

Fast and Optimal Pathfinding

Daniel Harabor

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

October 2014

Except where otherwise indicated, this thesis is my own original work.

Daniel Harabor
28 October 2014

To those who taught me.

Acknowledgments

During my candidature I have been fortunate to work with and receive guidance from a number of talented individuals. Foremost among these are my co-supervisors: Adi Botea, Alban Grastien and Philip Kilby. They have each given me so much of their time, their patience, their encouragement and their deep expertise that I feel I can never repay the debt. They are exceptional individuals and I have tried at all times to model myself according to their example. Working alongside them and learning from them has shaped me personally and professionally and I am humbled for having had the opportunity.

I owe another great debt to Sylvie Thiébaux, who has mentored me throughout my time at ANU and NICTA. Sylvie is a tireless champion who has opened many doors and given me many opportunities to interact with and present my work to eminent individuals from academic and non-academic backgrounds. She has taught me to be confident, to tell it how it is and to always persevere in the face of adversity. I am immensely fortunate to have had her in my corner.

A special thank you to Patrik Haslum for the endlessly stimulating discussions about AI Planning and Heuristic Search. Thanks also for the many cigarettes!

Thank you to my many friends from NICTA and elsewhere who made my time in Canberra so enjoyable. Special shout-outs to Chris Bennetts-Cash, Adam Rumbold, Ayman Ghoneim, Nina Narodytska, Debdeep Banerjee, Jason Li, Pattaraporn Khuwuthyakorn, Cindy Wang, Eddie Li and Zahra Zamani.

Thank you to my brother, Alex, who constantly reminds me there is a more amusing side to life. Some of my favourite moments from these last few years have been while visiting Melbourne and catching up with you.

I am grateful to my wife, Alina, who encouraged, supported and otherwise put up with me throughout this long endeavour. Her love has been like a light, shining brightly always and helping me find my way home.

Finally, to my parents, Daniel and Elena, who uprooted their lives for the sake of my brother and I. You taught me the importance of education, you taught me drive and ambition, you taught me respect, compassion, forgiveness and love. You raised me as a boy and then you shaped me into a man. You showed me the path and I follow it still. Thank you.

Abstract

Pathfinding (or navigating) from A to B is a common problem in Computer Science with broad practical applications in areas as diverse as digital entertainment, logistics and robotics. Pathfinding is made difficult when many variations, or symmetries, of the same path exist. Symmetry slows down search by forcing otherwise performant algorithms to waste time considering many equivalent states. We speed things up by developing new online and offline symmetry-breaking techniques that eliminate a large number of symmetric states. Our contributions are optimality preserving, memory efficient and can have a dramatic positive effect on algorithmic performance. They are especially well suited to speeding up pathfinding search on grid-maps of the type widely employed in computer games and robotics. Moreover, our work is largely orthogonal with a wide range of efficiency-improving techniques that have been previously described in the academic literature.

We investigate a number of novel symmetry breaking approaches. Rectangular Symmetry Reduction (RSR) identifies symmetric path segments during an offline pre-processing step. This approach is optimal, requires very little overhead (usually a few seconds of up-front time and a linear amount of memory) and it can improve the performance of classical pathfinding algorithms such as A* by several factors. Our second contribution, Jump Point Search (JPS), significantly improves on the performance of RSR and currently represents the state of the art for pathfinding on grid-map domains. In its online form JPS requires zero preprocessing, zero additional memory and always finds the shortest path. Our experiments show that JPS can consistently improve the performance of A* search by over one order of magnitude and more. In its offline form JPS reformulates the search space to achieve even better performance but requires an up-front investment of time. The algorithm has zero memory overheads when applied to graphs that are stored as an adjacency list. When applied to graphs stored as an adjacency matrix, the algorithm introduces a linear-sized memory overhead.

In addition to RSR and JPS we study the related any-angle pathfinding problem. Recently formulated but nevertheless well studied, this problem involves finding a shortest path in a grid-map domain but asks that the path is not constrained to the points of the grid. Though a range of approaches have been suggested there are no effective techniques, to date, that are both optimal and online. As a final contribution we give a first algorithm that can provably compute solutions having both of these desirable characteristics.

Contents

Acknowledgments	vii
Abstract	ix
List of Figures	xv
List of Tables	xvii
1 Introduction	1
1.1 Practical Considerations	2
1.2 Target Applications and Research Challenges	3
1.3 Contributions	4
1.3.1 Symmetry Breaking in Pathfinding Search	4
1.3.1.1 Rectangular Symmetry Reduction	4
1.3.1.2 Jump Point Search	5
1.3.2 Optimal Any-angle Pathfinding	6
1.4 Publications and Thesis Overview	6
2 Literature Review	7
2.1 The Single-agent Pathfinding Problem	7
2.1.1 Search Graphs	8
2.1.2 Paths and Instances	8
2.2 Types of Search Graphs	9
2.2.1 Grid Maps	9
2.2.2 Navigation Meshes	10
2.2.3 Roadmaps	11
2.2.4 Shortest Path Maps	11
2.2.5 Visibility Graphs	12
2.2.6 Comparative Analysis	12
2.3 Search Strategies	13
2.3.1 The Search Process	14
2.3.2 Blind Search	14
2.3.3 Informed Search	15
2.3.4 Local Search	15
2.3.5 Performance Enhancements	16
2.4 Implementation Enhancements	17
2.5 Abstraction	18

2.5.1	Approximate Abstractions	18
2.5.2	Road Network Hierarchies	18
2.6	Improved Heuristics	19
2.7	Symmetry Breaking	21
2.7.1	Approaches for Breaking State Symmetries	21
2.7.2	Approaches for Breaking Path Symmetries	22
2.8	Dominance Detection	24
2.9	Euclidean Shortest Paths	25
2.9.1	Any-angle Pathfinding Algorithms	25
2.9.2	Euclidean Shortest Path Algorithms	26
2.10	Summary and Discussion	26
3	Rectangular Symmetry Reduction	29
3.1	Path Symmetries	30
3.2	Symmetry Breaking In 4-Connected Grid Maps	31
3.2.1	Online Insertion	32
3.2.2	Optimality	33
3.2.3	Identifying Empty Rooms	34
3.3	Symmetry Breaking In 8-Connected Grid Maps	35
3.4	Further Enhancements	38
3.4.1	Perimeter Reduction	38
3.4.2	Online Node Pruning	39
3.5	Memory Requirements and Dynamic Environments	40
3.6	Experimental Setup	40
3.7	Results	41
3.7.1	4-Connected Grid Maps	43
3.7.2	Comparison to Swamps:	43
3.7.3	Comparison to Portal Heuristic:	44
3.8	Discussion	44
4	Jump Point Search	47
4.1	Notation and Terminology	48
4.2	Jump Points By Example	48
4.3	Neighbour Pruning Rules	49
4.3.1	Forced and Natural Neighbours	50
4.4	Algorithmic Description	50
4.5	Optimality	52
4.6	Weighted Grid Maps	55
4.7	Time and Space Requirements	56
4.8	Experimental Setup	56
4.9	Results	57
4.9.1	Comparison with Swamps	57
4.9.2	Comparison with HPA*	58
4.10	Discussion	59

5	Improving Jump Point Search	61
5.1	Introduction	62
5.2	Block-based Symmetry Breaking	62
5.2.1	Additional Considerations	64
5.3	Preprocessing	65
5.3.1	Properties	66
5.3.2	Advantages and Disadvantages	66
5.4	Improved Pruning Rules	67
5.5	Experimental Setup	68
5.6	Comparison with JPS 2011	69
5.7	Comparison with SUB	71
5.8	Analysis	71
5.9	Discussion	73
6	Any-angle Pathfinding	75
6.1	Introduction	76
6.2	Preliminaries	77
6.3	Principle of Anya	78
6.4	Algorithm	78
6.5	Correctness and Optimality	80
6.6	Discussion	82
7	Conclusion	83
7.1	Summary	84
7.1.1	Contributions to Symmetry Breaking in Pathfinding Search . . .	84
7.1.2	Contributions to Any-Angle Pathfinding	85
7.2	Future Work	86
7.2.1	Symmetry Breaking	86
7.2.2	Any-angle Pathfinding	87

