The Difficulty of Designing a General Heuristic Agent Navigation Strategy

Mikael Fors  Madelein Hermelin
6th June 2011

Abstract
We consider an abstract representation of some environment in which an agent is located. Given a goal sequence, we ask what strategy said agent - utilizing readily available algorithmic tools - should incorporate to successfully find a valid traversal route such that it is optimal in accordance with a predefined error-margin. We present four scenarios that each incorporate aspects common to general navigation to further illustrate some of the difficult problems needed to be solved in any general navigation strategy. Two reinforcement learning and four graph path planning algorithms are studied and applied on said predefined scenarios. Through the introduction of a long-term strategy model we allow comparative study of the result of the applications, and note a distinct difference in performance. Further, we discuss the lack of a probabilistic algorithmic approach and why it should be an option in any general strategy as it allows verifiably “good” estimated solutions, useful when the problem at hand is NP-hard. Several meta-level concepts are introduced and discussed to further illustrate the difficulty in producing an optimal strategy with an explicit long-term horizon. We argue for a non-deterministic approach, looking at the apparent gain of $\epsilon$-randomness when incorporated by a reinforcement learning agent. Several problems that may arise with non-determinism are discussed, based on the notion that such an agents’ performance can be viewed as a markov chain; possibly resulting in suboptimal paths concerning norm.

Keywords: Agent Navigation, Path Planning, Heuristics, Nondeterminism, Artificial Intelligence, Terrain Exploration Optimization
Contents

1 Introduction 1
   1.1 Framework .......................... 1
   1.2 Aim ................................ 2
   1.3 Demarcations .......................... 3
   1.4 Method ............................... 3
   1.5 Outline ............................... 4

2 Scenarios 4
   2.1 Scenario I .......................... 4
   2.2 Scenario II .......................... 5
   2.3 Scenario III ........................ 5
   2.4 Scenario IV ........................ 7

I Theory 8

3 Reinforcement Learning 8
   3.1 RPROP ................................ 9
   3.2 Q-Learning ............................ 11
   3.3 SARSA ................................ 11

4 Graph Algorithms 12
   4.1 Dijkstra\textquotesingle s Shortest Path Algorithm .... 12
   4.2 A* ..................................... 12
   4.3 D* ..................................... 15
   4.4 LPA* .................................. 16
   4.5 D* lite ................................ 17

II Analysis 21

5 Long-term strategy model 21
   5.1 Applied application of $\varphi$ .......... 22

6 Scenario Outcome 22
   6.1 Scenario I .......................... 22
   6.2 Scenario II .......................... 24
   6.3 Scenario III ........................ 25
   6.4 Scenario IV ........................ 25

7 Reducing chance of failure 27
   7.1 A Fuzzy View .......................... 28

8 A general strategy is non-deterministic 32

9 Conclusion 33
Acknowledgements

We would like to extend our gratitude towards Professor Andreas Hamfelt for being our supervisor. His input as well as the level of freedom we have enjoyed while working on this thesis have been greatly appreciated. Further, we are thankful to the Department of Information Technology of Uppsala University for providing us with GridWorld; a tool we utilized quite a lot. Special thanks also go out to Olle Gällmo for his excellent course in machine learning that sparked our interest in this topic (well, portions of what we have covered anyhow). We thank Lennart Salling for his wonderful course on automata theory, the knowledge gained through taking said course has proved valuable in many situations. Finally, we thank each other - yes, we are this sweet.
1 Introduction

Navigation is a very complex task that we, as humans, rarely consider. Very few of us ever think twice of how we solve the problem of suddenly finding our path blocked. We simply apply a solution - finding an alternate way - without thinking. When asked, we might reply that it is a logical approach; if you cannot go through it - go around it. While this might be the case, a more interesting question is how our search for an alternate path works. In computer science, the idea of transforming a situation into an abstraction is central, as it allows one to focus on the actual difficulties apparent in the problem, rather than other non-related pieces of information. However, while we can easily construct an abstract representation of some environment, populate it with an agent and define a goal sequence, we cannot reduce the notion of a general heuristic navigation strategy to a few explicit rules. There is a duality to the task at hand. On one hand we want the agent to utilize a heuristic approach, and on the other we also want to limit the behaviour due to meta-level constraints. Not only is the idea of navigation very general, which in itself is a problem as it is very difficult to explicitly describe a general situation with a limited number of rules; but it is subjective. This is especially apparent when we consider the wide range of utility an agent may have. We must therefore divide any abstraction of a navigation situation into two parts, one regarding the abstraction of the environment and one concerning the subjective aspects of the agent.

The composition of these two abstractions regarding the situation is central to the subjective problem that is to be solved. Through the subjective definition of the agents’ goals, we also eliminate the rather difficult task of explicitly stating what constitutes a good solution; it is implicitly defined. Further, this representation allows ut to consider many meta-level problems that, while not immediately apparent in the problem description, appear due to previously discussed subjective constraints. In essence this is a very interesting notion. By reducing the difficulty in constructing an abstraction of the situation as such, several meta-level problems arise. It appears as if one simply cannot avoid the complex nature apparent in the general problem.

1.1 Framework

Consider the set of all $\Gamma_1 \times \Gamma_2$ matrices with values in $\mathbb{Z}_2$ such that there is at least one non-zero and one zero index. That is, let

$$E_{\Gamma} = \{ E \subseteq \mathbb{R}^{\Gamma_1 \times \Gamma_2} \mid (\forall i, j) E_{ij} \in \mathbb{Z}_2) \land (\exists i \in \mathbb{Z}_{\Gamma_1+1}^+ \exists j \in \mathbb{Z}_{\Gamma_2+1}^+ : E_{ij} = 0) \land (\exists i \in \mathbb{Z}_{\Gamma_1+1}^+ \exists j \in \mathbb{Z}_{\Gamma_2+1}^+ : E_{ij} \neq 0)\}$$

then there are elements $e_i \in E_{\Gamma}$ for $0 \leq i < |E_{\Gamma}|$ such that they have indices $\iota_1 = (\alpha_1 \alpha_2)$ and $\iota_2 = (\alpha_3 \alpha_4)$, where $\langle \iota_1, \iota_2 \rangle$ is a traversable 0-valued path. Since any such path can be thought of as a curve, we denote it $C_{\iota_1, \iota_2}$, and say that
it is generated by $r(t) = x_1(t)e_1 + \cdots + x_n(t)e_n$, where $t$ is the step, $x_1 \cdots x_n$ denote the dimension functions and $e_1 \cdots e_n$ are (standard) basis vectors in $\mathbb{R}^n$.

1.2 Aim

The aim of this thesis is to demonstrate the difficulties in designing a general heuristic navigation strategy such that it is optimal, utilizing modern algorithmic approaches. This is done by defining four general scenarios that reflect typical navigational problems such that they are likely to arise and should thus be readily covered in any general strategy. We present several graph traversal algorithms that are typically utilized in current applications as well as two reinforcement learning algorithms and then apply these on the scenarios presented. The primary focus is on optimization in regard to the restrictions defined in the strategy model $\mathcal{M}$ utilized, viz. application, cost, redundancy and scenario success. We argue that a general strategy, in accordance with the analytic results we provide, requires a non-deterministic approach involving several artificial intelligence elements that form a basis which can then be trained using, for instance, a neural network to better comply with the dynamic scenarios presented to it.

NOT($s$) = \begin{cases} 0 & \text{if } s = 1 \\ 1 & \text{if } s = 0 \end{cases}

In this thesis we consider the general question of heuristic navigation for some agent in a partially solvable scenario set in an environment $e_i \in E$. By altering $g$ we study the task of static exploration, cycle optimization and path finding. Further, we introduce a matrix $A \in \mathbb{R}^{\Gamma_1 \times \Gamma_2}$ such that $f_A(e_i) : e_j$, where $e_j$ is $e_i$ with $k$ NOT indices, given the restriction of $e_{j_1j_2}$, where $A_{\Delta_i}.current\_position = (\frac{q_1}{q_2})$, always remaining unaltered. We study dynamic performance of said agent by invoking $f_A(e_i)$ during traversal.

We define a Long-Term Strategy Model $\mathcal{M}$ to be a goal definition with implicit traversal restrictions. A navigation strategy is said to be optimal iff there is a predefined - in $\mathcal{M}$ - constant $\varphi$ such that $|k_1 - k_2| \leq \varphi$, where $k_1$ is the cost-yield ratio of the agent with the current path and $k_2$ is the cost-yield ratio of a path proposed by utilizing an oracle$^2$ viz. a perfect path.

1. An oracle is a turing machine$^{18\ 41}$ that solves a decision problem in one step.
1.3 Demarcations

While discussed, applied strategies is an area too great for the scope of this thesis and will therefore not be considered on a detailed level. In addition, there may be additional optimized versions of covered algorithms but these will not be considered. We do not explore all possible outcomes as far as scenarios go and thus it is imperative to state the limitation of the scope of this thesis explicitly. That is, while the results as such may be considered accurate, they are so only in the context presented.

Due to time constraints not all algorithms are tested on all scenarios. We will, however, through mathematical and logical means, discuss their capabilities based on their algorithmic outlines. We will explicitly state when actual experimentation has been performed and we acknowledge the fact that we utilize code not written by us, which enables a possibility of error that is out of our control. However, efforts have been made to obtain code from reliable sources, i.e. sound sources such as the author(s) of the algorithms or academic professionals with a major interest in the field studied to minimize the chance of error. In light of this fact, we note that our findings are general and as such do not rely on exact performance of the algorithms; but rather the properties defined in their respective pseudocode. As these are obtained through reliable sources, viz. peer reviewed or otherwise verified means, the results can be thought of as just, while the exact performance results can soundly be questioned.

Further, while we include essential background information on several areas covered in the appendices, it should be noted that we do expect a certain basic level of mathematical knowledge from the reader (linear algebra, analysis of one and multiple variables, basic set theory, elementary logic, fundamental algebra). We also expect programming skills in some language and fundamental knowledge of complexity theory (we provide a brief description in the appendix).

1.4 Method

We construct a solid basis for our work by defining four scenarios upon which we perform the experimental portion of the thesis. We continue by defining a long-term strategy model to ensure a fair basis for comparative study of the proposed algorithmic approaches. The performance issues apparent with reinforcement learning agents is demonstrated through an applied experiment on scenario one. We discuss, from a theoretical point of view, the non-issue of extending the scenario with multiple goals; should they only require recurrent algorithm application as is apparent in scenario two. The lack of a probabilistic approach is illustrated as we discuss scenario three, in which NP-hardness renders our proposed algorithmic tools rather useless from a cost-gain ratio perspective. We conduct an experiment on scenario four, testing the dynamic capabilities of A*, LPA*, D* LITE, Q-learning and SARSA. We continue by discussing apparent meta-level issues that arise with the notion of a long-term strategy model. All claims made on a theoretical basis are supported by mathematical soundness and proof to ensure correctness in the context they are made. We state assumed limitations explicitly.
1.5 Outline

We begin by defining the four scenarios utilized in the study through descriptive and mathematical means. This is followed by Part I in which the algorithms studied are presented and explained, both through wording and by presenting their pseudocode. We also discuss resilient backpropagation to further illustrate how a neural network can be trained to be implemented in a reinforcement learning situation. This is followed by Part II in which we analyse the algorithms in accordance with the scenarios previously defined and illustrate the strengths and weaknesses of said algorithms. We show that while each algorithm does provide certain aspects that are desirable, a general strategy requires several candidate algorithmic approaches as none of the ones studied are ideal, viz. optimal in all cases. Further, we illustrate and discuss several meta-level difficulties that arise due to constraints defined in the long-term strategy model. We continue by arguing for a non-deterministic general strategy incorporating several artificial intelligence elements to provide a wide basis that can then be trained to comply with the scenario defined in the long-term strategy model utilized.

2 Scenarios

Let $e_{\Delta} \in \mathcal{E}_{\Gamma}$ be a partially solvable environment with an agent $A_{\Delta \Gamma}$ and let $\forall i \in \mathbb{Z}_{T_1+1}^+ \forall j \in \mathbb{Z}_{T_2+1}^+ : e_{\Delta ij} = 1 \iff e_{\Delta ij}$ is a wall. Since we define $\langle \alpha, \beta \rangle$ to be a 0-valued path in an environment, it follows that for every index $ij$ in $e_{\Delta ij} = 0 \iff e_{\Delta ij}$ is traversable. In an applied scenario this abstraction may not hold as angle of direction is introduced. Consider

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

where $A_{11} \rightarrow A_{12}$ is an invalid move, as is $A_{22} \rightarrow A_{12}$. However, in an applied situation there might be an angle $\alpha$ such that $A_{12}$ is traversable from either $A_{11}$ or $A_{22}$. We shall, however, not consider this fact when considering the abstract maps utilized in this thesis. To support more complex scenarios, we introduce $\varphi \subseteq \mathbb{Z}^+$ as the set of possible non-zero values present in any index, allowing additional terrain information in addition to traversability. As such

$$\forall i \in \mathbb{Z}_{T_1+1}^+ \forall j \in \mathbb{Z}_{T_2+1}^+ : \mathcal{E}_{\Gamma ij} \in (\varphi \cup \{0\}) \subseteq \mathbb{Z}_{(\max \varphi)+1}$$

2.1 Scenario I

Let $A_{\Delta \Gamma}$ be an agent with initial position $p$ and let $\langle p, g \rangle$, with $|g| = 1$ be the global goal for $A_{\Delta \Gamma}$. Since $A_{\Delta \Gamma}$ operates in $e_{\Delta} \in \mathcal{E}_{\Gamma}$, which was defined as partially solvable, $\langle p, g \rangle$ must exist, but it may not necessarily be unique. We shall consider $p_1 = \langle \alpha, \beta \rangle$ better than $p_2 = \langle \alpha, \beta \rangle$ iff $||p_1|| < ||p_2||$. As such the global goal may be fulfilled, yet not be ideal, should multiple paths be possible.

The first scenario is a very fundamental part of any navigational strategy used by an agent as it is equivalent to the simple process of path finding. As stated in the scenario definition, there may be various degrees of solutions; should multiple
paths be possible. Said degrees denote the norm of the path and while it may not always hold that smaller norm is better, for instance due to slope, we shall consider that so is the case for simplicity.

### 2.2 Scenario II

Let \( \langle p, g \rangle \) remain a global goal but for every point \( \phi \in \mathcal{W} = \{ \phi \in e_\Delta \mid \phi \text{ is interesting} \} \) that we encounter while finding \( \langle p, g \rangle \) we want to store the path \( \langle p, \phi \rangle \) and if new information is made available, possibly update all such paths.

