigences de calcul augmentent de façon exponentielle. Pour permettre de trouver un Chemin le Plus Court Robuste (RSP) dans ces cas, nous montrons comment une méthode de Monte Carlo Tree Search (MCTS) peut être utilisée. Cette méthode s'appuie sur des exemples antérieurs dans la littérature qui utilisent MCTS dans des jeux tels que les échecs et le go. De plus, dans des mondes plus vastes, certains nœuds et arêtes peuvent être élagués s'ils ne peuvent pas être utilisés le long d'un chemin optimal. Cet élagage peut réduire les exigences de calcul à la fois dans l'algorithme de VI et dans la méthode MCTS.

Acknowledgments

First and foremost, I would like to thank my thesis supervisor, Professor Aditya Mahajan. Over the past two years, he has guided and taught me both in the classroom and in my research work. He has shown me how to rigorously pursue and develop quality research.

During my studies, I worked extensively with Professor Silviu-Iulian Niculescu. He invited me to France, where he supervised and contributed to my work during my stay at the University of Paris-Saclay, CentraleSupélec, L2S Laboratory.

This thesis was reviewed by Professor Inna Sharf of McGill. I would like to thank her for her review and feedback regarding this work.

I am deeply grateful to McGill University and the ECE department, which have supported my research through the Graduate Excellence Fellowship. Additionally, I have received generous funding from the NSERC Alliance Mission Grant, the MITACS Globalink Award, and the McGill Graduate Mobility Award. The experimental results presented in this work were also partially facilitated by Compute Canada, which was incredibly helpful.

I would also like to thank my fellow students in the Systems and Control Lab, who have contributed to my work through instructive presentations and discussions. Special thanks go to Berk Bozkurt, Amit Sinha, Borna Sayedana, Raihan Seraj, and Reihaneh Ghoroghchian.

Reaching the point where I could attend McGill and begin my master's degree was made possible by many individuals, particularly the professors who introduced me to research and nurtured my interest during my undergraduate studies. I would like to thank Professor Jay Weitzen, Professor Kshitij Jerath, Professor Thanuka Wickramarathne, and Professor Jean-François Millithaler. During that time, Alok Malik also played a significant role in my development through our research collaborations.

Special thanks to my incredible group of friends who were always there in my hours of need: Daniel Abreu Fernandez, Renin Jose, Connor O'Rourke, Delbert Edric, Al W. Fox, Derek Houle, Aparna Fitch, Elizabeth Spy, Mathew Mayger, Asa Losurdo, and Mohamed Martini.

Finally, I would like to thank my family, who have encouraged me at every point in my life.

Contents

List of Figures

List of Tables

List of Acronyms

MDP Markov Decision Process

POMDP Partially Observable MDP

MCTS Monte Carlo Tree Search

SP Shortest Path

RSP Robust Shortest Path

SSP Stochastic Shortest Path

DSP Deterministic Shortest Path

DRSP Distributionally Robust Shortest Path

DP Dynamic Programming

VI Value Iteration

Spiral STC Spiral Spanning Tree Coverage

AI Artificial Intelligence

DNN Deep Neural Network

RL Reinforcement Learning

UCB Upper Confidence Bounds

UCT Upper Confidence applied to Trees

UCT Predictor + UCT

NPC Non-Player Character

SL Supervised Learning

RL Reinforcement Learning

RHS Right Hand Side

ZSG Zero-Sum Game

Chapter 1

Introduction

1.1 Motivation

To move from a start to an end state, a series of steps, actions, or transitions through intermediary states is required. This process forms a path from the initial state to the end state. Path planning is the process of selecting an optimal path. The optimality of this can be computed using various methods, with the most common being the selection of the path with the least cost. The cost of a path is influenced by the environment and other factors, such as the distance, time, or energy needed to traverse the path.

Path planning is important because when an entity moves from one location to another, it is typically desirable to minimize the cost. For example, many different paths exist between any two cities, but some are more costly than others. Time and resources can be saved by selecting a more efficient path.

Path planning is traditionally solved using algorithms such as Dijkstra's Shortest Path [2] or A* search [3]. These methods represent the world as a graph, where each node corresponds to a specific location and the edges represent transitions between adjacent locations. Costs are embedded in the model by assigning weights to the edges. Dijkstra's [2] and A* [3] algorithms find the series of edges between the start and end nodes that result in the lowest total cost. These methods assume that the world and the cost of moving through it are fully known, meaning path planning is conducted with perfect information. However, if this assumption does not hold, uncertainty exists in the world, and the algorithm must plan with imperfect information. In such cases, Dijkstra's and A* algorithms cannot guarantee the shortest path.

With the assumption of imperfect information, an outlook on the world is required. In such cases, a pessimistic or optimistic outlook can be used when choosing the path. A pessimistic outlook assumes the worst possible state of the world, while an optimistic one assumes the best-case scenario. The advantage of using a pessimistic outlook is that it establishes an upper bound on the cost of traversing the environment. In time-critical operations, this approach can be essential for selecting the path that will reach the destination the fastest, assuming the worst-case scenario. This upper-bound-focused method forms the Robust Shortest Path (RSP) [4], a robust generalization of the Stochastic Shortest Path (SSP) [5, 6, 7, 8].

When dealing with uncertain environments, there are two types of decision-making models used for path planning. One set determines the final optimal path given the uncertainty before the first action is taken. These solutions are often referred to as *offline path planning* algorithms [9]. The other method generates a general solution before the first action and updates it as observations of the environment are made. These are generally referred to as *hybrid path planning* [10], which will be the focus of the method proposed in this paper. A third type, called *online path planning* [11], requires no prior knowledge of the world before making the first observation. However, this approach is generally outside the scope of RSP.

Solving RSP problems can be done in an offline manner by developing a policy to find the optimal path. Whether uncertainty exists or not, this has traditionally been addressed using Dynamic Programming (DP) algorithms, such as Value Iteration (VI) or Policy Iteration (PI)[12, 4]. Under certain assumptions, these algorithms converge to the optimal policy, even in uncertain environments. These methods are generally used in three types of cases. The first is entirely offline, where the policy is determined before the path is taken[13, 14]. The second is a hybrid online/offline method, where part of the path is predetermined, but as more data is collected, the path is updated in real-time [10]. The third is entirely online, where the path is determined on the move, with no prior knowledge [15]. The terms "online" and "offline" refer specifically to path planning literature, not general AI online or offline learning.

In this work, we present an online/offline infinite horizon VI algorithm that computes the value of each node relative to the cost of reaching the terminal node, given the different possible states of the environment. Using these values, the optimal policy for the agent at each node/state combination is to move to the next node with the lowest cost, given the

possible states. As information about the environment is revealed, certain possible states are ruled out, altering the best-worst cost for each node. The policy then adjusts to take the next best action. Due to the time cost of this approach, we will demonstrate how to simplify the offline computation using Monte Carlo Tree Search (MCTS). Additionally, we will present a pruning method to reduce computational costs by limiting the number of possible graphs in a given environment and the number of nodes and edges within those graphs.

The remainder of this work is structured as follows: The rest of this chapter will cover motivating examples (Section 1.2) and a literature review on the path planning problem (Section 1.3). Chapter 2 presents the theoretical and algorithmic contributions of this work, along with a small-scale example. In this chapter, we introduce both the Robust Shortest Path algorithm and the pruning method. Chapter 3 discusses how MCTS can be used to generate approximate Robust Shortest Path solutions. Chapter 4 provides additional numerical examples in larger and more complex scenarios.

1.2 Claims of Originality

1. Online/offline infinite horizon VI algorithm to solve Robust Shortest Path problems.
2. Pruning method to reduce environments utilizing the minimax structure of Robust Shortest Paths to eliminate dominated edges, nodes, and graphs as permitted.

1.3 Motivating Examples

Let's consider an example for path traversal. A person is moving through a building. This is a known environment to them where they know all the possible passages. In this case, a person is moving through a building to retrieve an object at a location which is known. The person wants to minimize the time cost of the path to the object. This cost is a function of the time needed to find the object. Given the person has perfect information about the environment they know the cost of traversing each passage. The simplest way to compute this cost would then be the length of the path divided by the average speed of the person. Given the multiple possible paths and the cost associated with each one the person optimizes their time by taking the path which has the least costly traversal.

This cost analysis for each possible path was not very realistic. It assumed that every detail of the passages for the path was already known and nothing was uncertain. There were no objects out of place which could slow down the mission and there were no crowds of people possibly blocking the way. Effectively the building was static and entirely known. Now let's imagine there is uncertainty in the building. This uncertainty can come in two forms as aleatoric or epistemic uncertainty. The first is in the true state of the passages (aleatoric uncertainty) and the second is uncertainty in the actions the person might take (epistemic uncertainty). Let's return to the example with the person in the building. The uncertainty about the state of the hallways, doors, and location of obstacles along the path falls under the concept of aleatoric uncertainty. There exists multiple possible states of the building and since the person does not know the true state, the different state must be taken into account. Epistemic uncertainty could come in the form of uncertain action decision. This uncertainty does not fit in this type of problem since the agent will just follow the path. At every position along the path, there is one action to take to get to the next position. A different action would constitute a different path. The agent is not deciding on the action and each action results in a deterministic outcome.

With this aleatoric uncertainty, the path traversal problem becomes more realistic. There could be some objects blocking the way, doors could be locked or unlocked and possible slowdowns in hallways because of crowds. The cost analysis for each path now becomes much more complex due to the uncertainty. This makes the optimal path for path traversal more complex to find, especially prior to the start of the path. Given the time of day, or any other information of the building's uncertainty, the person might take a path they expect to be the quickest.

Let's now modify the example. Given the possible uncertainty in the building state, imagine there exists different parallel worlds with all the possible combinations of the uncertain factors. Now imagine the person is pessimistic. They do not know the true state of the uncertain factors in building, but they assume the worst-case. No matter which path the person takes, the building state is in the worst possible state. This makes the traversal or coverage as cost as much time as possible. If the person then wants to take the optimal path given this worst-state case, it is said they are taking the robustly optimal path. In this scenario, the person has multiple possible paths they could have followed. Each possible path contains a certain time cost to it until the person gets to the end. Depending on the state of the world, the optimal path is different. The robust optimization solution to this

would develop a policy across the different possible paths to determine the best one across all worst-case worlds. This would allow the person to know the maximum amount of time they would need for the path regardless of the situation.

In the aforementioned examples, the person traversing or covering the building was initially making decisions before executing the path. This type of path planning is considered offline path planning. This is generally done using all the knowledge that is known about the building's passages. In this case, that would mean the person already knows the map of the environment. This does not necessarily mean the true state, but can include the uncertainty as well. If the person moving around did not know the map of the building and was discovering all the state information only once moving through the passages, the agent would be operating in online path planning (again not to be mixed with online AI learning). There exists a third case called hybrid path planning which makes a general plan for the traversal prior to the mission start, but uses data collected while traversing the passages to update the path. This is what the person was doing in the previous example when the uncertainty was introduced.

1.4 Literature Review

The work presented in this thesis is centered around robust optimization in path traversal and this section will present literature on that topic. A similar subject related to this is path coverage. This will also be discussed alongside how a robustly optimal path traversal solution can be used for path coverage.

Path Traversal

Path traversal entails finding a path from a given start point to a given ending point. Traditional navigation solutions have often employed Dijkstra's algorithm [2], Bellemanford algorithm [16] or the A* search algorithm [3] to plan the shortest path in a given environment. These methods require representing the environment as a weighted graph $G = \{N, w\}$ with nodes N and weighted edges w. They generate optimal or suboptimal policies for traversing the world in the absence of uncertainty, therefore we categorize them as Deterministic Shortest Path (DSP) problems. DSP can be generalized to Stochastic Shortest Path (SSP) problems [17] when the value of any weight comes from a distribution

that can be unique to that edge. There is extensive literature on various algorithms designed to address SSP cases [5].

Given that values from an SSP problem come from distributions an expected cost can be formed for a given path. This entails the average value of each possible edge in the graph. This expectation is not a worst-case cost bound for a path. The difference between the worst-case and expected cost paths would be that the worst-case has the highest possible cost of the path instead of the average cost. In both cases, there exists uncertainty in the true value of each edge but these two cost models predict the edge costs differently. In many situations picking the path with the lowest expected cost might be ideal but in other cases the shortest worst-case path would be needed. This would mean finding the RSP [4]. In this case, robustness pertains to cost uncertainty, and RSP methods identify the shortest path assuming the worst-case environment state. In this kind of problem the edge uncertainty is modeled as an adversarial game between two agents: one aiming to minimize the path cost and the other seeking to maximize it.

The effectiveness of RSP methods fundamentally depends on the uncertainty model employed. Various types of aleatoric uncertainty exist in an RSP setting, such as uncertainty in the cost of edges [4, 18], or the next transition [12]. For brevity, this thesis focuses on *uncertain edge weights*, assuming known edge existence and transitions but with stochastic edge costs. A crucial consideration is the relationship between uncertainties on different edges, which may be independent [4] or correlated as a discrete set of scenarios [18]. This work addresses the latter case due to computational cost considerations. When edge cost uncertainties are independent, the number of potential cases grows exponentially, limiting the manageable complexity and size of the world under evaluation.

RSP cases can be then expanded to have stochastic and uncertain worst-case costs. This would mean the exists uncertainty of the true distribution of the weights. This means each edge has a series of possible distributions which could be the true one. This forms the case of Distributionally Robust Shortest Path (DRSP). There exists a multitude of methods to solve this particular case.

In the realm of RSP in area traversing only offline or hybrid solutions can really be used. There is, of course, online RSP *online path planning* [11], which assumes no prior knowledge of the world before observations. However, this is generally beyond the scope of RSP since the uncertainty of the world cannot be known if nothing about the world is known.

Having outlined the features of RSP, a model to set up a solution is required. Similar to SSP, RSP can be modeled as a Markov Decision Process (MDP) of the form S, A, P, R, where a state represents both the traversing agent's location and the current structure's uncertainty [19]. These MDPs are typically solved using Dynamic Programming (DP) methods such as VI or Policy Iteration, with solutions derived using finite horizons [20].

A simple example for the use case of Path Coverage was given earlier with a person moving through a building. Here are some additional use cases:

- **General Routing and Navigation** [4]

  When someone or something wants to get to a destination point, they usually want to take the fastest path. This limits time loss and possible energy costs costs. In the case of cars, given the information known about the streets and the possible congestion a path traversal algorithm could take this into account to make an optimal path. This can also be update live as route information changes and for example, accidents or traffic is reported.

  This type of case can also factor in a desire for a robust path if there is uncertainty about the path and there is great urgency in the time cost of the path. Imagine an ambulance is trying to drive to a hospital. While the fastest path outright to the hospital would the best, given there is uncertainty in the state of the road the path with the best worst-case guarantees would be best.

  Routing and navigation is not only unique to cars, but to robots and humans as well. Take the original example given in this thesis with a person navigating through a building. A person needs to navigate from point A to point B while the path is uncertain. They would therefore benefit from a path that assumes worst-case conditions.

- **Network Routing** [4]

  When messages are being sent across a data-network they need to be received as quickly as possible. A path needs to be planned to do this. Also, in big networks, a packet needs to go through many routers along a predetermined path. Given different uncertain states of congestion of these routers, the time a packet would need to arrive at its final destination may vary. Therefore, a path that takes these uncertain router states into account can reduce the traversal time. This uncertainty does exist because

even if the current state of each router is known before the packet is sent, this can evolve and change as the packet goes from node to node.

- **Route or Network Topology Design** [4]

  In the two previous cases of General Routing/Navigation and Network Routing an optimal path is found along a current road/network. This is in contrast to network or route design, in this case the worst-case path cost wants to be computed but in hypothetical cities. Effectively, when a city's transportation or a network is being designed, finding the RSP between two important points can be needed to know where to allocate more bandwidth. This case would have uncertainty in both the given factors being considered for the path and any other uncertainty being taken into account in the design.

Path traversal formed a problem that could be solved with exact guarantees using DP solutions. This gives an upper-bounded cost to traversing a space that can be used in a variety of important applications as previously described. As will be explained path coverage problems are more computationally taxing given the same space and could generally using a modified solution compared to path traversal. On top of this, the work presented forms missing literature about RSP with discrete uncertainty, with DP in infinite horizons, for path traversal. Therefore, this was the problem chosen for the sake of this thesis.

Path Coverage

Path Coverage does not have a single location as a final destination point. Instead it generally has one of two possible goals. The first type of goal is to go to a set of points in a given environment. The second type is stopping after something has been found. Imagine searching a space until a desired object is found. Until the object is found all possible areas need to be checked, but after finding the object nothing more has to be done. If a graph is used to represent the possible locations in the world of interest with edges representing the ability to move from one state to another, path coverage will go from one node to another until the goal is attained.