List of Figures

3.1	An example of path symmetry	30
3.2	Rectangular Symmetry Reduction on 4-connected maps	31
3.3	Examples of online insertion	33
3.4	Searching with A* vs. A* + RSR	34
3.5	Rectangular Symmetry Reduction on 8-connected maps	36
3.6	RSR enhancement: perimeter reduction	38
3.7	RSR enhancement: online pruning	39
3.8	Search time speedup: RSR	42
4.1	Examples of straight and diagonal jump points	49
4.2	JPS pruning rules	50
4.3	Types of turning points	54
4.4	Search time speedup: JPS vs. Swamps vs. HPA	59
5.1	Example of block-based jumping	63
5.2	Examples of JPS+: preprocessing and node insertion	65
5.3	Example of pruning intermediate jump points	67
5.4	Performance of improved JPS variants vs. original JPS	70
5.5	Search time performance: JPS+ vs. SUB	72
6.1	Illustration of Lemma 8	77
6.2	The problem with Theta*	77
6.3	Observable and non-observable successors	79
6.4	Illustration of Lemmas 9 and 10	81

List of Tables

3.1	RSR preprocessing requirements	41
3.2	Search time speedup: RSR vs PH-e.	44
4.1	Node expansion speedup: JPS vs. Swamps vs. HPA*	57
5.1	A comparative breakdown of how JPS and A* spend their time	62
5.2	Expanding random start nodes: JPS vs. improved variants	68
5.3	Preprocessing requirements: JPS+ vs. SUB	69
7.1	A summary overview of all RSR, JPS and their variants.	85

Introduction

Pathfinding is the name given to a broad class of related problems often appearing in Computer Science. In the canonical case the pathfinding problem asks that we navigate between an arbitrary pair of start and target locations drawn from a map. Such problems appear in a myriad of important and real-life contexts. For example:

- Pathfinding is at the heart of all personal GPS navigation devices.
- Pathfinding is used by transportation and logistics companies to improve performance and reduce operating costs.
- Pathfinding is central to the correct operation of personal and industrial robots.
- Pathfinding powers the AI systems of many modern computer games.

Just as there are many possible application areas for pathfinding there are equally many variations of the problem as well. Frequently we are asked to find a path which is optimal with regard to distance. In other cases a more desirable path can be one that minimises travel time or even travel cost. Sometimes it may not necessary – or even possible – to compute an optimal path: low-power or real-time computing devices place strict limits on the amount of resources (CPU, memory) that are available for navigation. In these cases near-optimal, bounded sub-optimal or indeed any path at all will often suffice. Other types of pathfinding problems include (but are not limited to) finding a path in a dynamic environment, finding a path in three or more dimensions, navigating in the presence of other moving entities and even chasing a moving target. All these topics have received extensive attention from both researchers and industrial practitioners.

In this thesis we will aim to compute distance-optimal paths in a discrete and static two-dimensional environment. Our work has possible applications in robotics and computer games. We will study a range of different approaches but in each case our objective will be (i) to find the shortest path, (ii) as quickly as possible and (iii) as economically as possible with respect to available resources such as memory and pre-computation time.

1.1 Practical Considerations

There are two common questions which every pathfinding practitioner must address at the outset: (i) how to construct a map (or search graph) that represents the operating environment in which we want to navigate and (ii) how to actually search the map for a path. We discuss each in turn.

Maps used for pathfinding are exactly analogous to their real-life cartographic counterparts: they document the salient features of an environment such as the locations of roads, the lengths of road segments, terrain type and elevation and the placement of both natural and artificial obstructions such as buildings or waterways. Depending on the application and setting some of these features may be more important or less important; still others we might choose to simply ignore. For example:

- In computer games the virtual environment is often discretised into a grid of cells, each of which is either traversable or non-traversable.
- In logistics and personal navigation road maps accurately describe the features of transportation networks but details about the rest of the world are omitted.
- In robotics, visibility graphs capture information about which points in the environment can be reached from one another by travelling in a straight line. Other information is often secondary from the perspective of a mobile robot.

Many more types of maps exist but for the purposes of pathfinding there is no single “best” choice. Each type of map emphasises different features of the physical or virtual environment and each has distinct strengths and weaknesses – characteristics which sometimes only become apparent in a particular application or setting.

Having decided on a particular type of map, pathfinding practitioners must also decide, often at the same time, how to actually search the map for a path. The academic literature is rich with works that describe different techniques for solving pathfinding problems. Examples include:

- Search algorithms inspired by the behaviour of real-life insects.
- Blind-search algorithms developed to systematically solve mazes.
- Informed-search algorithms that employ heuristic lower-bounds in order to search the most promising areas of the map first.

Some approaches always find the shortest path while others do not. Some come with performance and efficiency guarantees while others do not. Depending on the target application (and even map type) some techniques may dominate others but there is no single “best” technique that is preferable in all cases.

1.2 Target Applications and Research Challenges

In this thesis we study optimal pathfinding in the context of two-dimensional computer games. We focus on real-time strategy games (such as StarCraft, WarCraft et al.) and real-time role-playing games (Baldur's Gate, Dragon Age, etc.). In both cases the game map is often represented as a grid of traversable and non-traversable cells. Though our principal results are derived in the context of computer games they are equally applicable to other related application areas such as two-dimensional mobile robot navigation, finding shortest paths on printed circuit boards and indeed any setting where the operating environment can be described compactly in terms of fixed adjacency relations between nodes on a map.

Grid maps are popularly employed in computer games for several reasons: (i) they are simple to understand and apply; (ii) they are memory efficient (only one bit of storage is required per grid cell); (iii) they facilitate fast spatial reasoning by providing constant-time access to individual grid cells. At the same time, grids have two significant disadvantages. First, there are often many cells in a grid and the distance between them is small. Many steps may need to be taken before the target can be reached and this makes pathfinding surprisingly challenging. Second, computed paths must always pass through the fixed points of the grid (either the centre of each cell or else the corners). This means that characters navigating in a game world behave unrealistically: they always turn at fixed angles of 45 and 90 degrees and they follow paths which, although optimal with respect to the points of the grid, are not optimal with respect to the underlying game environment.

Our first research challenge, relating to the difficulty of pathfinding on a grid, is a topic that has been previously examined in the academic literature. Many techniques have been developed; they can be broadly categorised into one or more of: spatial abstraction, graph pruning and memory heuristics. Each of these approaches involves some pre-computation before pathfinding can begin and each of these approaches trades either memory or optimality for speed. Spatial abstraction techniques for example create a small approximation of the grid map that is faster to search but computed paths are not guaranteed to be optimal. Graph pruning and memory heuristics retain optimality but require significant investment in terms of additional memory and up-front pre-processing time: both scarce resources in computer games.

The second research challenge, relating to the optimality of grid paths, is known in the academic literature as the Any-Angle Pathfinding Problem. Given a grid map and two points upon it, we are asked to compute a shortest path that does not necessarily intersect the fixed points of the grid. A range of approaches exist, both optimal and approximate, including solutions to a more general form of the problem known as the Euclidean Shortest Path Problem. Optimal approaches can be typically characterised as very fast but requiring pre-computation and (sometimes extensive) memory overheads. These methods are not applicable in many computer game and robotics contexts as each time the map changes the pre-computation must begin anew. Approximate techniques on the other hand are fast, online and overhead-free but by their very nature do not yield distance-optimal paths.

1.3 Contributions

We address our first research challenge, related to the difficulty of pathfinding on grid maps, by developing Rectangular Symmetry Reduction and Jump Point Search: two very fast and in some cases very economical algorithms that can dramatically improve the performance of pathfinding search. Our central insight in this case concerns the novel application of symmetry elimination to pathfinding.

We address our second research challenge, related to the optimality of paths on a grid, by giving a new theoretical results which show that the Any-angle Pathfinding problem can be solved exactly and online. We also describe in detail the implementation and correct operation of a corresponding search algorithm, Anya, which appears especially well suited to pathfinding in computer games.

1.3.1 Symmetry Breaking in Pathfinding Search

In Chapters 3, 4 and 5 we speed up grid-optimal pathfinding in an entirely novel way: through the identification and elimination of path symmetries. Unlike the majority of works from the literature, in which a path is concretely defined in terms of a specific set of nodes and edges, we consider a path as an ordered sequence of (often) interchangeable steps or actions. By taking an alternative view of the problem we show that many of the paths between two points on a grid map are simply permutations of each other; i.e. they all involve the same set of actions and they all have identical cost.