In this scenario we have thus added an additional global goal to the agent. Namely to find, and maintain, a set of paths from the origin to any point of interest that is discovered while looking for a path to \( g \). We are especially keen on the notion of keeping said path list up to date, in the sense that if a shorter path is made available we wish to store it, rather than the previous and thus longer path.

It should be noted that in an actual applied situation the set \( \mathcal{W} \) is subjective in the sense that it is variable what points are of interest. However, this is not a problem per se, but rather just enforces the requirement of a clear mission definition. Should the situation be such that the environment is complex, i.e. it may be non-trivial to determine whether a given point is interesting without some sort of investigation by the agent, we must consider the notion that the time complexity of any algorithm may be greatly increased should the complexity of the investigation exceed that of the path finding algorithm. Accordingly, it may be the case that tests such that they require a constant, yet lengthy, timeframe to complete are required and the final performance may thus be rather difficult to generalize. We mention these limitations of a fair general analysis but we shall not consider them in our discussion as they add far too much complexity for the scope of this thesis.

### 2.3 Scenario III

Let \( w \) be a collection of points in \( e_\Delta \in \mathcal{E}_r \). We want to find a path \( c = (p, g_\alpha, \cdots, g_\omega, p) \) containing all points \( g_i \in g \), such that \( |c| \) is as small as possible, i.e. \( ||c| - |c_{\text{perfect}}|| \) is as close to 0 as possible.

In order to attack this problem, note that in every inner product space \( V \)

\[
|\langle v, w \rangle| \leq ||v|| \times ||w|| \quad \forall v, w \in V
\]

(1)

which can be generalized \([4]\) for our purposes as the triangle inequality \([17]\), stating \( |x| - |y| \leq |x + y| \leq |x| + |y| \) for vectors \( x, y \in \mathbb{R}^n \) (see Figure \([1]\)). That is, for any two points \( w_1, w_2 \in w : |w_1 - w_2| \leq |w_1 + w_2| \).

In accordance with the Cauchy-Schwarz inequality \([1]\) and thus the triangle inequality, it follows that in a situation such that we have an origin surrounded by points, it will always be better to go from point to point, rather than return to the origin in between points. However, in an actual applied situation this does not always apply as it may not be possible to take the direct route. While this might seem problematic, the Cauchy-Schwarz inequality still gives us an upper bound on the shortest path between two points, which we summarize in a lemma.
We have the three vectors
\[ x = (1.5, 0.7) \]
\[ y = (-1, 0.5) \]
\[ x + y = (1.5, 0.7) + (-1, 0.5) = (0.5, 1.2) \]
and
\[ ||(0.5, 1.2)|| \leq ||(1.5, 0.7)|| + ||(-1, 0.5)|| \]

Figure 1: Triangle Inequality

**Lemma 1.** In an applied situation, the shortest path between points \( x, y \) cannot exceed \( \sigma(0, x) + \sigma(0, y) \), where \( \sigma(\alpha, \beta) \) denotes the shortest path between \( \alpha \) and \( \beta \).

**Proof.** The Cauchy-Schwarz inequality tells us that \( |x + y| \leq |x| + |y| \), so if \( x\bar{y} \) is not directly traversable there is either a path \( p \) such that \( ||x\bar{y}|| < p < |x| + |y| \) or \( |x| + |y| \) is the shortest path as there was no such \( p \); meaning that \( |x| + |y| \leq p \). □

Using Lemma 1 we can conclude that we always know the path \( c \) such that it is the largest minimum path, namely the path \( (p, v_1, p, \cdots, v_j, w, p) \), where \( v_i \in \pi|w| \) \( P_j \in \text{permutations} \). Unfortunately, as we will now show, finding \( c \) such that it is minimal, is a NP complete problem [19].

To see why this is the case, consider the scenario where we know \( \sigma(w_\alpha, w_\beta) \) for all \( \alpha, \beta \in M = \{1, \ldots, |w|\} : \alpha \neq \beta \). Then we could create a graph \( G = (V, E) \) where \( V = w \cup \{\text{origin}\} \) and \( E = \{(v_\alpha, v_\beta) \mid v_\alpha, v_\beta \in V \land v_\alpha \neq v_\beta\} \) having weights \( \sigma(v_\alpha, v_\beta) \). This would be an ideal situation, because then all the best interconnecting paths are known and all that remains is to find an optimal path containing all vertices starting and ending at the origin using said edges. However, this is equivalent to the traveling salesman problem [20] which is known to be NP complete (see Figure 2). Hence, to find the optimal solution to the problem involving 60 points (including the origin) is equivalent to having to verify

\[ \frac{60!}{2} = \prod_{k=3}^{60} k \]

permutations, a number which exceeds \(^4 10^{80} \) - the number of atoms in the observable universe. The task of obtaining an optimal path \( c \) therefore has a non-polynomial worst-case time complexity and we formulate the goal in such a way that the output is a decent solution. That is, a solution that may not necessarily be optimal, but which gives a good estimate of a near-optimal solution.

\(^3\)That is, any of the possible permutations of the nodes; \( \pi(w) \).

\(^4\)By a factor of 41.6.
2 SCENARIOS

2.4 Scenario IV

Let $e_\Delta$ and $A_{\Delta r}$ like previously and let $A \in \mathbb{R}^{\Gamma_1 \times \Gamma_2}$ such that $f_A(e_i) : e_j$, where $e_j$ is $e_i$ with $k$ NOT indices, given the restriction of $e_{j_{\text{current}}}$, where $A_{\Delta r}.\text{current}\_\text{position} = \left( \begin{smallmatrix} q_1 \\ q_2 \end{smallmatrix} \right)$, always remaining unaltered. That is, by invoking $f_A(e_i)$, we perform a dynamic alteration of $e_i$.

For instance, let $e_\delta = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$, $q_1 = q_2 = 4$ and $A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$, then

$$f_A(e_\delta) = Ae_\delta = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

is a valid transformation as $e_{\delta_{44}} = (f_A(e_\delta))_{44}$ and both $e_\delta$ and $f_A(e_\delta)$ are partially solvable, given $g = \{(1, 1)\}$. However, note that there may be a $e_\delta$ that is not invertible, for instance if $\det(e_\delta) = 0$ [4]. Likewise, since it is possible that $\Gamma_2 \neq \Gamma_2$, $e_\delta$ might not be a square matrix. We can overcome the latter of these problems by noting that if $e_\delta \notin \mathbb{R}^{n \times n}$, we can perform the operation $f_A$ on a portion of $e_\delta$.

However, the fact that $e_\delta$ may not be invertible is problematic unless we define $f_A(x) = A + x$ rather than $f_A(x) = Ax$. Since it is preferable to discuss the environment alteration in terms of operations rather than individual index alterations, we will define $f_A$ to comply with the proposed changes, i.e. if $e_\delta = [010; 010; 000]$ then $f_A(e_\delta) = [000; 110; 100]$ is possible. That is, $f_A(x)$ can either be $Ax$ or $A+x$.

Letting $f_A(x) = A + x$ will always be a safe choice, as both the domain and range of $f_A(x) = A + x$ is $\mathbb{R}^{n \times m}$. To see why this is the case, note that $(A + x) \in \mathcal{P}_1 - A$ constant - and since polynomials are continuous at all points the result for the
3 REINFORCEMENT LEARNING

domain follows. To prove that \( f_A(x) = A + x \) has \( \mathbb{R}^{n \times m} \) as range, note that we can consider both \( A \) and \( x \) as numbers with an arbitrary number of prepended zeroes. Since \( (\mathbb{Z}, +) \) is a group,

\[
\forall c, x \in \mathbb{Z} \exists A \in \mathbb{Z} : A + x = c
\]

the range must be \( \mathbb{Z} \), should we consider \( A, x \in \mathbb{Z} \). That is, the range is equivalent to the domain; \( A, x \in \mathbb{R}^{n \times m} \Rightarrow \text{range}(A), \text{range}(x) = \mathbb{R}^{n \times m} \), completing the proof.

Part I

Theory

3 Reinforcement Learning

Originating from the fundamental concepts of animal training; reinforcement learning is based on the notion of reward and punishment. In general, the situation is such that the animal, or in our case the agent, gets rewarded for correct actions and punished for faulty actions. An agent receives feedback from the environment in which it is working through rewards - or lack thereof - along with possible inputs through any sensory inputs available to it. The learning process in reinforcement learning is thus heavily based on the notion of environment state; that is, what action is desirable - from a reward perspective - at a given point. Any learning must therefore commence through interaction with the environment and is, at least during the initial episodes, very much based on the notion of trial and error. Hence, the agent “discovers” what actions are desirable by making mistakes and - hopefully - finding at least some actions that yield an award.

![Figure 3: Reinforcement Learning.](image)

There are three main components of an agent in a reinforcement learning situation:

- **Policy.** The decision making function of the agent that is used to determine what actions to execute based on the current state. We may consider the policy as a set of tuples \((\alpha, \rho)\), where \(\alpha\) is an action and \(\rho\) a response.

- **Reward function.** Defines which actions are desirable through rewards and which are not. All rewards are immediate and represent the current
environment state the agent is in. The long run goal of any agent is to maximize the overall reward received throughout the run.

- **Value function.** Predicts future rewards and thus indicates what actions are favorable in the long run.

Engelbrecht [2] states that the value function is of particular interest, and more specifically the problem lies in “how the future should be taken into account”. A few models for how this can be done have been suggested by Kaelbling et al. [3].

\[
M_{\text{finite-horizon}} = E \left[ \sum_{t=1}^{n_t} r(t) \right]
\]  
\[
M_{\text{infinite-horizon}} = E \left[ \sum_{t=0}^{\infty} \gamma^t r(t) \right]
\]  
\[
M_{\text{average-reward}} = \lim_{n_t \to \infty} E \left[ \frac{1}{n_t} \sum_{t=0}^{n_t} r(t) \right]
\]

These models vary in how the general strategy is to be outlined. In essence we may consider the overall goal to center around maximizing the finite - and thus the somewhat immediate - future through model 2, or rather focus on an infinite time horizon as in equation 3. The model described in equation 4 is perhaps of greater interest in a general scenario, as we focus on a stable output, i.e. maximizing the average reward. To ensure an optimal policy one needs to determine an optimal value function, which is also suggested by Kaelbling et al. [3], shown in equation 5.

\[
V^*(s) = \max_{a \in A} \left\{ R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V^*(s') \right\}, s \in S
\]

It is important to stress that \( A \) is the set of all possible actions (and should thus not be confused with the previously defined agent \( A_{\Delta \Gamma} \)), \( S \) is the set of environmental states, \( R(s,a) \) is the reward function and \( T(s,a,s') \) is the transition function. Hence, one needs to define the models in terms of \( T \) and \( R \), which is quite challenging and will not be covered in this thesis as we will focus on model-free reinforcement learning when implementing the algorithms studied. It should, however, be noted that models tend to be very useful when they are mathematically sound, which obviously is a problem in itself as it may be very difficult to design models that are general enough for a wider application. Further, we note that the models for future rewards, while not utilized by us, are of interest as they suggest a flaw of reinforcement learning when it comes to allowing dynamic planning. That is, should the agent at any time want to change strategy, the data previously processed will be somewhat useless; which may prove problematic when we are dealing with dynamic scenarios.

### 3.1 RPROP

Resilient backpropagation, or RPROP for short, is a supervised learning method utilized in feedforward neural networks. Originally proposed by Riedmiller and Braun in 1993 [5], there have been several proposed improvements, for instance
3.1 RPROP

RPROP

1. NN weights are initialized to small random values.
2. Set $\Delta_{ij} = \Delta_{kj} = \Delta_0, \forall i = 1, \cdots, I+1, \forall j = 1, \cdots, J+1, \forall k = 1, \cdots, K$
3. $t = 0$
4. repeat
5. for each $w_{kj}, j = 1, \cdots, J+1, k = 1, \cdots, K$
6. if $\frac{\partial E}{\partial w_{kj}}(t-1)\frac{\partial E}{\partial w_{kj}}(t) > 0$
7. $\Delta_{kj}(t) = \min\{\Delta_{kj}(t-1)\eta^+, \Delta_{\text{max}}\}$
8. $\Delta w_{kj}(t) = -\text{sign}\left(\frac{\partial E}{\partial w_{kj}}(t)\right)\Delta_{kj}(t)$
9. $w_{kj}(t+1) = w_{kj}(t) + \Delta w_{kj}(t)$
10. elseif $\frac{\partial E}{\partial w_{kj}}(t-1)\frac{\partial E}{\partial w_{kj}}(t) < 0$
11. $\Delta_{kj}(t) = \max\{\Delta_{kj}(t-1)\eta^-, \Delta_{\text{min}}\}$
12. $w_{kj}(t+1) = w_{kj}(t) - \Delta w_{kj}(t-1)$
13. $\frac{\partial E}{\partial w_{kj}} = 0$
14. elseif $\frac{\partial E}{\partial w_{kj}}(t-1)\frac{\partial E}{\partial w_{kj}}(t) == 0$
15. $\Delta w_{kj}(t) = -\text{sign}\left(\frac{\partial E}{\partial w_{kj}}(t)\right)\Delta_{kj}(t)$
16. $w_{kj}(t+1) = w_{kj}(t) + \Delta w_{kj}(t)$
17. until stop conditions $== \text{true}$

Algorithm 1: RPROP

by Igel and Hüsken [6]. We shall only consider the original version, as our main focus is not on the ideal performance per se, but rather lies with the possibilities of applying Q-learning and SARSA on the scenarios studied. The method as such, centers around the notion of altering the weights based on the sign of the partial derivatives $\frac{\partial E}{\partial v_{ji}}$ or $\frac{\partial E}{\partial w_{kj}}$. If there is a sign change, the update value $\Delta_{ji}$ or $\Delta_{kj}$ is decreased by $\eta^-$ since the last weight update resulted in the algorithm jumping over a local minimum. Likewise, if the sign is retained, the update value is increased by $\eta^+$ to increase the rate of convergence. The following equations

$$\Delta v_{ji}(t) = \begin{cases} -\Delta_{ji}(t) & \text{if } k > 0 \\ +\Delta_{ji}(t) & \text{if } k < 0 \\ 0 & \text{otherwise} \end{cases}, \text{where } k = \frac{\partial E}{\partial v_{ji}}$$

$$\Delta_{ji}(t) = \begin{cases} \eta^+\Delta_{ji}(t-1) & \text{if } m > 0 \\ \eta^-\Delta_{ji}(t-1) & \text{if } m < 0 \\ 0 & \text{otherwise} \end{cases}, \text{where } m = \frac{\partial E}{\partial v_{ji}}(t-1)\frac{\partial E}{\partial v_{ji}}(t)$$

determine the actual weight updates, which translates into $v_{ji}(t+1) = v_{ji}(t) + \Delta v_{ji}(t)$. We present RPROP in its entirety in algorithm [1]. Note that we present this batch learning approach, which is offline, to further enhance the apparent traits of reinforcement learning. The algorithm itself demonstrates ideas central to the notion of artificial intelligence, which will be discussed later.
3.2 Q-Learning