In the case of imperfect state information, this type of problem is traditionally solved with algorithms such as depth-first search [21] and Spiral Spanning Tree Coverage (Spiral STC) [22]. These algorithms can use the partially known graph and find a coverage for it

with minimal prior information about the graph. However, they do not take into account costs associated with traversing edges. In the original office building example, these algorithms would tell the person how to move around an environment without considering the time it take between any two points. There are more sophisticated algorithms like Prim's algorithm [23, 24] or Kruskal's algorithm [24]. These take the cost of the graph edges into account and can give the shortest path to cover the space. However, these methods do not take uncertainty about the graph edges into account. The only methods in literature which could possibly take an uncertainty into account use various DP, Deep Neural Networks (DNN), or Reinforcement Learning (RL) to solve the problem [25]. These methods can be trained to develop solutions that take the uncertainty into account. These methods have drawbacks. Aside from DP these methods only give an approximation of the cost and not the true while DP can have a very expensive time complexity to its computation.

A simple example for the use case of Path Coverage was given earlier. Here are some additional use cases:

- **Spatial coverage (searching)**

  Spatial coverage and searching missions require all areas of a given world be investigated. An example of this is with farming robotics [26]. There exists a known map of the field to be covered and a machine needs to be sent out to distribute water and fertilizer. While the map is generally known, new obstacles may arise preventing movement through the space. A fence or tree might have collapsed onto the path, in which case the agent will need to compensate accordingly.

- **Finding operations (Rescue)**

  Finding operations such as rescue missions require a area to be covered until something is found. Then the mission can stop.

  Imagine that in a tunnel environments or buildings, a rescue team needs to find an person [27]. The location of the person is unknown inside the space. The rescuers know the original map of the environment, but a series of events have occurred which has change the dynamics. For example, a tunnel or passage could have collapsed. Given this uncertainty of the state of the tunnels an optimal path is needed to cover the area.

Given the collapsed passages, there is a possibility that some portions of the map are totally inaccessible. This would make it impossible for path coverage to be completed in those areas. In such cases, the definition of path coverage would be the coverage of the coverable space, where there exists a path from the starting point to the point in question.

Another case of this would be with electric grids [28]. In the case of regularly maintained electric grids, all the lines need to be verified to validate if any repairs are needed. This can also be necessary if there is damage to the network, but the location of the damage is unknown. In this case, an agent such as a drone needs to fly to inspect the lines until they are all inspected or the problem in question is found. This requires a path that covers the area of the electric lines.

The uncertainty in this type of case can come from the possible usable paths. While a drone can fly over most spaces, it would also need a good angle to properly view the electric lines. This path becomes variable when there is a possible storm and certain areas may become impassable for a drone.

- **Spatial Mapping (unknown environments) [29]**

  The ability to have a map of a previously unknown area generally requires the whole surface of the given space to be passed through and modeled. This is particularly true in indoor or underwater cases. This type of case lends itself well to online path coverage. In that case, very little prior information can be leveraged since there is no known information.

- **Software Security Testing [30]**

  Given a program, it can be broken up into a graph which shows the flow of the code. This develops a control graph. Depending on the state of the code, different branches of the graph might be used. These types of graphs are important in software testing because all sections and conditions of the code need to be validated for functionality and security reasons. Graph coverage algorithms are used in such cases. In these cases, there can be uncertainty of when a particular edge will be used. This uncertainty can make it take longer to test all following blocks with all possible inputs.

- **Traveling Salesman [25, 31]**

  The Traveling Salesman problem is a traditional graph theory where the objective is to find a path from a start point on a graph and go to all nodes without any repeat. This is not exactly the problem solved with path coverage, but if the coverage algorithm does not repeat any nodes it does solve this problem. This graph would then be a Hamiltonian cycle.

Path coverage, as presented above, forms another interesting problem to solve that can be used in many cases. It has an added issue where a state is a function of not just uncertainty and location but also the unique positions the agent has covered so far. This does increase the complexity of the problem which will grow faster than in the case of path traversal. The greater interest in path traversal was that a solution to that more common problem can be used to solve coverage cases. This would entail making the value of a state not related to the cost to reach a terminal node but the cost of arriving to and covering all as of yet not covered nodes. This last node to be covered is a terminal state. On top of this, if an approximation solution is found for robust path traversal it can equally apply in this case.

1.5 Notation Section

In the section we will specify the meaning of the symbols and conventions used in this this work.

1.5.1 Symbols

1. $\mathbb{R}$: set of natural numbers.
2. $\mathbb{N}$: set of integer numbers.
3. $\mathcal{C}$: a finite subset of positive real numbers which includes ∞ ($\mathbb{R}_{>0} \cup \{\infty\}$)
4. ∞: infinity.
5. min: function giving the minimum value in a set.
6. max: function giving the maximum value in a set.
7. argmin: function giving the element number in the set with the mimimum value.
8. argmax: function giving the element

number in the set with the maximum value.

9. $\mathcal{G}$: a graph.
10. $\mathcal{G}_\ell$: a graph $\mathcal{G}_\ell$ with $\ell \in \mathcal{L}$
11. $\mathcal{N}$: the set of possible nodes in $\mathcal{G}$.
12. n: a node from $n \in N$.
13. $\mathcal{N}^+(n)$: the set of out-neighbors of node n.
14. $\mathcal{N}_\ell^+(n)$: the set of out-neighbors of node n in graph $\ell \in \mathcal{L}$.
15. $\mathcal{N}_{\ell^*}^+(n)$: the set of out-neighbors of node n from the true graph ℓ^*.
16. L: the number of weighted graphs in a given environment.
17. $\mathcal{L}$: the set of weighted graphs in a given environment.
18. ℓ: a specific $\mathcal{G}$ from $\mathcal{L}$.
19. ℓ^*: the element number for the true $\mathcal{G}$ in L as chosen by nature.
20. w: a singular set of weighted edges in graph $\mathcal{G}$.
21. $w(n, m)$: a weighted edge between nodes n and m such that $n, m \in \mathcal{N}$.
22. $w_\ell(n, m)$: a weighted edge from set $\ell \in \mathcal{L}$ between nodes n and m such that $n, m \in \mathcal{N}$.
23. k: the number of nodes in a path.
24. n^s: the start node of a path.
25. n^d: the end node of a path.
26. $o = \mathcal{O}(\ell^*, n)$: the set of observation of the out-going neighbors from node n and the weight associated with going from node n to these next nodes.
27. Θ: the set of all possible observations the agent could make from any given node in G across all $\mathcal{L}$
28. A: the set of possible actions the agent can take.
29. π: a particular control law the agent can take with a policy $\boldsymbol{\pi} \in \boldsymbol{\Pi}$.
30. π^*: optimal control law the agent can take.
31. $\boldsymbol{\Pi}$: set of all possible Policies for the agent.
32. $\boldsymbol{\pi}$: a policy for the agent where $\boldsymbol{\pi} \in \boldsymbol{\Pi}$.
33. $\boldsymbol{\pi}^*$: the optimal policy for the agent.
34. $T_\ell^{\boldsymbol{\pi}}$: the first time when an agent starting at node n^s in graph $\mathcal{G}^\ell$ and following policy $\boldsymbol{\pi}$ reaches the destination n^d.
35. $J^{\boldsymbol{\pi}}$: the maximum cost of taking policy $\boldsymbol{\pi}$.
36. s_t: the true partially observed state of the agent at time t.

37. x_t: the information state of the agent at time t.
38. f: the function to update the state to s_{t+1}.
39. $\mathcal{F}_t$: the set of feasible states at time t.
40. ϕ: the function to update the set of feasible states to $\mathcal{F}_{t+1}$.
41. V: the unique bounded solution of a given fixed point equation.
42. B: Bellman update for the V.
43. Λ: function to develop the optimal $\boldsymbol{\pi}$ from a given V.
44. p: a particular path from n^s to n^d in a given graph.
45. C_{max}: the maximum cost of a given path across all possible graphs.
46. D_{min}: the minimum cost of a path across all possible graphs going through a particular edge.
47. Q: function giving the value of state action pair
48. N_s: the number of times a given state has been visited in MCTS.
49. N_{sa}: the number of times a given action has been taken at a given state in MCTS.
50. $\mathcal{C}$: modifier on the RHS of the UCB equation to proportionally incentive increased exploration.

1.5.2 Conventions

1. Any variable marked as $a_{1:t}$ indicates because the instance of that variable from all times $(a_1, \ldots, a_t)$ with $t > 1$.

Chapter 2

Robust Shortest Path

In this chapter, we present the RSP problem over weighted graphs with uncertain weight function and describe a DP solution for it. We then illustrate the algorithm via a small example.

2.1 Preliminaries on Graphs

Definition 1 (Weighted Directed Graphs) A weighted directed graph $\mathcal{G} = \{\mathcal{N}, w\}$ is given by a finite set of nodes $\mathcal{N}$ and a weight function $w : \mathcal{N} \times \mathcal{N} \mapsto \mathbb{R} \cup \{+\infty\}$.

The weight function w implicitly encodes the edges of the graph. In particular, for $n, m \in \mathcal{N}$, if $w(n, m) < \infty$ then (n, m) is an edge of the graph, otherwise it is not. For this reason, we do not explicitly model the set of edges in the graph.

Given a graph $(\mathcal{N}, w)$, the set of out-neighbors of a node $n \in \mathcal{N}$ is defined as:

$$\mathcal{N}^+(n) := \{m \in \mathcal{N} : w(n, m) < \infty\}. \tag{2.1}$$

As an example consider the graph shown in Figure 2.1 where $\mathcal{N} = \{0, 1, 2, 3\}$ and the weight w can be represented by a 4 matrix

$$w = \begin{bmatrix} \infty & a & \infty & \infty \\ \infty & \infty & b & \infty \\ \infty & \infty & \infty & c \\ d & \infty & \infty & \infty \end{bmatrix}. \tag{2.2}$$

Definition 2 (Paths and loopless paths) Given two nodes $n, m \in \mathcal{N}$, we say that there is a path between n and m if there exists an integer k and nodes $n_1, \ldots, n_k \in \mathcal{N}$ such that $n_1 = n$, $n_k = m$ and for all $i \in \{1, \ldots, k-1\}$, $n_{i+1} \in \mathcal{N}^+(n_i)$. A path is said to be *loopless*[1] if all the nodes in the path are unique.

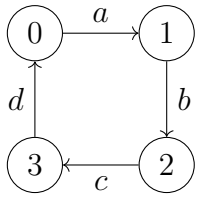

Fig. 2.1 Example directed Graph with node set $\mathcal{N} = \{0, 1, 2, 3\}$ and edge weight w

Take the nodes 1 and 3 from Figure 2.1. There exists a path between these two nodes which from $n = 1$ to $n = 3$. In this case $k = 4$ where the path is $\{0, 1, 2, 3\}$.

Definition 3 (Undirected Graph) A graph $\mathcal{G} = (\mathcal{N}, w)$ is said to be undirected if for all $n, m \in \mathcal{N}$, $w(n, m) = w(m, n)$.

2.2 Problem Formulation

As mentioned in the Introduction, we are interested in finding the shortest path from a source to a destination node when there is uncertainty about the weight function of the graph.

We model this uncertainty as follows. Let $\mathcal{C}$ be a finite subset of $\mathbb{R}_{>0} \cup \{+\infty\}$ that contains $+\infty$. There are L weighted directed graphs, $\mathcal{G}_1, \ldots, \mathcal{G}_L$, where for $\ell \in \mathcal{L} := \{1, \ldots, L\}$, $\mathcal{G}_\ell := (\mathcal{N}, w_\ell)$, $w_\ell : \mathcal{N} \times \mathcal{N} \to \mathcal{C}$. The true graph is G_{ℓ^*}, $\ell^* \in \mathcal{L}$, where the value of ℓ^* is not known to the agent. We are given source node $n^s \in \mathcal{N}$ and a destination node $n^d \in \mathcal{N}$ and it is assumed that the following assumption holds.

Assumption 1 *For each graph $\mathcal{G}_\ell$, $\ell \in \mathcal{L}$, there exists a path from n^s to n^d.*

[1] Loopless paths are also called paths without cycles.

The agent starts with the knowledge of just $\mathcal{G}_1, \ldots, \mathcal{G}_L$ and additional information about the local neighborhood becomes available to the agent as it traverses the graph. In particular, when the agent is at node n, it obtains an observation $y = (n, o)$, where

$$o = \mathcal{O}(\ell^*, n) := \{(m, , w_{\ell^*}(n, m)) : m \in \mathcal{N}^+_{\ell^*}(n)\}. \tag{2.3}$$

The set of all possible observations an agent could make is defined by the set:

$$\Theta = \{\mathcal{O}(\ell, n) : \ell \in L, n \in \mathcal{N}\} \tag{2.4}$$

The agent can construct the out-neighborhood of n from the observation o. In particular,

$$\mathcal{A}(o) := \{m \in \mathcal{N} : (m, c) \in o \text{ and } c < \infty\} = \mathcal{N}^+_{\ell^*}(n) \tag{2.5}$$

Based on the history of these observations, the agent chooses an action $a \in \mathcal{A}(o) = \mathcal{N}^+_{\ell^*}(n)$ to decide where to go next. Let o_t denote the observation of the agent at time t and a_t denote the action at time t. The actions are chosen according to a policy $\boldsymbol{\pi} = (\pi_1, \pi_2, \ldots, \pi_t, \ldots)$ where:

$$\pi_t : (n_{1:t}, o_{1:t}, a_{1:t-1}) \mapsto a_t, \text{such that } a_t \in \mathcal{A}(o_t). \tag{2.6}$$

Let $\boldsymbol{\Pi}$ denote the set of all such policies.

Given policy $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ and $\ell \in \mathcal{L}$, let $T^{\boldsymbol{\pi}}_\ell$ denote the first time when an agent starting at node n^s in graph $\mathcal{G}^\ell$ and following policy $\boldsymbol{\pi}$ reaches the destination n^d.

Definition 4 (Proper Policy) A policy $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ is called **proper** if for every $\ell \in \mathcal{L}$, $T^{\boldsymbol{\pi}}_\ell < \infty$.

Assumption 2 *There exists a policy $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ that is proper.*

Remark 1 For undirected graphs Assumption 1 implies Assumption 2. The same holds in a graph which has a finite-cost reverse edge for each finite-cost edge except for the edges going into the terminal node.

The cost of a policy $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ in graph $\mathcal{G}_\ell$, $\ell \in \mathcal{L}$, when the agent starts at n^s and ends at

n^d is given by:

$$J_\ell^{\boldsymbol{\pi}} := \sum_{t=1}^{T_\ell^{\boldsymbol{\pi}}} w_\ell(n_t, a_t). \tag{2.7}$$

The worst-case cost of a policy $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ when the agent starts at n^s and ends at n^d is given by:

$$J^{\boldsymbol{\pi}} := \max_{\ell \in \mathcal{L}} J_\ell^{\boldsymbol{\pi}} \tag{2.8}$$

With the remarks and assumptions above, we address the following optimization problem.

Problem 1 *Given $\mathcal{N}$, and graphs $\mathcal{G}_1, \ldots, \mathcal{G}_L$ and n^s, $n^d \in \mathcal{N}$, find a policy $\boldsymbol{\pi} \in \boldsymbol{\Pi}$ of the form in* (2.6) *to minimize $J^{\boldsymbol{\pi}}$ given by* (2.8)*, i.e., solve*

$$\min_{\boldsymbol{\pi} \in \boldsymbol{\Pi}} \max_{\ell \in \mathcal{L}} J_\ell^{\boldsymbol{\pi}}.$$

We now illustrate the model via an example.

Example 1 Consider the grid-world shown in in Figure 2.2, where the agent (shown by a red triangle) starts in the middle left side and has to go to the destination (shown as a green square) on the middle right side. Three walls surrounding the middle node are blocked. There are two blue doors around the destination, one of which is closed and the other open; but the agent does not know their status. The agent is interested in finding an policy which minimizes the worst-case cost of going from the source to the destination.

We model this scenario using the model presented in Sec. 2.2. We model the grid-world as an undirected graph with $\mathcal{N} = \{0, 1, \ldots, 8\}$ as shown in Figure 2.3. The uncertainty about the status of the blue edges in Figure 2.3 is captured by considering two possible states of the world $\mathcal{G}_1 = (\mathcal{N}, w_1)$ and $\mathcal{G}_2 = (\mathcal{N}, w_2)$ as shown in Figs. 2.4 and 2.5. When the weight function w_ℓ is known, we can easily find a path from the source to the destination which is shown in red. Hence Assumption 1 holds. Since the graph is undirected, in Figure 2.4 and 2.5, Assumption 2 holds.