We show that in the presence of path symmetry classical and usually very performant algorithms such as A* [Hart et al., 1968] will waste much time looking at permutations of all shortest paths: from the start node to each expanded node. To avoid this behaviour we develop a range of pathfinding techniques that can discard from consideration many equivalent paths and in the process dramatically speed up grid-based pathfinding. Improvements typically range from several factors to as much as two orders of magnitude. In addition to being very fast we show that symmetry breaking in pathfinding can be very economical: our approaches require little-to-no upfront pre-processing and little-to-no additional memory; making them especially well suited to pathfinding in computer games.

1.3.1.1 Rectangular Symmetry Reduction

In Chapter 3 we present Rectangular Symmetry Reduction (RSR): an offline symmetry breaking algorithm for grid maps. It is fast, memory efficient, optimality preserving and can, in some cases, eliminate entirely the need to search. RSR decomposes an arbitrary uniform-cost grid map into a set of empty rectangles and removes from each such rectangle all interior nodes. Macro edges are then added between selected pairs of perimeter nodes to facilitate provably optimal traversal through each rectangle. We develop two variants of RSR: one for 4-connected grid maps and the other for 8-connected grid maps; both common domains that often appear in the AI literature and in real-life computer games.

In addition to the basic algorithm we also develop two pruning strategies which can significantly reduce the number of nodes that need to be explored during search. The first enhancement is applied offline and allows us to discard, in many cases, nodes from the perimeter of an empty rectangle. The second enhancement is applied online and allows us to speed up node expansion by selectively evaluating either all neighbours associated with a node or only a small subset.

We evaluate RSR on a range of uniform-cost grid maps from the academic literature and find that it can improve the running time of A* search by several factors. In certain cases the speedup can be as much as one order of magnitude. When we compare RSR to Swamps [Pochter et al., 2010], a contemporary and state-of-the-art graph pruning technique, we find that the two approaches have complementary strengths and that they could be easily combined. We also identify a range of benchmark problems on which RSR dominates convincingly.

1.3.1.2 Jump Point Search

In Chapter 4 we present Jump Point Search (JPS): an online pruning strategy that deals with path symmetry by selectively expanding only certain nodes on a grid map which we call *jump points*. Moving from one jump point to the next involves travelling in a fixed direction while repeatedly applying a set of simple neighbour pruning rules until either an obstacle or a jump point is reached. Because we do not expand any intermediate nodes we can improve the search time performance of standard A* by an order of magnitude and more. JPS is unique in the pathfinding literature in that it has very few disadvantages: it is simple, yet highly effective; it preserves optimality, yet requires no extra memory; it is fast, yet requires no pre-processing. Further, our method is orthogonal to and easily combined with many speedup techniques from the literature. We are unaware of any other algorithm which has all these features.

In Chapter 5 we extend the ideas of JPS in order to make three further contributions: (i) we give a new and more efficient procedure for online symmetry breaking by manipulating “blocks” of nodes at a single time rather than individual nodes; (ii) we give new offline pre-processing technique that can identify jump points a priori in order to further speed up pathfinding search. (iii) we enhance the pruning rules of JPS, allowing it to ignore many intermediate nodes that must otherwise be expanded; On benchmark domains taken from real computer games we show that our enhancements can improve JPS performance by anywhere from several factors to over one order of magnitude. We also compare our approaches with SUB [Uras et al., 2013]: a very recent state-of-the-art pathfinding algorithm. We find that the two techniques have complementary strengths. In addition we show that there are large sets of benchmark instances where our improved JPS variants are faster.

1.3.2 Optimal Any-angle Pathfinding

In Chapter 6 we develop a new optimal algorithm for solving the Any-angle Pathfinding Problem. Our approach involves searching over sets of states represented as intervals. From each interval we select a representative point to derive a corresponding f -value for the set. We give theoretical results that show our algorithm, Anya, always returns an optimal path. Moreover it does so entirely online and without any pre-processing. To the best of our knowledge Anya is the first known algorithm to provide such features and guarantees. Anya answers an open question from the areas of Artificial Intelligence and Game Development: is there an any-angle pathfinding algorithm which is online and optimal? The answer is yes.

1.4 Publications and Thesis Overview

The remainder of this thesis is structured as follows:

- Chapter 2 surveys pathfinding results from academic and industry literature and provides a background for remaining chapters.
- Chapter 3 presents the Rectangular Symmetry Reduction algorithm. The contributions from this chapter have been previously reported in [Harabor and Botea, 2010; Harabor et al., 2011].
- Chapter 4 presents the Jump Point Search algorithm. The contributions from this chapter have been previously reported in [Harabor and Grastien, 2011; Harabor and Grastien, 2012].
- Chapter 5 presents various enhancements to the Jump Point Search algorithm. The contributions from this chapter have been previously reported in [Harabor and Grastien, 2014].
- Chapter 6 presents the new results for solving the any-angle pathfinding problem optimally and online. The contributions from this chapter have been previously reported in [Harabor and Grastien, 2013].
- Chapter 7 presents the conclusion of the thesis and a summary of future work.

Literature Review

Pathfinding is the problem of navigating from one location to another on a map. The topic is well studied in Computer Science and represents an active area of research in sub-fields such as Artificial Intelligence, Computational Geometry, Computer Graphics, Game Development and Robotics. Many different pathfinding techniques have been proposed with each such work solving the problem to some degree in a specific and often specialised context. In this chapter we identify broad themes from across the academic literature and review a range of both classical and more recent results. We focus specifically on two common variations of the single-agent pathfinding problem: finding a shortest path in a discrete search graph and finding a shortest path in a continuous plane.

We consider a wide range of approaches for constructing discrete search graphs including grid maps, road maps and other popular and successful methods. We then compare and contrast a variety of algorithms that have been developed for computing shortest paths in discrete graphs. These include: heuristic methods, abstraction techniques and search space pruning strategies. We also examine known geometric techniques for pathfinding in continuous planar environments. This variation of the single-agent pathfinding problem is particularly challenging as there are no known methods that can solve the problem optimally and entirely online (i.e. without any pre-processing). We compare and contrast a range of solutions, both exact and approximate, and discuss their various trade-offs.

2.1 The Single-agent Pathfinding Problem

Single-agent pathfinding is the problem of navigating a lone entity, for example a robot or a virtual character, from one point to another in a given operating environment. In the most common setting environments are two-dimensional Euclidean (i.e. flat) or three-dimensional Geodesic (i.e. curved) spaces. They can take the form of a *spatial network* (i.e. a set of connected points) or they can be described as a collection of traversable and non-traversable polytopes; the latter often being called obstacles. There are many variations of the single-agent pathfinding problem. These arise by adjusting certain parameters associated with the problem such as:

- The objective function. In the canonical case the aim is to minimise travel

distance but other objective functions (including more than one at a time) could also be used.

- The type of agent. In the canonical case agents are modeled as points but they could have arbitrary shapes, sizes and capabilities that restrict or enhance movement.
- The operating environment. In the canonical case agents operate in a (i) static (ii) fully observable (iii) discrete environment. Depending on the application however, any or all of these assumptions may need to be lifted.
- Solution quality. In the canonical case agents are required to find an optimal path. In some real-time or resource constrained settings however a near-optimal or bounded suboptimal path may be preferable.
- Path constraints. In the canonical case the agent is simply required to move from the start position to the target position without intersecting any obstacles. In other settings additional constraints may complicate this task; for example the agent may need to visit certain pre-specified locations before arriving at the target.

In this thesis we will focus on optimal single-agent pathfinding in both discrete and continuous environments. We assume the environment is static and fully observable and that the agent can be modeled as a point. We will employ an objective function that minimises travel distance and we do not have any path constraints.

2.1.1 Search Graphs

Regardless of the problem variation at hand, practitioners typically all begin by constructing a model of the operating environment $G = (V, E)$ known as a *search graph*. Here V is a set of permissible locations that an agent can occupy; these are often referred to as the nodes or *vertices* of the graph. Meanwhile E is the set of *edges* that connect adjacent vertices. Edges can be thought of as roads or corridors that an agent can travel along or actions that can be executed in order to transition the agent from one location to another. The cost associated with each such move is called the *edge weight*. Weights often represent distance travelled but they could stand for other types of metrics as well; e.g. travel time or fuel consumption. When the cost of moving between two vertices a and b can differ to the cost of moving from b to a the graph is said to be *directed*¹. Otherwise the graph is said to be *undirected*.