In Q-Learning\cite{21} we let the greedy\footnote{Note that by greedy we do not mean a greedy approach, but rather a \textit{maximum} as far as utility goes, viz. the Q-value should reflect the best possible outcome.} choice be recursively defined for every state $s$. The outcome is then saved as the states $Q-value$ in direction $d$, denoting the direction chosen by the agent. We let the goal return a reward which in turn yields a theoretical reward, in the sense that every step which brings the agent closer to said reward can be thought of as rewarding; albeit not directly so\cite{22}. In equation\cite{6} we state the ideal value of state $s$ assuming that the best action is taken initially, $Q(s, a)$ denotes the reinforcement value of taking action $a$ in state $s$.

\begin{equation}
V^*(s) = \max_a Q^*(s, a) \tag{6}
\end{equation}

We let $\eta$ denote the learning rate (as usual) and $\gamma$ is a value used to ensure that the sum is absolutely convergent (we may consider infinite grids in theory), viz. we only add a fraction of the optimal yield of the next state to the current.

\begin{equation}
Q(s, a) = Q(s, a) + \eta (r + \gamma \max_{a' \in A} Q(s', a') - Q(s, a)) \tag{7}
\end{equation}

3.3 SARSA

Unlike Q-LEARNING, SARSA - “State-Action-Reward-State-Action” as suggested by Rich Sutton (see\cite{23}) - does not consider the yield of the next state, $Q(s', a')$, to be greedy\footnote{Again, we note that by greedy we simply mean that the next posistions’ state-action tuples are considered from a \textit{maximum} yield perspective, not that the algorithm itself is greedy.}; viz. the Q-value for any action $a$ in a state $s$ is the yield of the action the agent will actually take. In essence, this results in the values obtained being affected by introducing concepts such as $\epsilon$-randomness. For instance, if there is stochastic variable\cite{24} $Y$ with a $n + 1$ state space $\Omega$, where $n = |\text{directions}|$ and let there be four directions: $N, E, S$ and $W$; then $\Omega = \{N, E, S, W, G\}$ where $G$ is the “greedy” choice. We let $p_Y(\neg G) = \epsilon \iff p_Y(G) = 1 - \epsilon = q$\cite{25}. Even with low values of $\epsilon$ the output will be quite different from that of the original Q-LEARNING suggested by Watkins\cite{21}, especially since the output is dynamic, viz. Q-values may decrease even in simple deterministic scenarios and not just generally \textit{increase} as with Q-LEARNING\footnote{Q-values can still be dynamic - viz. both increase and decrease - with Q-learning, but are usually less so than what is observed with Sarasa.}

\begin{equation}
Q(s_n, a_n) = Q(s_n, a_n) + \alpha (r_{n+1} + \gamma Q(s_{n+1}, a_{n+1}) - Q(s_n, a_n)) \tag{8}
\end{equation}

In\cite{8} we note that the Q-value for taking action $a_n$ in state $s_n$ is the current Q-value plus a fraction of the reward given in the next state added with a fraction of the Q-value of the next state-action tuple minus the current. This reflects the notion that the action-state tuple taken next reflects the current choice, rather than just having the “greedy” max as in\cite{7}.
4 Graph Algorithms

The task of agent navigation is strongly connected with the field of graph theory as it is beneficial to consider environments as graphs. This is due to possibilities of abstraction that a graph offers, as well as the fact that graphs are generally well understood, from a mathematical perspective. We present several algorithms, some which are based on each other, that offer viable solutions to the scenarios presented.

4.1 Dijkstra's Shortest Path Algorithm

Devised by the famous Dutch computer scientist Edsger Dijkstra in 1956 [26], the Dijkstra's shortest path algorithm is a fundamental building block for later developments in the field of path finding. The concept as such, concerning the outline of the algorithm, is that to find the shortest path between any vertex and a source vertex, it is sufficient to only visit each vertex once and to always prefer shortest paths. Likewise, it is only ever necessary to save the shortest subpaths discovered. That is, the general version of the algorithm generates a tree of shortest paths with the source as the root.

We analyze the complexity of the algorithm by first noting that it can only be applied on a weighted directed graph $G = (V, E)$ where $\forall e \in E: \text{weight}(e) \geq 0$. The reason for this is that if there is at least one edge $e'$ such that $w(e') < 0$ then there might be a cycle $c = \langle e_\alpha, \cdots, e_\alpha \rangle$ where $e' \in c$, resulting in some vertices having no shortest path from $v_{\text{source}}$. That is, $\lim_{n \to \infty} ||v_{s}, \cdots, v_{1}, \cdots, v_{t}|| = -\infty$ with $||c|| < 0$. In such a scenario it is obvious that the algorithm does not apply. The time complexity of the algorithm is defined by the implementation of the min-priority queue utilized - denoted $Q$ in our algorithm. The reason for this is that we perform three priority-queue operations on $Q$ during the algorithm. These are $\text{INSERT}$ on line 9, $\text{EXTRACT-MIN}$ on line 13 and $\text{DECREASE-KEY}$ in lines 25 to 30. Using aggregate analysis we note that we will perform both $\text{INSERT}$ and $\text{EXTRACT-MIN}$ $|V|$ times whereas $\text{DECREASE-KEY}$ will be called at most $|E|$ times. As such we can conclude that the total worst time complexity will be

$$O(|V| \times O(\text{EXTRACT-MIN}) + |E| \times O(\text{DECREASE-KEY})) \quad (9)$$

We may therefore conclude that the final complexity will depend on the worst time complexity of these two priority-queue operations. For instance, if $E = o(V^2/\lg V)$, i.e. $G$ is sparse, we can improve the runtime by implementing the min-priority queue using a binary min-heap, since we get $O(E \lg V)$ rather than $O(V^2)$ (which is the complexity of an ordinary array implementation). It is also possible to obtain $O(V \lg V + E)$ by using a Fibonacci heap. Generally, any implementation will depend greatly on the properties of $G$ and as such we consider 9 to be the best valid, albeit somewhat vague, worst time complexity estimation.

4.2 A*

A* [29] is one of the most popular search algorithms utilized to find the shortest path between two nodes. It is very similar to Dijkstra's, described in 4.1 but
Dijkstra\( (G, v_{\text{target}}, v_{\text{source}})\)

1. // Signature: Graph \( G \), Vertex \( v_{\text{target}} \), Vertex \( v_{\text{source}} \) → Vertex List \( A \)
2. for \( j = 0 \) to \( |G.V| \)
   3. // Variant: \( |G.V| - (j + 1) \)
   4. \( d[G.V[j]] = \infty \)
   5. \( p[G.V[j]] = \emptyset \)
   6. \( d[v_{\text{source}}] = 0 \)
   7. \( Q = G.V \)
   8. \( \text{count} = 0 \) // used to prove that loop ends
9. while \( Q.\text{count}() \neq 0 \)
10. // Variant: \( |G.V| - \text{count} \)
11. \( w = Q.\text{minpop}() \) // Pops \( v \in Q : d[v] = \min \)
12. if \( d[w] == \infty \)
13. return [] // Path does not exist, return empty list
14. if \( w == v_{\text{target}} \)
15. \( A = [] \) // Let \( A \) be an empty list
16. \( q = v_{\text{target}} \)
17. while \( p[q] \neq \emptyset \)
18. // Variant: \( |G.V| - |A| \)
19. \( A.\text{append}(p[q]) \)
20. \( q = p[q] \)
21. return \( A.\text{reverse}() \)
22. for \( i = 0 \) to \( |w.\text{adj}| \)
23. // Variant: \( |w.\text{adj}| - (i + 1) \)
24. \( \text{dist}_{\text{temp}} = d[w] + \text{distance}(w, G.V[i]) \)
25. // Where distance(\( \alpha, \beta \)) is the edge value between \( \alpha \) and \( \beta \).
26. if \( \text{dist}_{\text{temp}} < d[G.V[i]] \)
27. \( d[G.V[i]] = \text{dist}_{\text{temp}} \)
28. \( p[G.V[i]] = w \)
29. \( \text{count} = \text{count} + 1 \)

Algorithm 2: Dijkstra’s Algorithm. This version returns the shortest path between two vertices (i.e. terminates when \( v_t \) has been reached).

It maintains a heuristic cost estimate from the current node being expanded to the goal vertex. Essentially the algorithm traverses the vertices and expands valid vertices, saving the cost of reaching it - just like Dijkstra’s - in an array, so a lookup \( \text{cost}_\text{of}_\text{path}(v_s, v) \) can be performed \( \forall v \in V \in G \). For every vertex the predecessor is also saved so that a path can be reconstructed once the target has been reached. A* requires the heuristic estimate \( h(v_n) \) - denoting the cost from the current vertex \( v_n \) to the goal - to be less or equal to the actual distance; viz. the algorithm is admissible \([26]\). There are several ways this can be implemented, but the most common is the direct vector \( v_n \rightarrow v_g \) or using the manhattan distance method \([26]\).

---

8Given by \( d(a, b) = |a.x - b.x| + |a.y - b.y| \).


```
1 Closed = ∅
2 Open = {v_s}
3 C_f = empty mapset
4 g_score[start] = 0 // Distance from v_s along optimal path
5 h_score[start] = HeuristicEstimate v_s → v_g // From v_s to v_g
6 f_score[start] = G(n) + H(n)
7 while Open ≠ ∅
8 x = min f_score
9 if x == v_g
10 return reconstruct_path(C_f, C_f[v_g])
11 // Reconstruct so we get the shortest path
12 Open.Remove(x)
13 Closed.Add(x)
14 foreach y ∈ neighbour_nodes(x)
15 if y ∈ Closed
16 continue
17 tentative_g_score := g_score[x] + ||x, y||
18 if y ∉ Open
19 Closed.Add(y)
20 tentative_is_better = true
21 else if tentative_g_score < g_score[y]
22 tentative_is_better = true
23 else
24 tentative_is_better = false
25 if tentative_is_better == true
26 C_f[y] = true
27 g_score[y] = tentative_g_score
28 h_score[y] = heuristic_estimate_of_distance(y, goal)
29 f_score[y] = g_score[y] + h_score[y]
30 return failure // there is no existing path from startnode to goal
```

**Algorithm 3: A***

In accordance with Hart et al. [9], we let \( F(v_n) \) denote the selection value of vertex \( v_n \) and given that a lower value is desirable, we can define the function according to

\[
F(v_n) = G(v_n) + H(v_n)
\]

where \( G(v_n) \) is the cost of reaching \( v_n \) from \( v_s \). Letting \( H(v_n) = 0 \) results in not making use of the information available in the problem domain, i.e. we may not have a static predefined goal. However, this results in behaviour that does not guarantee that a minimal number of nodes are expanded. A common method utilized in dynamic scenarios, albeit far from ideal as will be shown later, is repeated application of \( A^* \) during runtime, viz. running the algorithm every time a change has been recorded (like robot movement).
A common situation in applied scenarios is that the agent is working in a world which is partially or fully unknown, viz. we do not know anything about the graph or what we know may change over time. One way to handle such situations is to restart the agent navigation algorithm repeatedly upon movement or allow the algorithm to generate a global path based on existing information available to it upon initialization and then alter said path once changes are discovered during physical traversal. However, these are not good options in the sense that they require extensive calculations and/or are generally not practical in applied situations unless the terrain to be covered is very limited, sc. small area with few obstacles.

D* was introduced by Anthony Stentz in 1993 [13][14] and is an algorithm designed to have the capability to - in an efficient and optimal way - find paths in an unknown and dynamic environment. The name is based on A* and work in a similar fashion with the exception that D* can handle cost changes during a path finding process. As such it is a dynamic version of A*, viz. Dynamic A* and hence the name. The proof of its soundness, optimality and completeness is outside the scope of this essay and is generally a rather difficult subject involving several advanced topics and will thus not be covered.

Let \( G \) be the goal state and for all states \( x \) let \( b(x) = y \) be a backpointer to the previous state \( y \). The arc cost between two states are denoted by \( c(x, y) \) and we say that two states are neighbours iff \( c(x, y) \lor c(y, x) \) are defined. Every state \( x \) has a tag, denoted \( t(x) \), which can be set to \( NEW \) is \( x \) has never been in the open-list, \( CLOSED \) if the state is no longer in the open-list and \( OPEN \) if said state is in the open list. Like A*, D* also makes use of an open-list which is used to keep track of states. D* also introduces an estimated cost of traveling from the current state \( x \) to the goal \( G \) defined by \( h(G, x) \). The previous cost function \( p(G, x) \) is the same as \( h(G, x) \) prior to insertion in the open-list, but once in there the previous cost function can be classified further as one of two types; \( RAISE \) or \( LOWER \) state. \( RAISE \) state occurs when \( p(G, x) < h(G, x) \) and \( LOWER \) when \( p(G, x) \geq h(G, x) \). As such, said classifications denote whether or not the cost is higher or lower than the last time the state was in the open-list. Whilst in the open-list, states are sorted by their key-function value - \( k(x) \) - defined as \( \min(h(G, x), p(G, x)) \) if the state for \( t(x) \) is \( OPEN \). Should \( t(x) \neq OPEN \), the function is undefined. A path is said to be optimal iff it consist of states that are minimal, letting \( K_{\min} = \min(k(x)) \) we can detect an unoptimal path by the fact that it will be greater than \( K_{\min} \).

The algorithm is performed by utilizing two main functions, one that computes the optimal path cost to the goal and one that modifies the arc costs if an inconsistency is discovered during the execution of the first function. Steintz (REF) denote said functions ProcessState and ModifyCost respectively. By iterating ProcessState until \( t(x) = CLOSED \), the state \( x \) that is finally obtained is the state from the open-list where \( \min(k(*)) \) - a key-function independent of its domain,
4.4 LPA*

LPA*, short for Lifelong Planning A∗, is an incremental version of A∗ (see 4.2) applicable on graphs where $E$ has a finite cardinality. It is primarily designed to be utilized on problems with dynamic edges, that is, edges that may be removed or added as well as have their costs altered over time. We present the original algorithm proposed by Koenig, Likhachev and Furcy [7] in 2004 and then discuss the implication of said algorithm as well as the properties it holds. Our primary interest in LPA* lies with the notion that D* lite is based on it (see 4.5).