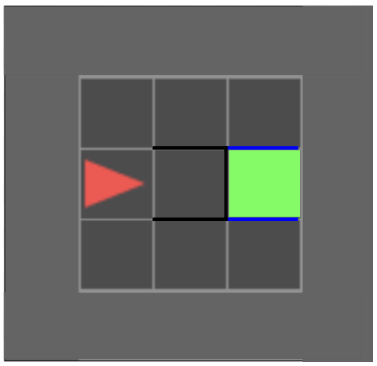

Fig. 2.2 Example 1 grid of 3×3 sample world.

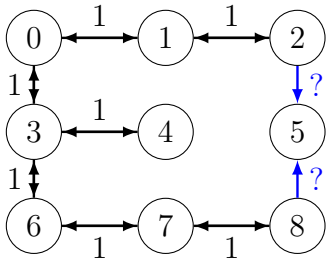

Fig. 2.3 Example 1 graph representation of Figure2.2.

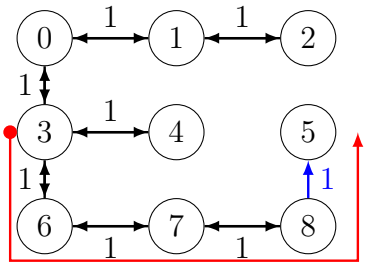

Fig. 2.4 Example 1 $\mathcal{G}_1$. Red line indicates shortest path.

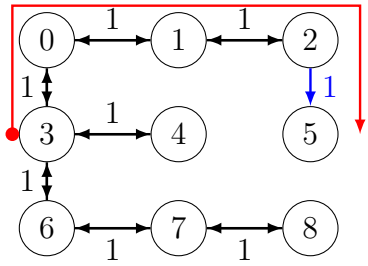

Fig. 2.5 Example 1 $\mathcal{G}_2$. Red line indicates shortest path.

2.3 Dynamic Programming Decomposition

2.3.1 Information State

The problem outlined in Problem 1 can be modeled as a uncertain dynamical system with a partially observed state $s_t = (\ell^*, n_t)$. The system starts in state $s_1 = (\ell^*, n^s)$ at $t = 1$ and the state evolves as:

$$s_{t+1} = f(s_t, a_t). \tag{2.9}$$

where the update function f is given by

$$f((\ell, n), a) = (\ell, a), \quad \ell \in \mathcal{L}, n \in \mathcal{N}, a \in \mathcal{N}_\ell^+(n). \tag{2.10}$$

At time t, the agents gets an observation $o_t = \mathcal{O}(s_t)$ where $\mathcal{O}$ is given by (2.3). The

action a_t is chosen accordingly to a policy $\boldsymbol{\pi} = (\pi_1, \pi_2, \dots)$ as given in (2.6) where π_t represents the control laws which output the specific action at a given state. The performance of the policy is given by (2.8).

Following [20], we can convert the above partially observable uncertain system to a fully observable uncertain system using an information state. The information state is given by $x_t := (\mathcal{F}_t, n_t, \mathcal{A}(o_t))$ where $\mathcal{F}_t$ is the set of feasible $\ell \in \mathcal{L}$ consistent with the history of observations and actions up to time t. The feasible set $\mathcal{F}_t$ evolves as follows:

$$\mathcal{F}_t = \phi(\mathcal{F}_{t-1}, n_t, o_t) \tag{2.11}$$

where the update function ϕ is given by

$$\phi(\mathcal{F}, n, o) = \{\ell \in \mathcal{F} : \mathcal{O}(\ell, n) = o\}, \quad \forall \mathcal{F} \in 2^{\mathcal{L}}, n \in \mathcal{N}, o \in \Theta \tag{2.12}$$

At $t = 0$, $\mathcal{F}_0 = \mathcal{L}$. At $t = 1$ an observation o_1 is made and therefore $\mathcal{F}$ updates as $\mathcal{F}_1 = \phi(\mathcal{F}_0, n_1, o_1)$. Thus, the information state starts at $x_1 = (\mathcal{F}_1, n_s, \mathcal{A}(o_1))$ and evolves as $x_{t+1} = (\mathcal{F}_{t+1}, n_{t+1}, \mathcal{A}(o_{t+1})) = (\phi(\mathcal{F}_t, n_{t+1}, o_{t+1}), n_{t+1}, \mathcal{A}(o_{t+1}))$. For all $t \geq 0$, $\mathcal{F}_t \subseteq \mathcal{F}_{t-1}$ as the uncertainty of the true ℓ^* can only reduce or stay constant with every given observation.

Remark 2 The cost of an outgoing edge at the current agent's node n_t with action a_t is $w_{\ell^*}(n_t, a_t)$. In (2.12), the sets $\mathcal{F}_t$ are constructed such that at each t all the possible $\ell \in \mathcal{F}_t$ are such that their out-neighborhood $\mathcal{N}_\ell^+$ is the same. Thus, for any feasible action $a_t \in \mathcal{A}(n_t)$, the agent knows the cost $w_{\ell^*}(n_t, a_t)$.

2.3.2 Dynamic Programming Solution

Theorem 1 *Suppose Assumptions 1 and 2 hold. Let V be the unique bounded solution of the following fixed point equation, called the dynamic program:*

$$V(\mathcal{F}, n, \mathcal{A}(o)) = \min_{a \in \mathcal{A}(o)} \max_{\ell \in \mathcal{F}} \Big\{ w_\ell(n, a) + V(\phi(\mathcal{F}, a, \mathcal{O}(\ell, a)), a, \mathcal{A}(\mathcal{O}(\ell, a))) \Big\}. \tag{2.13}$$

and let $\pi^*(\mathcal{F}, n, \mathcal{A}(o))$ denote the argmin *of the RHS of* (2.13)*. Then the time homogeneous policy $\boldsymbol{\pi^*} = (\pi^*, \pi^*, \dots)$, i.e., $a_t = \pi^*(\mathcal{F}_t, n_t, \mathcal{A}(o_t))$, is optimal for Problem 1.*

PROOF The DP solution follows from [20]. The analysis in [20] was for finite horizon models. Following the arguments in [32] these finite horizon results can be extended to infinite time horizons for stochastic shortest path problem under Assumptions 1 and 2.

2.3.3 Interpretation as a 2-Player ZSG

As the problem is solved as a minmax dynamic program, it can also be viewed as a 2 player ZSG. In this case player one is the agent which is trying to take actions to arrive at the destination while minimizing its total cost and player two is nature which is choosing the state of the world to maximize the total cost. The robustly optimal solution computed in Theorem 1 is a Minimax Equilibrium of this 2 player game.

2.3.4 Value Iteration

VI is an iterative algorithm for approximately finding the fixed point of a dynamic program. The algorithm is initialized at an arbitrary V_0. Then at each iteration $k \in \mathbb{N}$, an updated estimate is generated using

$$V_{k+1} = \mathcal{B}V_k, k \in \mathbb{N}. \tag{2.14}$$

where the Bellman operator $\mathcal{B}$ is defined as:

$$[\mathcal{B}V_k](\mathcal{F}, n, o) = \min_{a \in \mathcal{A}(o)} \max_{\ell \in \mathcal{F}} (w_\ell(n, a) + V(\phi(\mathcal{F}, a, \mathcal{O}(\ell, a)), a, \mathcal{A}(\mathcal{O}(\ell, a))). \tag{2.15}$$

We also compute a policy

$$\boldsymbol{\pi}_k = \Lambda V_k. \tag{2.16}$$

where the functional Λ is defined as:

$$[\Lambda V_k](\mathcal{F}, n, o) = \operatorname*{argmin}_{a \in \mathcal{A}(o)} \max_{\ell \in \mathcal{F}} (w_\ell(n, a) + V(\phi(\mathcal{F}, a, \mathcal{O}(\ell, a)), a, \mathcal{A}(\mathcal{O}(\ell, a))). \tag{2.17}$$

The algorithm terminates when $\|V_{k+1} - V_k\|_\infty < \epsilon$, for a pre-specified accuracy level ϵ. The policy $\boldsymbol{\pi_{k+1}}$ is returned as an approximately optimal policy.

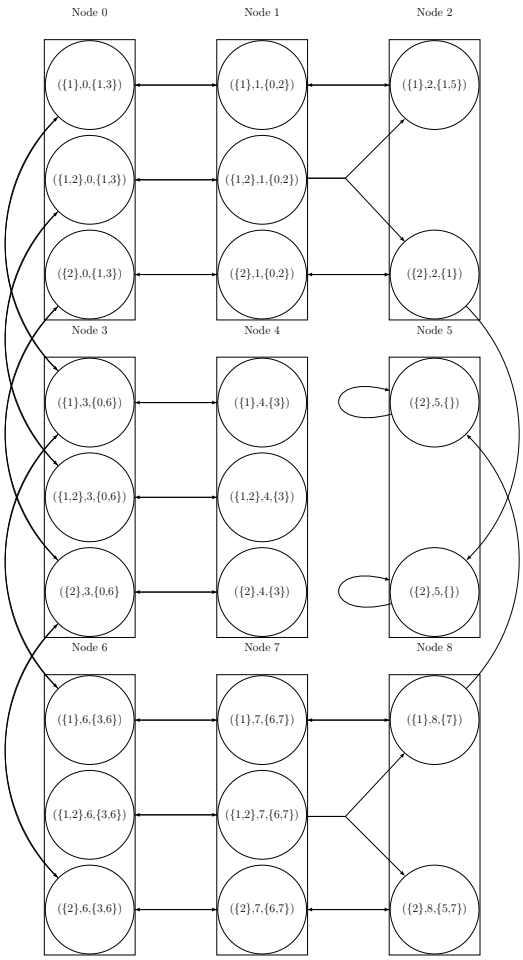

Fig. 2.6 Example 1 information state graph for example 1 where $\mathcal{L} = \{1, 2\}$. Bounding boxes at each node indicate the possible states at that node.

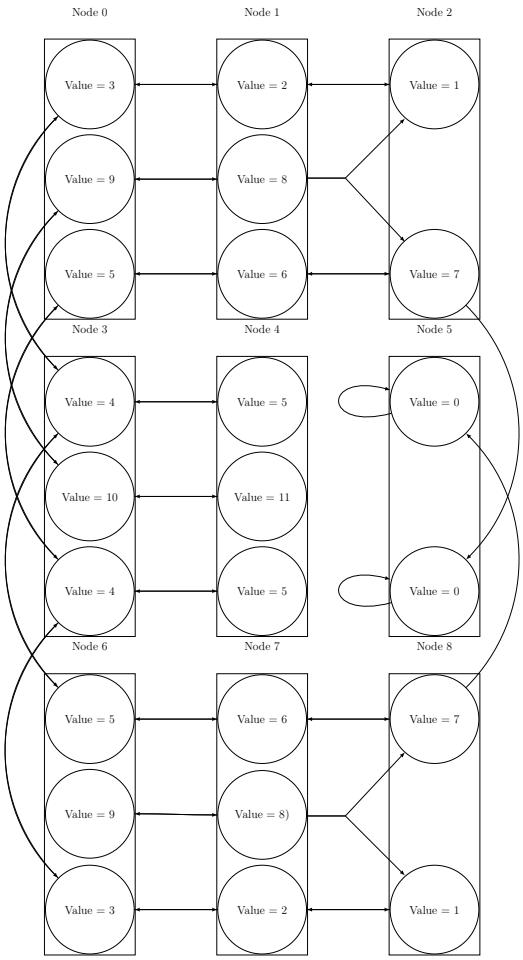

Fig. 2.7 Example 1 Value Figure.

Algorithm 1: Value Iteration

Input: ϵ (a small positive number)
Initialize: $V(x) \leftarrow 0$
repeat
    $V_{old} \leftarrow V$
    $V \leftarrow \mathcal{B}V$
until $|V_{old} - V| < \epsilon$
$V^* \leftarrow V$
$\boldsymbol{\pi}^* \leftarrow \Lambda V^*$
Output: V^* (approximately optimal value function) and $\boldsymbol{\pi}^*$ (approximately optimal policy)

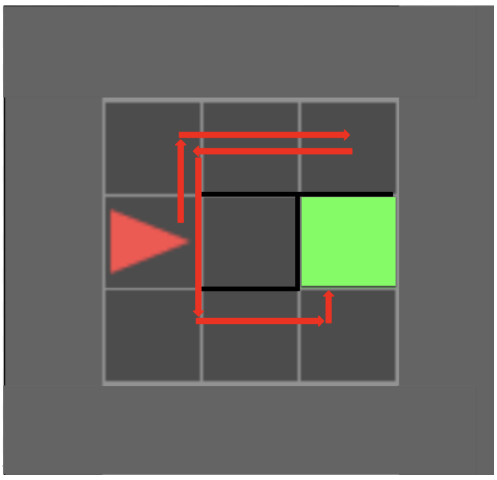

Fig. 2.8 Path taken by the optimal policy in Example 1 when the nature picks $\mathcal{G}_1$

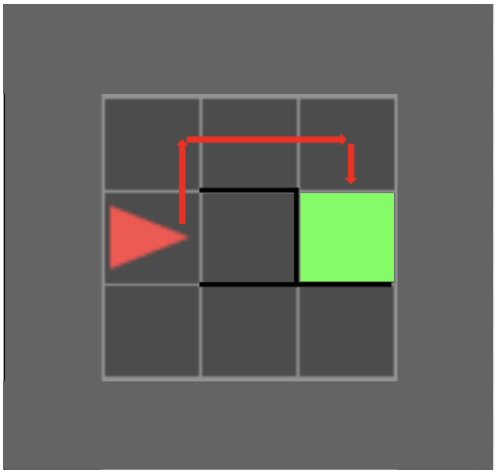

Fig. 2.9 Path taken by the optimal policy in Example 1 when the nature picks $\mathcal{G}_2$

2.3.5 Illustration of the DP Solution

Consider the example in Figure 2.2. Since $\mathcal{L} = \{1, 2\}$, there are $2^{|\mathcal{L}|} - 1 = 3$ possible values of $\mathcal{F}$ for each node n. For each $(n, \mathcal{F})$, the possible values of $\mathcal{A}(o)$ is fixed. Thus, there are $|\mathcal{N}| \times \left(2^{|\mathcal{L}|} - 1\right) = 9 \times 3 = 27$ possible values of information state. However, not all of them are feasible: at nodes $n \in \{2, 5, 8\}$, the belief state $\mathcal{F}_t = \{1, 2\}$ is not possible. Therefore, only $27 - 3 = 24$ states are feasible, which are shown in 2.6. Also note at states $(\{2\}, 5, \{5\})$ and $(\{1\}, 5, \{5\})$, which are the terminal states, the only action the agent can take is to stay at its current location. This action has cost of 0.

The out-going edges at each information-state in Figure 2.6 correspond to feasible ac-

tions at that information-state. Note that some of the actions lead to uncertain outcomes, e.g., information state $(\{1,2\},1,\{0,2\})$ and $(\{1,2\},7,\{6,7\})$. As can be seen in Figure 2.6, the policy for RSP from source $(\{1,2\},3,\{0,6\})$ to $\{(\{2\},5,\{5\}),(\{1\},5,\{5\})\}$, the terminal states, is optimal for the original optimization problem.

2.4 Node Pruning

The complexity of VI depends on the size of the set of all feasible information states. In this section, we propose a method to prune the graphs $\mathcal{G}_1,\ldots,\mathcal{G}_L$, which reduces the number of feasible information states and thereby improves the computational complexity of the algorithm. We start with some definitions.

For any path p in graph $\mathcal{G}_\ell$, $\ell\in\mathcal{L}$, let $C_\ell(p)$ denote the cost of the path. If the path contains an edge with infinite weight, then its cost is infinity. For any path p, define

$$C_{\max}(p)=\max_{\ell\in\mathcal{L}} C_\ell(p)$$

For any edge (n,m), $n,m\in\mathcal{N}$, let $D_\ell(n,m)$ denote the cost of the shortest path from the source to the destination in graph $\mathcal{G}_\ell$ that contains the edge (n,m). For any edge (n,m), define:

$$D_{\min}(n,m)=\min_{\ell\in\mathcal{L}} D_\ell(n,m).$$

Definition 5 (Looped Path) A path $p_\circ$ from n^s to n^d with a loop is a path of k nodes $n_1\ldots n_k\in N$ where there exists at least 2 nodes n_i and n_j where $n_i=n_j$. This means that there exists a set k' of nodes from $p_\circ$ which can also form a loopless path $p_{\circ'}$.