2.1.2 Paths and Instances

A *path* $\pi = \langle v_0, v_1, \dots, v_{k-1}, v_k \rangle$ can be interpreted as a walk in a search graph $G = (V, E)$. Each v_i is a vertex in V and each pair of adjacent vertices (v_i, v_{i+1}) are connected by an associated edge in E . When searching for a path we designate the

¹This includes the case when the cost in one direction is ∞

start location of the agent s and its destination or target location t . An *instance*, then, is an s - t pair that concretely define a pathfinding problem.

Once a path has been found its *quality* is typically evaluated in one of two ways: length or cost. The *length* of a path refers to the number of edges that comprise the path. The *cost* of a path refers to the total weight of all edges that comprise the path. An *optimal* path is the lowest-cost path between vertices s and t which can be found in G .

2.2 Types of Search Graphs

There are many ways to construct a search graph and in this section we discuss a broad range of popular approaches: grid maps, navigation meshes, road maps, shortest path maps and visibility graphs. All are examples of *explicit* search graphs. Explicit means that all vertices and all edges are enumerated apriori, before any pathfinding can begin. Such graphs arise in many settings including computer games [Davis, 2000; Tozour, 2002; Champandard, 2009], routing [Sanders and Schultes, 2005; Goldberg et al., 2006] and robot motion planning [Latombe, 1991; Choset et al., 2005]. Not all search graphs are explicit. Some are *implicit*, which means that the vertices and edges of the graph are identified on-the-fly during search. Implicit graphs appear in higher dimensional pathfinding settings [Lavalle, 1998; Bohlin and Kavraki, 2000] and related application areas such as AI Planning [Russell and Norvig, 2003].

When comparing different types of search graphs we will make reference to whether or not they preserve two important properties from the operating environment: *solution existence* and *solution optimality*. A search graph which preserves solution existence guarantees that (i) every non-obstacle point in the environment can be mapped to a vertex in the graph and (ii) if two non-obstacle points can be connected by a path in the operating environment then they can also be connected by a path in the search graph (though not necessarily the same path). A search graph which preserves solution optimality makes a similar but stronger guarantee: if a path between two points exists in the graph then there also exists in the graph a path which is cost optimal with respect to the operating environment.

As we will see not every search graph preserves existence or optimality however each representation has its own distinct advantages and disadvantages. Choosing the right one depends on the particular requirements of the pathfinding problem at hand. We discuss in turn a variety of different search graphs and then in Section 2.2.6 we compare them directly.

2.2.1 Grid Maps

A grid map is a tessellation of geometric shapes used to discretise an n -dimensional operating environment. The canonical case involves unit squares, called tiles, which are applied over a plane.² Each square tile has up to eight adjacent neighbours

²Other types of two dimensional shapes, such as hexes and tetes, are also used in this setting but less commonly; cf. [Yap, 2002].

and is marked as traversable or non-traversable depending on whether or not it intersects any obstacles. Agents are modeled as unit-size entities that occupy a single traversable tile and move from the centre of one such tile to the next. An alternative model involves point-size agents which travel in straight steps along the explicit edges of grid (and sometimes diagonally through tile interiors) [Nash et al., 2007]. In both cases straight steps incur a cost of 1. Diagonal steps, if permitted, incur a cost of $\sqrt{2}$. When diagonal moves are not permitted the grid map is said to be *4-connected*; otherwise it is *8-connected*.

Grid maps are popular for several reasons: (i) they are simple to understand and simple to apply (ii) they can be represented as a matrix of bits and stored efficiently (iii) individual nodes can be updated or queried in constant time. One significant disadvantage of grid maps is their fixed resolution. In many cases grids are too coarse to accurately model the underlying environment. Increasing the number of tiles is not always possible: there is always a corresponding increase in memory requirements and searching in a larger grid often makes pathfinding more challenging. Another disadvantage of grid maps is that they produce paths which are constrained to the points of the grid. Such paths may be not only aesthetically displeasing but they can also be longer than strictly necessary and may require post-processing to “smooth” them.

2.2.2 Navigation Meshes

A navigation mesh can be described as a low-fidelity model of an operating environment. Typically comprising a set of convex adjacent polygons, navigation meshes are often used in computer games to represent walkable and non-walkable surfaces in two and three dimensions [Snook, 2000; Tozour, 2002]. There are many approaches for constructing navigation meshes. One recent technique triangulates the environment [Demyen and Buro, 2006; Kallmann, 2010]. Another decomposes the map into voxels (three-dimensional pixels) which are then manipulated in order to construct a mesh of walkable polygons. Recursive spatial indexing algorithms can also be applied to produce navigation meshes. Quad Trees [Finkel and Bentley, 1974; Samet and Webber, 1985], for example, can be used to decompose an input map into a set of adjacent obstacle-free rectangular regions.

Meshes are popular because they are representationally complete and often more memory efficient than other search graphs e.g. grid maps. Another advantage is flexibility. For example, meshes can be hand-edited by game developers needing fine grained control over agent navigation. Meshes are also well understood by those in the game development industry and tools exist to aid in their creation; e.g. NAVPOWER and RECAST NAVIGATION. Disadvantages include:

- Meshes need to be recomputed/repaired if the environment changes.
- Computed paths often need to be smoothed or otherwise post-processed.
- Locating particular polygons involves search. In the case of Quad Trees each

such operation takes logarithmic time. In other cases, such as when a polygon ordering is not defined, linear search may be required.

2.2.3 Roadmaps

A roadmap is a set of connected points that are drawn from a given environment. Conceptually similar to *road networks*, which model automotive transportation systems, roadmaps are used to solve high dimensional pathfinding problems in the area of robotics. Many variants exist. The Probabilistic Roadmap (PRM) [Kavraki and Latombe, 1994], for example, is created by randomly sampling a configuration space (i.e. the operating environment) in order to build a connected graph. Another approach, known as the Reachability Roadmap (RRM) [Geraerts and Overmars, 2005] applies a grid tessellation in the first instance and then chooses points from the grid. Voronoi Diagrams [Aurenhammer, 1991] can also be regarded as a type of roadmap. In this case a network of edges is constructed which are all equi-distant from the two closest obstacles; the nodes of the network are found at the intersections of these edges.

Roadmaps are advantageous because they are an effective means of discretising large multi-dimensional search spaces into small graphs that can be searched efficiently.³ For example: using a roadmap any pathfinding problem can be factored into three smaller sub-problems: find a path from each of the start and target configuration to a point on the roadmap then search for a path on the roadmap that connects these two points. Disadvantages include:

- Some types of roadmaps are not complete. For example, when using a PRM the probability of a path being found, if one exists, increases with the number of sampled points [Barraquand et al., 1997]. By comparison RRM's have stronger guarantees but their applicability is limited to pathfinding in two and three dimensions.
- Roadmaps are applicable only in static environments. The introduction of new obstacles or changes to the placement of existing obstacles typically require the roadmap to be re-computed or repaired. A one-shot variation of the roadmap idea, known as Rapidly-exploring Random Trees, has been developed to address this limitation [Lavalle, 1998] but it is not as reliable as static roadmaps.
- Roadmaps compute only approximately shortest paths and they assume a single robot type (i.e. they are normally not reusable across different types of robots).

³An example of a search space with more than three dimensions is the operating environment of an industrial robot arm with $k > 3$ degrees of freedom. Such robots can be found in e.g. automotive manufacturing.

2.2.4 Shortest Path Maps

A Shortest Path Map (SPM) [Mitchell et al., 1987; Mitchell, 1997], is a tree whose nodes form a shortest path cover of an environment. The nodes of the tree are regions that are computed and organised with respect to a single source point. Each point interior to a region can be reached from the source by following an optimal path in the SPM that is fixed.