Let $G = (V, E)$ be a finite graph, then the finite set $S = V$ consists of all the vertices in $G$ and we denote the set of successors of vertex $s \in S$ by $\text{succ}(s) \subseteq S$. Likewise, we denote the set of predecessors of vertex $s \in S$ by $\text{pred}(s) \subseteq S$. Further, let $0 < c(s, s') \leq \infty$ denote the cost of moving from vertex $s$ to $s' \in \text{succ}(s)$. We let $s_{\text{start}}, s_{\text{goal}} \in S$ be the start and goal vertices respectively, and thus the purpose of LPA* is to find $\langle s_{\text{start}}, s_{\text{goal}} \rangle$.

$$g^*(s) = \begin{cases} 0 & \text{if } s = s_{\text{start}} \\ \min_{s' \in \text{pred}(s)}(g^*(s') + c(s', s)) & \text{otherwise} \end{cases} \quad (10)$$

In equation (10) we define $g^*(s)$, which returns the shortest path from $s_{\text{start}}$ to $s$. In [7] Koenig et al. demonstrate the effectiveness of LPA* by running an agent in a binary octagon gridworld, i.e. for every position there are up to eight adjacent positions and a position is either traversable or not, where the estimated distance is

4.4 LPA*

D* tries to find a sequence $a(x)$ that is the actual cost of traversing the cell and the $s(x)$ presumed cost. The algorithm can be described in six steps:

1. $G$ is placed in the open-list with $k(G) = h(G) = 0$. Let $S$ be the state where the agent starts.

2. Repeat PROCESSSTATE until $h(S)$ is $\leq K_{\text{min}}$. When found, we have a path from $S$ to $G$.

3. Follow the backpointers until we reach $G$ or and obstacle, viz. $s(x) \neq a(x)$.

4. If an obstacle is found, then $s(x) = a(x)$ and $c(x, *) \land c(*, x)$ are updated for all the affected neighbours. The alterations are put on the open-list with the MODIFYSTATE function.

5. PROCESSSTATE is then invoked until $K_{\text{min}}$ is equal or exceeds the $h(\ast)$ value of the state that currently contains the agent (a new optimal path needs to be found).

6. Go to step 3

viz. a candidate for a minimal cost path. The backpointers are then followed and error values in the arc costs are then updated by invoking MODIFYCOST to reflect the actual costs. The affected states are put in the open-list.
obtained by max\{\((a.x - b.x), (a.y - b.y)\)\} where \(a, b \in S\). The major fundamental idea behind LPA* is to, unlike A*, not recalculate unnecessary cells, i.e. cells which have not been altered since the previous update. However, it does share a great deal of aspects with A* as well; just like A* LPA* utilizes a nonnegative and consistent heuristic approximation \(- h(s)\) - of the goal distances of the vertices \(s \in S\) on which to focus its search. This obeys the triangle inequality (special case of equation 1), i.e. \(h(s_{goal}) = 0 \land \forall s \in S \forall s' \in succ(s) : h(s) \leq c(s, s') + h(s')\) where \(s \neq s_{goal}\).

Further, LPA* maintains an estimate of the \(g^*(s)\) values - \(g(s)\) - which denotes the estimated start distances for each vertex \(s \in S\) to \(s_{goal}\). In addition to this estimate, LPA* also maintains a second type of estimate of the start distances; denoted \(rhs(s)\). These are a one-step lookahead value based on the \(g\)-value that always satisfy the relationship

\[
rhs(s) = \begin{cases} 
0 & \text{if } s = s_{start} \\
\min_{s' \in pred(s)}(g(s') + c(s', s)) & \text{otherwise}
\end{cases}
\]  

For a definition of \(g(s)\), consider [10] where all occurrences of \(g^*(s)\) have been replaced with \(g(s)\). While A* maintains an open and a closed list containing the vertices that are to be expanded and those that should not be expanded respectively, LPA* only utilizes a priority queue which contains exactly those vertices that are locally inconsistent. These are denoted by keys found in the algorithm, and by study of said algorithm we note that LPA* always expands the vertex with the smallest key. Said key is defined as \(k(s) = [k_1(s) ; k_2(s)]\) for a vertex \(s \in S\), i.e. \(k(s)\) is a vector in \(\mathbb{R}^2\). The actual value of \(k_1\) and \(k_2\) is defined in \textsc{CalculateKey}(s) found in algorithm 4.

Koenig et al. perform several experiments on comparative performance, but due to the difficulty in comparing the operations of LPA* and A* on a fair basis, no conclusive results follow. Since we shall not consider LPA* as a viable algorithm as far as application is concerned, but rather as a theoretical base on which D* Lite is built on, we do not consider this a problem per se. Rather, we consider both of these algorithms - in essence bases on which D* and D* Lite are built upon - to both be viable approaches to the problem at hand; i.e. agent navigation in a gridworld. It should further be noted that Likachev and Koenig [8] have also proposed GLPA* in which the priority queue only contain those vertices \(s \in S\) which are locally inconsistent such that they have not been previously expanded. While they also experimentally show that GLPA* outperforms LPA* on grids, we note that actual ideal peak performance, while interesting, is not the main focus of this thesis and as such we consider the main differences between A* and LPA* to be our main interest, rather than exact performance.

4.5 D* lite

D* Lite - short for Focussed Dynamic A* Lite - is, unlike suggested by its name, not based on the D* algorithm but is rather a dynamic deviation of LPA*.
Algorithm 4: LPA*

\[\text{CalculateKey}(s):\]
1 \(\text{return } \left[\min(g(s), rhs(s)) + h(s); \min(g(s), rhs(s))\right]\)

\[\text{Initialize():}\]
2 \(U = \emptyset\)
3 \(\forall s \in S : rhs(s) = g(s) = \infty\)
4 \(rhs(s_{\text{start}}) = 0\)
5 \(U.\text{Insert}(s_{\text{start}}, [h(s_{\text{start}}); 0])\)

\[\text{UpdateVertex}(u):\]
6 \(\text{if } (u \neq s_{\text{start}}) rhs(u) = \min_{s' \in \text{pred}(u)}(g(s') + c(s', u))\)
7 \(\text{if } (u \in U) \ U.\text{Remove}(u)\)
8 \(\text{if } (g(u) \neq rhs(u)) \ U.\text{Insert}(u, \text{CalculateKey}(u))\)

\[\text{ComputeShortestPath():}\]
9 \(\text{while } (U.\text{TopKey}()) < \text{CalculateKey}(s_{\text{goal}}) \lor rhs(s_{\text{goal}}) \neq g(s_{\text{goal}})\)
10 \(u = U.\text{Pop}()\)
11 \(\text{if } (g(u) > rhs(u))\)
12 \(\quad g(u) = rhs(u)\)
13 \(\quad \forall s \in \text{succ}(u) : \text{UpdateVertex}(s)\)
14 \(\quad \text{else}\)
15 \(\quad g(u) = \infty\)
16 \(\quad \forall s \in \text{succ}(u) \cup \{u\} : \text{UpdateVertex}(s)\)

\[\text{Main():}\]
17 \(\text{Initialize()}\)
18 \(\text{forever}\)
19 \(\text{ComputeShortestPath()}\)
20 \(\quad \text{Wait for changes in edge costs}\)
21 \(\quad \forall \text{ directed edges } (u, v) \text{ with changed costs}\)
22 \(\quad \text{Update the edge cost } c(u, v)\)
23 \(\quad \text{UpdateVertex}(v)\)
present the original unoptimized version of the algorithm as proposed by Koenig and Likhachev in 2002 [16]. Unlike D*, D* lite is rather easy to comprehend due to its many similarities with LPA*. Koenig and Likhachev state this ease of comprehension as a major reason to adopt their proposed algorithm as it allows the user to understand and thus extend their work to better suit his or her needs. This in rather sharp constrast to just considering the algorithm as a black box; which according to Koenig and Likhachev is common practice with D*, albeit its vast popularity ranging from graduate level robot development to Mars Rover prototypes [38]. We particularly note that the many similarities between A* and LPA* (see section 4.2 and 4.4) are not, as shown in this section, a negative aspect but rather ensures us of the soundness of the proposed approach utilized by both algorithms. That is, a heuristic approach. However, we wish to put emphasis on the notion of the incremental properties of LPA* which serves to distinguish the mentioned algorithms. Further, we urge any reader not familiar with LPA* to study section 4.4 prior to reading this section, as several important functions defined there will reappear here.

D* lite is, as previously mentioned, based on LPA* with the main difference being that instead of moving from \( v_s \) to \( v_g \), a path \( \langle v_g, v_s \rangle \) is the target goal, viz. essentially a reversed version of LPA*. This means that the heuristic function \( h(s, s') \geq 0 \) needs to obey \( h(v_a, v_s) = 0 \) and \( h(v_a, s) \leq h(s_a, s') + c(s', s), \forall s \in S \) and \( \forall s' \in Pred(s) \). Note that since the agent moves, this property should apply on all vertices it starts from. Apart from this difference, minor adjustments are needed in the MAIN() procedure of algorithm 5 to reflect the necessity of moving the agent and then recalculating the priorities of the vertices in the priority queue accordingly. The reason for this is that since we are dealing with a dynamic situation, viz. the robot is moving and the terrain is dynamic, the heuristics change; as they are calculated based on the notion that \( v_s \) is the current agent position (which has been altered). Apart from this, the ideas presented in 4.4 apply.
\begin{algorithm}
\alglinenumber{1} \textbf{CalculateKey}(s):
\alglinenumber{2} \textbf{return} \left[\min(g(s), \text{rhs}(s)) + h(s_{\text{start}}, s) + k_m; \min(g(s), \text{rhs}(s))\right]

\alglinenumber{3} \textbf{Initialize}():
\alglinenumber{4} U = \emptyset
\alglinenumber{5} k_m = 0
\alglinenumber{6} \forall s \in S : \text{rhs}(s) = g(s) = \infty
\alglinenumber{7} \text{rhs}(s_{\text{goal}}) = 0
\alglinenumber{8} U.\text{Insert}(s_{\text{goal}}, \text{CalculateKey}(s_{\text{goal}}))

\alglinenumber{9} \textbf{UpdateVertex}(u):
\alglinenumber{10} \textbf{if}(u \neq s_{\text{goal}}): \text{rhs}(u) = \min_{s' \in \text{Succ}(u)}(c(u, s') + g(s'))
\alglinenumber{11} \textbf{if}(u \in U) : U.\text{Remove}(u)
\alglinenumber{12} \textbf{if}(g(u) \neq \text{rhs}(u)) : U.\text{Insert}(u, \text{CalculateKey}(u))

\alglinenumber{13} \textbf{ComputeShortestPath}():
\alglinenumber{14} \textbf{while}(U.\text{TopKey}() < \text{CalculateKey}(s_{\text{start}}) \lor \text{rhs}(s_{\text{start}}) \neq g(s_{\text{start}}))
\alglinenumber{15} \quad k_{\text{old}} = U.\text{TopKey}()
\alglinenumber{16} \quad u = U.\text{Pop}()
\alglinenumber{17} \quad \textbf{if}(k_{\text{old}} < \text{CalculateKey}(u)):
\alglinenumber{18} \quad \quad U.\text{Insert}(u, \text{CalculateKey}(u))
\alglinenumber{19} \quad \textbf{else if}(g(u) > \text{rhs}(u)):
\alglinenumber{20} \quad \quad g(u) = \text{rhs}(u)
\alglinenumber{21} \quad \quad \forall s \in \text{Pred}(u) \cup \{u\} : \text{UpdateVertex}(s)
\alglinenumber{22} \quad \textbf{else}:
\alglinenumber{23} \quad \quad g(u) = \infty
\alglinenumber{24} \quad \forall s \in \text{Pred}(u) \cup \{u\} : \text{UpdateVertex}(u)

\alglinenumber{25} \textbf{Main}():
\alglinenumber{26} s_{\text{last}} = s_{\text{start}}
\alglinenumber{27} \textbf{Initialize}()
\alglinenumber{28} \textbf{ComputeShortestPath}()
\alglinenumber{29} \textbf{while}(s_{\text{start}} \neq s_{\text{goal}}):
\alglinenumber{30} \quad s_{\text{start}} = \arg \min_{s' \in \text{Succ}(s_{\text{start}})}(c(s_{\text{start}}, s') + g(s'))
\alglinenumber{31} \quad \text{Move to } s_{\text{start}}
\alglinenumber{32} \quad \text{Scan graph for changed edge costs}
\alglinenumber{33} \textbf{if} any edge cost changed:
\alglinenumber{34} \quad k_m = k_m + h(s_{\text{last}}, s_{\text{start}})
\alglinenumber{35} \quad s_{\text{last}} = s_{\text{start}}
\alglinenumber{36} \quad \forall \text{ directed edges } (u, v) \text{ with changed edge costs:}
\alglinenumber{37} \quad \quad \text{Update the edge cost } c(u, v)
\alglinenumber{38} \quad \quad \text{UpdateVertex}(u)
\alglinenumber{39} \quad \text{ComputeShortestPath}()
\end{algorithm}

\textbf{Algorithm 5:} D* lite (unoptimized)
Part II
Analysis

In this part we analyse the algorithms discussed from a perspective that reflects the ideas introduced with the scenarios previously defined. In order to gain an insight into their, sc. the algorithms, respective strengths and weaknesses we further define the concept of a long-term strategy model, introduced in section 1.1. Such models will be central to this section as our intention is to further illustrate the difficulties faced when devising a general heuristic approach.

5 Long-term strategy model

When analysing the scenarios presented in section 2 it is imperative to do so from a mathematically sound perspective. That is, one needs some form of factor that enables a fair judging. We have previously noted that each of the algorithms described offer solutions to navigation problems of various nature. As such, it is not scientifically sound to compare them on general terms, i.e. without taking notice of what they offer in a grander perspective. To do so, we introduce the concept of a long-term strategy model - which essentially incorporates the very notion of what goal, and thus also what strategy, the agent should aim for in the long run. What constitutes as the “long run” is somewhat subjective, in the sense that as the scenario itself might be variable, viz. we may consider “long-term” to denote the horizon apparent in the mission description.

We present the variables that are to be defined in an long-term strategy model:

- **Mode.** The mode defines the objective of the agents current mission in the environment. For instance, to move from \( a \) to \( b \) while looking for evidence of life (Mars Rover).