We consider three types of pruning:

1. **Pruning of $\mathcal{L}$.** For every $\ell_\circ\in\mathcal{L}$, remove the graph $\mathcal{G}_{\ell_\circ}$ if there exists a graph $G_{\ell'}$, $\ell'\in\mathcal{L}$, $\ell'\neq\ell_\circ$, such that $w_{\ell_\circ}(n,m)\leq w_{\ell'}(n,m)$, for all nodes $n,m\in\mathcal{N}$.

2. **Pruning of edges in $\mathcal{G}_\ell,\ell\in\mathcal{L}$.** Take a node $n\in\mathcal{N}$ and a loopless path $p_\circ$ from n to the destination such that $C_{\max}(p_\circ)<\infty$. Consider an edge (n,m), which does not lie on $p_\circ$, such that

$$D_{\min}(n,m)>C_{\max}(p_\circ).$$

Then remove the edge (n, m) from all graphs $\mathcal{G}_\ell$, $\ell \in \mathcal{L}$. Note that removing an edge is equivalent to setting its weight to $+\infty$.

3. **Pruning of nodes in all $\mathcal{G}_\ell$, $\ell \in \mathcal{L}$.** Remove all nodes $n \in \mathcal{G}_\ell$ that do not belong to any finite-cost path from the source n^s to the destination n^d in any graph $\mathcal{G}_\ell$, $\ell \in \mathcal{L}$. Removing a node means that we remove it from the node set $\mathcal{N}$.

Note that the pruning steps can be repeated. So, when we start with a problem, we apply the pruning steps one by one, until they no longer lead to a simplification.

Proposition 1 *The three pruning methods described above do not change the optimal solution.*

Proof We will separately establish that each of the pruning methods do not change the optimal solution.

1. Fix an $\ell_\circ \in \mathcal{L}$. Suppose there exists an $\ell' \in \mathcal{L}$ such that $w_{\ell_\circ}(m, n) \leq w_{\ell'}(m, n)$ for all $m, n \in \mathcal{N}$. Then, for any policy $\boldsymbol{\pi}$, we have

$$J^{\boldsymbol{\pi}}_{\ell_\circ} \leq J^{\boldsymbol{\pi}}_{\ell'}$$

where we allow the right hand side to be infinity. Thus,

$$\max_{\ell \in \mathcal{L}} J^{\boldsymbol{\pi}}_\ell = \max_{\ell \in \mathcal{L} \setminus \{\ell_\circ\}} J^{\boldsymbol{\pi}}_\ell$$

Thus, nature can ignore $\ell_\circ$ while picking its action.

2. Let $p_\circ$ and (n, m) be as in the definition of pruning. Consider any information state $x = (\mathcal{F}, n, \mathcal{A}(o))$. Let $(n, m_\circ)$ be the first edge of $p_\circ$. Since $C_{\max}(p_\circ) < \infty$, we must have that $w_\ell(n, m_\circ) < \infty$ for all $\ell \in \mathcal{L}$. Therefore, $m_\circ \in \mathcal{A}(o)$ for all feasible observations o at node n. Define

$$Q(\mathcal{F}, n, \mathcal{A}(o), a) = \max_{\ell \in \mathcal{F}} \Big\{ w_\ell(n, a) + V(\phi(\mathcal{F}, a, \mathcal{O}(\ell, a)), a, \mathcal{A}(\mathcal{O}(\ell, a))) \Big\}. \tag{2.18}$$

Note that $Q(x, m) \geq D_{\min}(n, m)$ (because no path for n to the destination can cost less than $D_{\min}(n, m)$) and $Q(x, n) \leq C_{\max}(p_\circ)$ (because following the path $p_\circ$ is a

feasible policy, hence the optimal policy must be at least as good as $p_\circ$). The fact that $D_{\min}(n,m) > C_{\max}(p_\circ)$ implies that

$$Q(x,m) \geq D_{\min}(n,m) > C_{\max}(p_\circ) \geq Q(x,m_\circ).$$

Hence, action m is never optimal, and can therefore be eliminated

3. Under Assumption 2, an optimal policy will never visit a node that does not belong to any finite-cost path from the source to the destination. So, removing such a node, does not change the optimal policy.

We now present some examples to illustrate pruning.

Example 2 Consider the example of Figure 2.10, where there are 2 possible graphs, $\mathcal{G}_1$ and $\mathcal{G}_2$, that differ in the weight of the edge $(10, 15)$ which has value 1 in $\mathcal{G}_1$ or value $+\infty$ in $\mathcal{G}_2$. These two graphs are seen in Figures 2.11 and 2.12.

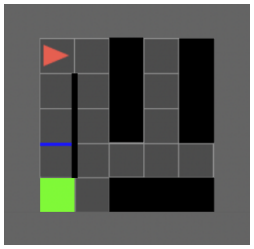

Fig. 2.10 Example 4 grid-world.

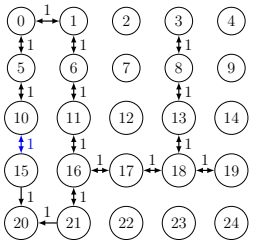

Fig. 2.11 Example 4 graph $\mathcal{G}_1$.

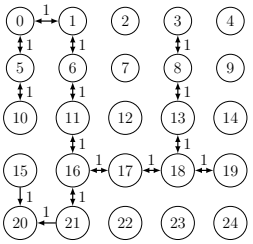

Fig. 2.12 Example 4 graph $\mathcal{G}_2$.

In this example, the edge weights of $\mathcal{G}_1$ are always less than or equal to $\mathcal{G}_2$ such that $w_1(n,m) \leq w_2(n,m)$. Therefore, the graph $\mathcal{G}_1$ can be pruned leaving only $\mathcal{G}_2$. In graph $\mathcal{G}_2$, let p_0 be the path through nodes $\{0, 1, 6, 11, 16, 21, 20\}$. All the edges which are not along this path can be pruned as they would lie on a path which costs more than the $C_{\max}(p_0) = 6$. This then allows edges $\{(0,5), (5,10), (16,17), (17,18), (18,19), (18,13), (13,8), (8,3)\}$. Due to this, many nodes such as $\{3, 5, 8, 10, 13, 15, 17, 18, 19\}$ can be removed as they no longer exist on a path from 0 to 20 meaning. The result of this pruning is the graph shown in Fig. 2.13.

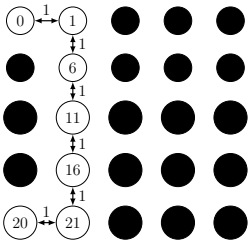

Fig. 2.13 Example 4 graph $\mathcal{G}_2$ after pruning.

Example 3 Consider the example of Figure 2.14. This environment contains 2 graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ as seen in Figures 2.15 and 2.16. In both cases there are 3 possible loopless paths the agent could take from $n^s = 1$ to $n^d = 7$. These paths go through the left, middle or right side of the map. Let these paths be defined as p_1, p_2, and p_3 respectively.

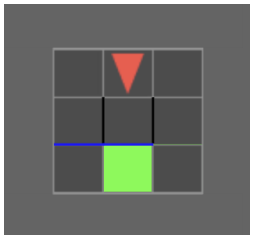

Fig. 2.14 Example 3 grid-world.

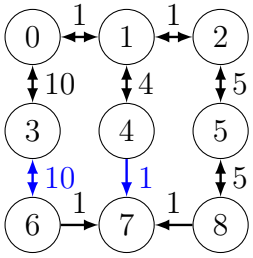

Fig. 2.15 Example 3 graph $\mathcal{G}_1$.

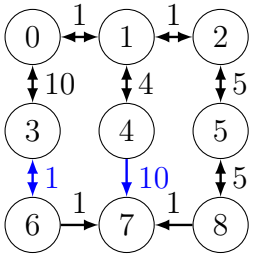

Fig. 2.16 Example 3 graph $\mathcal{G}_2$.

At start, no graph $\mathcal{G}_\ell$ can be removed since neither $w_1(n, m) \leq w_2(n, m)$ for all nodes $n, m \in \mathcal{N}$ nor $w_2(n, m) \leq w_1(n, m)$ for all nodes $n, m \in \mathcal{N}$ is true. Given the max cost of the right most path is $C_{\max}(p_0) = 12$, the edge $(0, 1)$ can be removed as the $D_{\min}(1, 0) = 13$. This then allows nodes $\{0, 3, 6\}$ to be removed as they would no longer be attainable along a path from $n^s = 1$ to $n^d = 7$. This also removes the uncertain edge $(3, 6)$ after which graphs G_1 and G_2 can be seen in Figures 2.17 and 2.18. In this current state, G_1 can be removed as $w_1(n, m) \leq w_2(n, m)$ for all nodes $n, m \in \mathcal{N}$. The resulting graph for Example 4 is Figure 2.18

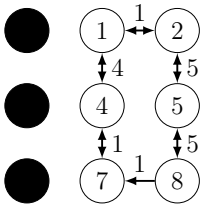

Fig. 2.17 Example 3 graph $\mathcal{G}_1$ after pruning.

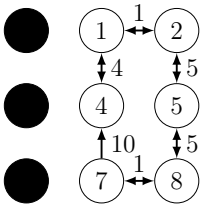

Fig. 2.18 Example 3 graph $\mathcal{G}_2$ after pruning.

Example 4 Consider the example of Figure 2.19 or its equivalent graph representations in Figures 2.20 and 2.21. In this example, either the top edge (2,5) or the bottom edge (5,8) does not exist and is valued at ∞.

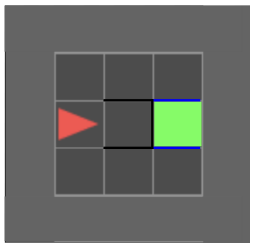

Fig. 2.19 Example 1 grid-world.

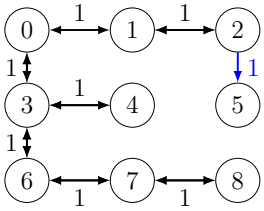

Fig. 2.20 Example 1 graph $\mathcal{G}_1$.

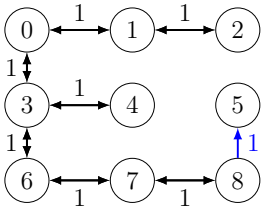

Fig. 2.21 Example 1 graph $\mathcal{G}_2$.

In this case, no graph can be removed, all nodes are attainable along a path $n^s = 0$ to $n^d = 5$ and no edge has a value $D_{\min}$ which is greater than a $C_{\max}(p)$ for any possible path p. Consider the edge $(3, 4)$ and path p which goes through nodes $\{3, 0, 1, 2, 5\}$. The value $D_{\min}(3, 4) = 6$ while there is no finite-cost $C_{\max}(p)$ as the worst-case of this path has a cost of ∞. The same is true for the path going through nodes $\{3, 6, 7, 8, 5\}$. Due to the lack of prunable graphs, edges and nodes nothing can be done in this case. The final graphs are the same as in Figures 2.20 and 2.21.

Chapter 3

Monte Carlo Tree Search

3.1 Upper Confidence Bounds

Upper Confidence Bounds (UCB) is a solution to a question in Reinforcement Learning (RL) on how to balance exploration and exploitation [33]. UCB works by ensuring that, at a given state, each action is explored at least once. After that, a proportional modifier is added to the current estimated value of each state/action pair. This modifier is based on how often a state/action pair has been used. It increases or decreases the estimated value such that a less frequently sampled action is explored more often by becoming the greedy action. This method contrasts with ϵ-Greedy algorithms, which take random actions with a certain probability, ϵ. This ϵ can either remain constant or decrease over time.

An example of the UCB equation is shown in (3.1). In this equation, an arbitrary Markov Decision Process (MDP) consists of states s and actions $a \in A(s)$. Here, $N_{sa}(s,a)$ refers to the number of times an action has been taken at a state, and $N_s(s)$ refers to the number of times a state has been visited. Given this, the UCB function in (3.1) selects the action that maximizes the next state's Q-value function, using argmax. This maximization is performed over the values of each state, which are modified proportionally to the number of times that action has been taken relative to other actions at that state. The term $\mathcal{C}$ modifies the value added, based on the state-action usage frequency.

$$UCB(s) = \underset{a \in A(s)}{\operatorname{argmax}} \left\{ \frac{Q(s,a)}{N_{sa}(s,a)+1} + \mathcal{C}\sqrt{\frac{2\ln(N_s(s)+1)}{N_{sa}(s,a)}+1} \right\} \tag{3.1}$$

If UCB was trying to minimize reward (with argmin_a) the whole term must be negated as seen in (3.3) [1].

$$UCB(s) = \underset{a \in A(s)}{\text{argmin}} \left\{ -1 \times \left\{ \frac{Q(s,a)}{N_{sa}(s,a)+1} \right\} - \mathcal{C}\sqrt{\frac{2\ln(N_s(s)+1)}{N_{sa}(s,a)+1}} \right\} \tag{3.2}$$

$$UCB(s) = \underset{a \in A(s)}{\text{argmin}} \left\{ -1 \times \left\{ \frac{Q(s,a)}{N_{sa}(s,a)+1} \right\} - \mathcal{C}\sqrt{\frac{2\ln(N_s(s)+1)}{N_{sa}(s,a)+1}} \right\} \tag{3.3}$$

In certain MCTS examples where there are two players playing in an adversarial manner, one player would want to maximize their reward while the other wants to minimize the opponent's reward. Because of this, both a maximizing UCB from equation (3.1) and a minimizing UCB from equation (3.3) to build a full MCTS algorithm.

3.2 Monte Carlo Tree Search [1]

Monte Carlo Tree Search (MCTS) is a method focused on collecting samples from an environment and building them into state/action trees, where each state has an associated value. MCTS is typically used in conjunction with Upper Confidence Bounds for Trees (UCT) to determine which action to take. In this case, UCT is analogous to UCB, with the different branches of the tree representing the actions. This approach has been applied in various parts of larger AI algorithms, one of the most notable examples being the AlphaGo algorithms [34, 35]. MCTS was used in both the training phase and the live implementations of the model.

In other instances, MCTS can be used to encourage exploration of less commonly sampled states of the game or to rapidly conduct policy evaluations. Each state has a specific value associated with it, and when a game reaches a particular state in the tree, the next action can be selected as the greedy maximum of the subsequent states.

This method breaks down into 4 steps:

1. Selection: Select the next node to take. The tree is descended using the UCT algorithm until a until it arrives at a node where there exists an action it has never taken.

2. Expansion: If a new action is taken then the tree is expanded with the new edge and

node.

3. Simulation: After the expansion the rest of the tree is followed or explored until a terminal node is reached.

4. Backpropagation: The rewards or cost accumulated during the last simulation, regardless if the tree was expanded or not, is backpropagated up the tree.

The selection algorithm in MCTS leads to one of two types of events: the Tree policy or the Default policy. In the former case, expansion is conducted to create a new leaf node, if possible, followed by a simulation of the value of that state. In the latter case, greedy actions are taken, with the values derived from the UCT method. In some instances, UCT is replaced by Predictor UCT (PUCT), which incorporates the belief about the next state [36]. This belief state represents the probabilities based on past observations of all possible next states. This process is illustrated in Algorithm 2. At the end of each iteration, backpropagation is used to update the value of each state. The combination of selection, expansion, simulation, and backpropagation ensures that, given enough time, the UCT method will reveal the true value of all states.

This method has been applied in various settings, such as multi-armed bandit problems and games like Scrabble, Bridge, Chess, and Go. These games benefit from MCTS because it accounts for the adversarial nature of Zero-Sum Games (ZSG)[1]. The tree can represent different nodes corresponding to the actions of different players. To model the games properly in the tree, MCTS uses a maximizing UCB(3.1) to determine actions at one player's nodes and a minimizing UCB (3.3) at the other player's nodes [37].

In all of these games, MCTS provides guarantees. As the number of samples in the tree grows to infinity, the policy derived from MCTS converges to the optimal state values.

Let's consider an example: a 2-person adversarial and sequential game. The first player takes an action, and the second player responds accordingly. Each player has only two actions to choose from. A tree representing this scenario is shown in Figure 3.1. Now, consider Figure 3.2, where the tree has not been fully explored. As a result, the MCTS algorithm explores the left side of the tree. Then, in Figure 3.3, backpropagation ascends the tree, updating the nodes as it progresses.

