ALEXANDRE BERGERON GUYARD

Finding Optimal Paths on Dynamic Road Networks

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en informatique
pour l'obtention du grade de Maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

décembre 2008

Résumé

Ce document examine différentes méthodes pour calculer des chemins optimaux sur des graphes dynamiques. Deux grandes approches sont comparées: l'approche déterministe et l'approche probabiliste. L'approche déterministe prend pour acquise une certaine connaissance préalable des changements à venir dans l'environnement. L'approche probabiliste tente de modéliser et traiter l'incertitude. Une variante dynamique de l'algorithme de Dijkstra est détaillée dans le contexte déterministe. Les paradigmes des *Markov Decision Processes* (MDP) et *Partially Observable Markov Decision Processes* sont explorés dans le cadre du problème probabiliste. Des applications et mesures sont présentées pour chaque approche. On constate une relation inverse entre la calculabilité des approches proposées et leur potentiel d'application pratique. L'approche déterministe représente une solution très efficace à une version simplifiée du problème. Les POMDP s'avèrent un moyen théorique puissant dont l'implantation est impossible pour des problèmes de grande taille. Une alternative est proposée dans ce mémoire à l'aide des MDP.

Abstract

This document examines different methods to compute optimal paths on dynamic graphs. Two general approaches are compared: deterministic and probabilistic. The deterministic approach takes for granted knowledge of the environment's future behaviour. The probabilistic approach attempts to model and manage uncertainty. A dynamic version of Dijkstra's algorithm is presented for the deterministic solution. Markov Decision Processes and Partially Observable Markov Decision Processes are analysed for the probabilistic context. Applications and measures of performance are given for each approach. We observe a reverse relationship between computability and applicability of the different approaches. Deterministic approaches prove a fast and efficient way to solve simpler versions of the problem. POMDPs are a powerful theoretical model that offers little potential of application. An alternative is described through the use of MDPs.

Avant-propos

Je tiens à remercier les personnes sans qui la réalisation de ce travail aurait été beaucoup plus ardue: M. François Laviolette pour ses idées et sa grande disponibilité ainsi que M. David Ouellet pour ses révisions, ses commentaires constructifs et sa contribution à la qualité des interfaces graphiques. Mes remerciements vont aussi à M. Éric Dorion pour ses révisions et ses commentaires judicieux, et, finalement, à messieurs Luc Pigeon et Alain Bergeron qui m'ont donné la possibilité d'effectuer ce travail.

Contents

List of Tables

List of Figures

Introduction

The objective of this research is to explore novel ways to compute optimal paths on road networks. The goal is simple: find a path from point A to point B. However, it gets trickier when we start to take into account the fact that the graph, or the environment in which we evolve, may change more or less unexpectedly. In a stable, fully predictable context, finding an optimal path is a simple problem. If we want to take into account future modifications to the environment, the problem quickly gets harder. Finally, if one wishes to consider the fact that not everything is known about the environment, and what changes may occur in the future, the problems gets even more complicated.

We will discuss different approaches to solve the three level of problems discussed above. We will see that there are efficient ways of finding solutions for the simpler, fully observable, deterministic problems. We will propose techniques to address the more complex problems, trying to find the best possible solution while dealing with a partially uncertain future in a reasonable amount of time.

Chapters 1 and 2 detail how the deterministic problem was approached and how it was implemented in an actual path finding application. Chapter 3 shows how the probabilistic aspects of the problem can be addressed using the Markov Decision Processes (MDP) framework. Chapter 4 discusses the use of Partially Observable Markov Decision Processes (POMDP) as a way to deal with uncertainty. Each approach will be described in detail: a theoretical background will be provided and the implementation of the approach will be discussed. Finally, the obtained results and limitations of every approach are detailed in order to identify solutions better suited to our problem.

Chapter 1

Deterministic Environment

This chapter explains how we model the environment to be able to compute shortest paths in a stable, predictable, deterministic context. The term deterministic means that although not everything may be known about the environment, the elements present in it have a location and behavior that is fully known.

This type of problem is somewhat similar to the one a military convoy faces when it has to pick a particular route in a city. At every intersection, the convoy will have to decide on which road segment to use. The available information about the road network (location of elements potentially impacting movement) will help the path planner to make a better decision. The more that is known about the road network, and how it will change in time, the better the chances are of planning a path correctly.

We start by describing a static environment, and detail an efficient solution for it. We will then explore ways to transform this basic static environment into a dynamic one, taking time into consideration. A solution for the dynamic version of the problem is also given.

1.1 Static Context

1.1.1 Environment

We start by representing the environment as a directed weighted multigraph where the weights of edges do not change in time. The graph is *directed*, meaning each edge goes from on vertex to another in an ordered fashion. The term *multigraph* indicates that multiple edges are

allowed from one vertex to another. In this graph, the edges represent road segments and the vertices represent the intersections, where the decision of which road segment to take will be made.

Définition 1. The basic static environment is comprised of a *Directed Weighted Multigraph*, a triplet $\{V, \overset{\frown}{E}, W\}$ where:

- $V = \{v_i \mid i \in I\}$ is the set of the graph's *vertices*;
- $\overset{\frown}{E}$ is the set of the graph's *edges* such that
$$\{(v, v) \mid v \in V\} \subseteq \overset{\frown}{E} \subseteq V \times V;$$
- $W \colon \overset{\frown}{E} \to \mathbb{R}^+ \cup \{\infty\}$ is the *cost function*, meaning, $W(\overset{\frown}{e})$ is the *weight* associated to edge $\overset{\frown}{e}$.
$W(\overset{\frown}{e}) = \infty$ means the edge $\overset{\frown}{e}$ is unusable.

This is the staring point that will allow us to compute initial solutions, and refine them. As we progress along this process, additions will be made to the basic environment, representing new concepts or refining existing ones.

1.1.2 Problem

Having a basic version of our environment, we are able to tackle a first version of the problem. Let us then consider $G = \{V, \overset{\frown}{E}, W\}$ a directed weighted multigraph. We consider vertex v_s as the "starting position"and vertex v_g, as the "goal position" we intend to reach. In graph G, we are looking for a path

$$P = \langle v_s = v_0, \overset{\frown}{e_1}, v_1, \overset{\frown}{e_2}, v_2 \ldots, v_{m-1}, \overset{\frown}{e_m}, v_m = v_g \rangle$$

for which the total weight of traversed edges is minimal. In other words, we wish to minimize

$$W\left(P\right) \overset{\text{def}}{=} W\left(\overset{\frown}{e_1}\right) + W\left(\overset{\frown}{e_2}\right) + \cdots + W\left(\overset{\frown}{e_m}\right) \tag{1.1}$$

We are looking for a path, going from v_s to v_g at a minimal cost. There are numerous existing solutions [7] [3] [12] for the static version of this problem.

1.1.3 Dijkstra's Shortest Path Algorithm

Let us consider Dijkstra's shortest path algorithm (Algorithm 1) to solve this static version of the initial problem. Dijkstra's algorithm computes the shortest path from a source vertex to a destination vertex on a directed weighted graph.

Algorithm 1 Dijkstra's Shortest Path Algorithm

function Dijkstra (G, v_s, v_g)
$G = (V, E)$ a directed weighted graph
V be the set of all vertices
$w(e_{i,j}) \in \mathbb{R}^+$ the weight associated with edge $e_{i,j} = (v_i, v_j)$
for all $v \in V$ **do**
  $cost(v) := infinity$ {Total cost to reach v from v_s}
  $previous(v) := undefined$
end for
$cost(source) := 0$ {Cost from v_s to v_s}
$Q := V$ {Q is the priority queue of non optimized vertices }
while Q is not empty **do**
  $u := extract\_min(Q)$ {Remove vertex from priority queue}
  for all v neighbour of u **do**
    $alt\_cost := cost(u) + w(u, v)$
    if $alt\_cost < cost(v)$ **then**
      $cost(v) := alt\_cost$
      $previous(v) := u$
    end if
  end for
end while
{Building the shortest path for v_s to v_g}
$P := \varnothing$ {Empty path}
$u := v_g$
while u exists **do**
  insert u at the end of P
  $u := previous(u)$
end while
return P

A typical implementation of Dijkstra's algorithm will have a worst-case running time of $O(V^2 + E)$ on a directed weighted multigraph G of V vertices and E edges. The costly part is the selection of the next vertex which takes $O(V^2)$[17] [5].

We now have an initial solution which provides an optimal path for a static environment. We have to modify this solution to take into account the time variable.

1.2 Dynamic Context

In this section we propose a generalization of the model described in Section 1.1 to a situation where the weights of the edges evolve in time.

1.2.1 Environment

In order for the algorithm to take time into account, we have to modify the environment as follows.

Définition 2. The dynamic environment in comprised of a *Directed Weighted Multigraph*, now a quatruplet $\{V, \widehat{E}, T, W\}$ where:

- $V = \{v_i \mid i \in I\}$ is the set of the graph's *vertices*;
- $\widehat{E}$ is the set of the graph's *edges* such that
$$\{(v, v) \mid v \in V\} \subseteq \widehat{E} \subseteq V \times V;$$
- $T := \mathbb{R}^+$ is the time variable t;
- $W : \widehat{E} \times T \to \mathbb{R}^+ \cup \{\infty\}$ is the *cost function*, meaning, $W(\widehat{e}, t)$ is the *weight* associated to edge $\widehat{e}$ at time t for $(\widehat{e}, t) \in \widehat{E} \times T$.
$W(\widehat{e}, t) = \infty$ means the edge $\widehat{e}$ is unusable at time t.

We have modified the W function to take time into account. Doing so means being able to access a structure capable of providing weight evaluations for particular edges at current and future times. Obviously the size of the new structure would largely depend on the granularity of the time variable we wish to consider. If we choose to consider a context

dealing in minutes over a period of one hour, the original structure will grow 60 fold. If the dynamism's precision is such that we get weight updates every 1/10th of second over a day, the initial structure would get 864000 times larger. These figures are arbitrary, but illustrate that the granularity of the time variable used has to be carefully calibrated. The problem becomes even more complex if one wishes to consider many possibilities, where the edge's weights would evolve differently through time (see figure 1.1). The details pertaining to the construction of such a structure are out of the scope of this research. We will suppose that such a structure is available to us.

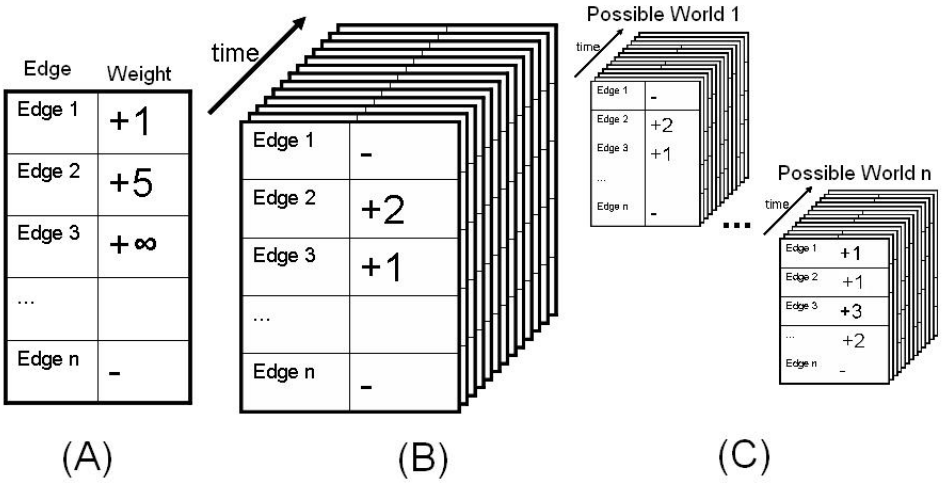

Figure 1.1: Evolving weight structure (A) Static (B) Dynamic (C) Dynamic with many possibilities

1.2.2 Problem

Now that our environment has been modified, we must modify the definition of our problem in order to take into account the time variable. We are still looking for a path

$$P = \langle v_s = v_0, \widehat{e_1}, v_1, \widehat{e_2}, v_2 \ldots, v_{m-1}, \widehat{e_m}, v_m = v_g \rangle$$

but, we wish to minimize the time taken to go from v_s to v_g. It will be beneficial to consider the result of our weight function as the time taken to traverse a particular edge at a particular time. Now that the weight is time based, we can consider the total weight, or time, to reach a particular vertex as the time it takes to reach the vertex preceding it plus that vertex's traversal time. Thus, we end up with an augmented version of Equation 1.1. We are looking to minimize:

$$W\Big(P, t_0\Big) \stackrel{\text{def}}{=} W\Big(\widehat{e_1}, t_0\Big) + W\Big(\widehat{e_2}, t_0 + W(\widehat{e_1}, t_0)\Big) + \cdots + W\Big(\widehat{e_m},\ t_0 + \sum_{i=1}^{m-1} W(\widehat{e_i}, t_i)\Big) \quad (1.2)$$

Remains to see the implications of these modifications to algorithm 1.

1.2.3 Dijkstra with a Time Constraint

In order for the algorithm to be able to use the new $W(\widetilde{E}, T)$ function, it will be necessary to keep track of how much time the traversal of each edge takes. While the first call to the function will be made at time t_0, all subsequent calls will have to be made for the specific time t_i at which the particular edge e_i will be reached. Algorithm 2 illustrates the actual changes made to the original algorithm.