Developed by practitioners working in the area of Computational Geometry, SPMs are highly efficient data structures: they can facilitate the extraction of a shortest path from the source to any point on the map in time logarithmic to the depth of the tree. Their primary disadvantage is that they are limited to queries originating at a single fixed source. Each time the source changes the SPM must be recomputed and each such operation requires $O(n \log_2 n)$ space and time. Another disadvantage is that techniques developed for computing SPMs are complex and difficult to implement in practice [Surazhsky et al., 2005].

2.2.5 Visibility Graphs

A visibility graph [Lozano-Pérez and Wesley, 1979] is a model of an operating environment based on line-of-sight. In the canonical case each vertex in a planar environment becomes a node in a graph. Edges are then added between any pair of nodes that are visible from one another. Often employed in robot motion planning and computer games, visibility graphs are popular due to their simplicity and ability to provide strong guarantees including completeness and (depending on the search strategy at hand) Euclidean optimality. Disadvantages include:

- Visibility graphs can be very large; in the worst case every vertex is visible from every other vertex and a quadratic number of edges is required. Tangent Graphs [Liu and Arimoto, 1992] and Silhouette Zones [Young, 2001] are two variant approaches that address this shortcoming in many practical cases but their memory requirements remain worst-case quadratic.
- Visibility graphs are static models and must be recomputed or repaired if the environment changes.
- Visibility graphs do not preserve solution optimality when applied to environments in three-dimensional space or higher.⁴

2.2.6 Comparative Analysis

Selecting a technique that models an operating environment as a search graph is not a straightforward task. We have discussed a number of popular approaches but none are ideal in every situation; each has their own set of advantages and disadvantages.

⁴In two dimensions an optimal path can only circumvent a polygonal obstacle by passing through one or more of its vertices. The problem in e.g. three dimensions is that an optimal path can cross a polyhedral obstacle at any point along its surface – not just at the vertices that define its faces. The same observation is true for obstacles in higher-dimensional spaces.

Shortest-path maps, for example, preserve solution existence and solution optimality and they can solve any shortest path query w.r.t a single fixed source in just logarithmic time. Unfortunately their applicability is limited: in many pathfinding settings, particularly computer games and robotics, it is common for the source location to change frequently and each time it does a new Shortest-path map must be computed requiring time $O(n \log n)$. Grid maps, by comparison, are much better suited in such contexts: though they only preserve solution existence they can be constructed in amortized linear time and operations on individual tiles can be performed in constant time. Unfortunately grid maps can require high resolutions to accurately represent an environment and this slows pathfinding search.

Roadmaps and Navigation Meshes are techniques that produce sparse graphs because they do not rely on cells of uniform size. They can be very fast to search but must be constructed as part of an offline preprocessing step. They must both be repaired (or recomputed) if the environment changes and they both have linear-time worst-case requirements for operations on individual nodes. A similar description is true of Visibility Graphs however in this case an additional and often prohibitive space overhead may be required.

In this thesis we will employ grid-based map representations. We do not claim this representation to be *the best* however, as we have discussed, it has certain advantages that simplify pathfinding search. Moreover, grids are very popular with academic and industry practitioners; often appearing in real-world applications and often employed as benchmark problems for search algorithms. Despite widespread use very few works from the pathfinding literature are explicitly concerned with grids as objects for study and analysis. In later chapters we will see that grids present a unique set of challenges when it comes to finding shortest paths and we will present a range of grid-based techniques that significantly improve the state-of-the-art for pathfinding search on grid-based domains. We focus not only on improving the efficiency of search but also the quality of computed solutions.

2.3 Search Strategies

Given a search graph $G = (V, E)$ and a pair of vertices $s, t \in V$, which correspond to the start and target positions of a single agent, our objective is to find a path in G from s to t . A large number of strategies have been developed for this purpose. Most fall into one of three distinct paradigms:

- *Blind-search algorithms* which explore the search space in a systematic fashion using a fixed node expansion order.
- *Informed-search algorithms* which explore the search space in a systematic fashion but employ sorting and ranking to ensure that the most promising nodes are expanded first.
- *Local-search algorithms* which employ heuristic decision-making or mimicry of natural processes in order to explore only select portions of the search space.

In the course of discussing each paradigm we will sometimes find it useful to refer to two important properties of search algorithms: *completeness* and *optimality*. A search algorithm is *complete* when it guarantees that it will return a path between the nodes s and t if one exists in G . Optimality is a stronger property which entails completeness. A search algorithm is *optimal* if: (i) it guarantees completeness and (ii) it guarantees that every returned path is a shortest path in G .

Note that both properties are defined only with respect to the current search graph and not the operating environment. This has some important ramifications. For example: in order for a complete algorithm to accurately report whether or not any path exists between s and t it will require a graph G that preserves solution existence. Similarly, a path returned by an optimal search algorithm is not guaranteed to be *truly* optimal in the environment unless G is constructed in a way that preserves solution optimality.

2.3.1 The Search Process

Looking for a path between two locations s and t is a process of repeatedly *expanding* and *generating* nodes from a search graph G . Expanding a node n is equivalent to performing an action that moves the agent from its present location to the location associated with node n . As part of this operation the *successors* of n (those nodes immediately adjacent to n in the graph G) become generated. This means they are evaluated in terms of cost and added to an expansion queue. This process continues until either (i) the target node is expanded and a path is returned or (ii) all the nodes in the expansion queue are exhausted and the algorithm returns failure.

2.3.2 Blind Search

Blind Search is a systematic search strategy that expands nodes according to a fixed order. Often applied to search problems involving small graphs and trees, Blind Search algorithms can sometimes find good paths, even optimal paths, much faster than more sophisticated search strategies. Two classical examples from the literature are Breadth First Search (BFS) [Moore, 1959] and Depth-First Search (DFS) [Russell and Norvig, 2003]. Both of these algorithms construct a search *tree* which has the start node s at its root. BFS proceeds by expanding all nodes at a given level of the tree before moving down to the next level. DFS meanwhile expands a node and immediately recurses down the tree by choosing one of its successors to explore next⁵.

Though BFS is a complete and optimal search strategy it requires an amount of space and time which increases very quickly; in some domains size of the search tree is exponentially related to branching factor and depth. This means that on challenging real-world problems, for example involving large graphs and long paths, BFS will often run out of time or out of memory well before any solution can be found. DFS, by comparison, requires an amount space that is only linear in the

⁵The selection strategy is usually fixed.

length of the longest branch of the search tree; however this strategy does not have any optimality or completeness guarantees. In the worst case DFS will explore all possible paths in the tree. This means that if the depth of the tree is unbounded, or if the tree contains repeated nodes, DFS may not terminate.

2.3.3 Informed Search

Informed search, sometimes known as best-first search, is a systematic strategy that assigns a priority to each generated node. The idea is rank nodes by merit and to always expand the most promising node first. In a typical implementation a priority queue, called the *open list*, is used to rank generated nodes. A related data structure, known as the *closed list*, keeps track of nodes already expanded.

Among the best known and most popular implementations of informed search is A* [Hart et al., 1968]. This algorithm computes, for every generated node n , a priority $f(n)$:

$$f(n) = g(n) + h(n) \quad (2.1)$$

The function $g(n)$ represents the sum of edge costs along the path from the starting node s to n . The function $h(n)$ is a heuristic that estimates the cost of the path from n to the target node t . Taken together these two values, $g(n)$ and $h(n)$, provide an estimate of the total cost of a path from s to t via n . A* can be shown to be complete, optimal and optimally efficient [Dechter and Pearl, 1985] but only if the heuristic function is both *admissible* and *consistent*.

- Admissible means that $h(n)$ always lower-bounds the cost of the path from n to t .
- Consistent means the estimated f -cost values computed at each node n and at each successor n' are non-decreasing when using the heuristic function h ; i.e. $h(n) \leq c(n, n') + h(n')$. A direct corollary arising from the consistency property is that along every path the f -value of each subsequent node is never smaller than its predecessor.
- Optimally efficient means that no algorithm which has access to the same information as A* can possibly expand fewer nodes than A*.

In terms of complexity, A* requires $O(|V|)$ space and computes a solution in worst-case time $O(|V| \log_2 |V|)$.