This example can be applied to an arbitrary game, and the same principles work in a game like Tic-Tac-Toe. Tic-Tac-Toe is a good small-scale example, as it has a limited

Algorithm 2: General MCTS approach

```
1: function MCTSSEARCH(s_0)
2:     create root node with state s_0
3:     while within computational budget do
4:         while bellow depth limit do
5:             TREEPOLICY
6:             DEFAULTPOLICY
7:         end while
8:         BACKUP
9:     end while
10:    return BESTCHILD
11: end function
```

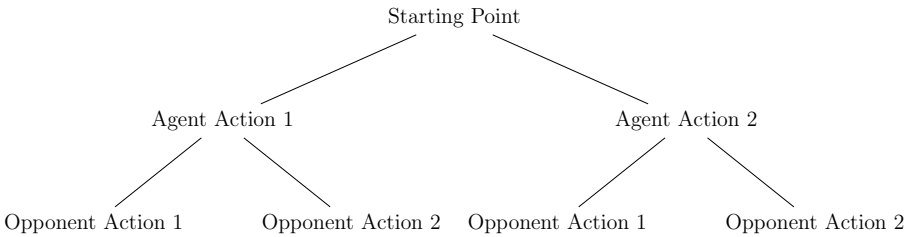

Fig. 3.1 Example Full MCTS Tree with the red line indicating discovery

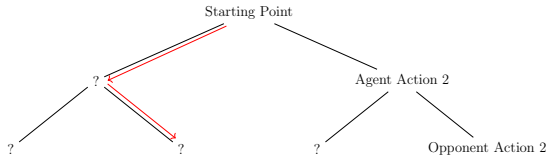

Fig. 3.2 Example of incomplete MCTS Tree

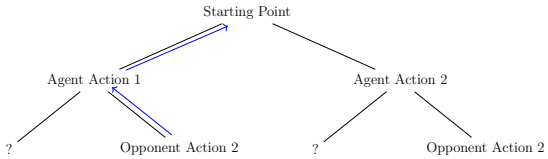

Fig. 3.3 Example MCTS Tree with the blue line indicating the backpropagation

number of actions that are played sequentially. The board for this game is shown in Figure 3.4. As the game progresses, there are fewer and fewer actions for each player to choose from. A condensed version of this tree is shown in Figure 3.5.

If MCTS were to be used for a game with a very large, but still finite, size, the state dimension problem associated with MCTS becomes apparent. As the action space grows, the state space also grows. As the state space grows, the total number of nodes in the MCTS tree increases, and it may become impossible to build and store the full tree. An example of this is chess, which has a larger board and more possible actions compared to Tic-Tac-Toe. The number of nodes required to fully build the MCTS tree for chess would be on the order of 10^{120}.

The storage issues of such trees can be mitigated by edge pruning as the tree is built one run at a time [38]. To address memory-related problems in MCTS, a class of memory-limited MCTS algorithms exists. Computer memory is typically limited to gigabytes, and only a small portion of that may be allocated to an MCTS tree, depending on the application. Video games, for example, may use MCTS trees to determine the next action of AI actors (commonly called Non-Player Characters (NPCs)), but these trees must not occupy too much space, especially if there are many different NPCs with separate trees. Dynamic memory allocation is not ideal due to the complexities of constantly pulling and pushing data to storage. Pruning and depth-limiting can help keep the tree to manageable sizes, but these are often crude and arbitrarily set solutions. More advanced methods, like "node recycling" or "garbage collection" [39], are used instead. These techniques operate a normal MCTS tree until the maximum memory limit is reached, after which unpromising nodes are removed, freeing up memory for further exploration. Unpromising nodes are those that do not terminate (given a maximum tree depth) or that consistently lead to game losses.

Depending on the game, another issue arises when the terminal state is not easily

reached. In Tic-Tac-Toe, the game ends after a maximum of nine actions, with a win, loss, or tie. This outcome can be backpropagated through the tree. However, in chess, a game could last much longer or even indefinitely, making it challenging to determine when and how to reward the system. In such cases, game length can be limited, similar to chess tournaments, with a reward or penalty applied based on the game state at the end.

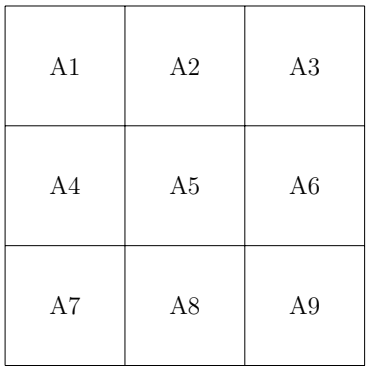

Fig. 3.4 Tic-Tac-Toe Game with grid indicating the action (A) to cover that space by either player

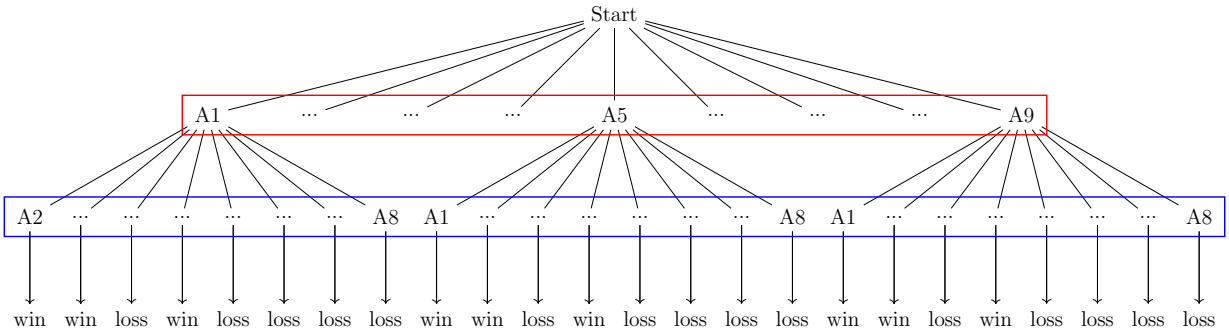

Fig. 3.5 Example MCTS Tree for Tic-Tac-Toe with alternating rows between the two players. This game is truncated: for the sake of space up to 7 rows are removed. The actions are the same as seen in Figure 3.4 for this game.

In the default utilization of MCTS, it is assumed that each state contains perfect game information. For example, when MCTS is applied to chess, the true state of the board is fully known. However, as in the distinction between MDPs (Markov Decision Processes)

and POMDPs (Partially Observable Markov Decision Processes), there is a difference when the state contains uncertainty due to partial observability. Examples of games with partial observability include Bridge or Poker, where each player observes some state information but not the full state. In these games, there is uncertainty about the opponent's cards and the remaining cards in the deck. This type of state information can be represented by an information state, which becomes the nodes in the MCTS tree. The same convergence guarantees hold for MCTS with these information states [36].

3.2.1 Alpha Go

Go is an ancient board game that dates back over 3,000 years. The game is played on a 19×19 board where players take turns placing black or white stones. The game results in a vast number of possible board configurations, as each tile can end up being occupied by either a black or white stone. Due to this immense state space, developing an algorithm capable of playing Go optimally has been a long-standing challenge [1]. Chess presents a similar challenge, as it also features a large board, numerous pieces, and many possible actions per turn, leading to an exponentially growing state space.

In Go, these complexities were addressed using memory-limited MCTS methods and tree pruning, alongside a neural network. The original AlphaGo algorithm [34] was built using several components: Supervised Learning (SL), Deep Neural Networks (DNNs), Reinforcement Learning (RL), and MCTS. Initially, an SL policy was trained on expert Go games, and this policy was further refined through self-play. The resulting policy network then generated a corresponding value network. During actual gameplay, the value of each subsequent state was derived from a combination of the value network and MCTS simulations based on the policy network. This approach created an AI that could compete at the highest levels of Go for the first time. The following year, a new version of AlphaGo, called AlphaGo Zero, was introduced [35], which omitted expert data and relied solely on MCTS for training through self-play.

3.3 MCTS+DNN Implementation

The algorithm outlined in the original AlphaGo paper [34] begins by leveraging expert knowledge to train a policy network, which is subsequently improved through self-play. This process then trains a value network, allowing an MCTS tree to be constructed without

requiring the value of each node to come solely from potential descendants reaching a terminal state. Instead, new nodes can be added to the tree, with their values derived from the policy and value networks.

In the case of Robust Shortest Path (RSP), this approach needs modification. In Problem 1, there was no expert path traversal information available to train a policy network. Therefore, in our MCTS+DNN implementation, a value network was constructed directly through self-play. This self-play enables both the agent and nature actors to develop a value estimation for each state using a single value network. During the expansion step, this DNN-derived value is utilized. As the exploration of the state space progresses, backpropagation along the tree updates the value of each state accordingly.

Since MCTS trees converge to the true solution only when the entire tree is fully built, it is not always feasible to construct the complete tree in every scenario. For this implementation, MCTS will expand the tree a hundred times, followed by backpropagation, before taking each action. In smaller worlds, such as in Figure 2.4, this may be excessive and can be scaled down; however, in larger-scale examples, this level of expansion will be necessary.

Chapter 4

Numerical Examples

This section is structured as follows. Section 4.1 provides an overview of the three algorithms being tested. These algorithms are evaluated on six benchmarks, discussed in Section 4.2. These grid-world benchmarks vary in complexity, incorporating different sizes, numbers of obstacles, and feasible sets. These environments were custom built for our work to specifically show how the algorithms work at different scales of size and uncertainty. The results are presented in Section 4.3, followed by a discussion in Section 4.4.

4.1 Baseline Algorithms Compared

3 Algorithms will be used in the comparison of results for this work. They are as follows:

1. **DNN + MCTS:** This method is described in Section 3.3, where a DNN is used to approximate next state values in conjunction with a finite-depth MCTS with a depth of 50. When a new expansion occurs in the MCTS tree, state approximation is performed, followed by backpropagation in the tree. For smaller models, the full MCTS tree is less than this depth, while for larger models, it exceeds this depth. Larger paths are truncated at 50 actions; it is assumed that after the last action is taken, the episode ends and incurs a terminal cost of 100.

2. **Modified Dijkstra's:** This heuristic method employs a modified Dijkstra's shortest path algorithm to find the cheapest path within the environment's possible graphs. The algorithm assumes that the true graph is the one with the cheapest possible path from start to end. As the agent follows this shortest path, it makes observations

similar to the other two algorithms. If it makes an observation inconsistent with the assumed true graph, it updates the set of possible graphs to align with this and all previous observations. The agent then re-runs Dijkstra's to find the new cheapest path among the updated set of possible graphs. This process is repeated until the terminal node is reached.

This method is heuristic in nature and does not guarantee finding the robust optimal path; however, it serves as a low-complexity benchmark for comparing the more sophisticated algorithms.

The VI, Modified Dijkstras and DNN + MCTS algorithms were implemented from scratch. The implementation of the modified Dijkstra's algorithm was based on the python library "NetworkX" [40]. The DNN for the MCTS model were built using Pytorch [41].

4.2 Benchmark Models

4.2.1 Model 1

Model 1 presents a 5×5 grid world, illustrated in Figure 4.1, alongside its equivalent graph shown in Figure 4.2. In this model, one edge has an uncertain existence: the edge between nodes 23 and 24 may either have a cost of 1 or be non-existent, indicating an infinite cost. All other edges have a consistent cost of 1. This world accommodates two possible weights for the true graph, denoted as $\mathcal{L} = \{1, 2\}$, as depicted in Figures 4.3 and 4.4.

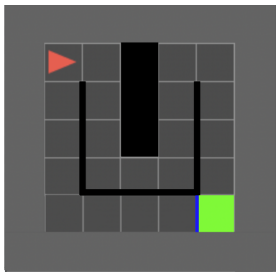

Fig. 4.1 5×5 model 1 grid-world with uncertain edges in blue

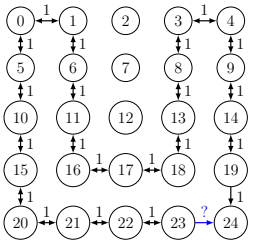

Fig. 4.2 Graph model 1

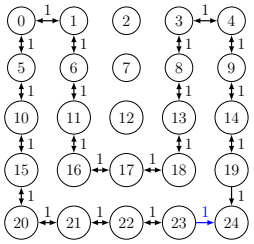

Fig. 4.3 Model 1 graph $\mathcal{G}_1$

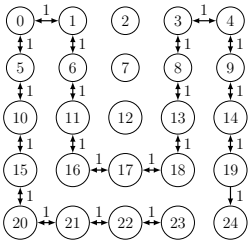

Fig. 4.4 Model 1 graph $\mathcal{G}_2$

4.2.2 Model 2

Model 2 presents a 5×5 grid world, as shown in Figure 4.5, along with its equivalent graph in Figure 4.6. In this model, there are 3 uncertain edges, with up to 2 of these edges potentially being non-existent. All existing edges are assigned a cost of 1. This world accommodates 7 possible true weights for the graph, denoted as $\mathcal{L} = \{1, 2, 3, 4, 5, 6, 7\}$. The three graphs where there are 2 blocked edges are illustrated in Figures 4.7, 4.8, and 4.9.

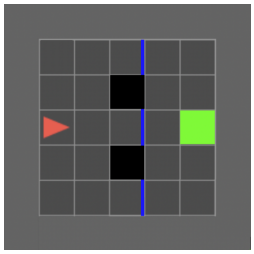

Fig. 4.5 Model 2 grid-world with uncertain edges in blue

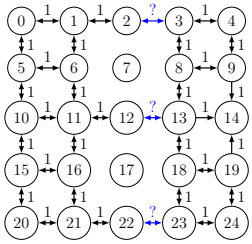

Fig. 4.6 Model 2 graph

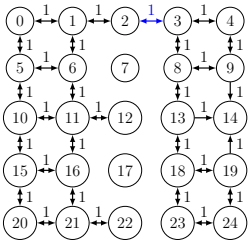

Fig. 4.7 Model 2 graph $\mathcal{G}_1$

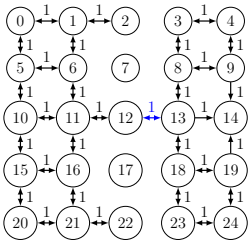

Fig. 4.8 Model 2 graph $\mathcal{G}_2$

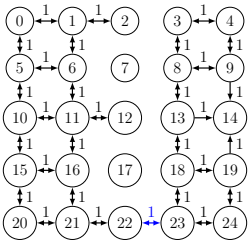

Fig. 4.9 Model 2 graph $\mathcal{G}_3$

4.2.3 Model 3

Model 3 presents a 4×7 grid world, as shown in Figure 4.10, along with its equivalent graph in Figure 4.11. In this world, there are 4 uncertain edges, each representing a door with a unique cost for opening it, which is initially unknown. The costs of the doors are drawn from the set $\{1, 5, 10, 20\}$. All other edges are valued at 1, except for the edge from nodes 12 to 13, which has a value of 7. This setup results in the set $\mathcal{L} = \{1, 2, \ldots, 24\}$, with 3 sample cases illustrated in Figures 4.12, 4.13, and 4.14. For this model, we will present only 3 sample graphs. Models 5 and 6 will also include more than 3 such graphs, but it becomes impractical to present all of them as the number of sets increases.

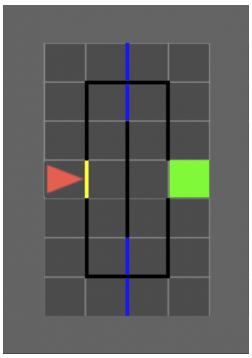

Fig. 4.10 Model 3 grid-world

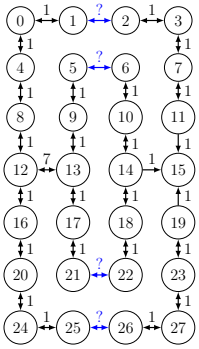

Fig. 4.11 Model 3 graph

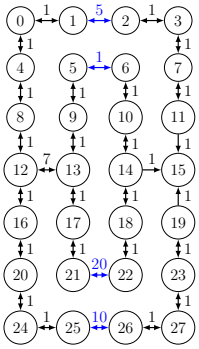

Fig. 4.12 Model 3 graph $\mathcal{G}_1$

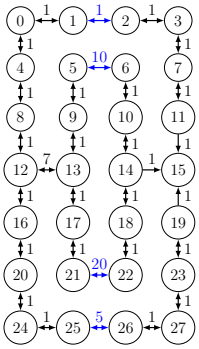

Fig. 4.13 Model 3 graph $\mathcal{G}_2$

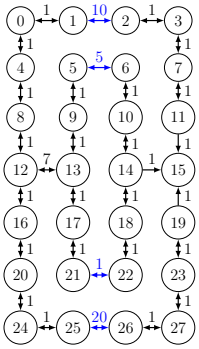

Fig. 4.14 Model 3 graph $\mathcal{G}_3$

4.2.4 Model 4

Model 4 presents a 7×7 grid world, as seen in Figure 4.15, along with its equivalent graph in Figure 4.16. In this world, the certain edges are directed, with some edges valued at 10 while others are valued at 1. The values of 3 pairs of directed edges are initially unknown. There are 3 possible graphs: either only edge $(40, 41)$ is untraversable, $(47, 48)$ is untraversable, or both are untraversable. In the latter case, edge $(27, 34)$ costs 3 instead of 1. This results in the set $\mathcal{L} = \{1, 2, 3\}$, with 3 sample cases illustrated in Figures 4.17, 4.18, and 4.19.