Algorithm 2 Dijkstra's Shortest Path Algorithm - Dynamic Version

function DynamicDijkstra (G, v_s, v_g, t_0)
$G = (V, E, T)$ a directed weighted graph
$T = \{t_i\}$ the time variable
V the set of all vertices
$w(e_{i,j}, t_k) \in \mathbb{R}^+$ the weight associated with edge $e_{i,j} = (v_i, v_j)$ at time t_k
for all $v \in V$ **do**
  $cost(v) := infinity$ {Total cost to reach v from v_s starting at t_0}
  $previous(v) := undefined$
end for
$cost(source) := 0$ {Cost from v_s to v_s}
$Q := V$ {Q is the priority queue of non optimized vertices }
while Q is not empty **do**
  $u := extract\_min(Q)$ {Remove vertex from priority queue}
  for all v neighbour of u **do**
    $alt\_cost := cost(u) + w(u, v, cost(u))$
    if $alt\_cost < cost(v)$ **then**
      $cost(v) := alt\_cost$
      $previous(v) := u$
    end if
  end for
end while {Building the shortest path for v_s to v_g}
$P := \varnothing$ {Empty path}
$u := v_g$
while u exists **do**
  insert u at the end of P
  $u := previous(u)$
end while
return P

1.2.4 On the Usability of the Dynamic Algorithm

The deterministic assumption guarantees that, if all is known of the environment, the function $cost(v_i)$ will return the exact time it takes to travel from v_s to v_i. However, if there exists unknown elements that would impact the cost of a particular path, there is a chance that the computation of $cost(v_i)$ might be inaccurate $\forall i$. This has a big impact when using the computed path in a practical situation. What this actually means is that from the moment when we reach v_i at time t_i and $cost(v_i) \neq t_i - t_0$, we must restart our computation from scratch using v_i as the start vertex. This consideration is paramount, since in practice, depending on the granularity of our time variable, the chances of $cost(v_i)$ being exactly the same as $t_i - t_0$ may be relatively slim. If we assume that our computation of $cost$ is still of decent quality, the algorithm remains usable if it can run quickly. By quickly, we mean fast enough that it can yield a result well under the smallest traversal time of an edge. Actual use of the algorithm has shown that it does perform well inside this condition as we will see in the next chapter.

Intelligent Transportation Systems

Interesting vehicle routing results were obtained by Kim et al.[11]. In this work, the urban environment is modeled as a graph in a fashion similar to the one described in this chapter. In this case, the researchers were interested in measuring the impact of historical and real-time traffic data on vehicle routing. They put a particular focus on evaluating the impact of traffic data on the likelihood of reaching a destination on time. Their goal is to minimize the idle "buffer time" often added to a planned route in order to reach a destination on time. Two algorithms are detailed to determine the optimal departure time using traffic data. The first algorithm focuses on minimizing the total cost, the second trying to minimize vehicle usage. More study of this work would be of interest. Although it is not the focus the work described in this document, accurate parameterization of departure time would be a great addition to the implementation discussed in Chapter 2. As such, efforts to integrate this research in our work would be relevant.

Chapter 2 will give more details on this solution by showing how it was implemented as a path finding solution in an actual application. Implementation will be further discussed and measures of performance will be given.

Chapter 2

Implementation of the Deterministic Solution

In this chapter, we describe an actual implementation of the deterministic approach. This was accomplished at the Defense Research & Development Canada - Valcartier research center, as part of the SCIPIO research project. The SCIPIO research project aims at providing army commanders with a slew of capabilities, among which the ability to capture the commander's wishes; to dispatch resources; to fuse data; and to provide optimal path calculations in an evolving urban context. Path calculations are made by the Optipath module, which computes optimal paths in real time from two locations in a city, trying to avoid elements which would impede movement. This work has been integrated as a part of the SCIPIO - Optipath module. It was but a steppingstone toward a very complex and elaborated solution, but is, nonetheless, still used and validated in that context. We will discuss how it performs, and how its performance compares to an implementation of a standard Dijkstra algorithm.

We will focus on the performance of the system. Performance will be assessed both from an algorithmic, and a "real life" or benchmark perspective. The benchmark analysis will be made on the implemented algorithm as well as on the communication mechanism used for client-server communication: Javaspace/Jini.

2.1 Algorithmic Complexity Analysis

We refer to the algorithm described in **algorithm** 2 as the Optipath algorithm. As seen before, the Optipath algorithm is a modification of the well known Dijkstra algorithm. While a lot

of additions and customizations have been made, algorithmically, a lot remains the same. A typical implementation of Dijkstra's algorithm runs in $O(V^2 + E)$ on a directed weighted multigraph G of V vertices and E edges. The costly part is the selection of the next vertex which takes $O(V^2)$. Our implementation takes advantage of the fact that our graph actually represents a road network. Such a graph generally has between 2 and 4 edges to every vertex (4 to 8 when using unidirectional edges) making it a rather sparse graph. While $O(n^2)$ is the best possible complexity for dense graphs, it can be significantly improved upon for sparse graphs[9].

Fibonacci heaps support the arbitrary deletion from a n-item heap in $O(logn)$ amortized time and all other standard heap operations in $O(1)$ amortized time [8]. Implementing vertex selection with a priority queue using a Fibonacci heap makes the time complexity of our algorithm $O(VlogV + E)$[5].

2.2 Execution Analysis

To start our analysis of the algorithm's execution we begin by comparing the performance of Optipath with Dijkstra's algorithm. We'll continue by analyzing the response time of the algorithm on instances of different sizes.

All the tests were conducted on a Intel Pentium 4 HT, with a 3,20GHz CPU, 1,00GB of RAM and 512MB of cache. The same setup has been used for all the measures presented thereafter.

2.2.1 Optipath vs. Dijkstra's Algorithm

We compared the two algorithms on a graph representing the region of Quebec. This graph covers the entire region north of the St-Lawrence river and in comprised of 15422 vertices and 28294 edges. Each vertex is the equivalent of an intersection on the road network and each edge is the road segment between them. We did 1000 calls to both algorithms searching for shortest paths between 2 random vertices on the graph.

In table 2.1, we see the average runtime on instances of about 15000 vertices with about 1,83 edges to a vertex. Both runtimes are pretty close to one another. Actually, the observed standard deviation (both around 120% of the average) tells us that the variation on observed runtime was such, that both values can be considered equivalent. Moreover, the average

	Dijkstra	Optipath
Average	102.33 ms	90.71 ms
Variance	15242.37	11218.06
Standard Deviation	123.46	105.91

Table 2.1: Dijkstra and Optipath Runtimes

response time (about 1/10th of a second) is near real-time execution, and the differences observed between the two algorithms can be attributed more to the external factors affecting the execution (Java's memory management, CPU resources allocation) than to actual differences in the algorithms performances.

2.2.2 Optipath Performance on Various Size Graphs

In order to generalize our analysis of Optipath's performance, we had to be able to run it on graphs of different sizes. To do so, we generated artificial graphs that were representative of a generic urban environment. We built square shaped graphs of nodes that are linked with all neighbour nodes that share the same line or column. Each edge was assigned a random weight between 1 and 100. Figure 2.1 illustrates this artificial graph structure.

We, once again, made 1000 calls to the Optipath algorithm on graphs of roughly 10k, 25k, 50k and 100k nodes (table 2.2). The configuration of the test machine did not allow us to go any higher (a 100k node generated graph took all but the totality of the memory).

Vertices	Edges	Avg. Runtime	Std. Deviation	SD / AR
10000	39600	30.735 ms	19.50824	0.63472393
24964	99224	91.502 ms	62.85347	0.686908155
49729	198024	208.513 ms	137.1862	0.657926364
99856	398160	509.669 ms	349.5637	0.685864159

Table 2.2: Optipath Runtimes on Different Sized Graphs

Once again, the standard deviation shows that the execution was affected by external factors (Java's memory management, CPU resources allocation). However, on 1000 iterations, the deviation/average runtime ratio was nearly the same for every sized graph. Since our average number of edge per vertex (close to 4) was higher than in the case of the Quebec City map (1,83), our runtimes are slightly higher. It makes sense since the algorithm had more edges to explore. At half a second for a graph of 100 000 vertices, the algorithm is drifting away from real-time execution. However a comparison of the runtime with the asymptotic time complexity analysis of the algorithm provides us with interesting information.

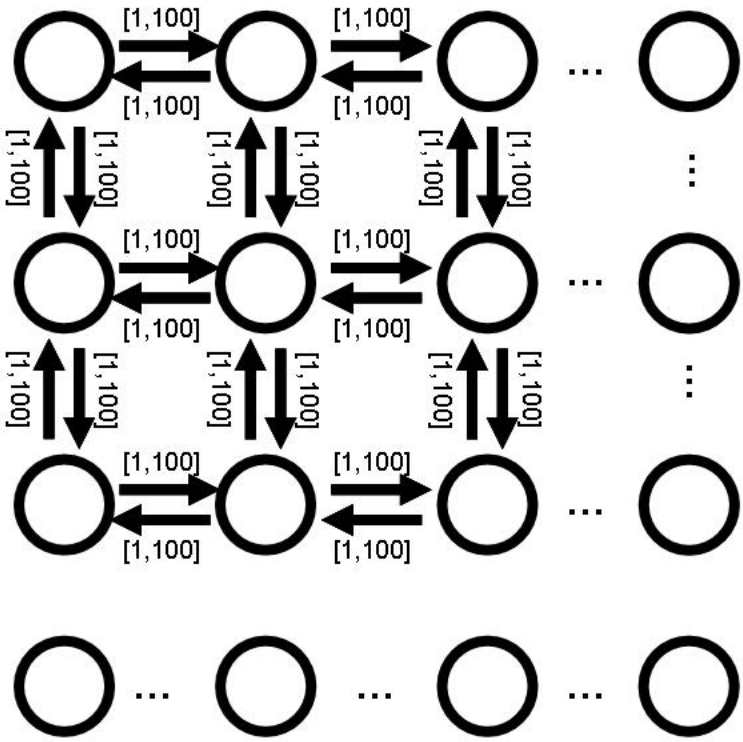

Figure 2.1: Illustration of the structure used to analyse algorithmic performance on different sizes graphs

Vertices	Edges	Avg. Runtime	$(E + VlogV)$	TC / AR
10000	39600	30.735 ms	79000	2589.881243
24964	99224	91.502 ms	208998.551	2284.087244
49729	198024	208.513 ms	431581.7051	2069.807183
99856	398160	509.669 ms	897377.50666	1760.706472

Table 2.3: Optipath Runtimes and Time Complexity Graphs

We see that as the size of the instance grows, the time complexity $(E + VlogV)$/Runtime ratio slowly decreases (table 2.3). While the ratio is still linear, the slight decrease may indicate that to every execution of the algorithm is linked a certain extra runtime that is dependant of the instance size. As the size of the problem grows, this extra independent runtime gains impact. This extra runtime could become an issue on large scale problems. However, large scale urban scenarios graphs could hardly become larger than 350k vertices with less than 4 edges per vertex.[1] Since this runtime is directly related to CPU and memory performance, it can be drastically reduced with the use of a more powerful computer.

[1]The graph representing the entire Toronto region spanning from Niagara Falls and comprising the entire West bank of Lake Ontario has less than 350k vertices.

2.2.3 Optipath Performance When Adding Threats

In this practical use of the algorithm, the changes in edges traversal costs in time $w(e_i, t_j)$ is dictated by the presence of threats in the system. The term threat can designate any factor that will have an impact on the time it takes to traverse a particular road segment. For this implementation of the algorithm, we considered two types of threats: static, and dynamic. Static threats refer to elements which have the same effect on traversal cost over time, for instance, a roadblock. Dynamic threats refer to elements whose effect on edge traversal costs will change over time, for instance mobile threats (vehicles) or evolving threats (gas cloud). We will discuss the impact of these two types of threats on performance.

When threats are present on the graph, the current implementation allows to get updated weights for all edges in time $O(1)$. It is basically just a matter of querying a multi-dimensional array for the current weight of a particular edge, which takes just about no time. Therefore, the execution of the algorithm is not affected by the addition of threats. Data analysis confirms that fact. However, the issue lies is the creation of a threat: the actual building of the multi-dimensional matrix.

In the case of a static threat, time doesn't affect the threat, since it does not move or change in time. In this case, we are left with a vector of edge weights ($E \times 1$ matrix). Obviously, the building of such an array can be done in $O(E)$. For dynamic threats, the matrix becomes larger as the threat lasts. We end up with a matrix which can be built in $O(E \times T)$ where T is the actual number of time units the threat is computed for. We'll see how this translates into performance in the actual implementation.

Static Threats

Let's first examine the time needed to build a static threat. The data shows the time taken to instantiate the matrix, to assign values to it, and the combined value of both (table 2.4). Note that the times mentioned in this analysis are obtained using the System.currentTimeMillis() method of the Java API. Values of zero are cited when returned by the method. This does not mean that the runtime is zero, but simply that it is lower then computable by the the method. It is obvious that in the case of a single data vector (single column matrix), the building of the threat can be done almost instantly. When we reach threat sizes of over 100k edges, we start to see computation times emerge. They are however, a non factor since their creation time remains very low, and threats of those sizes stand little chances of being created (they would span a larger portion of a gigantic graph (remember that the Quebec city graph has about 28k edges, and the city of Toronto/Ontario Lake west bank graph has under 350k). However,

things become a little more complicated in the case of a dynamic threat.