2.3.4 Local Search

Local search is the name given to a broad class of non-systematic search algorithms. The central idea is to employ heuristic decision-making to drive search but avoid the memory and time overheads associated with blind or informed strategies. In the case of pathfinding, local search algorithms are typically strategies where: (i) search is limited to nodes in the immediate vicinity of the agent; (ii) search returns

only a partial path toward the goal. Local search algorithms are often used in real-time settings where pathfinding agents operate in dynamic or limited information environments, may be subject to limited computing resources and are expected to react within a constant time-bound. Such agents typically apply local search in an iterated manner, interleaving very short path planning episodes with path executions steps.

Bug algorithms [Choset et al., 2005] are prototypical examples of local search applied to pathfinding. Inspired by insects from the natural world, these methods navigate in a straight line toward the goal until an obstacle is encountered. They then circumnavigate the obstacle (either partially or entirely) in order to find the best “leave” point from which to continue toward the goal. Bug algorithms are very fast, require little memory and, though incomplete and suboptimal, are surprisingly effective on a range of problems (e.g. search spaces containing only convex obstacles).

More sophisticated examples of localised search techniques can be found among the members of a large family of algorithms based on LRTA* [Korf, 1990]. Such methods employ a variation of informed search that typically expands only a fixed number of nodes or all nodes up to some fixed f -value threshold. Developed with iterated execution in mind, these techniques re-use information (e.g. heuristic values) from previous path planning episodes to improve decision making during the current episode. Though not usually optimal (in a global sense) algorithms based on LRTA* are, with few caveats, guaranteed to find a path if one exists. Recent works of this type includes: RIBS [Sturtevant et al., 2010a], kNN-LRTA* [Bulitko et al., 2010], LRTA*+sub [Hernández and Baier, 2011] and f-LRTA* [Sturtevant and Bulitko, 2011].

2.3.5 Performance Enhancements

A variety of methods have been developed to enhance the performance of the search strategies we have discussed this far. Such efforts are typically focused in one of three areas:

- Reducing the size of the search space. Typical examples include spatial abstraction, graph pruning, symmetry breaking and other approaches that can compact sets of states. We discuss a number of representative works in Section 2.5 and Section 2.7. Often such approaches involve a pre-computation step before any pathfinding can begin.
- Better heuristic functions. Typical examples include: database-driven heuristics, weighted heuristics and heuristic portfolios. An overview of such strategies is the subject of Section 2.6.
- Code and memory optimisations. These are discussed in Section 2.4.

Each approach has different strengths. Reducing the size of the search space allows pathfinding solutions to be found more quickly and permits larger search spaces to be explored. Better heuristics help to focus the search process toward promising choice-points and often provide further guidance in the form of lower-bounds.

Specialised data structures and other code and memory and memory optimisations facilitate machine-efficient operations for fetching, sorting and storing sets of states that need to be explored during search. These methods are not mutually exclusive. Indeed, as we will see, they can be and often are very successfully combined.

2.4 Implementation Enhancements

An efficient implementation of any given search strategy can have a dramatic positive effect on running time and memory requirements. The same is true in the case of the contra-positive. Such works usually focus on one of three areas:

1. More efficient data structures for use during search.
2. Optimising common instructions and common numerical operations.
3. Optimising memory access patterns.

We review a number of representative works from the literature which focus on each of these issues. In keeping with the scope of this thesis we focus on strategies for improving the performance of best-first search algorithms such as A* [Hart et al., 1968] and Dijkstra's Algorithm [Dijkstra, 1959]. A standard implementation of either of these algorithms typically employs a Binary Heap for efficiently inserting, deleting and updating elements from the open list; e.g. [Stout, 1996; Botea et al., 2004]. Though operations to elements on the heap have logarithmic complexity (quite efficient indeed!) a number of works have identified cases where this running time can be further improved.

Sequence Heaps [Sanders, 1999] are data structures organised around small sorted sequences of elements which are sized to fit into fast cache. The main idea is to create a cache-aware data structure which is optimised for insert and delete operations ⁶. A different optimisation technique involves implementing the open list as a rotating array of stacks [Cazenave, 2006]. Applicable in cases where the range of f -values in the open list is bounded by a relatively small value, this approach achieves constant-time insert, delete and update operations. If the elements in a stack need to be sorted by some secondary criterion (e.g. highest g -value) the computation requirements are higher but still better than a standard Binary Heap since sorting is limited to the elements in a single stack.

Fast Expansion [Sun et al., 2009] is a recent optimisation technique that can avoid unnecessary operations on the open list. The idea is a simple recursive suggestion: during node expansion if a successor has an f -value as good as the best node on the open list, expand the node immediately and place it directly on the closed list.

A number of works aim to speed up search through better memory organisation; e.g. [Rabin, 2000; Higgins, 2002; Cain, 2002; Park et al., 2004; Cazenave, 2006]. These recommendations include:

⁶Elements which need to be updated can be deleted and inserted anew.

1. Memory pre-allocation.
2. Lazy initialisation of data structures.
3. Bit-packing to reduce the size of data structures.
4. Mapping data structures to different parts of CPU cache.
5. Keeping a table of failed search results so they can be avoided in future.

2.5 Abstraction

Spatial abstraction is a technique often employed in pathfinding to improve the efficiency of search. Abstraction involves dividing an arbitrary map into distinct areas, much in the same way cities are often divided into suburbs. The idea is simple and natural: by representing each area as a single location (or as a small set of points) one can quickly identify an *abstract path* that begins in the same area as the starting location and terminates in the same area as the target location. Once such a path has been found it can be refined by solving a small set of sub-problems in the original map; each one involving the traversal of only a single area. In most cases this factored approach is much more efficient than searching for a path using only the original map. Another advantage, important in many realistic settings, is that the agent can begin moving as soon as the first sub-problem is refined. Finally, abstractions are, in general, entirely independent from any particular map representation and also from the the search strategy at hand.

The literature is rich with works that study the effectiveness of different abstraction techniques. Some of these are concerned with building memory efficient approximations of arbitrary maps. Others works study specialised abstractions, developed for particular domains such as road networks. We discuss both of these approaches.

2.5.1 Approximate Abstractions

Often studied in the literature of Artificial Intelligence and Heuristic Search, approximate abstraction methods are simple but very efficient decompositions of arbitrary two dimensional maps. A typical example is HPA* [Botea et al., 2004]. This algorithm takes as input a given map⁷ and returns a multi-level abstract graph. Each level of the abstraction is constructed by first dividing the map into square clusters, all of equal size, and then adding to the abstract graph nodes and edges to represent intra-cluster and inter-cluster transitions. Other works proceed in an analogous manner; e.g. PRA* [Sturtevant and Buro, 2005], MMA [Sturtevant, 2007] and HAA* [Harabor and Botea, 2008]. These abstractions are attractive for several reasons: they are small and efficient to search, they require little additional memory, they preserve completeness and, because the abstract path can be refined piecemeal, they allow agents to start moving long before the entire concrete path has been identified – an important

⁷The authors study grid maps but the technique is easily generalised to arbitrary planar graphs

advantage in many real-world settings including computer games. The price for all these benefits is that, in most cases, computed solutions are not optimal. Additionally the final path, once refined, may need further post-processing to improve its cost and aesthetic appeal [Pinter, 2001; Botea et al., 2004].

2.5.2 Road Network Hierarchies

Another line of work, originating in the literature of Algorithmics, is concerned with efficient and optimality-preserving abstractions; usually in the context of pathfinding on road networks. One popular example of such a work is TRANSIT [Bast et al., 2006, 2007]: a pathfinding algorithm that combines spatial decomposition with an all-pairs database of exact costs between a distinguished set of nodes called *access nodes*. Identified during a preprocessing step, these nodes are interesting because they form the backbone of a “highway network”; i.e. they appear on a great many number of shortest paths. The existence of such nodes is a well known and often exploited phenomenon; see e.g. Highway Hierarchies [Sanders and Schultes, 2005, 2006], Highway-Node Routing [Schultes and Sanders, 2007] and Contraction Hierarchies [Geisberger et al., 2008].