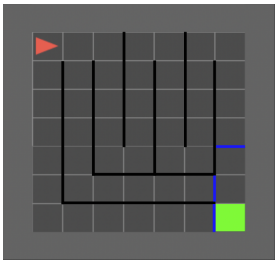

Fig. 4.15 Model 4 grid-world with uncertain edges in blue and edges of cost 7 in yellow

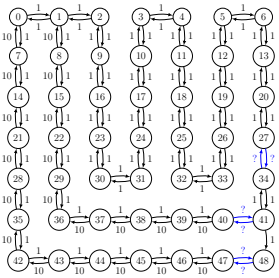

Fig. 4.16 Model 4 graph

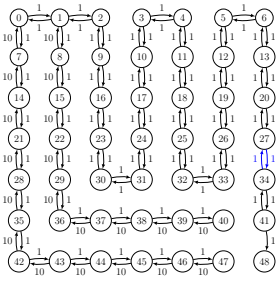

Fig. 4.17 Model 4 graph $\mathcal{G}_1$

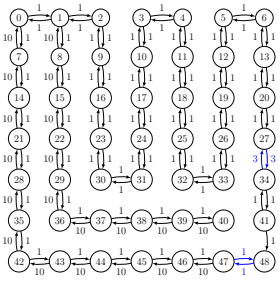

Fig. 4.18 Model 4 graph $\mathcal{G}_2$

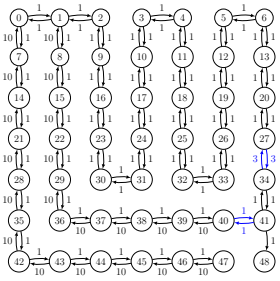

Fig. 4.19 Model 4 graph $\mathcal{G}_3$

4.2.5 Model 5

Model 5 presents a 7×7 grid world, as seen in Figure 4.20, along with its equivalent graph in Figure 4.21. In this world, there are 9 uncertain edges, which exist in 27 possible configurations. In each case, 2 edges do not exist in each row, resulting in 3 possible obstacle states per row. Given there are 3 rows, the set is defined as $\mathcal{L} = \{1, 2, \ldots, 27\}$. Each edge that exists in the graph has a weight of 1. Three sample graphs can be seen in Figures 4.22, 4.23, and 4.24.

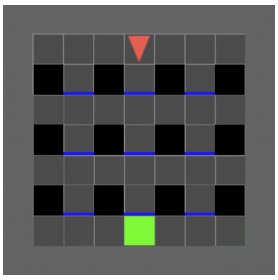

Fig. 4.20 Model 5 grid-world with uncertain edges in blue

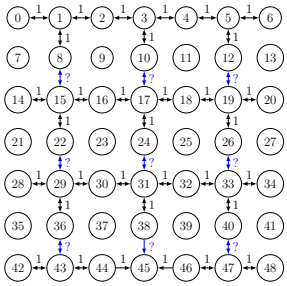

Fig. 4.21 Model 5 graph

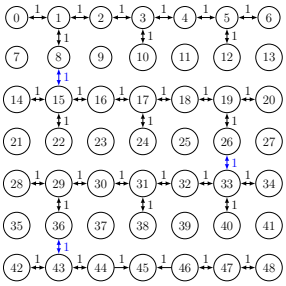

Fig. 4.22 Model 5 graph $\mathcal{G}_1$

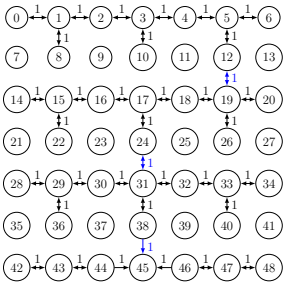

Fig. 4.23 Model 5 graph $\mathcal{G}_2$

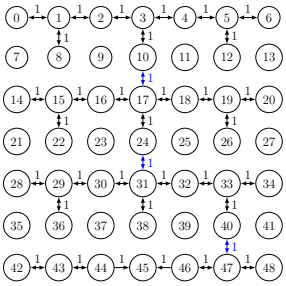

Fig. 4.24 Model 5 graph $\mathcal{G}_3$

4.2.6 Model 6

Model 6 presents a 10×10 grid world, as seen in Figure 4.25, along with its equivalent graph in Figure 4.26. In this world, there are 8 uncertain edges. The value of each uncertain edge

must be within the set $\{1, 2, 5\}$, with the total cost of all the uncertain edges equaling 30. Aside from the uncertain edges, there exists a path of edges along the outside of the world, where all edges are valued at 2, as illustrated in Figure 4.25. Additionally, the edge $(89, 99)$ is valued at 4, while all other existing edges are valued at 1. This creates a feasible set of 168 elements, where $\mathcal{L} = \{1, 2, \ldots, 168\}$. Three sample weighted edge graphs can be seen in Figures 4.27, 4.28, and 4.29.

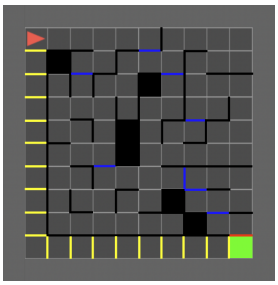

Fig. 4.25 Model 6 grid-world with uncertain edges in blue, edges of cost 2 in yellow and edges of cost 4 in red.

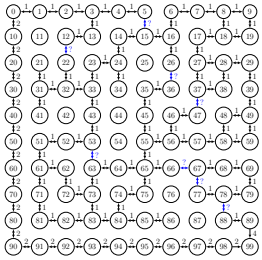

Fig. 4.26 Model 6 graph

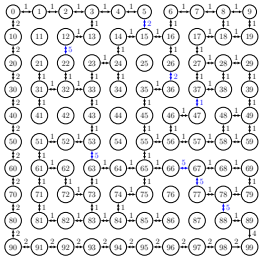

Fig. 4.27 Model 6 graph $\mathcal{G}_1$

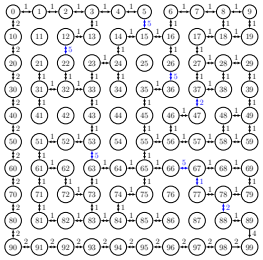

Fig. 4.28 Model 6 graph $\mathcal{G}_2$

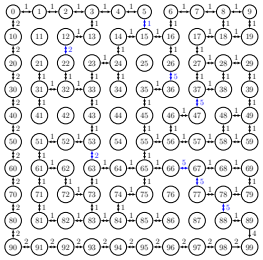

Fig. 4.29 Model 6 graph $\mathcal{G}_3$

4.2.7 Pruning Models

The following Figures 4.30, 4.31, 4.32, 4.33, 4.34, and 4.35 display the modified pruned versions of models $1-6$. The nodes that are removed due to pruning are shown in black. It

can be seen only the models in Figures 4.30, and 4.35 have actually been modified from the pruning. This is because models 2, 3, and 4 do not have any nodes that could be removed per the pruning rules in Section 2.4.

In the cases of model 1, the pruned nodes are removed for similar reasons as in Example 4. There are graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ such that $w_1(n,m) \leq w_2(n,m)$. This removes graph $\mathcal{G}_1$. Then edges:

$$\{(0,5),(5,10),(10,15),(15,20),(20,21),(21,22),(22,23)\}$$

in $\mathcal{G}_2$ can be removed as they never exist on a path which has a $D_{\min}$ through any of those nodes which costs less than the $C_{\max}(p_0)$ where:

$$p_0 = \{0,1,6,11,16,17,18,13,8,3,4,9,14,19,24\}.$$

Finally, after all the pruning conducted so far, nodes:

$$\{2,5,7,10,12,15,20,21,22,23\}$$

never exist on a path from n^s to n^d and can be removed.

The nodes removed in model 5 are removed since they are never on any path from n^s to n^d. There are just 12 nodes which meet this requirement. They are:

$$\{7,9,11,13,21,23,25,27,35,37,41,\}.$$

There are no other edges or graph that can be removed due to the structure of this model. For example, there exists no finite-cost $C_{\max}$ for any p_0 as they would all have a worst-case value of ∞. In the case of model 6, the edge

$$\{(7,8),(46,47),(57,58),(84,85),(85,86)\}$$

never exists on a path which has a $D_{\min}$ lower than the $C_{\max}(p_0)$ which goes along all nodes valued at 2 forming path:

$$p_0 = \{0,10,20,30,40,50,60,70,80,90,91,92,93,94,95,96,97,98,99\}.$$

The fact that edges $\{(7,8),(46,47),(57,58)\}$ are be pruned means nodes:

$$\{8,9,17,18,19,27,28,29,37,38,39,46,47,48,49,58,59\}$$

are no longer attainable along a path from $n^s = 0$ to $n^d = 99$. Along with this, the removal of edges $\{(84,85),(85,86)\}$ removes all edges going to nodes $\{85,86\}$ which allows these two nodes to be pruned. Finally, nodes $\{11,25,42,44,54,76,87\}$ in no graph have finite-cost edges that can be removed. The pruning of uncertain edge $(37,47)$ reduces the uncertainty. There exists no best-case or worst-case path for the agent to take through this edge. Nature would therefore always play its least costly edge between these two nodes. This ends up reducing the uncertainty from 168 possible combinations to 21.

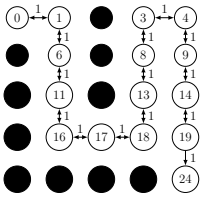

Fig. 4.30 Model 1 pruned with $\mathcal{G}_1$.

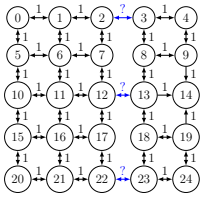

Fig. 4.31 Model 2 pruned with $\mathcal{G}_1$.

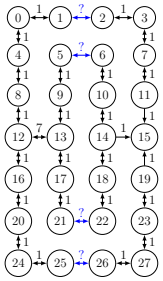

Fig. 4.32 Model 3 pruned with $\mathcal{G}_1$.

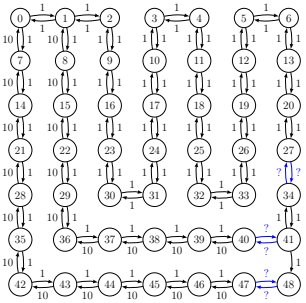

Fig. 4.33 Model 4 pruned with $\mathcal{G}_1$.

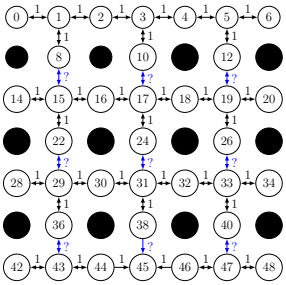

Fig. 4.34 Model 5 pruned with $\mathcal{G}_1$.

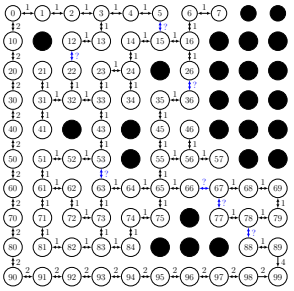

Fig. 4.35 Model 6 pruned with $\mathcal{G}_1$.

4.2.8 Size Comparison

Tables 4.1 and 4.2 show a breakdown of the number of nodes, certain edges, uncertain edges, and number of graphs in each Model. Across the models the number of edges, nodes, or graphs are increased. This is mitigated by the pruning of each model.

Table 4.1 Size tables for Models 1-3

	Model 1	Model 1 Pruned	Model 2	Model 2 Pruned	Model 3	Model 3 Pruned
Number of Nodes	25	15	23	23	28	28
Number of Certain Edges	21	14	35	35	26	26
Number of uncertain Edges	1	0	3	3	4	4
Number of Graphs	2	1	3	3	24	24

Table 4.2 Size tables for Models 4-6

	Model 4	Model 4 Pruned	Model 5	Model 5 Pruned	Model 6	Model 6 Pruned
Number of Nodes	49	49	49	37	100	74
Number of Certain Edges	48	48	33	33	96	75
Number of uncertain Edges	3	3	9	9	8	7
Number of Graphs	3	3	27	27	168	21

4.3 Results

The results will be presented to show the policy evaluation from the 6 different models across the 3 different algorithms, with and without node pruning. The overall methods will be summarized in Table 4.3. For the DNN + MCTS method, an additional graph for each model world will present the results from the training after each episode, averaged over 5 parallel runs. The time to run one instance of each algorithm is shown in Table 4.4.

Both the VI and Modified Dijkstra's methods have deterministic outputs, resulting in no variation between runs. The only variation arises from the 5 different random start seeds used in the DNN in the DNN + MCTS model. This variation will be represented by the standard deviation in the DNN + MCTS results.

Table 4.3 Policy Evaluation of Different Models

	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
Value Iteration	14	16	N/A	30	N/A	N/A
Value Iteration Pruned	14	16	N/A	30	N/A	N/A
Modified Dijkstra's	28	16	26	250	42	41
Modified Dijkstra's Pruned	14	16	26	250	42	41
DNN + MCTS	14 ± 0	16 ± 0	26 ± 0	30 ± 0	34.8 ± 1.8	36 ± 0
DNN + MCTS Pruned	14 ± 0	16 ± 0	26 ± 0	30 ± 0	34.8 ± 1.8	36 ± 0

Table 4.4 Computational Time Comparison of Different Models (Seconds)

	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6
Value Iteration	1.36	13.12	N/A	4.20	N/A	N/A
Value Iteration Pruned	1.34	2.28	N/A	4.63	N/A	N/A
Modified Dijkstra's	1.12	1.25	1.82	1.61	69.37	376.72
Modified Dijkstra's Pruned	1.28	1.63	4.08	1.98	117.78	117.68
DNN + MCTS	10.52	12.03	429.05	215.02	1,890.58	20,100.57
DNN + MCTS Pruned	11.77	13.73	432.79	218.17	1,938.99	755.02

4.3.1 DNN + MCTS Training Curves

The following figures represent the policy training curves for the DNN + MCTS. The values presented in the graphs indicate the policy of the agent under its worst possible conditions, specifically the value that is highest across all possible graphs. In all of the graphs, the line labeled "Optimal Cost" is derived either from the VI algorithm or by inspection.

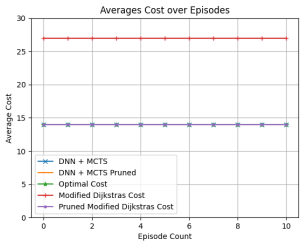

Fig. 4.36 Model 1

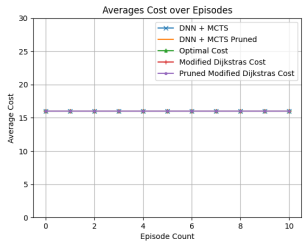

Fig. 4.37 Model 2

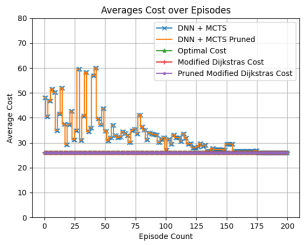

Fig. 4.38 Model 3

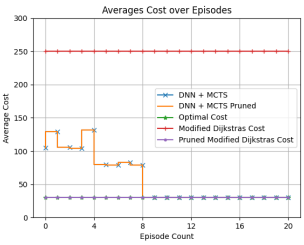

Fig. 4.39 Model 4

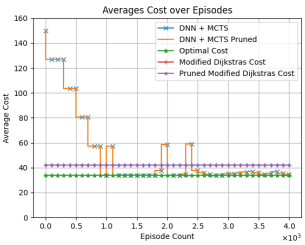

Fig. 4.40 Model 5

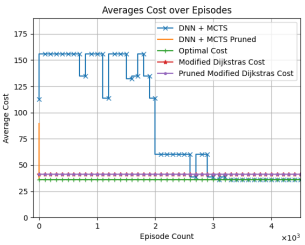

Fig. 4.41 Model 6

4.4 Summary of Results

Given the results of the three algorithms across the different models, Tables 4.3 and 4.4 show that robust algorithms like the VI and DNN + MCTS models can find the optimal robust solution, unlike greedy algorithms such as Modified Dijkstra's. The only cases where Modified Dijkstra's finds the optimal policy are when the shortest path in the worst-case graph is identical to that in the best-case graph, with or without pruning. This is evident in Models 1, 2, and 3. If there is unrecoverable deviation from the best-case and worst-case graphs in a greedy manner, then algorithms like Modified Dijkstra's do not succeed in finding the optimal path. They can find a finite-cost path, but it might significantly deviate from the optimal cost.