Edges	Init (ms)	Assign (ms)	Total (ms)
1	0	0	0
10	0	0	0
100	0	0	0
500	0	0	0
1000	0	0	0
5000	0	15	15
10000	0	0	0
25000	0	16	16
50000	0	0	0
100000	62	0	62
250000	141	0	141
500000	312	16	328

Table 2.4: Time to Build a Single Time Slot Threat

Dynamic Threats

We will examine the performance in the same way we did before. We will have initialization, assignment, and total runtimes for threats of sizes 10 to 500000 edges. However, we will have such a table for different amounts of time slots. Starting with 10 we will try to go up to 25k slots for every size threat. Detailed tables for every amount of time slots are detailed in Appendix A. We will look at the combined result in the form of charts. We will see that when the size of our 2D matrix ($E \times T$) nears 250M units, we reach the memory limit (1024MB), and are unable to obtain results, in which case, no more results are displayed on the chart. Let us quickly go over, and describe each chart.

Figure 2.2 shows the time required to initialize the matrices for every sized threats. The size of the threat is on the X axis, and the time required is on the Y axis. Each line on the graph represents a different amount of time slots, ranging from 10 to 25k.

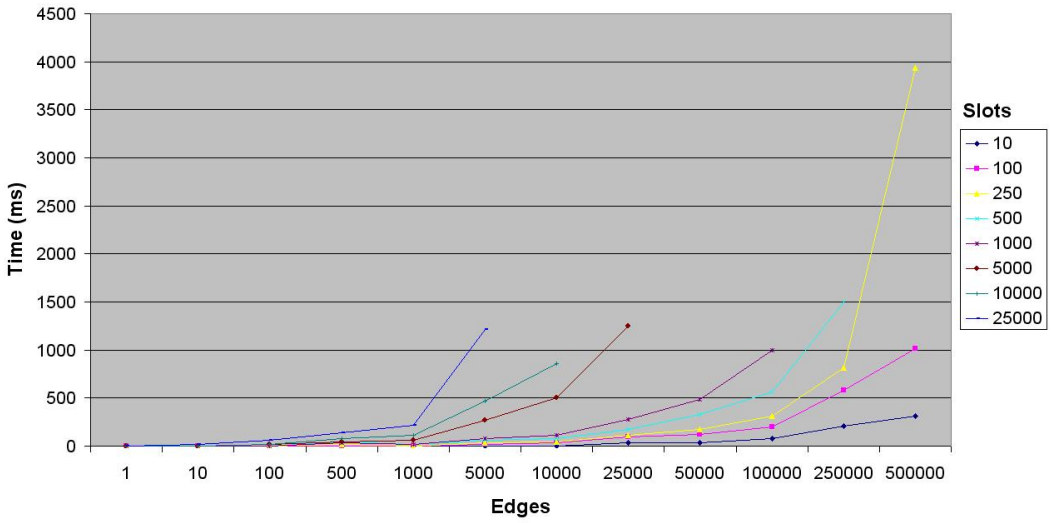

Figure 2.2: Time Required to Initialize a Threat

Figure 2.2 shows the time required to assign values to the matrices for every sized threats.

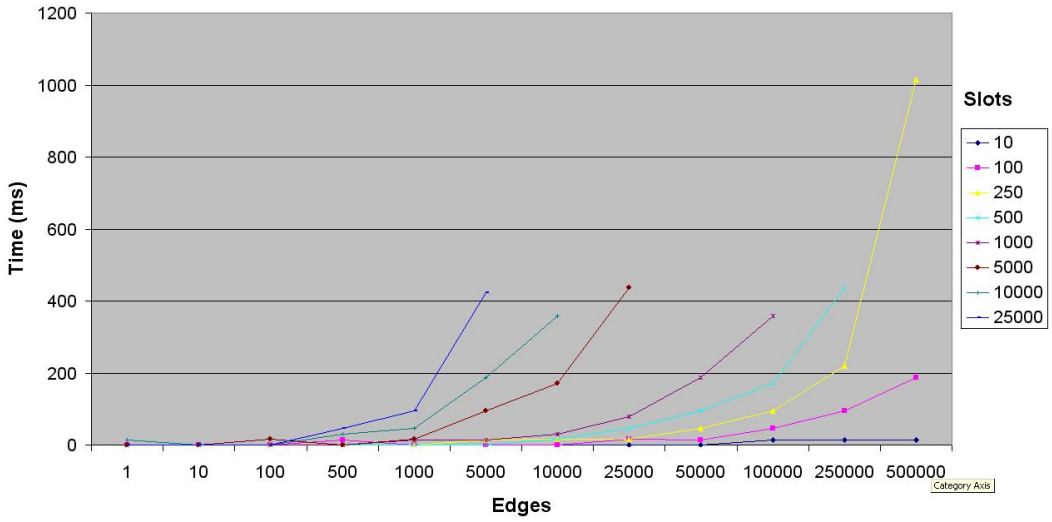

Figure 2.3: Time Required to Assign Values for a Threat

Figure 2.2 shows the total time required to assign values to the matrices for every sized threats.

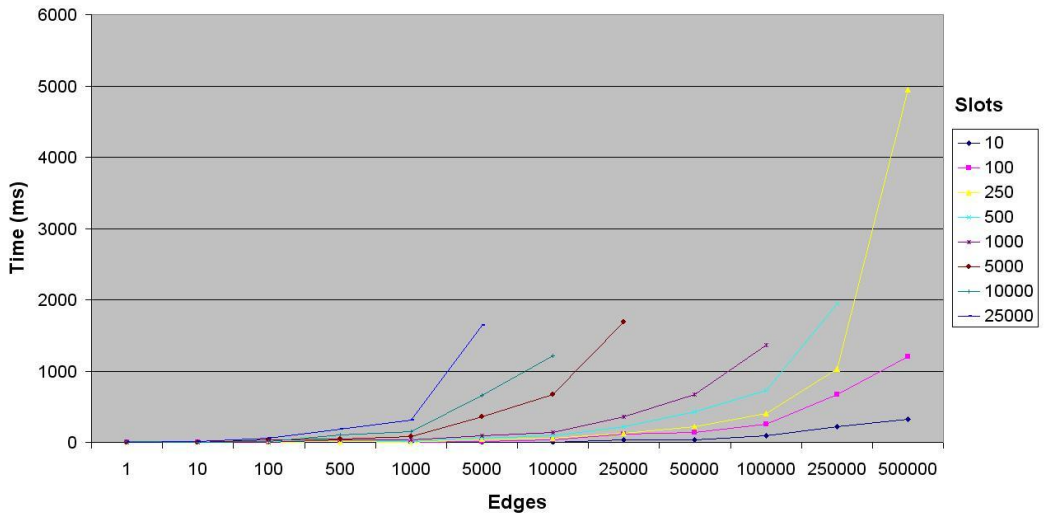

Figure 2.4: Total Time Required to Create a Threat

Several interesting facts emerge from all this data. First, we can confirm that the execution time is a direct function of the number of edges and the amount of time slots ($O(E \times T)$). Roughly, the time to instantiate a matrix cell is 1.5×10^{-5}ms. It follows that instantiating huge threats spanning several hundreds of thousands of edges can be done inside a few seconds (125M edges takes roughly 1.6 seconds) even on a modest computer.

Building larger threats would only be a matter of upgrading the hardware. Since threat creation is linear, double the memory and you can double the size of your largest threat. In this case, memory limitations appear when ($E \times T$) passes 125M. This figure is affected by both the size of the threat and the number of time slots it has. This means that the larger the threat, the larger the threat matrix. Moreover, it means that the more often we update the threat status the larger the threat matrix gets. Even though current implementation yields decent performance on more than large and precise enough threats, there are ways to circumvent potential problems.

Let us consider a threat that would exceed our current system's capabilities. If we had for instance a threat spanning 10k edges with 25k time slots ($E \times T = 250M$). Such a threat would cover over a third of the Quebec City region, and could be updated every second for almost 7 hours. In practice, receiving threat updates every second would be less than likely. Threat updates with a frequency of less than a second would be even less likely. This means that we could relax the frequency of updates to, let's say every 2 seconds. This would result in having freed half the memory, which could then be used to compute a larger scale threat (20k edges). Suffice to say, the implementation is very efficient for our current needs, and could easily be scaled to accommodate larger scaled contexts.

Transfer Rates

We now have a better understanding of the way things work on the server side and from the algorithmic perspective. However, when the system is in use, not everything happens on the server. Users actually interface with the system through a client that is remote from the server. Even on the server side, not all components of the system are necessarily located on the same machine. This means that a number of different computers have to interact together. The speed at which they are able to communicate the information is key to the overall execution speed of the system.

System communication is accomplished with the help of Javaspaces/Jini technology. Whether they be path requests, path answers, threats or the graph itself, everything passes through Javaspaces. We will look at a few sample transfer times in order to get a grasp of the possible impact the transfer rates have on system performance.

Obviously, transfer rates are largely dependant on the network system used to carry the information. From the physical support (wired, wireless), to the transport and network protocols (TCP/IP). These considerations are beyond the scope of our research, and so network testing and optimization will not be done here. However, it is possible to configure the software used to implement client-server communication (Javaspaces/Jini). We will test our actual implementation/configuration with all components on a single machine to abstract the network related components.

Various types of information are transferred across the system. This information can be divided in two main categories: initialization information, and execution information. Basically, initialization information is the preliminary information the system needs to be able to work. We mainly think of the information as regarding the road network (the graph), and the threats present on the graph. Initialization information can be rather large, however it is mostly only transferred once at the start of the application. Execution information is essentially composed of path requests (sent from the client to the server), and shortest path itinerary (returned by the server to the client). Execution information is small, and is sent back and forth throughout the execution. We will examine performance for both types of information.

We can characterize Javaspaces/Jini's communication scheme by dividing it in two main steps: serialization of the information, and communication/transaction. Simply put, first we package the information (serialization), then we send it (communication/transaction). For the purpose of this experimentation, we have generated different sized graphs, and serialized them to disk. As seen on table 2.5, the sizes of the serialized graphs grow in direct proportion with the size of the graph, which makes perfect sense. There is a minimal, negligible, amount

Size of Graph	Number of Vertex	Size on Disk
0×0	0	2KB
10×10	100	214KB
100×10	1000	2172KB
100×100	10000	22236KB
100×1000	100000	222966KB

Table 2.5: Sizes of Serialized Graphs on Disk

Size on Disk	Time to Serialize	Time to Transfer	T. to T.− T. to S.
2KB	0ms	813ms	813ms
214KB	237ms	1218ms	981ms
2172KB	2053ms	3125ms	1072ms
22236KB	20678ms	21654ms	976ms
222966KB	216031ms	217254ms	1223ms

Table 2.6: Time Needed to Transfer Different Sizes of Files with Javaspaces/Jini

of size (2KB)which is taken up by information relevant to the serialization itself. We can safely conclude that a serialized graph roughly takes up 2,2KB per vertex. The graphs used for the serialization tests had an average of about 3.6 edges per vertex which is above what is encountered in urban contexts, and more than sufficient to be representative data.

Let us then examine the time required to communicate these different sized graphs through our Javaspaces/Jini implementation. Here, we basically timed ten transfers of every sized graph to present an average transfer time. Once again, we communicated through Javaspaces/Jini, but on a local machine, without the use of the network. On table 2.6, we also present the time required to serialize the object before actually beginning the transfer. We finally show the difference between transfer time and serialization time.

We first observe that the time taken to transfer the files is, once again, in direct proportion with the size of the file. When we cross reference this information with the serialization times we discussed earlier, we quickly realize that most of the time goes to the serialization of the object. The difference between the time to transfer and the time to serialize indicates that the transaction for every sized graph takes roughly 1 second. The rest of the time is taken to serialize the object.

This is a very interesting piece of information. While a 1 second overhead on every transfer is of little importance for the larger initialization data, it may be more of an issue for the smaller sized execution information. It has a small overall impact if, at the launch of the application, the use of Javaspace/Jini adds a few seconds to allow the transfer of all necessary

data. However, when the software is in use, the 1 second overhead is added on every time we send or receive path information. The information contained in a path request is rather small (< 10KB), and the path result closer to 100KB. We are talking about serialization times of less than 150ms. The addition of a 1 second transfer overhead multiplies this time by a factor of almost 10.

Applicability

The results detailed in this chapter show that the deterministic approach is very efficient in addressing problems in an environment which provides a somewhat accurate prediction of future edge weights. The small execution times make it a viable solution for real-time execution, allowing for very fast recalculations when encountering unforeseen events. However, this approach remains strictly deterministic, taking knowledge of the environment for granted. We will now discuss approaches which provide a better representation of uncertainty.

Chapter 3

Dynamic Path Finding in a Stochastic Environment

In the previous chapters, we have considered solutions based on a deterministic assumption. More precisely, we considered every element present in the system to be fully predictable. Changes in weight of every edge could be anticipated in time. Even though this approach yields interesting results, we want to try and generalize our problem in such a way that we are still able to efficiently compute shortest paths, even if the changes in edge weights are not completely known in advance. We want to move from a deterministic environment to a stochastic one. To do so, we will examine a mathematical framework called Markov Decision Processes (MDP). In this chapter, we will define the MDP model, detailing the problems it solves, and possible solutions for them. We will then see to what extent it can be applied to our path finding problem.