Algorithms such as TRANSIT are attractive because they preserve solution optimality and because they are very fast; query times do not typically depend the number of nodes in the graph but some other parameter such as the number of access nodes. Disadvantages include an often expensive preprocessing step to compute the set of access nodes (exponential time in the optimal case and poly-time otherwise Abraham et al. [2013, Revised May 2014]), the introduction of a significant memory overhead (e.g. TRANSIT requires an amount of space quadratic in the number of access nodes) and degraded performance when answering certain types of queries (e.g. shortest path rather than shortest distance queries and so-called “local” queries which are not handled by the pre-processed data and must be solved online using classical graph search). Another disadvantage is that these algorithms appear to have limited applicability. Recent theoretical work [Abraham et al., 2010, 2013, Revised May 2014] has shown that TRANSIT-like approaches are best suited to graphs having low degree and low *highway dimension*. These properties are typically true in road networks but they are not true for general planar graphs. For example, when TRANSIT and other similar algorithms are applied to grid graphs they exhibit large increases in memory consumption and their performance can be strongly impacted in a negative way [Antsfeld et al., 2012].

2.6 Improved Heuristics

In the context of optimal search a heuristic h is a lower-bound cost-estimation function defined over states. Its purpose is to estimate cost-to-go; i.e. to answer questions of the form “how far is this node from the target?”. Heuristics are useful if they can be computed efficiently (i.e. poly-time or better) and if they are accurate. A popular

default when pathfinding in the plane is h_{SLD} or the Straight Line Distance heuristic. h_{SLD} is consistent, admissible and runs in constant time (see Section 2.3 for a definition of these terms). In the absence of obstacles h_{SLD} is perfect but in more complex environments it can dramatically underestimate optimal distances between two arbitrary points; e.g. as seen in [Goldberg and Harrelson, 2005].

Many works focus on improving the accuracy of h_{SLD} without negatively impacting its running time. This approach usually translates into faster search algorithms. However, there are limits. A number of theoretical models [Pohl, 1977; Helmert and Röger, 2008] have shown that optimal search strategies employing even almost perfect heuristics, i.e. those having only a small additive constant for error, must expand, in the worst case, an exponential number of nodes before reaching the goal. A similar result can also be derived for the average case [Pearl, 1984]. The models used in these theoretical works often employ certain simplifying assumptions which nevertheless align well with domain models that appear in pathfinding search; e.g. constant branching factor, uniform-cost edges and a singleton goal state. Despite such seemingly discouraging results researchers have demonstrated that, in many pathfinding domains of practical interest (e.g. [Sturtevant, 2012]), better heuristic estimates can dramatically improve the performance of optimal search.

One idea for improving heuristic accuracy is to take into account domain-specific movement rules. h_{MD} (Manhattan Distance) and h_{OD} (Diagonal or Octile Distance) are two such heuristics; they retain all the properties of h_{SLD} but provide better lower-bound estimates when pathfinding in square grids. h_{MD} computes a lower-bound on the number of horizontal and vertical steps still needed to reach the target node. h_{OD} computes a similar lower-bound but assumes that diagonal steps are also allowed. Similar domain-specific heuristics have been developed for pathfinding in hexagonal grids [Yap, 2002] and more broadly for optimally solving large combinatorial puzzles [Korf and Taylor, 1996].

Another approach to improving heuristic accuracy involves pre-computing distance databases. A large family of related algorithms, sometimes called *memory heuristics*, has been described in the literature [Goldberg and Harrelson, 2005; Björnsson and Halldórsson, 2006; Sturtevant, 2007; Felner and Sturtevant, 2009; Goldenberg et al., 2010; Anderson, 2010; Yap et al., 2011]. ALT [Goldberg and Harrelson, 2005] is a typical representative. During a preprocessing step ALT selects from the map a set of *landmark* nodes and then computes a database of optimal distances from each node to every landmark. Given such a database it can be shown that admissible estimates between a pair of nodes can be computed by subtracting the distances from the two nodes at hand to a fixed landmark. ALT takes as its heuristic estimate the maximum difference over all landmarks. Further enhancements to this type of approach involve the combination of distance databases with spatial abstraction. Such examples include the Gateway Heuristic [Björnsson and Halldórsson, 2006] and the Portal Heuristic [Goldenberg et al., 2010]. Memory heuristics are very fast, typically constant time, and can improve the performance of optimal search by anywhere from several factors to an order of magnitude. Their primary disadvantages are often lengthy pre-processing times and significant space overheads.

In a recent paper Rayner et al. [2011] describe a technique for computing entire families of Euclidean heuristics. Their approach, which generalises the ALT technique, involves embedding a spatial network graph into a manifold. Computing the embedding requires solving a constrained optimisation problem that minimises heuristic error. The authors show that their approach produces very accurate results in practice but solving the optimisation problem can require significant preprocessing time and space.

SILC [Sankaranarayanan et al., 2005] and Compressed Path Databases (CPDs) are two very powerful memory-based techniques which are related to but which are not instances of memory heuristics. Rather than computing a database of distances between selected pairs of points however (i.e. a heuristic) these approaches constructs a pathfinding oracle using the results of an exact all-pairs shortest-path computation. A CPD for example can be described as a highly compressed set of tables that store the first move on the optimal path between any pair of nodes in a graph. Through a simple process of recursive look-ups CPDs can extract any optimal path in near-linear-time and without employing any state-space search. SILC and CPDs are among the fastest pathfinding techniques today but because they require substantial amounts of memory and pre-computation they are not applicable in many popular pathfinding settings, including games.

2.7 Symmetry Breaking

Symmetry is a naturally occurring phenomenon that arises whenever we are forced to enumerate permutations of elements in a set. Symmetry forces search algorithms to waste time and prevents real progress toward the goal. In the context of graph search, two kinds of symmetries can arise: *state symmetries*, which are equivalences between the individual nodes of a graph, and *path symmetries*, which are equivalences between ordered sequences of nodes. For an example of each consider the GRIPPER domain: a simple but challenging series of problems from the area of AI Planning. A typical instance involves a robot tasked with moving a series of identical balls between two adjacent rooms. The robot has two arms (the eponymous grippers for which the problem is named) and can move freely between the rooms but is restricted to manipulating only one ball at a time. To solve this problem a typical systematic search algorithm will enumerate every permutation of balls and grippers, despite the fact that such configurations are equivalent. For instance: picking up a ball with the left gripper yields a state equivalent to picking up a ball with the right gripper. Similarly, the sequence of actions necessary to move a single ball from one room to another is independent and interchangeable with the sequence of actions that moves any other ball; thus it doesn't matter in which order the balls are selected as every interleaving of the set of all such action sequences leads to the same state.

2.7.1 Approaches for Breaking State Symmetries

A range of methods for reducing state symmetries have been described in the literature of search. Though different in detail each approach involves a preprocessing step that takes a given problem description and converts it into an equivalent form that is symmetry free or symmetry reduced.

One recurring idea for breaking symmetries involves the application of concepts from group theory: two nodes in a graph are considered symmetric if they can both be mapped to the same symmetry group. Emerson and Sistla [1996] describe such an algorithm in the context of Model Checking. Their method proceeds by mapping every node in a problem graph to a canonical representative from its associated symmetry group. The result of this process is an equivalent *quotient graph* that consists only of these canonical representatives. Being symmetry free this graph is much smaller than the original and consequently much easier to search. Other algorithms using similar principles have also been proposed; for example in Constraint Programming [Crawford et al., 1996; Roney-Dougal et al., 2004] and AI Planning [Pochter et al., 2011]. In each case the authors report a significant improvement in search performance for a variety of practical problems. Unfortunately this approach to symmetry breaking reduces to repeatedly testing for graph isomorphism; a difficult problem for which no general polytime algorithm is known.

A faster and more practical strategy for dealing with state symmetries involves considering only a subset of the permutations in each symmetry group. This is known as *partial symmetry breaking*. The idea is to trade time spent identifying symmetry groups up-front against the risk of considering multiple symmetric states during search. Examples of such techniques include: computing only a subset of the permutations in each symmetry group (e.g. as in [Crawford et al., 1996]), computing only an approximately canonical representative for a given symmetry group (e.g. as in [Pochter et al., 2011]) and looking for *symmetric transitions* rather than symmetric states (e.g. as in [Fox and Long, 1999, 2002; Rintanen, 2003]).

2.7.2 Approaches for Breaking Path Symmetries

Path symmetries occur during search when considering sequences of independent and interchangeable transitions. Applied separately these transitions do not necessarily lead to a symmetric state but given a set of such transitions every possible interleaving always yields the same state. Enumerating such interleavings is pointless: each one involves searching a sub-graph that is isomorphic to all the rest and each one yields a state that is equivalent to all the rest.