- **Reliability.** We define the reliability of the long-term strategy model to reflect the risk awareness of the agent, that is, the degree of how imperative failure avoidance is. Essentially this tells us whether or not the agent should value redundancy and operation continuality as high as the main objective, or possibly even greater. Reusing our example of the Mars Rover, we note that reliability is very important, as if something goes wrong it results in high monetary cost.

- **Vision.** The initial data available to the agent as well as how new data is obtained. For instance, initial terrain information might come from satellite surveilance data and the agent might have the capacity to see one index in all adjacent directions.

- **Limitations/Restrictions.** Variables that limit the agents performance. In an applied scenario this includes resources such as fuel and physical limitations of the agent itself, viz. engine power and terrain gradient.

\[ \text{That is, the maximum slope gradient the agent can traverse.} \]
5.1 Applied application of $\varphi$

In section 1.1 we defined a curve[17] - i.e. a path - $C_{\iota_1,\iota_2}$ to be optimal iff $|k_1 - k_2| \leq \varphi$, where $k_1$ is the length of $C_{\iota_1,\iota_2}$ given by

$$
\int_{C_{\iota_1,\iota_2}} dt = \int_{\iota_1}^{\iota_2} r(t) dt = \int_{\iota_1}^{\iota_2} x_1(t)e_1 + \cdots + x_n(t)e_n dt
$$

and $k_2$ is the length of the curve given by an oracle (i.e. a perfect path). In an applied scenario such a definition will be rather useless as an oracle will not be available, as such there are several ways one can estimate a perfect path. For instance, it is possible to utilize the vector $v = \iota_1 \iota_2$ and let $k_2 = kv$ for some scalar $k \in \mathbb{R}$. The important aspect of $k_2$ is not that it is necessarily absolutely correct in an applied situation, but rather that it is a good enough estimate to allow measurement of success regarding path quality. Obviously said scalar should depend on the quality of the terrain, sc. traversability, and should be updated as the terrain is explored. Letting $k = 1$ during the initialization and then updating it as terrain is discovered, according to some set of rules, would then result in a convergence of the scalar to a reasonable value. How this would be implemented more precisely requires additional research and experimentation. Note the similarity to the heuristic functions found in some of the graph algorithms studied.

6 Scenario Outcome

6.1 Scenario I

This scenario is very fundamental as it involves the basic notion of path finding. In the scenario description we note that we shall consider shorter norm better (noting that we can always find an exact actual norm through vector augmentation) which contradicts the ideas present in non-deterministic agents; however, we disregard this for now. Essentially this scenario will be solved equally well by all algorithms that do not incorporate reinforcement learning, e.g. $A^*$, $D^*$, $LPA^*$ and $D^* \text{ lite}$, as they are all based on Dijkstra's. Take special note on the fact the scenario describes a static environment. However, should we consider the scenario such that the environment is unknown, there will be some subtle - yet interesting differences in performance.

First we note that $A^*$ needs to recalculate more indices than $LPA^*$ (and thus also $D^* \text{ lite}$) \footnote{It might be necessary to divide $[\iota_1, \iota_2]$ into $n$ parts. These would then be integrated independently and then summed together to return the length of the curve.} when run in an online situation (repeated application of $A^*$...
is a “lower” class of incremental search). Koenig and Likhachev [11] show that D* LITE outperforms D* in mapping unknown terrain and that said difference in performance seems to increase with an increase in sensory input range. Since these two algorithms are based on LPA* and A* respectively, it seems probable that the performance difference is connected with the incremental properties of LPA*. At the same time, the very same authors state the difficulties faced when trying to prove that LPA* outperforms A*, so we are unable to draw any final conclusions regarding these two algorithms [7] and we will simply state that they are at the very least equal in performance.

With Q-LEARNING and SARSA, however, the situation is somewhat different. A tool - GridWorld - written and utilized by the Department of Information Technology of Uppsala University was used to simulate the reinforcement agents. The tool supports an agent in a quadratic gridworld with binary grid values running some reinforcement learning algorithm. We ran tests with both Q-LEARNING and SARSA, using various parameter values for learning rate and $\epsilon$ randomness.

Figure 4: Q-learning versus SARSA on a $5 \times 7$ grid with $\eta = 0.9, \epsilon = 0.1$

The first apparent difference compared to the other algorithms analysed in this thesis, is that the reinforcement learning algorithms require a lot of movement. Recall (see section 3) that learning is achieved by initial trial and error; a trait that is obviously increased when utilizing a large $\epsilon$ randomness value. As such the agent will move about a lot, which is a problem in any applied scenario - especially with long-term strategic models such that there is limitation regarding movement; e.g. fuel. The application supports worlds up to $20 \times 20$, which is very limited when we consider any real world application, and when running the agent on the largest possible environment the number of steps is huge (4.85 million moves to reach the goal 50000 times). We also found that Q-LEARNING managed to converge to shortest paths where as SARSA generally performed poorly with the same number of steps, viz. more steps were required to obtain the same number of episodes. We present some findings in Figure 5.

Looking at the results in Figure 5 it follows that the proposed algorithms are far from feasible in direct application within an applied scenario as they require too
many moves to be effective. Obviously the convergent paths they find will be very good, but the initial time spent finding them is just too long. A possible solution to this problem is to utilize another algorithm to find a working path and then, once some initial data is available, let the reinforcement learning agent take over. That way, the number of steps taken will be greatly reduced where as the output in the long run will be very close to ideal.

6.2 Scenario II

In the second scenario we wish to record all points $\phi \in \mathcal{W}$ that we encounter, where $\mathcal{W}$ is the set of all interesting points, as well as maintain paths to said points from the origin. Essentially this scenario does not differ much from Scenario I, apart from the notion of maintaining a set of paths; rather than just one. Implementing this using any of the heuristic graph algorithms, i.e. not the reinforcement learning algorithms, is very straightforward; we simply perform one search for each goal during each iterative step. After encountering an interesting point, we add it to our goal list.

For the reinforcement learning agents, the situation is more complicated. Since an agent is only ever motivated to learn to find a path to a goal through rewards, we must add reward values for finding an interesting point and then reward the agent every time it encounters it again. This is problematic as we must decide on the relationship between the reward for the various points $\phi \in \mathcal{W}$ and the global goal. Actual steps required will not differ from an ordinary implementation, given that we do not implement greedy choices only as path convergence tend to occur given a sufficient number of episodes. A problem discussed by Sprague and Ballard [12] is that while utilizing techniques for multiple goal implementations - e.g. modular Sarsa - tend to work rather well - i.e. the output converges in experiments - there is no proof that absolute convergence is guaranteed.

In conclusion, we note that the complexity of the algorithms will not increase even after proposed modifications as the scenario only requires increased iteration. Since $O(n \times f) = n \times O(f) = O(f)$, the complexity remains the same. However, it is doubtful if using reinforcement learning agents is a good idea, given the lack of proof of guaranteed convergence.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Episodes</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sarsa</td>
<td>$5 \times 7$</td>
<td>500</td>
<td>18544</td>
</tr>
<tr>
<td>Sarsa (/w guidance)</td>
<td>$5 \times 7$</td>
<td>500</td>
<td>8696</td>
</tr>
<tr>
<td>Q-Learning</td>
<td>$5 \times 7$</td>
<td>500</td>
<td>15349</td>
</tr>
<tr>
<td>Q-Learning (/w guidance)</td>
<td>$5 \times 7$</td>
<td>500</td>
<td>7884</td>
</tr>
<tr>
<td>Sarsa</td>
<td>$20 \times 20$</td>
<td>500</td>
<td>117920</td>
</tr>
<tr>
<td>Q-Learning</td>
<td>$20 \times 20$</td>
<td>500</td>
<td>48533</td>
</tr>
</tbody>
</table>

Figure 5: GridWorld experiment results. Guidance signifies an initial expert guidance for 5 episodes.
6.3 Scenario III

This scenario is, as described in section 2.3, a variant of the traveling salesman problem and as such NP-hard. Even if we find good paths between each node, the main problem remains, viz. solving the traveling salesman problem. The scenario is therefore very interesting as it illustrates a large flaw with the graph algorithms studied; namely, that there are situations where traditional algorithmic approaches simply do not apply. We note that said algorithms - sc. the graph algorithms studied - can only be utilized in an efficient manner to obtain paths between each point in the set we are to visit.

The reinforcement learning agents will face a similar problem. While we could easily add variable rewards that depend on the distance the agent has traveled, the calculations needed would still be non-polynomial as this is equivalent to a brute force approach (trial and error, essentially). So in essence, the reinforcement learning approach fails short as well. This is to be expected as the problem at hand is NP-hard. However, a general strategy would need to be able to cope with situations such as the one presented in this scenario. So the question at hand is how this can be achieved without resorting to a brute force approach that would most certainly result in the agent performing calculations that would require millions of years to complete. The answer lies with the distinction of finding a solution, a “good enough” solution and an optimal solution. We do not want to just find any solution, it might be the worst possible out of several thousand possible candidates, and finding the optimal requires too many calculations to be practical. Hence the solution is to find a cycle that is “good enough”.

The weakness of the graph algorithms was that they were unable to provide any solution to how we go from a set of vertex interconnecting paths to a “good enough” cycle, where as the reinforcement learning algorithms fail to provide a fast strategy for finding said paths. As such we utilize the graph algorithms to find the paths and then apply a probabilistic metaheuristic algorithm such as ACO (ant colony optimization, see [10]) to obtain a solution that is close-to, but not optimal. The conclusion for the scenario is therefore that all proposed algorithms fall short due to different reasons but the result being that the problem is too difficult to solve in an orderly fashion while maintaining a polynomial complexity. A general heuristic strategy requires probabilistic algorithm capabilities to overcome difficulties in NP-hard problems. This does, as will be discussed later, introduce other problems.

6.4 Scenario IV

The final scenario introduces the concept of invoking an operation - \( f_A(x) \)- defined by the matrix \( A \) on the data abstraction of the environment; \( x \). This is defined to result in a graph that still remains partially solveable, viz. it is still possible to fulfill at least one goal. Such a scenario is very feasible in any applied situation as terrain tends to be non-static. The problem introduced is that of dynamic edges, viz. edge weights may be altered and edges may be removed or added during traversal. We note that both D* and D* LITE were designed with this in mind,
and it is not surprising that they perform the best.

We generated a $20 \times 20$, $50 \times 50$ and $80 \times 80$ grid with two possible paths from $v_s$ to $v_g$ (see appendix C for an example of how this can be done) and let the agent go from $v_s$ to $v_g$, then we blocked both of the paths and generated a new one without clearing any agent data. The agent was then asked to move from its current position - $v_g$ - to $v_s$. We recorded the number of vertices it had to expand, viz. the number of steps required, to return. This was repeated five times for all algorithms. We then used the average result for each map. Initially, we intended to repeat this experiment within GridWorld to obtain data regarding the reinforcement learning agents. However, given that one would then have to train said agents prior to the test, the result would be biased. That is, if the new paths were to consist of several vertices that were also part of the original paths, then the agent would have an advantage whereas if the new paths were to consist mainly of new vertices, viz. indices previously occupied by walls, then the reinforcement agent would have to resort to trial and error. As such, we decided not to include reinforcement learning agents in this experiment. We simply note that the behaviour of a reinforcement learning agent is somewhat more environment dependent, viz. the actual scenario is not as important\(^{12}\) compared to the graph traversal algorithms studied.

Another major difficulty experienced was that the availability of working code for D* turned out to be non-existant. Due to uncertainty of how to fairly implement some of the sets proposed by Stentz in [13][14], we decided to utilize data made available in [15] by Koenig and Likhachev. There is a large uncertainty in this data due to test conditions not being fully know, viz. there is no evidence to suggest that the results in [15] can be directly transferable to the conditions proposed by us. The data is thus not presented as a result, but rather as an estimated value based on the lower bound of the results found by Koenig and Likhachev. We denote said data with an asterix to further distinguish it from the other results. D* will be considered to perform equally well as D* lite in applied situations due to the uncertainty. We note, however, that there are results showing that D* lite outperforms D* in certain situations [11][15].

We used the C code for D* lite made available by Koenig and Likhachev\(^{13}\) but any implementation of the pseudocode in [5] should suffice. The number of vertex expansions and heap accesses for the algorithms were recorded\(^{14}\). For A* we used code made available by Justin Heyes-Jones\(^{15}\). The LPA* code was obtained through Sven Koenigs research webpage\(^{16}\). In the table below we have also in-

\(^{12}\)Obviously the scenario still has an impact, just not as much.
\(^{13}\)[http://idm-lab.org/code/dstarlite.tar](http://idm-lab.org/code/dstarlite.tar) (accessed the 21th of april 2011 18:16)
\(^{14}\)For heap accesses we utilized Valgrind, see [http://valgrind.org](http://valgrind.org)
\(^{16}\)Code: [http://idm-lab.org/code/lpastar2.tar](http://idm-lab.org/code/lpastar2.tar) - slightly modified, viz. we did not use
cluded the range of the data, as for some runs we had extreme results, viz. very small or very large numerical results in comparison with the average.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Tuple (vertex expansions, heap accesses)</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>D* lite</td>
<td>(11802, 16149)</td>
<td>(2170, 1047)</td>
</tr>
<tr>
<td>D*</td>
<td>(17703, 28260*)</td>
<td>(N/A)</td>
</tr>
<tr>
<td>A*</td>
<td>(26505, 32181)</td>
<td>(4281, 3056)</td>
</tr>
<tr>
<td>LPA*</td>
<td>(22415, 33410)</td>
<td>(5101, 9518)</td>
</tr>
</tbody>
</table>

Figure 6: Scenario IV Experiment Results. Data denoted by * is a rough estimate (not actual results) and the error margin is very likely quite large.

Our results support the claims of Koenig and Likhachev, viz. that LPA* generally outperforms A* [7]. However we also noted fewer heap accesses by A*. Seeing as how we did not use any optimized versions of the algorithms we must therefore conclude it as rather likely that these results do not necessarily reflect reality in an applied situation where optimized versions would be utilized. With this in mind, we can only reinforce the notion of the performance gain in the dynamic situation demonstrated by D* and D* lite, viz. dynamic algorithms are better at dynamic situations. Note also the very large variance with LPA* regarding heap accesses, the result of a particular run where the number of accesses was very small. While we feel that the results can not be viewed as an exact measurement of performance, we have illustrated the apparent utility of having dynamic graph traversal algorithms in the algorithmic toolbox, i.e. there is certainly a reason to include such algorithmic tools in a general approach.