3.1 Markov Decision Processes

Markov Decision Processes (MDP) is a mathematical framework which allows decision-making in contexts where the decision outcomes are not fully known in advance, but nevertheless quantified by probabilities. Figure 3.1 illustrates the MDP agent-environment dynamic. In a MDP, an agent (the decision maker) is located in a state of the environment it interacts with. Being in a specific state, the agent will take an action. Following that action, the agent will find itself in a new state of the environment and will receive a reward. The agent must learn to reach a desired state, or goal state, while maximizing the rewards it collects along the way. Learning will occur by trying different possible actions for each state

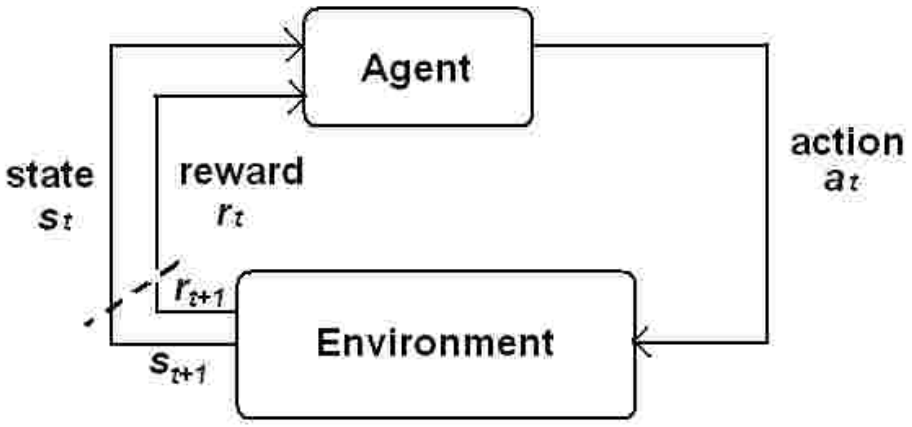

Figure 3.1: The agent-environment interaction

over many episodes, or iterations on the problem.

3.1.1 MDP Definition

Définition 3. Formally, a MDP is a tuple: (S, A, P, r) where:

- S is the set of states;
- A is the set of possible actions;
- $P^a_{s,s'} = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the transition probability of reaching state s' when taking action a in state s;
- $r_a(s, s')$ is the immediate reward received for reaching state s';

s_t, and s_{t+1} represent particular states $s \in S$ encountered in a specific, discrete time step $t = 0, 1, 2, 3, ...$ The objective of the MDP is to maximize the expected cumulative rewards

over a potentially infinite horizon:

$$\sum_{t=0}^{\infty} \gamma^t r_{a_t}(s_t, s_{t+1}). \tag{3.1}$$

$0 < \gamma \leq 1$ is called the discount-factor. It determines the value of future rewards. If $\gamma = 0$ the agent will only consider immediate rewards. The closer γ is to 1 the more the agent considers rewards received later. With $\gamma = 1$, the agent will consider rewards received at every time step as being equally desirable or important [19].

3.1.2 MDP Solutions

The solution of a MDP consists of a policy $\pi(s, a)$, a function that maps each state $s \in S$, and action $a \in A$ to a probability of taking action a in state s. A policy is said to be *deterministic* if at each state s there is always an action that has probability 1 of being chosen (all the others having probability 0). The objective here is to find a policy that maximizes at each state s, the expected cumulative rewards received, when starting in s, and following π[1].

The value function, the value of a state s, under a policy π is the expected cumulative discounted reward an agent can receive when starting at state s and acting following π. It can be defined as:

$$V^{\pi}(s) = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} \tag{3.2}$$

Note that, in an MDP, optimal policies always exists, and at least one of them is deterministic (see [19]). A property of the value function is that it satisfies a particular recursive relationship:

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\} \\ &= E_{\pi}\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s\} \\ &= \sum_{a} \pi(s, a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_{t+1} = s'\}] \end{aligned}$$

[1] If the agent "follows π", when in state s, it will choose action a_i according to the probability distribution $\pi(s, -)$.

$$= \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma V^{\pi}(s')] \tag{3.3}$$

where $R^{a}_{ss'} = E\{r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\}$ is the expected value of the next reward. Equation 3.3 is called the Bellman equation for V^{π}. It expresses a relation between the value of a state and the value of its successor states. This equation forms the basis for different techniques that allow the computation of V^{π} including the Q-Learning algorithms which we will see in detail later.

Among the various solutions that allow the computation of the optimal policy π^*, we find the Policy Iteration and Value Iteration approaches. Both are members of the Dynamic Programming family of algorithms and are based on the Generalized Policy Iteration (GPI) principle. In this approach, algorithms move alternatively between policy evaluation and policy improvement. Policy evaluation involves a computation of the expected value of states, when following a particular policy $V^{\pi}(s)$. Policy improvement uses the values computed in order to find an improved policy by selecting actions yielding higher expected values. If a better policy is found, new values can be recomputed and the process starts again. The Policy Iteration algorithm involves exact value computation for every state $s \in S$ which is very computationally expensive, and would make it a poor choice in practice. The Value Iteration algorithm is similar to Policy Iteration, however it limits policy evaluation to a given amount of steps. This approach is less costly, and would be a better choice for our problem. That being said, we have opted to implement Q-Learning algorithms. Q-Learning algorithms are members of the Temporal Difference (TD) learning algorithms, which are the flagship algorithms of the Reinforcement Learning world. TD Learning is a combination of Dynamic Programming and Monte Carlo Methods. This means that, as a member of the TD Learning algorithm family, Q-Learning will execute policy evaluation by looking at a limited number of states, like the Value Iteration algorithm. Q-Learning algorithms will also have the power to learn from episodes (raw experience), without looking at the entire environment, like Monte Carlo methods. On top of that, we will see that the Q-Learning algorithm works directly on state/action (s, a) pairs, which makes the extraction of paths from the computed policy a little faster. We detail two versions of the Q-Learning algorithm in the next section.

3.1.3 Q-Learning

Q-Learning algorithms try to find an optimal policy π^*, which maps each state $s \in S$ and action $a \in A$ to a probability of taking action a in state s, that maximizes the cumulative rewards obtained when reaching the goal state. The optimal policy π^* is obtained by evaluating action-value functions, also called Q-values. For every pair (s_t, a) we associate a Q-value, $Q^{\pi}(s_t, a)$, which represents the potential payoff from state s_t, of taking action a and then

acting according to the policy π. A deterministic policy π, can easily be derived from Q^π by choosing the action a that yields the highest Q-value as the single possible one for every state s.

To find an optimal policy π^*, we need to compute the optimal value function V^* for every state:

$$V^*(s) = \max_\pi V^\pi(s)$$

Optimal policies also imply optimal Q-values for all $s \in S$ and $a \in A$:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a)$$

We can write Q^* using V^*:

$$Q^*(s,a) = E\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\}$$

We know from Equation 3.3 that V^π satisfies the Bellman equation. For V^*, this equation can be rewritten:

$$\begin{aligned}
V^*(s) &= \max_{a \in A(s)} Q^{\pi^*}(s,a) \\
&= \max_a E_{\pi^*}\{R_t | s_t = s, a_t = a\} \\
&= \max_a E_{\pi^*}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\} \\
&= \max_a E_{\pi^*}\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a\} \\
&= \max_a E\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\} \qquad (3.4) \\
&= \max_{a \in A(s)} \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s')\right] \qquad (3.5)
\end{aligned}$$

Equations 3.4 and 3.5 correspond to the only fixed point of the Bellman Equation. For Q^*, the Bellman equation is:

$$Q^*(s,a) = E\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a\} \qquad (3.6)$$

$$= \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \max_{a'} Q^*(s', a')\right] \qquad (3.7)$$

By repeating iterations on the problem, we refine the Q-values, slowly converging towards Q^*, the optimal state-action function. The closer we get to Q^*, the closer we are to finding π^* since we can directly obtain an optimal policy from Q-values by simply choosing the action with the highest Q-value in every state. Equations 3.6 and 3.7 are the basis for the two algorithmic approaches which are described next: one-step Q-Learning and $Q(\lambda)$.

One-step Q-learning

Algorithm 3 shows the Q-learning algorithm in procedural form:

Algorithm 3 Q-learning algorithm

```
Initialise Q(s, a) arbitrarily
for each episode do
    Initialize s
    for all steps in the episode do
        Choose a from s using policy derived from Q
        Take action a, observe r,s'
        Q(s_t, a) ← Q(s_t, a) + α[r_{t+1} + γ max_a Q(s_{t+1}, a) − Q(s_t, a)]
        s ← s'
    end for
end for
```

After taking a particular action, this algorithm will try to update the Q-value by adding the observed immediate reward r_{t+1} and the difference between the best observable Q-value at s_{t+1} and the current Q-value.

$$Q(s_t, a) \leftarrow Q(s_t, a) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a)]$$

The α parameter is called the *step-size parameter*, which basically affects the rate of learning. It takes into account the number of episodes, or iterations, the algorithm has made. If we consider n to be the number of the current iteration, setting $\alpha = 1/n$ will ensure that the earlier iteration will have a stronger impact on the learning process. If we set $\alpha = 1$, the agent will continue its learning process at the same rate. It has been shown ([6] [10]) that given particular values of α, γ, and the specifics of the action selection method (which will be discussed later), Q-values will iteratively converge towards optimal $Q^*(s, a)$ and the optimal policy.

This first version of the Q-learning algorithm is interesting, but somehow limited by the fact that it only updates one Q-value for every step it takes into the solution space. Indeed, one-step Q-Learning, updates the value for s_t when it reaches s_{t+1}. As a result, a large number of episodes will be necessary in order to update Q-values for every s_t. It might be interesting to update more than the single Q-value for s_t. Instinctively, getting a better Q-value for state s_t could also lead to better Q-values for states used earlier in the

episode($s_{t-1}, s_{t-2}, s_{t-3}, ...$). Updating many Q-values in a single step may lead to convergence in a smaller amount of episodes. However, since updating many Q-values at a time takes longer, each episode will take longer to compute [19]. This is the general idea which lead to the algorithm we will describe next: $Q(\lambda)$.

$Q(\lambda)$ algorithm

In this section we describe the $Q(\lambda)$ algorithm, as described by Watkins [20]. In this approach, we try to update many Q-values as we go deeper into the episode. To do so, we evaluate an action by looking at the immediate reward and the potential reward of the next state and modify the Q-values of several states encountered before. Algorithm 4 gives the $Q(\lambda)$ pseudocode:

Algorithm 4 $Q(\lambda)$ algorithm

Initialise $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s, a.
for each episode **do**
  Initialize s, a
  for all steps in the episode **do**
    Take action a, observe r,s'
    Choose a' from s'
    $a^* \leftarrow argmax_b Q(s', b)$ (if a' ties for the max, then $a^* \leftarrow a'$)
    $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s, a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    for all s, a **do**
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      if $a' = a*$ **then**
        $e(s,a) \leftarrow \gamma\lambda e(s,a)$
      else
        $e(s,a) \leftarrow 0$
      end if
      $s \leftarrow s'$;$a \leftarrow a'$
    end for
  end for
end for

$e(s,a)$ represents the eligibility trace of the state-action pair (s,a). An eligibility trace is a record of the amount of times a state-action pair has been encountered. As we can see at the start of Algorithm 4, the eligibility traces are initialized at 0 for every state-action

pair, meaning that none of them has been encountered yet. As we progress through an episode, every time a state-action (s, a) pair is encountered, its eligibility trace is incremented $e(s, a) = e(s, a) + 1$. This mechanism helps identify the state-action pairs which were partly responsible for the cumulative reward obtained. The "eligible" state-action pairs, are assigned proper credit (or blame) depending on the outcome of the episode when the Q-values are updated $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$.

Results obtained using this approach will be discussed in the following sections.

3.2 Application of the MDP Model

We will refer to Definition 3, and see how it maps to the path finding problem. Let S be the set of states, representing our decision making points. On a road network, we consider the intersections as decision points, where we choose which way to go, or action to take. We then fix A as the set of possible actions, representing every road segment one can choose to take at any intersection. The transition probability, $P^a_{ss'}$, still represents the probability of reaching state s' when taking action a in state s. In the context of our problem, this would translate as the probability of reaching *intersection2* from *intersection1*, when taking the road that goes from *intersection1* to *intersection2*. The immediate rewards, $r_a(s, s')$, are given for reaching state s' from s by taking action a. We chose to reflect the traveling time in the rewards using negative values. This means that actions that take a lesser amount of time will yield the better rewards (less negative rewards). We use discrete time units (t = 0, 1, 2, 3, ...), so if we were to take action a, and reach state s' from s in 4 time steps, our reward r would be -4. The actions that lead to the goal state s_g will yield large positive rewards. This works well with the maximization objective of the MDP (Equation 3.1). In the case of our problem, if we were to reach the goal state s_g in n time steps, the objective would be to go from the starting state s_0 to the goal state s_g and to maximize:

$$\sum_{t=0}^{n} \gamma^t r_{a_t}(s_t, s_{t+1}) \tag{3.8}$$

Obviously, the presence affecting factors will impact both transition probabilities and rewards. By affecting factor, we designate any element that could impact our normal movement speed. Such a factor could be a traffic jam, the presence of construction, a roadblock, the presence of a crowd, or an enemy vehicle.