As the environment scales up with larger graphs and more possible configurations, the limitations of VI become apparent. Building all required feasible states eventually becomes unfeasible, as the time taken to create all possible states increases. This was why Models 3, 5, and 6 could not be tested, as even different combinations of feasible states could not be adequately stored in memory at the required scale. Even though Model 4 is larger than Model 3 in terms of the scale of the graph as it does contain less uncertainty. Due to this, VI is able to find the solution to Model 4.

Across Models 1, 2, 3, 4, and 6, the DNN + MCTS solution consistently found the optimal solution. The only exception is in Model 5. In Figure 4.40, after about 3000 episodes, the average path value varied between 34 and 38, with the final value of the path taken being 34.8. While Model 5 does not have the largest graph, Model 6 features a larger graph and a greater number of possible configurations due to uncertainty. Yet, the DNN + MCTS algorithm was able to always find the optimal path in Model 6, unlike in Model 5. One reason for this is that in Model 5, more opportunities exist to deviate slightly from the optimal path to a sub-optimal one. Model 5 has the most nodes with three or more edges along the optimal path, and at each of these nodes, there are only slight differences in cost when taking any sub-optimal edge. Since the DNN + MCTS model provides approximations for the value of each edge, this opens the door to taking slightly sub-optimal paths. In contrast, Model 6's optimal path strictly follows edges valued at 2, with no nodes along this path having two or more edges. This error could be reduced with further DNN model training or more extensive exploration of the MCTS tree.

Across different graph sizes and uncertainties, in some cases, the DNN + MCTS algorithm found the optimal cost with randomly initialized DNN weights. This was true for Models 1 and 2, which had small graphs and a lower value for L. In this work, each MCTS was given the same amount of expansion rounds regardless of the size of the space. In these cases, the full MCTS tree was built each time before the first action was taken. If the total number of expansions had been severely limited, the DNN + MCTS method would not have found the solution using random DNN weights in Models 1 and 2. In larger-scale Models 3-6, the full tree could never be built, so the algorithm relied more heavily on DNN approximations.

Across all examples shown, the Modified Dijkstra's algorithm is considerably faster at finding solutions. The algorithm generally takes orders of magnitude less time to run, as seen in Table 4.4. Unfortunately, the paths found can perform significantly worse than those

in the other two models, as shown in Table 4.3. Between the three algorithms—VI, Modified Dijkstra's, and DNN + MCTS—the latter is most capable of handling increasingly larger graphs with higher levels of uncertainty while still finding an optimal or nearly optimal path.

4.5 Detailed Discussion of Results

Model 1

The policies for the VI, Modified Dijkstra's, and DNN + MCTS algorithms in $\mathcal{G}_1$ can be seen in Figures 4.42, 4.43, 4.44, 4.45, 4.46, and 4.47. In this case, all three algorithms did not find the optimal path in any of the non-pruned scenarios (Figures 4.42, 4.43, and 4.44). This is partly due to the structure of the uncertainty and how nature will play. It is always optimal for nature to block the only uncertain edge if the agent approaches. Therefore, since a greedy algorithm like Modified Dijkstra's does not take this into account, it follows the shortest path in the best-case scenario, which ends up being impassable.

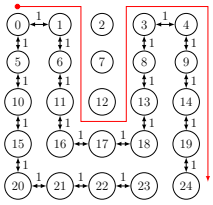

Fig. 4.42 Model 1 with $\mathcal{G}_1$ with VI solution.

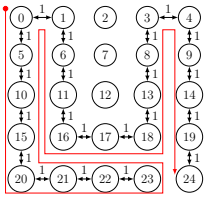

Fig. 4.43 Model 1 with $\mathcal{G}_1$ with Modified Dijkstra's solution.

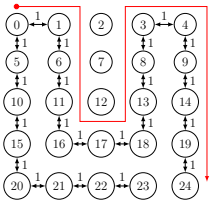

Fig. 4.44 Model 1 with $\mathcal{G}_1$ with DNN + MCTS solution.

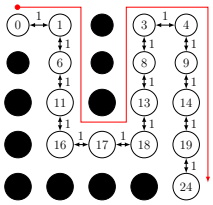

Fig. 4.45 Model 1 pruned with $\mathcal{G}_1$ with VI solution.

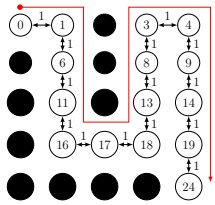

Fig. 4.46 Model 1 pruned with $\mathcal{G}_1$ with Modified Dijkstra's solution.

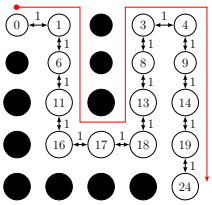

Fig. 4.47 Model 1 pruned with $\mathcal{G}_1$ with DNN + MCTS solution.

The pruning method reduces the overall uncertainty in this model to a single possible $\mathcal{G}_1$, as seen in Figures 4.45, 4.46, and 4.47. While this does not significantly impact the computation times for any of the models, as seen in Table 4.4, the time it takes to run just the VI algorithm in this case is similar to running the VI with the pruning method. This is due to the fact that Model 1 is not large and there are not many graphs to be removed, so the benefit of running pruning with VI is very small. A larger impact is seen in the policy evaluation of the Modified Dijkstra's algorithm, where reducing the uncertainty allows Modified Dijkstra's to find the correct solution, as shown in Figure 4.46, resulting in identical values for all models, as seen in Table 4.3.

The training curves for the DNN + MCTS in Model 1 can be seen in Figure 4.36. Given the small dimensions of the environment and the fact that there are only 2 possible graphs, $\mathcal{G}_1$ and $\mathcal{G}_2$, the MCTS method was able to find the solution with randomly initialized DNN values, both with and without pruning.

Model 2

The policies for the VI, Modified Dijkstra's, and DNN + MCTS algorithms in $\mathcal{G}_1$ can be seen in Figures 4.48, 4.49, and 4.50. In this case, all three algorithms found the optimal paths with and without pruning. This was consistent across all possible $\mathcal{G}_\ell$, even when nature is not using the worst-possible graph, as seen in $\mathcal{G}_1$ and $\mathcal{G}_2$. This is partly due to the structure of the uncertainty and how nature will play. It is always optimal for nature to block the first two unique uncertain edges that the agent will approach. In this case, it is optimal for the agent to start moving towards one of the three obstacles. When that

is blocked, the agent moves to the next closest until the final, initially uncertain edge is visited. This is the solution found by the VI Algorithm. This is identical to a greedy strategy, meaning algorithms like Modified Dijkstra's will find the same solution as the VI and DNN + MCTS Algorithms.

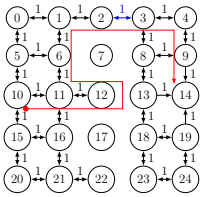

Fig. 4.48 Model 2 with $\mathcal{G}_1$ with the VI, Modified Dijkstras and DNN + MCTS solutions.

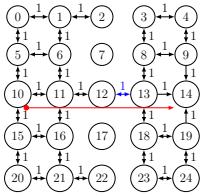

Fig. 4.49 Model 2 with $\mathcal{G}_2$ with the VI, Modified Dijkstras and DNN + MCTS solutions.

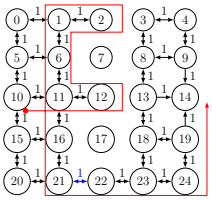

Fig. 4.50 Model 2 with $\mathcal{G}_3$ with the VI, Modified Dijkstras and DNN + MCTS solutions.

The pruning method does not impact this model at all. No nodes can be removed, and therefore running the pruning method slightly increases the total time cost of the algorithms, as seen in Table 4.4, with no impact on the path values, as shown in Table 4.3.

The training curves for the DNN + MCTS can be seen in Figure 4.37. For similar reasons as in Model 1, the MCTS method was able to find the solution with randomly initiated DNN values.

Model 3

The policies for the VI, Modified Dijkstra's, and DNN + MCTS algorithms in $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_3$ can be seen in Figures 4.51, 4.52, and 4.53. In this case, only 2 algorithms, Modified Dijkstra's and DNN + MCTS, could actually solve the policy. There exist 24 possible graphs, which means that for VI, there are $2^{24} \approx 1.6 \times 10^7$ feasible sets. This is then compounded by the 28 possible nodes, creating $2^{24} \times 28 \approx 4.7 \times 10^9$ possible states for the agent to exist in. As previously explained, this cannot be loaded into the memory of the computer.

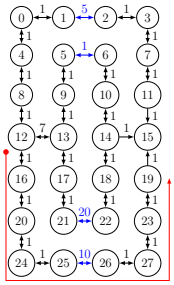

Fig. 4.51 Model 3 pruned with $\mathcal{G}_1$ for both the Modified Dijkstra's and DNN + MCTS solution.

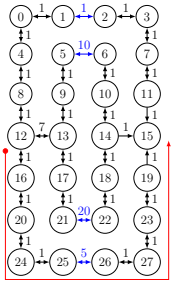

Fig. 4.52 Model 3 pruned with $\mathcal{G}_2$ for both the Modified Dijkstra's and DNN + MCTS solution.

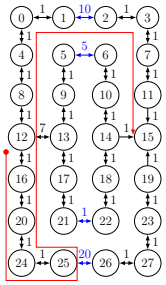

Fig. 4.53 Model 3 pruned with $\mathcal{G}_3$ for both the Modified Dijkstra's and DNN + MCTS solution.

Between the two models that were run, Modified Dijkstra's and DNN + MCTS, both were able to find the optimal policy. Regardless of the initial direction of the agent, it is always optimal for nature to use the highest edge cost obstacle at the first uncertain edge the agent approaches. Nature will then play the next highest remaining edge at each subsequent turn. This leads the agent's optimal policy to first move to the closest obstacle and then to the next one after observing the cost of the first uncertain edge. This is because moving from the first uncertain obstacle to the next may incur a lower cost than crossing the obstacle valued at 20.

The pruning method does not impact this model at all; therefore, running the pruning method followed by the algorithms actually slightly increases the total time cost, as seen in Table 4.4, and does not affect the path values, as shown in Table 4.3.

The training curves for the DNN + MCTS can be seen in Figures 4.38. In this case, the randomly generated DNN weights for episode 0 are insufficient for the algorithm to find the optimal path. Given that there are $\sim 4.7 \times 10^9$ possible states for the agent, the MCTS tree has this many possible nodes. This causes the algorithm to depend significantly on the approximation values from the DNN, which can only improve after several training episodes, unlike in Models 1 and 2.

Model 4

The policies for the VI, Modified Dijkstra's, and DNN + MCTS algorithms in $\mathcal{G}_1$, $\mathcal{G}_2$, and $\mathcal{G}_3$ can be seen in Figures 4.54, 4.55, 4.56, 4.57, 4.58, and 4.59. Model 4 is structurally similar to Model 1. There is one longer path that has no obstacles, while there are shorter paths that contain obstacles. The optimal policy is successfully identified by both the VI and DNN + MCTS algorithms. However, since the Modified Dijkstra's algorithm does not account for uncertainty, it greedily assumes that the shortest path will not have any obstacles. Nature consistently chooses to block the two shorter paths, which forces the agent to take a path that navigates through the top side of the graph.

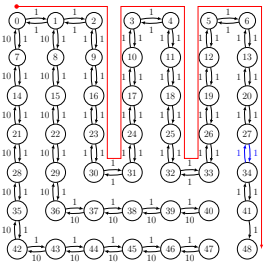

Fig. 4.54 Model 4 with $\mathcal{G}_1$ with VI and DNN + MCTS solution.

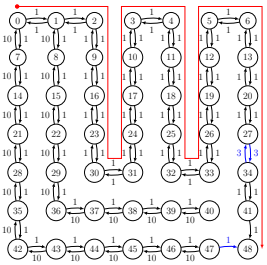

Fig. 4.55 Model 4 with $\mathcal{G}_2$ with VI and DNN + MCTS solution.

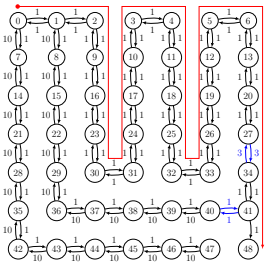

Fig. 4.56 Model 4 with $\mathcal{G}_3$ with VI and DNN + MCTS solution.

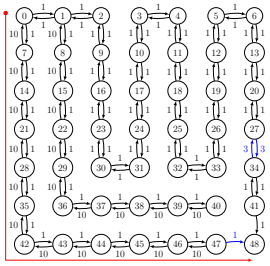

Fig. 4.57 Model 4 with $\mathcal{G}_1$ with Modified Dijkstra's solution.

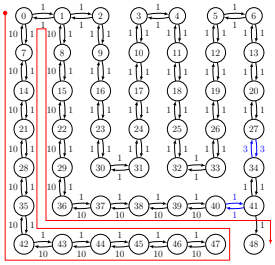

Fig. 4.58 Model 4 with $\mathcal{G}_2$ with Modified Dijkstra's solution.

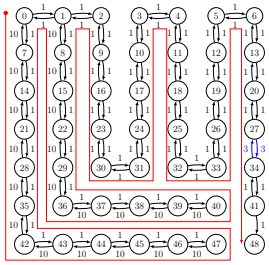

Fig. 4.59 Model 4 with $\mathcal{G}_3$ with Modified Dijkstra's solution.

As shown in Table 4.3, Modified Dijkstra's is never able to find the optimal policy, as the pruning method has no impact on this model. The cost path taken by Modified

Dijkstra's is valued at 250, which is considerably worse than the optimal solution valued at 30.

In this model, pruning does not change the scenario at all and therefore increases the computation time, as seen in Table 4.4. The training curves for the DNN + MCTS can be seen in Figure 4.39. In this case, the randomly generated DNN weights for episode 0 are insufficient for finding the optimal path. However, after 8 episodes, the DNN + MCTS successfully values the states, allowing it to identify the optimal path.

Model 5

The policy solutions for Model 5 given $\mathcal{G}_1$ are represented in Figures 4.60, 4.61, 4.62, 4.63, 4.64, and 4.65. These results are only for the Modified Dijkstra's solution and the DNN + MCTS solution. This is due to the fact that in Model 5, there exist 27 possible graphs. This would create $2^{27} \approx 1.34 \times 10^8$ feasible sets, which is then compounded by the fact that there are 37 nodes that the agent can reside at. Therefore, the total number of state values that the VI solution would need to find is on the order of $2^{27} \times 37 \approx 4.97 \times 10^9$. Just like in Model 3, it was not possible for this number of states to even be loaded into memory, and if it were, the computation time would have been on the scale of days or weeks. Due to this, VI was not used to find a solution in this case.

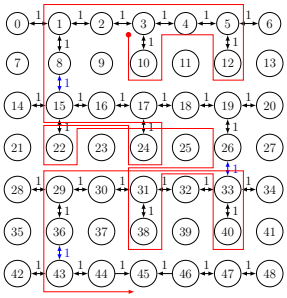

Fig. 4.60 Model 5 with $\mathcal{G}_1$ using Modified Dijkstra's solution.

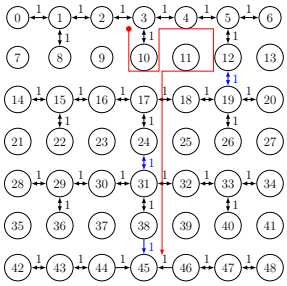

Fig. 4.61 Model 5 with $\mathcal{G}_2$ using Modified Dijkstra's solution.

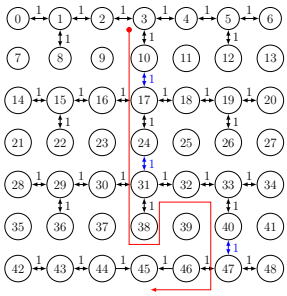

Fig. 4.62 Model 5 with $\mathcal{G}_3$ using Modified Dijkstra's solution.