7 Reducing chance of failure

As stated in section 2, we introduce \( \varphi \subseteq \mathbb{Z}^+ \) as the set of possible non-zero values present in any index, allowing additional terrain information in addition to traversability. Let \( \varphi_\xi = \{0\} \cup \varphi \). Consider an applied instance of any of the scenarios previously defined and let each index of the abstract data type utilized by the agent hold a reference to some equivalence risk class. By definition, we let an untraversable index have the highest risk, whereas a traversable and “safe” index would have the lowest risk value. We can thus define the chance of failure - i.e. the risk - of a given index as \( \frac{e_{\Delta ij}}{\max_{\xi}} \), recalling that \( \max_{\xi} \geq 1 \). For instance, if \( e_{\Delta ij} = 1 \) and the highest risk class utilized is 5, we get \( \frac{1}{5} = 0.2 \); i.e. there is a 20% chance of failure associated with trying to traverse \( e_{\Delta ij} \). We can therefore calculate the success rating of a given path \( v = (v_1, \ldots, v_n) \) by

\[
P_{\text{success}}(v) = 1 - \left(1 - \frac{v_1}{\max_{\xi}}\right)\left(1 - \frac{v_2}{\max_{\xi}}\right)\cdots\left(1 - \frac{v_n}{\max_{\xi}}\right)
\]  

(12)

We can easily find a path of minimum risk, viz. a path which has the highest probability of success, by letting the edge weights reflect their respective risk rating. For instance, one can apply a slightly modified version of Dijkstra’s randomized maps.
vertex values are multiplied in accordance with equation \[12\]. We would then prefer small values, as the total success is given by \(1 - \kappa\) where \(\kappa\) is the vertex value of \(v_t\) obtained from the source, sc. \(v_s\). Obviously the other algorithms discussed could be utilized in a similar fashion through slight modification as they are based on ideas central to Dijkstra.

Apart from finding a minimum risk path, we can also find a s-t cut\(^{17}\) which defines a - as there might be several such cuts of the same value - most risky section of our explored terrain. Such information is very important when considering a general strategy as it, not unlike a long-term goal for a reinforcement learning agent, is imperative to make non-greedy choices to maximize the output on a larger horizon. For instance, if there are unexplored indices in the vicinity of such a minimum\(^{18}\) s-t cut, then it may prove ideal to explore them in order to increase the maximum flow of the graph. Note that essentially all that has to be done to convert our initial graph \(G = (V, E)\) to a flow network \(G' = (V, E')\) is to let \(E' = \{(v_\alpha, v_\beta) = \gamma \mid (v_\alpha, v_\beta) \in E \land \gamma = \max_{\varphi} + 1 - |(v_\alpha, v_\beta)|\}\), viz. invert the values so that a high risk edge translates into a low flow. We add one to this flow value to ensure that there is a non-zero maximum flow in the network, sc. when \(e \in E\) has \(|e| = \max_{\varphi}\).

A problem with this approach is the lack of the notion of distance. Several “close”\(^{19}\) indices with low risk values might pose a risk on a meta level. If one of them were to be altered in an applied situation, for instance due to a flood, then the probability that the other low risk indices are affected increases as their distance from the affected index decreases. It is therefore also of interest to find, and possibly eliminate, narrow - in a geometric\(^{20}\) sense - portions of explored terrain. Maintaining redundancy as far as possible paths are concerned should be a high priority with a long-term horizon, as a dynamic environment is to be expected. Utilizing Ford-Fulkersons\(^{21}\) maximum flow algorithm with all edges having capacity one allows us to find the total number of paths between any subset of the environment graph.

### 7.1 A Fuzzy View

When considering an actual applied scenario it is somewhat obvious that the level of abstraction found in the models and scenarios discussed within this thesis may not typically reflect reality, viz. classification based on sensory input may be difficult to perform. Say we have an agent \(A_{\text{robot}}\) in an environment \(\mathcal{E}_T\), with \(T\) dynamic - viz. unknown, with a defined long-term strategy model \(M\). Let \(M.vision\) be such that we have sensory inputs \(\mathcal{S} = \{s_1, s_2, s_3\}\)\(^{21}\) and say the agent is able to “see”

\(^{17}\)An s-t-cut \(C = (A, B)\) of a graph \(G = (V, E)\) is a division such that \(v_s \in A\) and \(v_g \in B\) and \(A \cup B = V\) and \(A \cap B = \emptyset\).

\(^{18}\)Viz. it involves “breaking” as few edges as possible.

\(^{19}\)This is subjective, but should reflect the scenario, viz. the “closeness” should be a function dependent on the terrain size.

\(^{20}\)Could be any p-norm, however regard it as the “normal” Euclidian 2-norm.

\(^{21}\)The actual type of sensor is irrelevant to the argument.
only adjacent indices. As such, for every adjacent index \( i \), we get an input \( S_i \), which we then need to classify as belonging to a particular risk equivalence class. We can imagine this is done by a function \( f(s_1, s_2, s_3) = x \), where \( x \) is said risk class. However, in any applied situation there will be elements of uncertainty, for instance due to sensor limitations or other external effects. We must therefore ask ourselves if a static membership classification is really a sufficient approach to the problem.

A fuzzy set \( A \) over some space of points \( X \), is characterized by a membership function \( f_A(x) \in X \) with \( f_A(x) \rightarrow [0, 1] \). As such, the value \( 0 \leq f_A(x) \in \mathbb{R} \leq 1 \) denotes the “grade” of membership of \( x \) in \( A \), viz. the closer \( f_A(x) \) is to 1, the higher the grade of membership in \( A \). Should we consider an ordinary set \( S \) defined as a fuzzy set, there are only two possible values of \( f_S(x) \), sc. \( f_S(x) = 1 \) when \( x \in S \) and \( f_S(x) = 0 \) when \( x \notin S \). Equivalently, a fuzzy set may be fully binary, viz. only having the end-points of \([0, 1]\) as possible values for \( f_A(x) \). Such a set is typically referred to as a simple set\(^\text{22}\).

Zadeh \[30\] points out that while the membership function of a fuzzy set resembles a probability function when \( |X| \leq \aleph_0 \), viz. countable, or a probability density function when \( X \) is continuous, there are several differences. For instance, a state space \( \Omega \) for a given (discrete) stochastic variable is fully known, which may not be the case for \( A \), where we only know that it is a bounded region; analogous to a “continuous” state space\[24\]. The distinction being that with a fuzzy set membership, there is always a graduality; which is very rarely the case for the state space of a continuous stochastic variable. Further, as acknowledged by Engelbrecht \[2\], a probability is only valid prior to the outcome of an event, viz. once the event has occurred there is no longer any fragment of uncertainty to it; it is known. The same does not apply to a fuzzy membership, it will retain a level of membership. Engelbrecht also states that while a probability will, given that it is independent, adhere to the law of large numbers \[24\][25], a fuzzy membership will always be static. That is, if \( f_A(x_k) = 0.85 \), then that is always the case. However, as will be noted later, other researchers suggest a fuzzyness that applies to the degree of membership as well. Even when introducing such ideas, there is a distinction between probability and fuzzyness, as the other distinctions still apply.

The empty fuzzy set is defined according to \( \forall x \in X : f_A(x) = 0 \), which makes sense seeing as how no element in the domain has any degree of membership, viz. there can be no elements in \( A \). We say that two fuzzy sets \( A_0 \) and \( A_1 \) are equal iff \( \forall x \in X : f_{A_0}(x) = f_{A_1}(x) \), and denote the equality \( A_0 = A_1 \) or \( f_{A_0} = f_{A_1} \). Further, the complement of a fuzzy set \( A \), denoted \( \bar{A} \), is defined as \( f_{\bar{A}} = 1 - f_{A} \).

The union of two fuzzy sets \( A_0 \) and \( A_1 \) is denoted \( A_0 \cup A_1 \) and is itself also a fuzzy set. The membership function of \( B = A_0 \cup A_1 \) is given by \( \forall x \in X : f_B(x) = \max(f_{A_0}(x), f_{A_1}(x)) \) and is often denoted \( f_B = f_{A_0} \lor f_{A_1} \). The binary union relation

\[ \text{It is also commonly known as a crisp set.} \]
7.1 A Fuzzy View

is associative, viz.

\[(A \cup (B \cup C) = (A \cup B) \cup C) \iff (\max(f_A, \max(f_B, f_C)) = \max(\max(f_A, f_B), f_C))\]

A similar situation is true for the intersection as well, the difference being that the membership function utilizes min rather than max, i.e. let \(B = \overline{A_0} \cap A_1\), then \(f_B = \min(f_{A_0}, f_{A_1})\). We denote the intersection, reusing the previous definition of \(B\), as \(f_B = f_{A_0} \land f_{A_1}\). The intersection of two disjunct fuzzy sets is empty. Analogous to the union operator, the intersection is associative.

In [30], Zadeh notes that it is incorrect to treat fuzzy membership as a binary relation. Instead one can define two values \(\alpha, \beta \in (0, 1)\) such that \(0 < \beta < \alpha < 1\). A three-valued truth value can then be constructed for the membership of \(x\) in \(A\) according to the table below.

<table>
<thead>
<tr>
<th>Value</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>(f_A(x) &gt; \alpha)</td>
</tr>
<tr>
<td>F</td>
<td>(f_A(x) &lt; \beta)</td>
</tr>
<tr>
<td>U</td>
<td>(\beta &lt; f_A(x) &lt; \alpha)</td>
</tr>
</tbody>
</table>

That is, we let \(\alpha\) denote the lower limit boundary for membership and \(\beta\) denote the upper limit for non-membership. The truth-value \(U\) denotes the outcome that \(f_A(x)\) is uncertain as far as classification goes, viz. one cannot explicitly state any membership threshold given that the predefined boundaries fail to contain it. This goes against the principle of \(P \lor \neg P\) found in classical logic, as there is a third alternative, sc. uncertain - \(U\). Thus, one cannot apply the law of excluded middle [27], common to approaches that follow classic logics. This might, of course, not be negative when we consider application of fuzzy sets, as fuzzy logics have application when the situation that is to be modeled - be it applied or otherwise - is somewhat uncertain or there is a gradient of granularity involved.

For instance, consider a scenario where we wish to make a machine “understand” sentences such as ‘very’, ‘kind of’ and ‘fairly’. One approach, utilizing fuzzy sets, would be to introduce boundary values for each of these words, e.g. a binary function \(g(x, \text{‘fairly’}, Y) = f_Y(x) > c\), with \(c\) being a predefined (possibly fuzzy) constant and \(g\) having range \(\{0, 1\}\). Engelbrecht[2] states that one of the more important applications, and thus also concepts, of fuzzy logics is the ability to work with words, rather than numbers, viz. linguistic variables. Essentially the notion of working with subjective words rather than numerical values is only possible if we introduce rules for the mathematical machine - sc. computer - to “translate” (map) them to values in a meaningful and dynamic fashion. Since this is the very strength of fuzzy sets, the application is apparent. We can therefore consider words such as ‘tall’ and ‘fast’ to actually have meaning when prepended to an object in the universe that is the current domain.

\[23\text{This is the law of excluded middle.}\]
One should, however, pay attention to the uncertainty that is built into fuzzy logics, viz. \( P \lor \neg P \lor U \), and note that there may be objects which cannot be evaluated as 'fast' or 'tall' or whatever descriptive and subjective property we wish to bestow upon them. The problematic nature of this reality is easily illustrated through a fuzzy rule, say "if the person is tall, then he is good at basketball". In classical logic [27] we could translate this to the general form \( P \rightarrow Q \) and easily evaluate it. However, given that both or either of \( P \) and \( Q \) can take on uncertain values, we need to redefine the outcome as well. For instance, is it valid - given the context - to evaluate \( F \rightarrow U \) as true? We can defend such a proposition, under the circumstance that a false premise allows us to prove anything; but would such a claim apply when \( P \lor \neg P \) is not valid? As such, we note that while fuzzy logics certainly has various applications, it also introduces a lot of new - albeit quite interesting - problems.

Using the definitions of complement, union and intersection several identities that hold for regular sets can be extended to also include fuzzy sets. Zadeh lists some of the more interesting:

- De Morgan’s laws. \( \overline{A \cup B} = \overline{A} \cap \overline{B} \) and \( (A \cap B) = \overline{A} \cup \overline{B} \)

- Distributive laws. \( C \cap (A \cup B) = (C \cap A) \cup (C \cap B) \) and \( C \cup (A \cap B) = (C \cup A) \cap (C \cup B) \).

The ability to define these laws, viz. the notion that they apply to fuzzy sets, is not only good in applied situations, but it also assures us of the soundness of the theory itself. That is, it seems like rigorous systems such as ZFC [31] for simple (crisp) sets may be constructed to work with fuzzy sets. Gottwald [32][33] examines ideas presented by Weidner [34][35] and Prati [36] on membership notation in his struggle to introduce such a rigorous axiomatic system. Weidner introduces an alternative version of the simple membership predicate

\[ \in (x, y, w) \iff [x \in y] = w \]

meaning that the degree of membership of \( x \) in \( y \) is \( w \). Prati extends on this notion by introducing a quaternary membership predicate to denote fuzzy membership. By writing \( \in (X, Y, Z, W) \), we mean that \( X \) belongs to \( Y \) to a degree of \( Z \), with regards to the valuation class \( W \). A valuation class is essentially a dynamic valuation structure that allows the fuzzy object, as defined by the fuzzy set, to incorporate its own boundaries for membership degrees, viz. object-defined values for \( \alpha \) and \( \beta \) as previously discussed in the truth-value outcomes. Obviously one may define additional degrees in addition to just the two. Prati also discusses the possibility of allowing said valuation structure to be fuzzy, i.e. the membership degrees themselves are not static, which is an interesting aspect, as it introduces additional fuzzyness to the classification domain. One should also note that such fuzzyness

\[ ^{24}\text{Notation: } A^C = \overline{A} \text{ is the complement of } A. \]

\[ ^{25}\text{Regard these as evaluation brackets.} \]
A general strategy is non-deterministic

Apart from Q-learning and SARSA we have mainly considered deterministic transition model algorithms for static searches. However, it is obvious that this will not suffice when dealing with a general heuristic strategy, as long-term strategic models incorporating modes which require extensive agent movement will benefit from optimizations in the long run. Such optimizations will require the introduction of stochastic processes in the navigation strategy, viz. making non-greedy choices using $\epsilon$ randomness or similar. As a general strategy will also incorporate some gradient of risk limitation in conjunction with the notion of $\varphi_c$ equivalence classes (see section 7), a non-deterministic agent - due to the stochastic processes - will demonstrate a behaviour that might seem non-ideal. That is, avoiding areas of high risk may promote taking non-ideal paths, i.e. making non-greedy choices and eventually having the greedy path converge to something that is not necessarily the shortest path. This is especially apparent when the long-term strategy model has a limitation of the agent movement, e.g. fuel. As such there is a resulting tradeoff that needs to be balanced. However, it might be extremely difficult to find an optima and as such we note that non-determinism introduces the possibility of variable tuning hardness, viz. the cost of finding a perfect agent setting might exceed that of the apparent gain of utilizing it over a suboptimal one.