We are trying to reach the goal state s_g, in the least possible amount of time. The presence of affecting factors has to be reflected on the rewards. If an affecting factor is present in state

s', then the rewards $r_a(s, s'), a \in A, s \in S$ should reflect it by yielding a negative reward lower than the one reflecting normal travel time. For instance, if an affecting factor was to slow traffic down by 50% on a given road segment (s, s') which normally would take 5 time steps to cross ($r_a(s, s') = -5$) , then the affected reward should be $r_a(s, s') = -10$. We have chosen to have the transition probabilities be fully deterministic ($P^a_{ss'} = 1$, if a=going from s to s'). Thus, the stochastic aspect of the problem is solely reflected on the rewards. If an affecting factor is present in the environment with a particular probability, it will be reflected on the model by modifying particular rewards in the model according to the affecting factor's probability of occurrence. [2]

We end up with a single MDP, modeling the environment. Affecting factors are reflected on the rewards, and we can use the algorithms of Sections 3.1.3 and 3.1.3 to obtain a policy reflecting an optimal path. If new events (affecting factors) occur unexpectedly in the environment during the execution of a policy, a new MDP model will be built, taking the new elements into account, and a new policy will be computed. This makes MDP a valid choice in the planning phase, before the actual execution of the computed path. However, in the next section, we will see that MDP results can also be used for decision support during execution.

Both One-Step Q-Learning and $Q(\lambda)$ algorithms were implemented in order to solve the path finding problem. Detailed results, and measures of performance will be given in the next section.

3.3 Results and Applications

We will go into more detail about implementation of the MDP approach. We will discuss performance in terms of manageable problem sizes and quality of results, compared with results obtained with the deterministic approach. We will finally discuss certain potential applications for the MDP model.

[2] The choice of having deterministic transition probabilities does not render the MDP approach unusable. We could have opted to affect the transition probability, as well as the reward function, in which case, the agent could have found itself in an unexpected state after having taken an action leading towards an affecting factor. The MDP approach and its guarantees of convergence would still have been valid. One could chose to model a different type of affecting factor using non-deterministic transition probabilities. After proper parameterization of the algorithm (see Section 3.3.1), it is very likely that the results discussed later in this chapter would still be valid. Experimentation, and parameterization, using various types of affecting factors would certainly be of interest for anybody looking at practical application of this theory. The general idea remains to reflect real-world behavior as accurately as possible.

3.3.1 Results

The one-step Q-learning and $Q(\lambda)$ algorithms were implemented and tested on a square shaped environment structure similar to the one shown in figure 2.1. The algorithms were implemented as described in sections 3.1.3 and 3.1.3.

Parameters Impacting the Solution

We will first take a look at the algorithms' performance in terms of the obtained solution's quality. It is first necessary to mention that the quality of results obtained using these algorithms is highly dependant on numerous factors. The backup function's parameters α and γ, the amount of exploring starts, the rewards attributed to every state, problem size, the number of iterations executed, and the action selection method all have an impact on the quality of the solution. The exact study of all these factors and their exact impact on the algorithm is the object of numerous studies ([1][2]) and is beyond the scope of this research. We will discuss some of them, trying to highlight the impact they have on the solution.

As mentioned before, the step-size parameter, α regulates the rate of learning of the agent. If set to 1, the agent will consider its early iterations as being just as important as its latter ones. For our problem, we found that setting $\alpha = 1/n$, n being the iteration's number, yielded acceptable results. This allows for earlier iteration to have a stronger impact on the agent's learning. As the agent makes more iterations, and (typically) moves from exploration to exploitation, the rewards obtained will have a lesser impact on the solution.

For the discount factor γ, we found that setting values closer to 1 yielded better results. High γ values allow for the agent to consider rewards received later in the episode, or iteration, more strongly than it would have with a lower γ value. A low γ value would cause the rewards obtained later in the episode to have a lesser impact on the value of a state. For long episodes, this would make the "goal" reward have a small impact on the computed value, and would lead to finding the optimal path much more slowly. Our problem is one of path planning; the quality of a chosen path is dictated by the overall quality of the choices constituting it. It is not enough for a particular path to have excellent beginning actions if those actions lead to negative cumulative rewards in the long run.

It is obvious that rewards have an impact on the results we obtain. "Goal" rewards must be high enough that the agent will have enough incentive to head towards them. Negative rewards must be deterring, but not so much that an agent would try and avoid them to a point where it would be unable to reach its goal. In our experimentation, we found that setting

a "goal" reward -100 times the amount of the time step negative reward (for example 100 and -1) yielded good results. As for the value attributed to negative rewards associated with affecting factors, they should vary based on the risk relative to that particular affecting factor. If an affecting factor represents a dangerous element that we want to avoid at all costs, a very strong negative reward ($\rightarrow -\infty$) should be assigned to it, ensuring that the agent will try to avoid it. For our experimentation the impact of the affecting factor was reflected by distributing negative rewards ranging from -2 to -5 (regular steps gave rewards of -1) to the states which were affected.

The action selection method determines if the agent is going to choose exploration or exploitation. By choosing exploitation, the agent selects the action that yields the best expected. By choosing exploration, an agent will select an action that may not yield the best rewards in hope of discovering a new, even better solution in the long run. We considered 3 approaches: greedy, ϵ-greedy and softmax. In case of the greedy approach, the agent will always go for the action which yields the highest expected reward, systematically favoring exploitation. The drawbacks of this approach are obvious. An agent which never looks for new solutions will have a high chance of being caught in a local optimum, and never converge towards the optimal solution. An initial solution to this problem is to have the agent choose the greedy (exploitation) action with a certain probability ϵ and otherwise trying a different random action with a probability of $1 - \epsilon$. This approach, called ϵ-greedy, seemed like a much better option than the greedy approach. The major drawback of ϵ-greedy is that if it chooses to explore, it will do so by choosing a random action among available ones with equal probability. It is as likely to choose the worst action as it is to select the second best. A solution to this problem is to choose an action with a probability based on its expected value. The greedy action will still have the highest probability of being chosen, and all other actions will have a probability of being picked reflecting their value. This is called the softmax action selection method, it chooses action a at time t with the probability $\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}}$. τ is a positive parameter called the temperature. High temperatures will cause the probabilities for every action to be closer to one another. If $\tau \rightarrow 0$, the softmax will become similar to the greedy approach. For this research, all approaches were tested, softmax was ultimately chosen, having the τ variable move from a high value to 0 provided an efficient way to have the agent switch from exploration to exploitation.

Exploring starts designate the process of starting an iteration from a state that is not the designated start position. With a probability p, a state is chosen at random as the start state of an iteration. This could be seen as an alternative way to ensure a certain level of exploration by the agent, as calculating a path from a different starting point will necessarily lead to the exploration of new states. This has a very intricate effect on the results observed. Interestingly enough, we found that keeping exploring starts between 0.5 and 0.75 (starting from random state in 50% and 75% of the cases respectively) tended to aim the agent towards the optimal

solution faster than by always starting from the same point.

Finally, the different sized problems will be best resolved using different combinations of parameters. The main idea, is that correctly choosing parameters for a task is complex which requires knowledge and experience with a problem. Once a particular context of application is identified, one should take an appropriate amount of time to test out different parameters and identify favourable ones. As mentioned earlier, a lot of work is being done on the impact of the various parameters on Q-learning algorithms. Such work would be totally justified in the case of our problem. For the remainder of the discussion, we will show results obtained using various algorithm parameters. Obviously, the parameters used will have an impact on the solution obtained, but this impact is not the center of this research, and will therefore not be discussed in any more detail.

Solution Quality

We first need to validate the fact that the MDP approach is indeed capable of computing optimal paths correctly. To do so, we will start by checking if we can obtain the same results using the MDP approach as we did with the deterministic (Dijkstra) approach. We will see that the MDP approach is indeed capable of computing shortest paths, but in much slower times. Later, we will analyze various additional results which can only be obtained with the MDP approach, and which are of interest. We will see that in the planning phase, before the execution of the path, the MDP framework can yield results helping decision support and allowing path fusion. It is important to note that although these results, as well as the following ones, are shown on a grid type graphical device, it is not meant to reflect the implementation. Indeed, the MDP approach can be implemented using a graph environment similar to the one described in earlier chapters. Each vertex would represent a state, and edges $e = (v_i, v_j)$ would represent potential actions, for instance, going from v_i to v_j. We have chosen to use grids because they seem to offer a better graphical depiction of our various results.

To validate the path computed using the MDP approach , we tested the $Q(\lambda)$ algorithm against Dijkstra's algorithm in a simple case, where no affecting factor was present, as shown in figure 3.2. The figure shows a standard environment where the agent has to move from the green square on the left, to the state marked with the letter "G" on the right. The optimal policy is illustrated through the use of arrows. The arrows point in the direction the optimal policy would dictate. Therefore, by following the arrrows, we follow the policy. We first see, (A), the optimal policy (straight line) as computed by Dijkstra's algorithm. (B) shows the results obtained by 40 iterations of the algorithm. We notice that 40 iterations were enough to obtain the optimal path from the start to the goal state. Red areas on the figure identify

states where the computed policy is sub-optimal. Blue areas identify the unexplored states. We finally see that after 200 iterations, we have generalized the optimal path calculation to the entire graph.

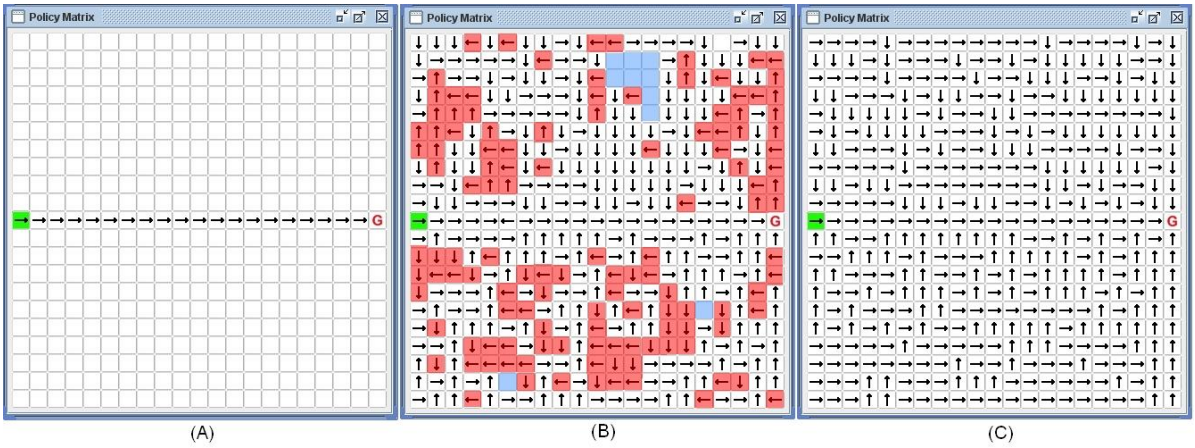

Figure 3.2: Optimal Policies as computed by (A) Dijkstra's algorithm and (B)$Q(\lambda)$ in 40 and (C) 200 iterations

We can see that the MDP framework is indeed capable of providing optimal paths calculations on a problem of small size (441 states). Remains to see if it can scale to larger sized-problems efficiently. We have done experiments on problems of 400, 2500, 10000, 22500, 45369, and 100489 states. We first noticed, that the number of iterations required in order to obtain an accurate optimal path calculation from start to goal state (not generalized to every state) seemed to grow according to the square root of the number of states under 2 conditions: limited episode size and precomputed shortest path incentive.

In our implementation, we first limited the maximum length of an episode to two times the length of a precomputed Dijkstra shortest path. This eliminated the chance of having an episode that would last for a very long time. In such an episode, the agent might explore useless parts of the environment. The drawback of this solution is that it caused the agent to often be unable to reach the goal in time, to miss the reward, and therefore not learn properly. It is somewhat obvious that making the episodes shorter doesn't make any sense if more episodes are required to learn the optimal policy.

We therefore added an "incentive" for the agent to look for better solutions from the start. We used results obtained from our deterministic algorithm as a heuristic for the MDP algorithm. Instead of initializing Q-values arbitrarily for each state/action pair, we initialize the state/action pairs of the Q-learning algorithm with positive values that reflected the shortest path computed by the deterministic algorithm. This way, when being greedy, the agent is likely to explore states closer to the deterministic optimal. The assumption is that

the deterministic policy will be better than a random one. With these two modifications to the algorithms, we are able to compute optimal policies in over 90% of cases in a number of iterations in the vicinity of the square root of the number of states.

Performance

We will quickly take a look at the actual time required to compute a policy using MDP algorithms for an environment without any affecting factors. To do so, we have timed algorithm iteration on problems of size 400, 2500, 10000, 22500, 45369, and 100489 states. Table 3.1 shows the tests results as executed on a 2.16Ghz T7400 processor with 1GB of assigned memory.

Number of States	Average Episode (Iteration) Duration	Total Time
400	0.596ms	0.001192sec
2500	137ms	6.85ec
10000	189ms	18.9sec
22500	311ms	46.65sec
45369	819ms	174.447sec (2min54sec)
100489	17809ms	5645.453sec(1h34min6sec)

Table 3.1: Average runtime of an episode

We see the average time it takes to run an episode on different sized environments. Since we have observed that the required number of iterations to obtain the optimal policy is roughly equivalent to the square root of the number of states, we are able to provide an approximate total required runtime of the algorithm in column 3 ($\sqrt{|S|}*$average episode duration). The longest time required for the problem of size 100489 may be due to the fact that a problem of this scale causes the operating system to swap some memory which slows down the process. The availability of more memory could solve this problem. It is worth mentioning that the presence of affecting factors would have little impact on the algorithm execution time. Indeed, the impact of affecting factors on rewards is computed at the instantiation of the MDP from the environment. During the algorithm execution, it has no measurable impact. This is why we only provided measures for environments without any affecting factor.