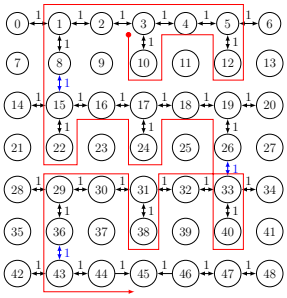

Fig. 4.63 Model 5 with $\mathcal{G}_1$ using DNN + MCTS solution.

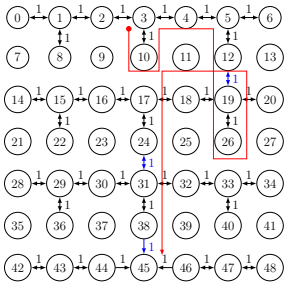

Fig. 4.64 Model 5 with $\mathcal{G}_2$ using DNN + MCTS solution.

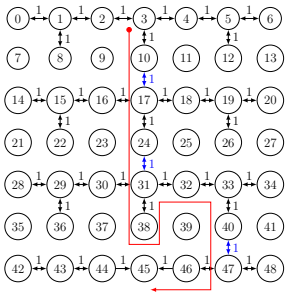

Fig. 4.65 Model 5 with $\mathcal{G}_3$ using DNN + MCTS solution.

Regardless of this limitation, Modified Dijkstra's and DNN + MCTS are usable in this case. The costs of their paths can be seen in Table 4.3. Between these two solutions, only DNN + MCTS finds the optimal solution given the uncertainty in the space. Note that Modified Dijkstra's could have found the optimal solution in graph $\mathcal{G}_1$ as seen in Figure 4.60. When the agent was at node 15, it had two greedy options. It could have moved down to node 22 to try to take the path through nodes $\{22, 29, 36, 43, 44, 45\}$, which costs identically to the path going through node 24, which would go through nodes $\{16, 17, 24, 31, 38, 45\}$. Depending on the implementation of Modified Dijkstra's, it will choose differently between those two options. These two paths are identical to a shortest path algorithm such as Modified Dijkstra's, which does not take uncertainty into account. The other graphs $\mathcal{G}_2$ and $\mathcal{G}_3$ represent cases where the path is not the worst-case given the policies of Modified Dijkstra's and DNN + MCTS. Because of this, in graph $\mathcal{G}_3$, the two algorithms find identical paths, and in $\mathcal{G}_2$, Modified Dijkstra's finds a shorter path. But as can be seen in the results from the worst-case policy evaluation in Table 4.3, Modified Dijkstra's does not outperform DNN + MCTS.

The training curves for the DNN + MCTS model can be seen in Figure 4.40, where over time the DNN's prediction improves, allowing for the optimal value to be found with the MCTS algorithm. It takes both algorithms about 3000 training episodes before finally settling down to their final average value. This final average value was not exactly the optimal solution. The final result is 34.8 ± 1.8. Over the course of the last 1000 training episodes, various runs tend to overfit, causing inaccurate actions to be taken along the path. The final output of the 5 runs was $\{34, 34, 34, 34, 38\}$. By inspection of the model, it can be seen that the optimal value for the worst-case path is $\{34\}$. Four of the five runs

had found this value. Given considerably more training episodes, the final five runs could all converge to this value of 34.

Model 6

The policy solutions for model 6 given $\mathcal{G}_1$ are represented in Figures 4.66, 4.67, 4.68, and 4.69. These results are only for Modified Dijkstra's solution and for the DNN + MCTS solution. This is due to the fact that in model 6, there exist 168 possible graphs. This would create $2^{168} \approx 3.74 \times 10^{50}$ feasible sets, which is then compounded by the fact that there are 93 nodes that the agent can reside at. Therefore, the total number of state values that the VI solution would need to find is on the order of $2^{168} \times 93 \approx 3.48 \times 10^{52}$. Just like in models 3 and 5, it was not possible for this number of states to even be loaded into memory, and if it were, the computation time would have been on the scale of days or weeks. Due to this, VI was not used to find a solution in this case.

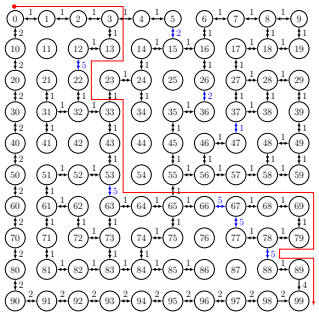

Fig. 4.66 Model 6 with $\mathcal{G}_1$ with Modified Dijkstra's solution.

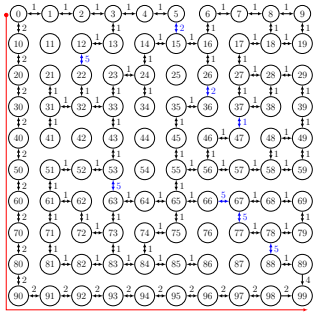

Fig. 4.67 Model 6 with $\mathcal{G}_1$ with DNN + MCTS solution.

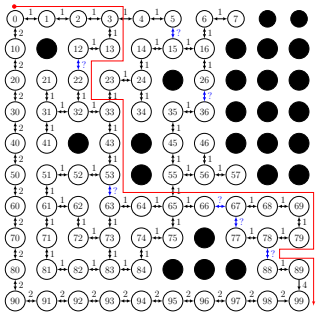

Fig. 4.68 Model 6 with pruned $\mathcal{G}_1$ with Modified Dijkstra's solution.

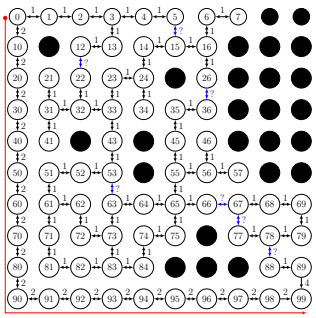

Fig. 4.69 Model 6 with pruned $\mathcal{G}_1$ with DNN + MCTS solution.

On the other hand, Modified Dijkstra's and DNN + MCTS are usable in this case. Between these two solutions, only DNN + MCTS finds the optimal solution given the uncertainty in the space. The way that this environment is constructed is set up such that it is never optimal in the worst-case graph $\mathcal{G}_{\ell^*}$ to go through the middle of the graph, just like Modified Dijkstra's did in Figures 4.66 and 4.68. In the worst cases, the optimal path can be seen in Figures 4.67 and 4.69, with a path valued at 36. Modified Dijkstra's, or another greedy type of algorithm, will go through the middle because in the best cases it would only cost 34. However, in the worst case, it will cost 38. Only the DNN + MCTS method was able to take the worst-case optimal path. These results can be seen in Table 4.3.

It takes about 3100 training episodes for the unpruned DNN + MCTS method to find the optimal solution. Over time, the training curve can be seen in Figure 4.41. Pruning, in this case, affects the training time for the DNN + MCTS method considerably, as it is then able to find the optimal solution within a hundred training episodes. There is no effect on the result of the Modified Dijkstra's solution, as the pruning in this environment does not remove any nodes or edges that the sub-optimal path taken by the Modified Dijkstra's method traverses. The effect seen in both algorithms from the pruning is that the computation time required decreases for both, as seen in Table 4.4.

Chapter 5

Conclusion

This thesis has focused on the development of algorithms to solve a Robust Shortest Path (RSP) problem using a hybrid planning model in a discrete set of possible uncertain worlds by taking advantage of local information revelation. This problem differentiates from other RSP problems, as other models entail entirely offline planning over a set of uncertain states that have some distribution.

To solve this uncertain path-planning challenge, a Value Iteration (VI) algorithm was developed to compute a path-planning solution in a hybrid offline/online manner. The path policy depends on the agent's current information state, which is based on the feasible states of the world. While the policy is computed offline, the information state is updated as local information is revealed. The policy then uses this updated state to determine the next action until the terminal state is reached. This algorithm is guaranteed to compute the optimal robust policy in a given world but has the limitation of requiring increasingly longer compute time and resources as the uncertainty and size of the space increase.

Due to the computational intensity of VI, a more efficient approach combining Deep Neural Networks (DNNs) with Monte Carlo Tree Search (MCTS) was implemented, inspired by the algorithm used in AlphaGo. In this approach, a DNN is first trained to estimate the value of states, allowing MCTS to approximate state values without having to fully construct the entire search tree. By leveraging the DNN's predictive capabilities, and a backpropagation for the MCTS which takes the worst-case value into account, MCTS is able to significantly reduce the computational overhead typically required for exhaustive tree exploration, streamlining the process while maintaining accuracy. This allowed the

examples in the thesis, which were not solvable by VI due to computational and time constraints, to be solved. While this MCTS algorithm has no guarantee to compute the optimal robust path until a full MCTS tree is built, it is able to find the solution in the larger examples shown in this thesis without building the full tree. Across the 6 main examples shown in this thesis time-efficient algorithms like Dijkstra's were shown to not consistently find the robustly optimal solution. In some cases Dijkstra's solution matched the solution found by VI but in other cases, the solution found by Dijkstra's catastrophically diverged from the optimal. In cases where the robust solution is needed VI and DNN+MCTS can find the solution.

5.1 Future Work

There are several directions in which the work done in this thesis could be expanded. One area involves potential cost stochasticity. The work in this thesis deals with discrete uncertainty sets, where the value of an edge can take on values from a specified set. If each value instead comes from a distribution, solving the problem becomes more complex because the feasible set becomes a belief set. This type of problem could still be approached using the same hybrid path planning method (DNN + MCTS) if it is capable of capturing this stochastic cost information.

Another area for future work involves testing the algorithms developed in this thesis in real-life routing or navigation scenarios, such as with robots or communication packets. The examples from the Minigrid world demonstrate a variety of cases where the algorithm is effective, and it is likely that the same would hold true outside of these simulated instances.

The problem described in this thesis was focused on one agent traversing a space. This can be extended to multiple agents traveling to either the same or different goals. As they progress through the space, they can share data on the explored space. Each action taken by a given agent would aim to decrease not only their traversal path but also the cost of the other agents as well. The solution to such a problem would require a more extensive state and observation model which takes the transition and observation of each agent into account.

A final potential area for future work relates to the close similarities between the problems encountered in robust path planning for traversal and coverage scenarios. The work done in this thesis can be extended to the case of path coverage. The DNN + MCTS

method could manage the complexity related to coverage history without the exponential increase in computational requirements that the VI method would entail.

References

[1] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[3] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[4] G. Yu and J. Yang, "On the robust shortest path problem," *Computers & operations research*, vol. 25, no. 6, pp. 457–468, 1998.

[5] D. P. Bertsekas and J. N. Tsitsiklis, "An analysis of stochastic shortest path problems," *Mathematics of Operations Research*, vol. 16, no. 3, pp. 580–595, 1991.

[6] D. P. Bertsekas *et al.*, "Dynamic programming and optimal control 3rd edition, volume ii," *Belmont, MA: Athena Scientific*, vol. 1, 2011.

[7] D. P. Bertsekas and H. Yu, "Stochastic shortest path problems under weak conditions," *Lab. for Information and Decision Systems Report LIDS-P-2909, MIT*, 2013.

[8] D. Bertsekas, "6.231 dynamic programming and stochastic control, fall 2011," 2011.

[9] P. Kamkarian and H. Hexmoor, "A novel offline path planning method," in *Proceedings on the International Conference on Artificial Intelligence (ICAI)*, p. 10, The Steering Committee of The World Congress in Computer Science, Computer . . . , 2015.

[10] T.-W. Zhang, G.-H. Xu, X.-S. Zhan, and T. Han, "A new hybrid algorithm for path planning of mobile robot," *The Journal of Supercomputing*, vol. 78, no. 3, pp. 4158–4181, 2022.

[11] L. Schmid, M. Pantic, R. Khanna, L. Ott, R. Siegwart, and J. Nieto, "An efficient sampling-based method for online informative path planning in unknown environments," *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 1500–1507, 2020.

[12] D. P. Bertsekas, "Robust shortest path planning and semicontractive dynamic programming," *Naval Research Logistics (NRL)*, vol. 66, no. 1, pp. 15–37, 2019.

[13] A. F. DePavia, E. Tani, and A. Vakilian, "Learning-based algorithms for graph searching problems," in *International Conference on Artificial Intelligence and Statistics*, pp. 928–936, PMLR, 2024.

[14] M. M. Pascoal and M. Resende, "The minmax regret robust shortest path problem in a finite multi-scenario model," *Applied Mathematics and Computation*, vol. 241, pp. 88–111, 2014.

[15] D. Tschirky, "An information-theoretic approach to the robust shortest path problem," Master's thesis, Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2012.

[16] A. V. Goldberg and T. Radzik, *A heuristic improvement of the Bellman-Ford algorithm*. Stanford University, Department of Computer Science, 1993.

[17] M. Guillot and G. Stauffer, "The stochastic shortest path problem: a polyhedral combinatorics perspective," *European Journal of Operational Research*, vol. 285, no. 1, pp. 148–158, 2020.

[18] M. C. M. N. d. M. Resende, *The robust shortest path problem with discrete data*. PhD thesis, 2015.

[19] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[20] D. Bertsekas and I. Rhodes, "Sufficiently informative functions and the minimax feedback control of uncertain dynamic systems," *IEEE Transactions on Automatic Control*, vol. 18, no. 2, pp. 117–124, 1973.

[21] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[22] Y. Gabriely and E. Rimon, "Spiral-stc: An on-line coverage algorithm of grid environments by a mobile robot," in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)*, vol. 1, pp. 954–960, IEEE, 2002.

[23] M. Iqbal, A. P. U. Siahaan, N. E. Purba, and D. Purwanto, "Prim's algorithm for optimizing fiber optic trajectory planning," *Int. J. Sci. Res. Sci. Technol*, vol. 3, no. 6, pp. 504–509, 2017.

[24] H. J. Greenberg, "Greedy algorithms for minimum spanning tree," *University of Colorado at Denver*, 1998.

[25] C. S. Tan, R. Mohd-Mokhtar, and M. R. Arshad, "A comprehensive review of coverage path planning in robotics using classical and heuristic algorithms," *IEEE Access*, vol. 9, pp. 119310–119342, 2021.

[26] T. Oksanen and A. Visala, "Coverage path planning algorithms for agricultural field machines," *Journal of field robotics*, vol. 26, no. 8, pp. 651–668, 2009.

[27] B. Ai, M. Jia, H. Xu, J. Xu, Z. Wen, B. Li, and D. Zhang, "Coverage path planning for maritime search and rescue using reinforcement learning," *Ocean Engineering*, vol. 241, p. 110098, 2021.

[28] G. J. Lim, S. Kim, J. Cho, Y. Gong, and A. Khodaei, "Multi-uav pre-positioning and routing for power network damage assessment," *IEEE Transactions on Smart Grid*, vol. 9, no. 4, pp. 3643–3651, 2016.

[29] M. Zhou and J. Shi, "An uncertainty-driven sampling-based online coverage path planner for seabed mapping using marine robots," in *2022 IEEE/OES Autonomous Underwater Vehicles Symposium (AUV)*, pp. 1–7, IEEE, 2022.

[30] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia, "Pathafl: Path-coverage assisted fuzzing," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pp. 598–609, 2020.

[31] R. Sun, C. Tang, J. Zheng, Y. Zhou, and S. Yu, "Multi-robot path planning for complete coverage with genetic algorithms," in *Intelligent Robotics and Applications: 12th International Conference, ICIRA 2019, Shenyang, China, August 8–11, 2019, Proceedings, Part V 12*, pp. 349–361, Springer, 2019.

[32] D. Bertsekas, *Reinforcement learning and optimal control*, vol. 1. Athena Scientific, 2019.

[33] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, pp. 235–256, 2002.

[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*, vol. 529, no. 7587, pp. 484–489, 2016.

[35] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[36] D. Silver and J. Veness, "Monte-carlo planning in large pomdps," *Advances in neural information processing systems*, vol. 23, 2010.

[37] P. R. Williams, J. Walton-Rivers, D. Perez-Liebana, and S. M. Lucas, "Monte carlo tree search applied to co-operative problems," in *2015 7th Computer Science and Electronic Engineering Conference (CEEC)*, pp. 219–224, IEEE, 2015.

[38] S. J. Smith and D. S. Nau, "An analysis of forward pruning," in *AAAI*, pp. 1386–1391, 1994.

[39] E. Powley, P. Cowling, and D. Whitehouse, "Memory bounded monte carlo tree search," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 13, pp. 94–100, 2017.

[40] A. Hagberg, P. J. Swart, and D. A. Schult, "Exploring network structure, dynamics, and function using networkx," tech. rep., Los Alamos National Laboratory (LANL), Los Alamos, NM (United States), 2008.

[41] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.