In essence a good strategy does not only constitute good results - sc. optimal or suboptimal but “good enough” - it should also have a modest complexity along with heuristic and, if possible, incremental capabilities. In a comparative sense, looking at complex living organisms, synthetic agents have superior memory capabilities and as such are much more capable in applying incremental strategies. While this enables us to make very solid heuristic algorithms, there is a hardware limitation; viz. finite state machines versus biological neural networks. This can be illustrated further by considering facial recognition, a task that a human
can rather easily solve in a timespan that - due to the slow signal traversal between neurons - cannot exceed 100 layers of nodes \[26\] but which machines have large problems solving. Recall that electrical circuits, utilized in such machines, allow communication speeds that exceed that of their biological counterparts by a very large factor \[29\]. In essence the central problem in the field is how one is to utilize the strategies apparent in nature most efficiently. That we cannot translate these directly is a result of the - as previously mentioned - difference in hardware. Along with the very notion that it is quite possible that there are more efficient strategies than those developed by natural selection, this results in various difficulties when trying to implement a general heuristic strategy that while possibly being suboptimal, is “good enough”.

In accordance with the non-determinism of introducing stochastic processes - e.g. \(\epsilon\) randomness - in the general strategy, we are dealing with a situation that possesses a markov property. As such we may represent the entire gridworld traversal as a markov chain\[43\], a view that comes naturally as it essentially describes a non-deterministic finite automata. This view is not only beneficial from a mathematical perspective, but rather it enables us to see the general strategy as somewhat more chaotic than what one might otherwise expect. In essence, it follows that due to the agent’s online interaction with the environment, there is no best static strategy as far as exact directions go. Rather, in accordance with introducing a stochastic process, there is a long-term benefit of allowing non-greedy actions. To illustrate this, consider an agent that discovers a wall that it needs to pass, either by moving left or by moving to the right. If no further terrain information is available, i.e. the long-term strategy model has a very limited vision variable, both choices are as good. Should we always invoke a left policy, viz. always moving left when both right and left are as viable, then one might argue that this is in fact a decent algorithmic approach. This is because the probability that one makes a bad decision will always be 33% if we consider the probability of the left direction being longer to be independent. However, this does not necessarily apply when we consider a long-term strategy model that has a mode which requires the agent to perform extensive movement in the terrain for a longer period of time. In such a scenario the cost of making random choices and possibly “stumbling” across better paths, may very well be less than that of always making a greedy choice. For instance by taking an occasional right turn, even when a left path has been discovered. The agent may very well fail to find a path to the goal by doing this, but there is also a probability that a shorter path is found. Again, this is a tradeoff situation and it greatly depends on the long-term strategy model utilized.

9 Conclusion

We have shown the difficulty in designing a general heuristic strategy by defining four scenarios, each illustrating common navigation problems likely to occur in applied situations, and then applying covered algorithms on them. Scenario I could be solved efficiently utilizing graph algorithms while reinforcement learning

\[26\] Commonly known as the 100-step rule.
required extensive trial-and-error. This result illustrated a weakness of the reinforcement learning approach, viz. while the agent certainly did find optimal paths it did so in a manner that is not efficient nor ideal in an applied situation. By examining scenario II we showed that the additional condition $|g| > 1$ added no additional complexity to the agent, viz. the number of goals is not critical to the performance per se; but rather the fashion of how these are to be traversed. We introduced NP-hardness in scenario III which revealed a flaw in a non-probabilistic approach as the agent must function even when dealing with a difficult task. A probabilistic approach, as discussed, can yield a beneficial cost-gain ratio concerning computation and suboptimal performance, sc. near-optimal or “good enough”. However, by allowing probabilistic algorithmic approaches, we also introduce nondeterminism which may - potentially - result in suboptimal behaviour; especially when a more complex abstract gridworld is utilized. However, we argued that such suboptimal results are essential when discussing general approaches, considering the “no free lunch” theorem [12]. In essence, it follows that by favoring generalization we must also accept poorer performance in specific situations and tasks.

Further, in scenario IV we introduced dynamic situations - which reflect actual application - by invoking environment alteration during traversal. Experimentation revealed the benefits of dynamic algorithms such as D* lite and D* as well as the reinforcement learning algorithms. We continued by discussing the effects of a long-term strategy model, which incorporates the additional parameters of any navigational task, e.g. limitations and degree of target performance. By combining a scenario, with a sample environment, along with a specific long-term strategy model we were able to describe very specific and complex situations that should reflect most - if not all - applications. However, the added complexity resulted in several meta-level difficulties, some of which were discussed. The added dimension of more complex environment abstractions allowed us to consider risk limitation, optimality estimation and redundancy optimization. We noted that essentially all of these meta-level problems introduced the problem of tuning hardness, viz. the possibility that the computational costs of finding ideal parameters of the agent could exceed the benefits of utilizing them over an estimated set of parameters. As such, in accordance with the non-determinism introduced with probabilistic possibilites, we noted that generalization essentially requires suboptimal performance at certain situations, assuming $N \neq NP$. This is certainly not something ground-breaking, considering the approach favoured by natural selection in nature; a good example is the navigational performance of a bumblebee trying to fly through a window when trapped indoors. While said bumblebee certainly will utilize somewhat of a greedy bruteforce approach at first, there will be a stochastic process involved; seeing as how the insect will make a non-greedy choice eventually and - possibly - get out.

Our findings as such, we argue, point towards the benefits of non-determinism and stochastic subprocesses involved in the agent navigation. However, as noted in other research (for instance [37]), there is no apparent silver bullet that can be utilized when designing the strategy itself. As such one can consider the apparent benefits of reinforcement learning algorithms - especially the lack of a performance
CONCLUSION

limit - and speculate in how such approaches can be incorporated as a system of voter agents or possibly a central decision making agent that could then be utilized in carrying out the strategy. Having illustrated the apparent difficulty in designing such a strategy in this thesis, we cannot help but wonder how such a decision making intelligence could - and should - optimally be outlined, viz. what aspects are to be considered and how can one rate them according to importance (which, obviously is a dynamic function). More research is required in this field, as we are essentially discussing the main focus of the artificial intelligence field; the construction of actual intelligence of some sort that possesses properties similar to what can be viewed as self-awareness to cope with situational complexity.

Apart from this interesting aspect, one must also consider more basic difficulties that appear when considering environments in $\mathbb{R}^3$. One such interesting problem is how one can utilize topological data obtained a priori, as it can rather easily be converted into a vector field $\bar{F}$ (see [17]), which can then be examined through analytical means. For instance, say $\bar{F}$ is conservative, then we may not have to consider the gradient of the terrain as much as we might otherwise have to. Obviously, there are additional topics that are of interest and as there are readily available analytical tools for flux integration and line integrals, the problem of moving from $\mathbb{R}^2$ to $\mathbb{R}^3$ may primarily be how to utilize said tools, rather than how to solve the problems per se. Reducing the problem space, i.e. applying restrictions apparent when considering the result of an analytical analysis in a “fuzzy” manner, viz. allowing an error-margin, will always be beneficial and while the apparent gain will be greater in $\mathbb{R}^3$, it is still possible to apply some of these techniques in $\mathbb{R}^2$. In any case, three-dimensional space should ideally be basis for the strategy utilized, something that is not true for the scenarios and problems dicussed within this thesis.

With the discussed limitations that follow the generalization in mind, we may ask whether a general strategy is ever desirable in any applied situation. There is no definite answer given the wide range of possible applications. However, one can argue for the neccesity of generalization when considering a broader range of utilization, viz. should agents become plentiful in everyday life. That is, consider the introduction of more versatile robots with less restricted work domains, we ask whether it will be reasonable to spend time on specific navigational strategies for each and every model or if a broader - possibly not universal - strategy could apply? Again, this is purely speculation but it seems beneficial if the latter is the case. The field is obviously of theoretical value, given that a greater understanding of how - in principle - rather simple tasks are solved in a general manner allows optimization that could possibly result in superior behaviour of artifical agents in comparison with natural ones. However, while such speculation certainly is interesting, one must consider the possibility that non-determinism introduced with stochastic subprocesses could be the best strategy if generalization is the primary goal. While such a reality could limit the performance in specific situations, it has a certain appeal to it, as it would reinforce the notion that nature - through natural selection - is a process that is optimal by not being optimal. A rather interesting notion, from a philosophical point of view as well as technical; seeing as how genetic algorithms is a very interesting field illustrating the apparent strenghts of
combining a mathematical and technical approach with ideas and concepts plentiful in nature.

Our conclusion is thus that a general heuristic navigation strategy is very difficult to design, given the possibility of the superiority of applying stochastic processes to the main navigational process, viz. introducing non-determinism. We have illustrated this difficulty through reasoning and experimentation of four rather general scenarios utilizing a long-term strategy model to further describe the complexity of any applied navigational task. There are, as previously mentioned, several areas which require additional research and possibly advancements in hardware to fully comprehend and analyse. While we assume \( N \neq NP \), we note that even if that is not the case; the task of generalization is still a difficult one and our conclusion is therefore valid in any case. While we have argued for a non-deterministic approach, we acknowledge the fact that this is just speculation on our part and that additional experimentation is required. Given that the field, sc. navigation strategy design, is a very broad one, incorporating several branches of computer science, mathematics, biology and physics we also state explicitly that our conclusions should be viewed in the context they are made, viz. a rather limited one.
Part III
Appendix

A Graph Theory

In this section we introduce and discuss several main concepts of graph theory required to fully comprehend the basics of graph problems. We begin by defining what a graph is mathematically (see [10] for this entire section), but to do so we first need to establish what the two basic building blocks of graphs are.

**Definition 1 (Edge).** An edge is a tuple that joins two vertices. If \( V \) is the set of all vertices (see definition below), then the set of all edges, \( E \), is defined as
\[
E \subseteq V \times V
\]

**Definition 2 (Vertex).** A vertex is a node that may be connected to other vertices by edges.

With these fundamental definitions we may construct a graph by adding vertices and then possibly connecting them using edges. However, since it is desirable to be able to discuss a graph with more precision, we utilize the standard mathematical definition.

**Definition 3 (Graph).** A graph \( G = (V,E) \) is a tuple consisting of the two sets \( V \) and \( E \), where \( V \) is the set of all vertices and \( E \) the set of all edges such that they are in \( G \).

With this definition we gain the ability to refer to edges - \( e \in E \) - and vertices - \( v \in V \) - separately, as well as a unit by simply using \( G \). It should be noted that while the mathematical definition is often very practical when discussing graphs, the actual implementation may vary. We discuss different implementations at the end of this section.

When working on problems that can be transformed into a graph counterpart it is often useful to exploit certain characteristics to obtain efficient algorithms. We present a few of the more important such properties, beginning with vertex degree.

**Definition 4 (Vertex degree).** Let \( G = (V,E) \) be a graph. The degree of a vertex \( v \in V \), \( \deg(v) \), is the total number of edges \( e \in E : e = (v,w) \lor e = (w,v) \) for any \( w \in V \). We will denote the degree of \( v \in V \) by \( \delta(v) \).

The concept of vertex degree may seem very obvious and trivial, but - as illustrated by the Köningsberg problem [11] - knowledge of what kind of interconnection the vertices have is often extremely useful since it allows us to deduce other interesting properties.

**Lemma 2.** Let \( G = (V,E) \) be a graph, then \( \sum_{v \in V} \delta(v) = 2 \times |E| \). This is often referred to as the **Handshake Lemma**.
Proof. Since it holds that $\forall e \in E$ $e$ connects two vertices, and every vertex $v \in V$ has exactly $\delta(v)$ such connections, every edge must add two degree "points" in total. The result then follows.

There is a very interesting result of the Handshake Lemma. Let $V_{\text{odd}} = \{v \in V \mid \delta(v) \equiv 1 \pmod{2}\}$ and $V_{\text{even}} = \{v \in V \mid \delta(v) \equiv 0 \pmod{2}\}$. Then, since $V = V_{\text{odd}} \cup V_{\text{even}}$, it follows that

$$\sum_{v \in V} \delta(v) = \sum_{v \in V_{\text{odd}}} \delta(v) + \sum_{v \in V_{\text{even}}} \delta(v) = 2 \times |E|$$

As every term in the sum of the even edges is even, the entire summation of the even edges must also be even. Since $2 \times |E|$ is an even number, it follows that the sum of the odd vertices must also be even. However, since we need an even number of odd terms to get an even sum, it follows that there are an even number of vertices with odd degree.

**Definition 5** (Path). A path between vertices $u, v \in V$ is said to exist iff there is a vertex sequence $v = (u, \ldots, v)$ such as for every two adjacent vertices $w_i, w_{i+1}$ in $v$ there is an edge $e = (w_i, w_{i+1}) \in E$. We denote a path from $u$ to $v$ with $\langle u, v \rangle$. A path such that $v$ only contains unique vertices is said to be simple.

**Definition 6** (Walk). A walk in a graph $G = (V, E)$ is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $v_i$ and $v_{i+1}$ are adjacent ($1 \leq i \leq k - 1$). That is, a path is a walk where all vertices are distinct.

**Definition 7** (Eulerian Walk). A Eulerian Walk is a walk such that it contains every edge in $E$ exactly once. A graph $G = (V, E)$ contains a Eulerian Walk iff it is connected and there are zero or two vertices of odd degree. If there is a pair of odd degree vertices, then the Eulerian walk must start at one of them, otherwise the walk can commence from any vertex.

The concept of a Eulerian Walk is vital to the understanding of graph traversal since it effectively informs us that the degree of a, potentially, very minor subset of $V$ may still affect the properties of the entire graph. When discussing the complexity of graph traversal algorithms, and graph algorithms in general, this fact will become very apparent as it limits the efficiency of many algorithms.

**Definition 8** (Cycle). A Cycle is a path $\langle v, v \rangle$, viz. a path that starts and ends at the same vertex.

**Definition 9** (Hamiltonian Cycle). A Hamiltonian Cycle is a path $\langle v, v \rangle$ which contains every vertex $v \in V$ for a graph $G = (V, E)$.