The first conclusion that can be drawn is that the demonstrated runtimes make it impossible to use this approach in a real-time environment. While Dijkstra's algorithm yielded results well inside a second for problems of over 40000 states, we see that it is not the case here. However, if the displayed times are nowhere close to real-time, they are still short enough that the computed results could be used in a different context. Indeed, if real-time performance is needed during the execution of a path, slower times are acceptable in the planning phase.

In the next section, we will discuss MDP applications for planning, among which are path fusion and decision aid applications.

3.3.2 Applications

We will now discuss possible new applications of MDPs in the context of our path planning problem. We will look at the added value of MDP compared to deterministic approaches. At this point, it is obvious that the MDP framework does not offer the type of performance necessary to replace deterministic algorithms during path execution. This means that if a user requires a new path calculation while actually on the field, MDP will not be fast enough. However, in practice, before a user goes into the field to follow a particular path, there is a planning phase. It is during that planning phase that the system is setup in order to be able to provide accurate calculations during execution. MDPs could be of interest during that planning phase. The added value of using MDPs revolve around two aspects: precomputed policy generalization and decision aid. For the policy generalization application, we will initialize the Q-values of the Q-learning algorithm in order to reflect different already computed paths. As the algorithm iterates, it will generalize the precomputed policies and compute Q-values reflecting them for states surrounding the precomputed paths. In the case of the decision aid application, we will use the Q-values computed offline during the planing phase. During the execution, if an unforeseen event occurs, preventing the user from following the path, using the precomputed Q-values, an alternative will be available right away. Although these applications do not represent answers to our original problem, they offer new possibilities, or functionalities that are of interest.

Generalization of Precomputed Policies

In the previous section, we discussed how precomputed paths could be used as heuristics to lead the agent in the right way. It turns out this method is not only useful to obtain an optimal policy faster, but it also provides a way to generalize various precomputed paths' policies to the states surrounding them. Indeed, it is possible to precompute a certain number of paths and modify the Q-values for the related state/action pairs in order to reflect the initial calculations. For instance, we could use a deterministic approach to compute a simple shortest path from start to end. We could then initialize the Q-values of the state/action pairs which reflect that path with positive values (for instance +1), and all the other Q-values to 0. Obviously, if we were to follow the Q-values before running the algorithm, we would be following the policy computed with the deterministic approach. As mentioned in section 3.1.3 the Q-values can be initialized arbitrarily and, theoretically, over an infinite number of iterations, the algo-

rithm would converge towards an optimal solution. However, by the very nature of our action selection method (in the case of our implementation softmax), action/state pairs with higher Q-values have a higher probability of being used. This means that the algorithm would start by exploring states in, and around the precomputed path. Having an agent iterate over such a modified environment would have the states surrounding the original precomputed paths explored much more quickly. The precomputed path's would have an impact on the Q-values of the states surrounding them and the original paths would be generalized to other surrounding states. The same approach could be used with numerous precomputed paths. Obviously, as the agent iterates over the problem, it gradually moves towards the optimal policy and, possibly, away from the originally computed path. Therefore, time is of the essence since the moment at which we decide to stop the iterations will determine the level at which the precomputed paths will stop being having an impact and the optimal policy will take over. Stopping early will mean having results closer to precomputations, while stopping late will mean being closer to the optimal policy. The appropriate number of iterations would vary from one context to another.

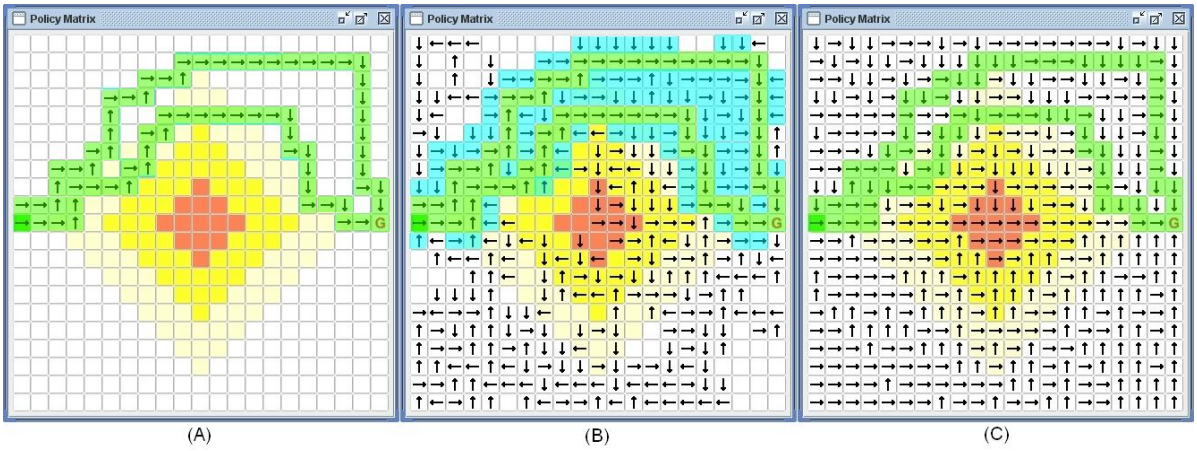

Figure 3.3: Fused Precomputed Paths

Figure 3.3 displays a square environment where the agent wants to go from the green state of Figure 3.3 (A) to the state marked with a "G" (on the right). Each white state is a "regular" state which gives an immediate reward of -1. The goal state gives a reward of $+100$. The various colored states (light yellow, yellow and orange) are impacted by an affecting factor and give rewards of -2, -3 and -5 respectively (from light yellow to orange). Figure 3.3(A) shows the two paths precomputed using the deterministic approach (trying to completely, and slightly avoid the affecting factor). A first precomputed path is set to go around the affected states, while the second precomputed path is only allowed to cross slightly affected states (light yellow states yielding a rewards of -2). Figure 3.3 (B) shows the generalization result after 50 iterations of the Q-learning algorithm. For every state, the arrows point in the direction of the highest Q-value. We see that the original paths are

still present (shown in green) and that most of the states surrounding the paths (we show in blue the states that are between or surrounding the precomputed paths that either reflect the optimal or the precomputed policy) have been computed to either head towards one of the precomputed paths or the optimal policy. Notice that the lower half of the figure has some unexplored states (white squares), as well as numerous states with a "bad policy" (a policy heading left or towards the bottom). This shows how the use of precomputed paths can lead to early exploration of a specific part of the environment, and generalize precomputed policies to nearby states. After 200 iterations, Figure 3.3 (C) shows that the computed policy now totally ignores the precomputed paths to show the optimal policy in almost every state.[3] This shows the importance of correctly timing the number of iterations in order to reach the desired goal: local exploration and precomputed path generalization, or optimal path computation.

We just saw how we could use the Q-Learning algorithm to generalize precomputed paths. We saw that by initializing specific state/action pairs with positive values and by iterating over the problem, the Q-learning algorithm will gradually generalize the pre-initialized Q-values and propagate them to surrounding states. We will now see another alternative application of the MDP framework which deals with decision aid.

Decision Aid: Alternative Action

Another possible application uses the various computed Q-values to provide the user with more than one potential action for each state. We have seen that the Q-value function reflects the expected value for a state/action pair $Q(s, a)$. Until now, we have shown the results of our policies as matrices of arrows pointing in the direction of the action providing the highest expected reward. However, in some cases, the Q-values of a particular state may be very close to one another, highlighting the fact that, for that state, two different actions have very similar expected cumulative rewards. We can take advantage of this information that is always calculated by the Q-learning algorithm. More specifically, we can specify a certain threshold within which various Q-values are considered equally good. If many Q-values have a relative difference less than the specified threshold, they will be considered equally good alternatives and potentially both be suggested to the user.

[3] In this case, the "optimal policy" evaluated does not to consider the various yellow states and would not be optimal in reality, however, it helps to show the evolution of the policy over the iterations.

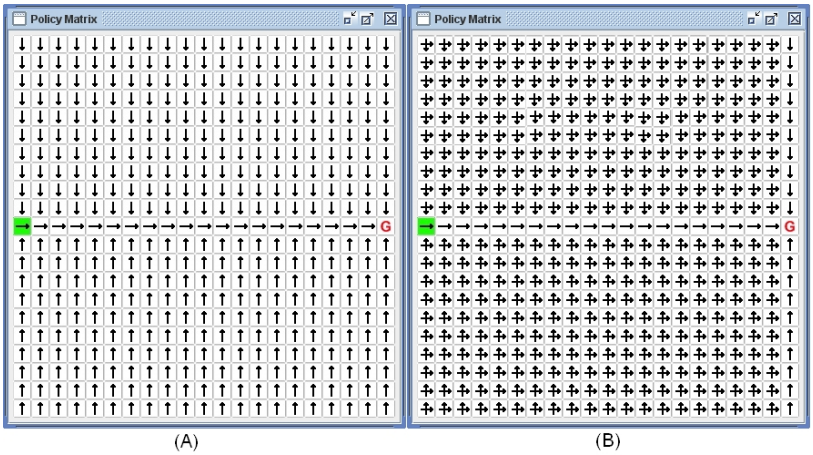

Figure 3.4: Generalized Policy

Figure 3.4 gives a simplistic illustration of this approach. (A) shows a simple policy to go from start to goal node. This is actually the same problem shown in figure 3.2 after 10000 iterations. In (B) we consider Q-values that are less than 20% apart from one another. The result is a generalized policy. The user is now presented with up to two alternatives which are considered equally good. This could be applied to more than two alternatives. One could also imagine ranking every action and providing the user with all positive alternatives in decreasing order.

As explained before, when pre-calculated before the true excursion, this application could be used during path execution if an unforeseen event occurred. If, for any reason, following the original policy became impossible, the user could instantly be suggested the precomputed second best action to take. This would be a useful decision aid tool for situations when an alternative action has to quickly be chosen. Referring to figure 3.4, let's imagine that, for some unforeseen reason, during the execution of the path, we realize it is impossible to reach, or use, the top half of the environment. Following the suggested policy (heading on top of the yellow area) would then be impossible. Using this approach, a second alternative, leading under the yellow area, would be presented to the user who could react instantly.

Stochastic Shortest Paths

It is interesting to note that somewhat similar decision aid results were obtained by Bertsekas et al. [4] studying a different problem they called Stochastic Shortest Paths. The Stochastic Shortest Path (SSP) problem is described in an environment similar to the one detailed in definition 1. It is a generalization of the standard shortest path problem where, at each vertex,

one must select a probability distribution over all possible successor vertices out of a given set of probability distributions. Note that if every probability distribution assigns a probability of one to a single action (or successor vertex), a single deterministic shortest path solution is obtained. Bertsakas et al. describe this approach in detail and propose a model based on Markov Decision Processes. The result is an environment where, for each state, each potential action is given a probability reflecting its likelihood to lead to the goal state. These probability distributions are similar to the alternatives obtained using the method described in the preceding section. However, our decision-aid results are obtained using the results of a Q-Learning algorithm stopped early in its computation. In their SSP approach, Bertekas et al. solve the MDP completely, which would, instinctively, possibly require more time. Further study of this work would be highly relevant. It would be of interest to compare the decision-aid results obtained using SSP with the results discussed in this chapter. It would give us a measure of accuracy for our results at different stages of computation as well as a potential interesting alternative for decision aid.

We have seen the added value of using the MDP framework through various applications, namely: precomputed path generalization and decision aid. This MDP approach nevertheless has a major drawback that follows from the needed Markov assumption. In the MDP framework, an agent has no memory of past event, its decisions (i.e., its policy) is only based on the current state of the MDP it is in. Hence, if for example, in the MDP, we identify various possible locations for a given affecting factor, the fact the agent has encountered it in some position at the beginning of its excursion will not be taken into account in the future. The agent will still try to avoid the other possible locations, even if we all know that they must free of affecting factor. To have an agent which can take into account past events, we have to enrich our model to what is called a Partially Observable Markov Decision Process (POMDP). In the next chapter, we will formally defined this new paradigm, and how it would address the shortfalls of MDPs.

Chapter 4

Model Limitations

We have examined the performance and applicability of MDPs in comparison with deterministic approaches. We saw the added value of using the MDP framework. We will now try to examine where the MDP framework falls short in comparison to a more powerful model: Partially Observable Markov Decision Processes (POMDPs). We will see a quick overview of POMDPs. We will take a look at the formal POMDP definition and generally discuss avenues to solve them. We will finally see how a solution obtained with a POMDP would differ from a solution obtained with a MDP. It will become obvious that, although MDP algorithms can be used to solve some cases of POMDPs, some problems can only be handled by POMDP algorithms.

4.1 Partially Observable Markov Decision Process

Typically, POMDPs are used for choosing actions when parts of the environment are unknown and when the environment is partially observable. An example of POMDP would be a robot, trying to find its way out of a room without knowing where it is located. If the robot knows the room to be 3 states wide, by 3 states high, some reasoning can occur. For instance, not knowing where it is at first, the robot can choose to try to go up twice, and then go right twice. If there are no obstacles, it will then be certain to have reached the upper right corner of the room.

4.1.1 POMDP Definition

POMDPs are similar to MDPs, but while the MDP agent knows exactly where it is located, the POMDP agent has to figure out its exact location as it moves about. Similarly to MPDs, the general idea is to progress in the environment, adapting the decision making process as the knowledge of the environment increases. The agent moves in the environment by choosing an action within a set of possible ones. For each action taken, it receives a reward and some additional informations (called observation). As information about the environment adds up, and over numerous iterations on the problem, the agent hopefully develops a better policy in order to maximize its cumulative reward. In addition, the POMDP agent has to keep track of the level of uncertainty relative to its location in the environment, making decisions accordingly. Let us see the formal definition of a POMDP framework.

Définition 4. A Partially Observable Markov Decision Process is a tuple: $\{S, A, \Omega, T, O, R, \gamma\}$ where:

- S is the set of states;
- A is the set of possible actions;
- Ω is the set of possible observations;
- $T(s, a, s')$ is the transition function. T gives the probability of reaching s', having taken action a in state s;
- $O(s', a, o)$ is the observation function. O gives the probability of observing o, having taken action a and reached state s';
- $R(s, a)$ is the reward function. R specifies the reward associated with performing action a in state s;
- γ is the discount factor ($0 < \gamma < 1$). γ specifies how the learning agent favours immediate rewards over future rewards (covered later);

The new aspect introduced in POMDPs is the partial knowledge of the environment. When the agent takes an action, it is rarely completely certain of the actual state it is in. The agent must take this fact into account as it makes decision. To keep track of this partial knowledge of the environment, the agent uses a belief state b. A belief state is a $|S|$-dimensional vector where the i^{th} element represents the probability of being is state s_i. B, the belief space, is the set of all possible belief states. Every time the agent takes an action, it receives a reward, as well as an observation o. The observation is used to update the belief state.

Application of the POMDP Model

We will refer to the formal definition of a POMDP given in Definition 4, and see how we can map an adapted version of our problem to it. The set of states S, represents our decision making points, the intersections of the road network for every possible affecting factors configuration in the environment. More specifically, the set of states S of the POMDP will be a set of intersection/affecting factor pairs $S = \mathcal{I} \times \mathcal{F}$ where $\mathcal{I}$ is the set of intersections, and $\mathcal{F}$ is the set of possible affecting factors in which we add the the element $nothing$. Hence, a pair (i, f) will indicate that affecting factor f is at intersection i, and ($i, nothing$) will indicate that intersection i is free of any affecting factor. As in the MDP framework, A, the set of all possible actions, will be every road segment one can choose to take at any intersection. The transition function $T((i, f), a, (i', f'))$ is similar to the transition probability defined for MDPs, and will only depends on i, a and i'. It represents the probability of reaching intersection i' when taking action a in intersection i. The transition function will remain deterministic ($T(s, a, s') = 1$, if a =going from s to s'). The reward function R specifies the immediate reward associated with performing an action a to state (i', f'). The reward function reflects, as it did with MDPs, the time required to take a particular action, to go from one state to another. Of course, if f' is $nothing$, the reward function will return -1, otherwise, it will return a larger negative value. The Observation Function O returns the probability of receiving an observation, having taken an action a, and reached a state (i', f'). In our model, the agent will be able to observe (receive an observation) an affecting factor if it is in the same intersection, or in an adjacent position. Thus, the set Ω, given in the POMDP definition, will contain all such possible observations. Recall that observations are used to update the belief state as discussed in Section 4.1.1. An example of this would be reaching a particular intersection on the road network and observing an enemy vehicle at the next intersection. Such an observation would change our belief state, since we would have new information about the location of a particular affecting factor. In addition, we would also become aware that the given affecting factor is not located elsewhere. Such an observation will decrease the partial observability of the system.

On top of affecting the Reward Function, the affecting factors also have an impact on the

observations. The observations remain unchanged and we are in a totally observable model (MDP), as long as no affecting factor is added to the environment. If affecting factors are present in the environment, the agent will likely encounter intersections which will yield observations allowing to refine the belief state, to gain additional knowledge on the affecting factors. Thus, such states are either intersection where the affecting factor is present or intersections where the affecting factor was believed to be present. As we saw in the example given earlier, new observations are used to update our belief state.

At this point we have to analyze this theoretical model to see if it is applicable to the dynamic path finding problem.

4.1.2 On the Usability of POMDP

We will now go over general solutions for POMDPS, and take a look at their usability for our path finding problem. We will quickly come to realize that, although the POMDP framework represents a great model for the path finding problem, the performance of the current algorithms is such that practical use of POMDPs seems impossible.

There exist many algorithms to solve POMDPs. In the context of this research we have evaluated three of them: Exact Value Iteration (EVI), Point-Based Value Iteration (PBVI) and Stochastic Search Value Iteration (SSVI). EVI was first presented in the 1970s [18], and was refined ever since. The general idea of EVI is to use dynamic programming to build the optimal policy. EVI first builds an optimal policy for an horizon of 1, then uses this policy to iteratively compute optimal policies for following horizons. This algorithm covers the entire problem space, computing exact values of for every belief state of the problem. PBVI [15] is an alternative solution to EVI. This approach computes the values at specific belief points rather than over the entire belief space, therefore requiring less computation. Finally, SSVI [13] looks deeper into the solution set and uses an action selection method which allows to focus exploration only on particular sequences of action.

The main drawback of these algorithmic approaches is their complexity, most approaches can only solve problems in the order of 10 to 1000 states. It is important to ask ourselves if we can rely on these approaches to solve our problem in practice. EVI execution has been shown [15] to grow exponentially based on the number of states ($|S|^2$) at every algorithm iteration, and POMDP algorithms typically require a exponential number of iterations. PBVI is a more scalable solution than EVI as it only explores a subset of belief points. However, using PBVI, it is possible to handle problems in the order of 10^3 states, which is one order of magnitude larger than problems solved using Exact Value Iteration [16]. To evaluate the applicability

of SSVI, we can refer to metrics accumulated on a problem counting 12545 states [13]. In this instance, SSVI yielded results in a time of 796 seconds (over 13 minutes). As discussed in Chapter 2, the number of states present in a real-life size problem range from 50000 to over 500000. Moreover, these figures don't even reflect the impact of the various potential affecting factors, which strongly influence the size of the POMDP state set, and performance of algorithms. The discussion stops here, as it becomes clear that computing times for real-life sized problems would become unacceptably high,, even if we want to limit ourselves to off-lines precalculations. .

Until improved POMDP algorithms are developed, we will have to rely on the deterministic and MDP approaches to solve our path finding problem. Nevertheless, in the next section, we will see how solutions obtained using the POMDP framework could differ from results computed using MDPs, highlighting the power of the POMDP approach, and limitations of the MDP approach. We will also show that some interesting results can sometimes be obtained by partially solving the POMDP (by solving it, without considering its non observable part).

4.2 MDP and POMDP Solutions

We will examine two examples of POMDPs. In the first one, we have an environment in which the agent is faced with a choice between two paths, one of which is safer, but longer than the other. We will see how POMDP would solve the problem. We will also see that in this particular case, using the MDP Q-Learning algorithm (called QMDP [14] in this context) could yield interesting results[1]. The second example is somewhat similar to the first. However, we will add a third possible path to the agent's potential choices. This second example will require the agent to have some memory of the past elements encountered in the episode. In this second case, we will see that to correctly solve the problem, we have to completely solve the POMDP, in the sense that the QMDP-Learning algorithm will not lead to the most suitable solution.

Choosing the Safest Path

In this first example, we want the algorithm to choose between two paths of different lengths. On one of these paths an affecting factor which has a major impact on the usability of the

[1]This algorithm, known by the name of QMPD, consists of applying a Q-Learning algorithm without factoring in the non-observability of POMDP. Of course, this approach offers guarantee of convergence

state (basically renders it unusable, and yields a large negative reward) will randomly appear. The decision of which path to choose will occur early in the episode and will possibly only be reversible by backtracking along already explored state. One path will allow to go around the risky state while the other will only go directly through it. The goal will be to find the shortest path that offers good chances of being able to avoid the affecting factor.

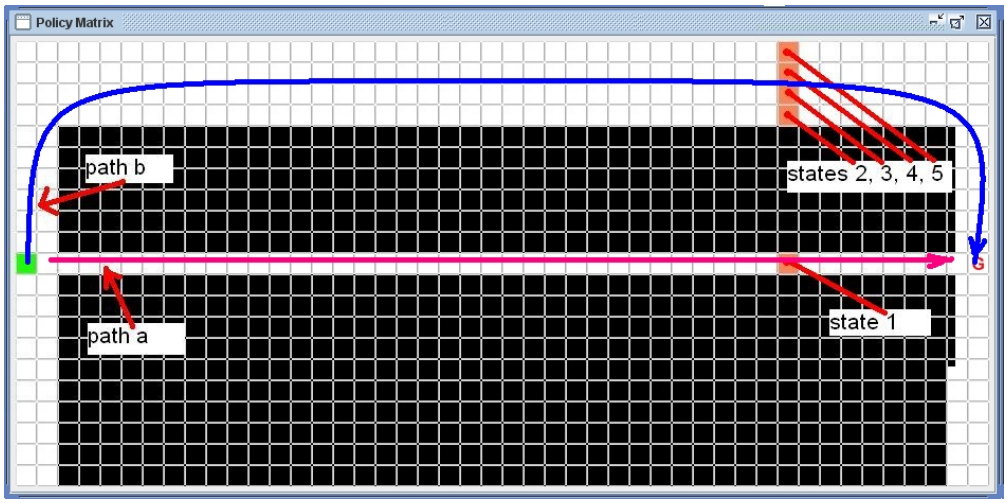

Figure 4.1: Illustration of the Sample Problem Environment

Figure 4.1 shows an illustration of such an environment. There are two possible ways of reaching the goal, using *path a* or *b*. The shortest path goes straight to the end state (*path a*), while the other goes up and around the environment (*path b*). Obviously, the shortest path is the straight one. However, we must also consider the orange states (*states 1* to *5*) as having an equal probability ($1/5$) of having a strong affecting factor. In this example, the state with the affecting factor is unusable (impossible to cross). In the absence of risk, *path a* offers a much better reward than *path b*. In the event *state 1* has the affecting factor, going down *path a* means encountering the affecting factor, receiving a high negative reward, and having to backtrack to use *path b*. In the cases where *states 2* to *5* have the affecting factor, going down *path b* will mean encountering the risk with a 0,25 probability, and being able to go around it in 3 extra actions. The objective is finding an action policy that will lead down *path b*. It is possible to configure and solve this problem POMDP approach. We would have to make a "copy" of the state set for every possible position of the affecting factor (in this case 5). We would then add a particular initial state which would branch to every possible "copy" of the environment.

We could use a deterministic or MDP approach to solve this problem. However, since the impact of the affecting factor would be equally distributed across the five orange states for every iteration, they would only select *path a* as the optimal choice (as it is shorter, and the affecting factor would have the same negative impact on every possible path). A

POMDP algorithm, on the other hand, would run the different iterations with the affecting factor impacting a different single orange state every time (chosen randomly) and would uncover a different solution. Let us examine the expected cumulative value of taking *paths a* or *b* that a POMDP algorithm will extract after a certain number of iterations. For *path a*, 4/5th of the time, the agent would encounter no threat, and would reach the goal in 48 steps. In 1 case out of 5, the agent would encounter the risky state, have to backtrack, and reach the goal in a minimum of 126 steps. For *path b*, 4/5th of the time, the agent would encounter the threat, and go around in (in 1 to 3 steps) and reach the goal in an average of 63.5 steps. In 1 case out of 5, the agent would encounter no threat, and reach the goal in an average of 62 steps. For *path a* we obtain an expected 63.6 steps($0.8 * 48 + 0.2 * 126 = 63.6$). For *path b*, we have an expected 62.45 steps ($0.8 * 63.5 + 0.2 * 62 = 63.2$). The expected number of steps required would be slightly lower for *path b* than *path a*, and this difference would grow if the amount of states required to turn back increased. A POMDP environment would allow to iteratively converge towards the smallest expected number of steps, as opposed to a deterministic or MDP framework.

It is interesting to note that, here, the QMDP algorithm will give the desired solution. Hence, in such case, the problem will remain tractable, even if we modelized it as a POMDP. For simple problems like this one, where only one affecting factor is present and only a limited number of options are offered to the agent, the QMDP approach could yield interesting results. We briefly tested this theory by using the $QMDP$-Learning algorithm for this problem. We had an affecting threat randomly appear on one of the orange states at every iterations. The algorithm selected *path b* as the optimal in a lot of the cases. We didn't go into a lot of effort to correctly parameterize the algorithm or to effectively measure its performance, but more research on the topic would be justified. However, the QMDP algorithm would be useful only in some very limited particular cases, such as the one described in this example. For example, as soon as a second affecting factor was added to the environment, the results became inconclusive. This puts a damper on the potential practical use of QMDP for our path finding problem in general.

We will now look at a slightly more complex example, in which we clearly see added vallue of the POMDP framework, and the impossibility to simply use the QMDP algorithm to solve it.

A Third Possible Path

This new example is similar to the previous one. However, in this new example, we will add a third path (*path c*) which connects *path a* and *b*. Because of this third path, the agent will have an opportunity to switch between *path a* and *b* after having made the initial choice. The

choice to use *path c* will be based on knowledge of the affecting factor and its position. We will see that, in this context, making the correct decision requires to have some accumulated information, or memory, of the environment. We will see that only real POMDP algorithms are fit in such a context.