**Definition 10** (Connected). A graph $G = (V, E)$ is said to be connected if $\forall u, v \in V : \langle u, v \rangle$.

**Definition 11** (Fully connected graph). A graph $G = (V, E)$ such that $\forall v \in V : \delta(v) = |V| - 1$ is called a fully connected graph.

**Definition 12** (Planar Graph). A graph $G = (V, E)$ is said to be planar if it is possible to draw it without any edges crossing.
A GRAPH THEORY

A.1 Undirected Graphs

An undirected graph is a graph \( G = (V, E) \) such that there is no distinction between \( e_1 = (\alpha, \beta) \in E \) and \( e_2 = (\beta, \alpha) \in E \). Recall that in accordance with set theory a set cannot contain duplicate elements. As such it follows that if \( (\alpha, \beta) \in E \) then \( (\beta, \alpha) \) cannot be in \( E \) as well because \( \{ (\alpha, \beta) \} = \{ (\beta, \alpha) \} \) (due to no distinction being made).

**Lemma 3.** A fully connected undirected graph has exactly \( T_{|V|−1} \) edges.

**Proof.** Since there are \(|V|\) vertices and each one has \(|V|−1\) edges and every edge is shared between two vertices, it follows that there are exactly \( \frac{|V|^2−|V|}{2} = \sum_{i=1}^{V−1} i \) = \( T_{|V|−1} \) edges.

\( \square \)

A.2 Graph Implementations

There are two general fundamental methods of implementing a graph; through an adjacency matrix or through adjacency lists. We present the method utilized in this thesis, viz. representation through an adjacency matrix.

A.2.1 Adjacency Matrix

In an adjacency matrix we represent all edges \( e = (\alpha, \beta) \in E \) in a graph \( G = (V, E) \) by a \( R^{|V| \times |V|} \) matrix. That is, given \( e = (\alpha, \beta) \in E \) we let \( A_{ij} \), \( A \in R^{|V| \times |V|} \) where \( i = \alpha, j = \beta \), denote whether or not there is an edge between \( \alpha \) and \( \beta \). We illustrate this with an example, let

\[
A = \begin{bmatrix}
0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

then \( A \) is equivalent to the graph in Figure 7. It should be noted that if the graph is not a digraph, then \( A \) is symmetric, i.e. \( A^T = A \).
An adjacency matrix suffers greatly from the fact that it may be very memory inefficient. Since we will always allocate $|V|^2$ indices we will be wasting memory unless $|E| \in O(|V|^2)$, i.e. $G$ is fully or near fully connected. We do have $O(1)$ lookup time for any edge, however, so in an applied situation one would have to carefully consider the constraints on the work being performed as well as the performance requirements. Further, as noted previously, since an undirected graph is a symmetric matrix we are in fact storing twice the required data when working with non-digraphs.
Algorithm Evaluation

When evaluating the efficiency of some algorithm we will be focusing on the time complexity. There are several ways to evaluate the time complexity and we define them below.

**Definition 13** (Big Oh). We say that a function \( f(n) \in O(g(n)) \) if \( 0 \leq f(n_k) \leq cg(n_k) \), \( c \in \mathbb{R} \), beginning at some \( k \)-th value of \( n \). If \( f(n) \in O(g(n)) \) we say that \( g(n) \) is an asymptotic upper bound on \( f(n) \).

**Definition 14** (Big Omega). We say that a function \( f(n) \in \Omega(g(n)) \) if \( 0 \leq cg(n_k) \leq f(n_k) \), \( c \in \mathbb{R} \), beginning at some \( k \)-th value of \( n \). If \( f(n) \in \Omega(g(n)) \) we say that \( g(n) \) is an asymptotic lower bound on \( f(n) \).

**Definition 15** (Theta). We say that a function \( f(n) \in \Theta(g(n)) \) if \( f(n) \in O(g(n)) \land f(n) \in \Omega(g(n)) \). If \( f(n) \in \Theta(g(n)) \) we say that \( g(n) \) is an asymptotically tight bound on \( f(n) \), i.e. \( f(n) \) grows just like \( g(n) \).

**Definition 16** (Little Oh). We say that a function \( f(n) \in o(g(n)) \) if \( f(n) \in O(g(n)) \land f(n) \neq \Theta(g(n)) \). That is, \( f(n) \) grows slower than \( g(n) \).

**Definition 17** (Little Omega). We say that a function \( f(n) \in \omega(g(n)) \) if \( f(n) \in \Omega(g(n)) \land f(n) \neq \Theta(g(n)) \). That is, \( f(n) \) grows faster than \( g(n) \).

### B.1 Asymptotic analysis

When studying the asymptotic efficiency of an algorithm, we are concerned with how the input size effects the running time, i.e. increase the runtime. Given a function \( f(x) \) that is called \( n \) times, we can obtain an estimate of the worst case scenario, timewise, by \( K(f(x)) \times n \) where \( K(f) \) returns the worst time complexity of \( f \).

### B.2 Amortized analysis

Amortized Analysis is often used when we have a sequence of operations that take different time depending on how the structure looks and when it is invoked. Generally, it is utilized when one is concerned with the average performance in the worst case. So when applied, the focus is on the overall cost for a sequence of operations. Usually this type of analysis is applied when dealing with data structures such that they have properties which prevent the worst case scenario from occurring unless a certain condition is true. Such a condition typically prevents a non-pessimistic analysis through study by asymptotic analysis. For instance, consider a function \( f(n) \) that has a stack implementation and for every call it pops \( n \) items from the stack and pushes a single item on it.

\[
\begin{aligned}
\text{F}(n): \\
1 & \quad \text{stack.pop}(n) \quad \text{// pop n items} \\
2 & \quad \text{stack.push}(1)
\end{aligned}
\]

Suppose we call \( \text{F}(n) \) \( n \) times, then the worst case scenario is that we pop \( n \) items from the stack and push one back and the upper bound would be \( O(n^2) \). Looking
at the structure, however, enables us to conclude that it is only possible to get $O(n^2)$ if we preceed the $f(n)$ call with $n-1$ $N(0)$ calls. Hence, if we want to perform an analysis that is more bounded, we must utilize amortized analysis and there are three different methods to do so.

### B.2.1 Aggregate method

The Aggregate method is when we determine the upper bound on $T(n)/n$ so that the cost of every operation is the average cost for the whole sequence. Consider, for the examples below, a function with a stack as the underlying implementation. Let $POP(S)$ and $PUSH(S, x)$, where $S$ is the stack and $x$ is the object pushed on the stack, be defined. Both of these operations take $O(1)$ to complete. A sequence of $n$ of said operations will thus have a running time of $\Theta(n)$. Further, define $MULTIPOP(S, k)$, where $k$ is the number of objects popped, as a sequence of $POP(S)$ calls. We can then conclude that a sequence of $n$ PUSH, POP and MULTIPOP operations on a stack with 0 objects would have, as the worst case scenario, $O(n^2)$ since $O(MULTIPOP) = O(n)$. However, using the same argument as in the Amortized Analysis section, it can be noted that this is indeed not a tight bound. Obviously it is not wrong to claim that the upper bound is $O(n^2)$, but it does not reflect reality that well.

Searching for a tighter bound is done by noting that the total cost of the sequence is at most $O(n)$ because an object can at most be poped once each time it has been pushed onto the stack. So for every call of MULTIPOP($S, n$) we need to perform PUSH $n$ times. So obviously the cost cannot exceed $O(n)$ if we call PUSH $n$ times (which we do, as defined by the sequence). Each operation in said sequence would thus take $\frac{O(n)}{n} = O(1)$.

### B.2.2 Accounting method

The accounting method determines the cost by saving any “prepaid credit” when an action is performed that has a lower cost than the estimated cost. This credit can then be spent on actions that have a higher cost than their estimation. The idea is that in the end the credit will be zero as the number of times we overestimate a cost is equal to the times we underestimate it. It is imperative that the sum of amortized cost for individual operations making up the whole sequence of operations does not exceed the upper bound for the total cost. We can then state the total credit saved is the difference between the total amortized analysis and the total actual cost. It must be ensured that the total cost is never negative and that it hold for any sequence of operation said that data structure.

For instance, consider the previously defined function $f(n)$. We know that the upper bound $O(n^2)$ is not tight. Let the cost for every operation be 2, then by studying the function we know that the first time we must call $F(0)$, so that one object is pushed onto the stack. The actual cost of this operation is 1, and we thus receive 1 credit for future operations. Giving every object this credit enables us to pop it later and we thus obtain $O(n)$, which is a tighter bound.
C Map Generation

We present a simple $\Theta(mn)$ python function that can be used to generate sample environments. Note that the function helper($d$) is only used due to the line length limitation of the code listing box.

Code Listing 1: map_gen (py)

```python
'''
Generate map
'''
import random

def helper( d):
    return int(not random.randrange(1 + d))

def map_gen( m, n, d):
    # we generate a random graph based on d...
    A = [[helper(d) for p in range(n)] for q in range(m)]
    i=0;j=0
    # ...then we clear one path to ensure
    # traversability from (0,0) to (m,n)
    while 1:
        if i >= m:
            while not (j >= n - 1):
                j += 1
                A[i - 1][j] = 0
            break
        if j >= n:
            while not (i >= m - 1):
                i += 1
                A[i][j - 1] = 0
            break
        A[i][j] = 0
        if not random.randrange(2):
            j += 1
        else : i += 1
    return A
```

We also present a simple function to draw said environment in code listing 2.

Code Listing 2: draw_env (py)

```python
'''
Draw Environment
'''
import Image
import ImageDraw

def draw_env( A, path, bgc, wallc, pathc, target ):
    g = Image.new('RGB', (len(A[0]), len(A)))
    gh = ImageDraw.Draw(g)
    gh.rectangle([(0,0), (len(A[0]), len(A))], fill = bgc)
    for c in [(i,j) for i in range(len(A[0])) for j in range(len(A))]:
        if A[c[0]][c[1]]: gh.point(c, wallc)
    for p in path: gh.point(p, pathc)
    g.save(target, 'PNG') #or whatever 24+bpp-format
```
## D Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>{···}</td>
<td>Set</td>
<td></td>
</tr>
<tr>
<td>∪</td>
<td>Union</td>
<td>( {a} \cup {b} = {a, b} )</td>
</tr>
<tr>
<td>(</td>
<td></td>
<td>(a, b)</td>
</tr>
<tr>
<td></td>
<td>(</td>
<td>a</td>
</tr>
<tr>
<td>(\exists)</td>
<td>Exists</td>
<td>There is at least one</td>
</tr>
<tr>
<td>(\forall)</td>
<td>For All</td>
<td></td>
</tr>
<tr>
<td>(\in)</td>
<td>Member</td>
<td></td>
</tr>
<tr>
<td>(\notin)</td>
<td>Not Member</td>
<td></td>
</tr>
<tr>
<td>(\Rightarrow)</td>
<td>Implies</td>
<td></td>
</tr>
<tr>
<td>(\rightarrow)</td>
<td>Gives/Maps</td>
<td></td>
</tr>
<tr>
<td>(\Leftrightarrow)</td>
<td>Equivalence</td>
<td></td>
</tr>
<tr>
<td>(\land)</td>
<td>And</td>
<td></td>
</tr>
<tr>
<td>(\lor)</td>
<td>Or</td>
<td></td>
</tr>
<tr>
<td>(\mathbb{P}_k)</td>
<td>Polynomials of degree (\leq k) (subspace of (\mathbb{R}^k))</td>
<td></td>
</tr>
<tr>
<td>(\mathbb{Z})</td>
<td>The Integers</td>
<td></td>
</tr>
<tr>
<td>(\mathbb{Z}^+)</td>
<td>The positive Integers</td>
<td></td>
</tr>
<tr>
<td>(\mathbb{Z}_k)</td>
<td>The set of all Integers modulo (k)</td>
<td></td>
</tr>
<tr>
<td>(\mathbb{R})</td>
<td>The Real Numbers</td>
<td></td>
</tr>
<tr>
<td>(\mathbb{R}^{m \times n})</td>
<td>The set of all (m \times n) matrices</td>
<td></td>
</tr>
<tr>
<td>(f_A(x))</td>
<td>The function defined by the matrix (A)</td>
<td></td>
</tr>
<tr>
<td>(A_{ab})</td>
<td>Index (ab) of (A)</td>
<td>Column (a) Row (b)</td>
</tr>
<tr>
<td>([0 ; 1 ; ; ; 1 ; 0])</td>
<td>Matrix</td>
<td>“;” denotes row break</td>
</tr>
<tr>
<td>{...</td>
<td>...}</td>
<td>Such that</td>
</tr>
<tr>
<td>(\subseteq)</td>
<td>Subset equal</td>
<td>(\leq) for sets</td>
</tr>
<tr>
<td>(\subsetneq)</td>
<td>Subset not equal</td>
<td>(&lt;) for sets</td>
</tr>
<tr>
<td>(\langle a, b \rangle)</td>
<td>Path or inner product</td>
<td>(depends on context)</td>
</tr>
<tr>
<td>(\vec{ab})</td>
<td>The Vector from (a) to (b)</td>
<td></td>
</tr>
<tr>
<td>(\sum_{k=a}^{n}(k+1))</td>
<td>Sum</td>
<td>((a + 1) + (a + 2) + \cdots + (n + 1))</td>
</tr>
<tr>
<td>(\prod_{k=a}^{n}(k+1))</td>
<td>Product</td>
<td>((a + 1)(a + 2)\cdots(n + 1))</td>
</tr>
<tr>
<td>(!)</td>
<td>Factorial</td>
<td>(3! = 1 \times 2 \times 3)</td>
</tr>
<tr>
<td>(\min)</td>
<td>Minimum</td>
<td></td>
</tr>
<tr>
<td>(\max)</td>
<td>Maximum</td>
<td></td>
</tr>
<tr>
<td>(\lim)</td>
<td>Limit</td>
<td></td>
</tr>
<tr>
<td>(\int)</td>
<td>Integral</td>
<td>(\int_0^a f(x) , dx = F(x)</td>
</tr>
<tr>
<td>(\dfrac{\partial}{\partial x})</td>
<td>Partial Derivative</td>
<td>(\dfrac{\partial(3x+y)}{\partial x} = 3)</td>
</tr>
<tr>
<td>(T_n)</td>
<td>The (n:\text{th}) triangle number</td>
<td>(T_3 = 1+2+3 = 6)</td>
</tr>
<tr>
<td>(\lg)</td>
<td>Logarithm</td>
<td>(usually radix 2, may vary)</td>
</tr>
</tbody>
</table>
References


REFERENCES