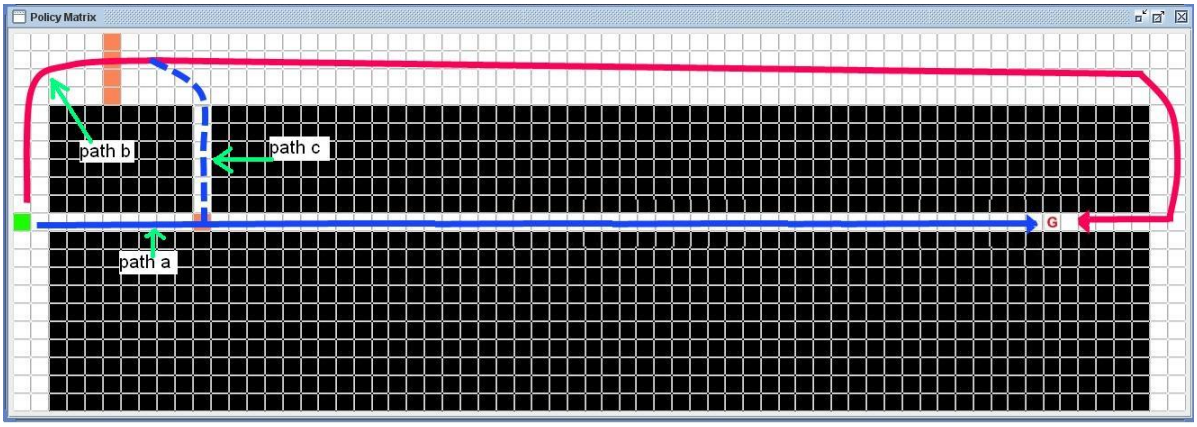

Figure 4.2: Modified Sample Problem Environment

Figure 4.2 illustrates the example environment. An affecting factor is present on one of the 5 orange states and is yielding a large negative reward, because of that, it is effectively "'blocking" the state. An agent can only observe the affecting factor if it is in the same state as the affecting factor or in an adjacent state. The objective is to go from the green state (on the left) to the goal state (marked with a "G"). Like in the first example, it is possible to model this environment, by building a "copy" of the system for each possible affecting factor position (5 copies in this case) and by adding a new initial state which would branch towards every "copy".

The initial choice is to either take *path a* or *path b*. We notice that it is possible to take a third path, *path c*, that goes through the single orange state to reach the goal faster. As far as *paths a* and *b* are concerned, the expected cumulative reward is higher for *path b* than it would be for *a*, and a QMDP algorithm might, in time, uncover the appropriate policy (using *path b*). It gets somehow trickier later on, when the agent, having initially chosen *path b*, has to pick between staying on *path b* or taking a shortcut with *path c*. When executing an episode, if the agent initially picks *path b*, the choice of switching for *path c* should be affected by whether or not it has encountered the affecting factor. If the agent has encountered the affecting factor on *path b*, then *path a* and *c* are risk free. This means that the agent should choose *path c* after having encountered the affecting factor. The MDP framework (and QMDP algorithm on the POMDP framework) is not fit to handle such a level of complexity. A MDP algorithm will consider different alternatives and evaluate their expected values. It will not keep track of the relationship between them. If encountering a particular state affects the likelihood of encountering future environment configurations, our standard MDP definition becomes unfit.

This is referred to as the Markov Property, which all states of the environment must possess in order for MDPs to be functional. The Markov property is summarized as:

$$P^{a}_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \tag{4.1}$$

This means that the next state we reach depends only on the state and action at the current moment. The fact that we encountered some particular state in the past cannot be taken into account. In other words, a MDP system is memoryless. It only considers the current state and action.

POMDPs offer a way to keep track of the past events encountered in the current episode with the use of the belief function. As seen in this chapter, POMDPs keep track of the level of knowledge of the environment using belief states. Every time the POMDP agent takes a step, its belief state is updated based on its observations. This updated belief is then used to evaluate the expected cumulative reward of states. In the case of our example, in the beginning the agent has a belief that the affecting factor can be in any of the orange states. If the agent observes the affecting factor during an iteration, it will update its beliefs, now knowing exactly where the risk is located. If it encounters the affecting factor on *path b*, it will know *path a* and *c* to be risk-free and act accordingly.

MDPs offer the flexibility to deal with uncertainty to a certain extent. However, being memoryless, MPDs can only successfully tackle environments where the actions taken do not impact the knowledge of the environment. POMDP solve this problem with the use of belief states. Unfortunately, the poor scalability of current POMDP algorithms makes them impossible to apply to our problem.

Conclusion

Three Approaches

We have considered three approaches to take on the problem of path finding in dynamic environments. We first detailed a deterministic approach, effectively capable of dealing with predictable dynamism. Probabilistic approaches were also presented under two paradigms: Markov Decision Processes and Partially Observable Markov Decision Processes.

The deterministic approach proved to be an efficient way to discover single paths in an accurately predicted environment. Yielding results well within a second on problems of considerable size, this approach is to be favoured when little time is available to make a decision.

MDPs allow dealing with stochastic affecting factors distributed in environments where the various states do not impact each other: where the Markov Property is respected, thus with the important drawback that with this model, the agent has no memory of past events. MPD algorithms take (at least) a few minutes to provide results on problems of practical sizes. This makes them a bad option for live use. However, they can be useful during the planning phase, which precedes the execution of a plan. On top of being able to compute paths, they offer path fusion, decision aid and safest path capabilities, which are of interest.

POMDPs offer the greatest theoretical potential to deal with uncertainty. They can keep track of many "possible worlds" and compute optimal policies accordingly. Through the use of belief states, POMDP algorithms will keep track of their actual knowledge of the environment and plan accurate policies based on it. Unfortunately, at this point, POMDP algorithms can only manage problems of sizes much smaller than the ones we deal with in practice. Future research may lead to the discovery of a new algorithm that can solve larger sized problems and make POMDP the approach of choice.

There seems to be a trade-off between the computability and range of applicability within the three approaches. Indeed, deterministic approaches offer excellent practical performance

but few ways to deal with uncertainty. POMDP can theoretically manage unknown environments, but are practically unusable. The way we propose to use them, MDPs stand in between, computing optimal paths and providing results that are usable in the planning phase.

Future work

In order to be able to use MDPs effectively, work remains to be done in order to identify in which way the MDP algorithm should be parameterized in order to provide the best possible results. Variants of the algorithms detailed in this work should also be studied and compared. Having done so, practical experiments could be conducted in order to further validate, and possibly better tune the MDP approach to the problem.

Obviously, a lot of research is needed to produce a robust, applicable POMDP algorithm. Using the algorithm on a selected subset of the environment could be an interesting approach. Uncertainty would be effectively managed in a small part of the environment, and an alternate approach could be used for the remaining part.

Bibliography

[1] H. Aljibury and A. Arroyo. Creating q-table parameters using genetic algorithm. Florida Conference on Recent Advances in Robotics, 1999.

[2] Cline B.E. Tuning q-learning parameters with a genetic algorithm. http://www.benjysbrain.com/ThePond/Tuning.pdf, 2004.

[3] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[4] D.P. Bertsekas and J.N. Tsitsiklis. An analysis of stochastic shortest path problems. *Mathematics of Operations Research*, 16:580–595, 1991.

[5] P.E. Black. Dijkstra's algorithm. From Dictionary of Algorithms and Data Structures http://www.nist.gov/dads/HTML/dijkstraalgo.html, 2006.

[6] P. Dayan and T.J. Sejnowski. Td() converges with probability 1. In *Machine Learning*, pages 295–301, 1994.

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

[8] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–616, 1987.

[9] L. Goodman, A. Lauschke, and E.W. Weisstein. Dijkstra's algorithm. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/DijkstrasAlgorithm.html.

[10] T. Jaakkola, M.I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation*, 6:1185–1201, 1994.

[11] S. Kim, ME Lewis, and CC White III. Optimal vehicle routing with real-time traffic information. *IEEE Transactions on Intelligent Transportation Systems*, 6:178–188, 2005.

[12] J.B. Kruskal. On the shortest spanning subtree of a graph and the travelling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

[13] F. Laviolette and L. Tobin. A stochastic point-based algorithm for pomdps. In *Canadian Conference on AI*, pages 332–343, 2008.

[14] M. L. Littman, A. R. Cassandra, and L. P. Kaelbling. Learning policies for partially observable environments: Scaling up. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 362–370. Morgan Kaufmann, 1995.

[15] J. Pineau, G. Gordon, and S. Thrun. Point-based value iteration: An anytime algorithm for pomdps, 2003.

[16] J. Pineau, G. Gordon, and S. Thrun. Anytime point-based approximations for large pomdps. *Journal of Artificial Intelligence Research*, 27:335–380, 2006.

[17] S. Skiena. *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Addison-Wesley, Reading, MA, 1990.

[18] E.J. Sondik. *The Optimal Control of Partially Observable Markov Processes*. PhD thesis, Stanford University, 1971.

[19] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.

[20] C.J.C.H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992.

Appendix A

Detailed Tables of Time Required to Build Threats

Here, we give detailed tables of the time required to build threats for different amounts of time slots. Starting with 10 we will try to go up to 25k slots for every size threat. When the size of our 2D matrix ($E \times T$) nears 250M units, we reach the memory limit (1024MB), and are unable to obtain results, which will be indicated by a "-".

Table A.1 represents a dynamic threat spanning 10 time slots, and affecting from 1 to 500k edges. Initialization, value assignation, and total times are shown.

Edges	Init (ms)	Assign (ms)	Total (ms)
1	0	0	0
10	0	0	0
100	0	0	0
500	0	0	0
1000	0	0	0
5000	0	0	0
10000	0	0	0
25000	31	0	31
50000	32	0	32
100000	78	15	93
250000	204	15	219
500000	313	15	328

Table A.1: Time to Build a 10 Time Slots Threat

Table A.2 represents a dynamic threat spanning 100 time slots, and affecting from 1 to 500k edges. Initialization, value assignation, and total times are shown.

Edges	Init (ms)	Assign (ms)	Total (ms)
1	0	0	0
10	0	0	0
100	0	0	0
500	0	15	15
1000	0	0	0
5000	16	0	16
10000	31	0	31
25000	94	16	110
50000	125	15	140
100000	203	47	250
250000	578	94	672
500000	1016	187	1203

Table A.2: Time to Build a 100 Time Slots Threat

Table A.3 represents a dynamic threat spanning 250 time slots, and affecting from 1 to 500k edges. Initialization, value assignation, and total times are shown.

Edges	Init (ms)	Assign (ms)	Total (ms)
1	0	0	0
10	0	0	0
100	0	0	0
500	16	0	16
1000	0	0	0
5000	31	15	46
10000	47	16	63
25000	109	16	125
50000	172	47	219
100000	312	94	406
250000	812	219	1031
500000	3938	1015	4953

Table A.3: Time to Build a 250 Time Slots Threat

Table A.4 represents a dynamic threat spanning 500 time slots, and affecting from 1 to 500k edges. Initialization, value assignation, and total times are shown.

Edges	Init (ms)	Assign (ms)	Total (ms)
1	0	0	0
10	0	0	0
100	0	0	0
500	47	0	47
1000	16	0	16
5000	62	0	62
10000	78	16	94
25000	172	47	219
50000	328	94	422
100000	562	172	734
250000	1500	437	1937
500000	-	-	-

Table A.4: Time to Build a 500 Time Slots Threat

Table A.5 represents a dynamic threat spanning 1000 time slots, and affecting from 1 to 500k edges. Initialization, value assignation, and total times are shown.

Edges	Init (ms)	Assign (ms)	Total (ms)
1	0	0	0
10	0	0	0
100	0	0	0
500	31	0	31
1000	16	15	31
5000	79	15	94
10000	110	31	141
25000	281	78	359
50000	484	188	672
100000	1000	359	1359
250000	-	-	-
500000	-	-	-

Table A.5: Time to Build a 1000 Time Slots Threat

Table A.6 represents a dynamic threat spanning 5000 time slots, and affecting from 1 to 500k edges. Initialization, value assignation, and total times are shown.

Edges	Init (ms)	Assign (ms)	Total (ms)
1	0	0	0
10	0	0	0
100	15	16	31
500	47	0	47
1000	62	16	78
5000	265	94	359
10000	500	172	672
25000	1250	438	1688
50000	-	-	-
100000	-	-	-
250000	-	-	-
500000	-	-	-

Table A.6: Time to Build a 5000 Time Slots Threat

Table A.7 represents a dynamic threat spanning 10000 time slots, and affecting from 1 to 500k edges. Initialization, value assignation, and total times are shown.

Edges	Init (ms)	Assign (ms)	Total (ms)
1	0	15	15
10	0	0	0
100	16	0	16
500	78	31	109
1000	109	47	156
5000	469	188	657
10000	859	359	1218
25000	-	-	-
50000	-	-	-
100000	-	-	-
250000	-	-	-
500000	-	-	-

Table A.7: Time to Build a 10000 Time Slots Threat

Table A.8 represents a dynamic threat spanning 25000 time slots, and affecting from 1 to 500k edges. Initialization, value assignation, and total times are shown.

Edges	Init (ms)	Assign (ms)	Total (ms)
1	0	0	0
10	15	0	15
100	63	0	63
500	140	47	187
1000	219	94	313
5000	1218	422	1640
10000	-	-	-
25000	-	-	-
50000	-	-	-
100000	-	-	-
250000	-	-	-
500000	-	-	-

Table A.8: Time to Build a 25000 Time Slots Threat