Studies that deal with path symmetries can be found across the literature of Artificial Intelligence. In the area of Heuristic Search for example practitioners often study combinatorial puzzles such as Sliding Tile and Rubik's Cube. Both are highly symmetric domains where interleavings of available actions often yield the same state. To improve the efficiency of search algorithms for such problems Taylor and Korf [1993] propose constructing automata that detect redundant interleavings. Their approach is complete and optimal but suffers from two drawbacks: (i) it does not

prune any nodes from the search space; only transitions (ii) its effectiveness depends on a preprocessing step that samples the search space; if the sample is not representative of the search space more broadly then the pruning power of the automaton is reduced. Recent work has extended this approach to the more general problem of move pruning in AI Planning Burch and Holte [2012].

Automaton-based pruning can be regarded as a type of canonical ordering over sequences of transitions in the search space. A similar idea, which appears in the area of AI Planning, is Commutativity Pruning [Haslum and Geffner, 2000]. This algorithm deals with interleavings of independent and interchangeable actions by only considering actions sequences that respect a given lexicographic order. Like Taylor and Korf [1993]’s work this approach focuses on pruning redundant sequences of transitions. It has the same advantages (complete, optimal) but unlike that work it requires no preprocessing.

A rich source for works dealing with path symmetries is the area of Model Checking where practitioners reason about the operation of concurrent transition systems. In this context the state of a system (sometimes called its *configuration*) is defined in terms of its execution history; an ordered sequence of *events* where each event represents a concurrent computation undertaken by one or more independent components of the system. The objective of model checking is to automatically determine whether specifications of such systems guarantee certain properties; e.g. that certain events will occur or that no deadlocks exist. Though verification is NP-hard in general, practical algorithms are further hampered by the fact that many of the events which can occur at some point in time are independent from and interchangeable with many other events.

A Mazurkiewicz Trace [Mazurkiewicz, 1986] (or simply a *trace*) is a conceptual approach to concurrent model checking that avoids isomorphic sub-graphs. Trace semantics define an equivalence class $[h]$ for execution histories of a concurrent system. Each history in a given equivalence class is identical to all the others save for the order in which interchangeable and independent computations occur. A number of different strategies for searching in the space of Mazurkiewicz Traces have been proposed [Esparza and Heljanko, 2008]. All of them define a precedence relation \prec over histories. This is combined with a strategy such as depth-first search in order to explore the space of all possible traces. The idea is that only one interleaving from each equivalence class is ever considered: the one that is \prec -minimal. Provided that \prec is well-founded and preserved by extensions it can be shown that this strategy is both sound and complete. Moreover, if \prec is a total order then the maximum length of any history is linearly bounded.

A more general approach, which is similar to but which does not instantiate Mazurkiewicz Trace semantics, is Persistent Sets. This is a family of related algorithms [Overman and Crocker, 1982; Valmari, 1989; Godefroid and Wolper, 1991; Godefroid, 1996; Wehrle and Helmert, 2012; Alkhazraji et al., 2012] that all proceed in a similar manner: they construct in each state s a set of enabled transitions T such that every non-empty sequence of transitions composed using only elements of T is independent from (i.e. can be interleaved with) every enabled transition outside T .

During search only transitions found in T are followed; all other transitions enabled in the current state are ignored. Like Mazurkiewicz Traces, Persistent Sets are both sound and complete. They differ in two important ways: (i) Persistent Sets do not require the specification of any precedence relation; (ii) Persistent Sets do not guarantee that every state that is reached during search has a lex-minimal or even unique execution history. This latter property means that a search algorithm employing Persistent Sets may explore several interleavings of the same set of transitions instead of just one.

A similar idea, which addresses some of the drawbacks associated with Persistent Sets, is Sleep Sets [Godefroid and Wolper, 1993; Godefroid, 1996]. A Sleep Set is a set of transitions which are enabled in a state s but which will not be executed. Sleep Sets are passed from one state to another during search. The idea is to avoid executing any transitions which are independent and interchangeable with the last transition taken in order to reach the current state. For example: suppose a state s' is reached by applying transition t in a preceding state s . The sleep set of s' is the sleep set of s plus all enabled transitions in s that are independent and interchangeable with t and minus all transitions that are not. Sleep Sets are sound and complete but less powerful than Persistent Sets. However, two approaches are orthogonal and can be combined. The result is a pruning strategy that is at least as effective and often more effective than either approach applied in isolation.

2.8 Dominance Detection

In many pathfinding domains, and especially in gridmaps, arbitrary problem instances rarely have only a single optimal solution. These solutions can be symmetric to each other (c.f. Section 2.7) or they can simply be equivalent in terms of cost. Pathfinding instances that have many equivalent solutions are more difficult to solve; even for efficient search algorithms such as A^* . In such cases, before even one optimal path can be found, A^* must usually generate (and often expand) every node on every other equivalent optimal path. This is wasteful and prevents A^* from making real progress toward the target. A number of works can be found in the literature of Heuristic Search that identify and eliminate equivalent paths. Such algorithms are known as *graph pruning* or *dominance detection* techniques. We briefly outline them here.

Definition 1 A path π dominates another path π' if $\text{cost}(\pi) \leq \text{cost}(\pi')$. If $\text{cost}(\pi) < \text{cost}(\pi')$ then π is strictly dominant.

The Dead-end Heuristic [Björnsson and Halldórsson, 2006] and Swamps [Pochter et al., 2009, 2010] are two graph pruning techniques that can eliminate from consideration many dominated and strictly dominated paths. Both rely on an offline preprocessing phase that decomposes a given map into obstacle-free areas. During online search each of these two algorithms consults the decomposed map in order to identify areas that are relevant for the instance at hand and areas that can be ignored

entirely because exploring them could not improve the cost of any path. Recent work shows that a limited type of Swamp detection can also be performed online however that algorithm prunes fewer nodes than its offline counterpart Sharon et al. [2013].

MSA* [Bolanca, 2009] is another recent pruning technique which is orthogonal to both Swamps and the Dead-end Heuristic. Where those algorithms identify areas that can be ignored during search, MSA* generates, on the fly, macro-edges that facilitate fast travel through areas that do have to be searched. It relies on a preprocessing step that decomposes a given grid map into a series of empty rectangular rooms. The key idea is that each such room can be traversed in a single macro step. MSA* is similar in spirit to Rectangular Symmetry Reduction; an alternative pruning algorithm that we introduce in Chapter 3. While our work can consistently speed up A* search, MSA* is only able to achieve this in some cases. It is also worth noting that MSA* uses a different empty room decomposition method from the one described in our work.

A different technique for pruning the search space is to identify *dead* and *redundant* cells [Sturtevant et al., 2010b]. Developed in the context of learning-based heuristic search, this method speeds up search only after running multiple iterations of an iterative deepening algorithm. During each iteration the algorithm stores the g -value associated with each expanded node and prunes from subsequent iterations all nodes that were found to have zero g -cost-increasing neighbours.

Fast Expansion [Sun et al., 2009] is another related work that speeds up A* pathfinding search. This algorithm detects when a successor of the current node has an f -value that dominates the f -value of the best node on the open list. The idea is to expand such nodes immediately and avoid unnecessary operations on the open list.

2.9 Euclidean Shortest Paths

Many of the optimal pathfinding algorithms we have discussed to now are optimal only in a graph-theoretic sense. In this section we review the state of the art for computing Euclidean shortest paths: i.e. shortest paths with respect to the underlying operating environment of an agent and not a graph-based discretisation. We discuss two popular formulations for this problem: the Any-angle Pathfinding Problem, which asks for a Euclidean shortest path given a set of obstacles drawn from a grid, and the more general Euclidean Shortest Path Problem, which asks for a Euclidean shortest path given an arbitrary set of obstacles.

2.9.1 Any-angle Pathfinding Algorithms

The Any-angle Pathfinding Problem appears in robotics and computer video games. It involves finding a shortest path between an arbitrary pair of points on a two-dimensional grid map but asks that movement along the path is not artificially constrained to the points of the grid.

Within the game development community a simple and popular solution to the Any-angle Pathfinding Problem is *string pulling* [Pinter, 2001; Botea et al., 2004]. The idea is to compute a grid-optimal path in the first instance and smooth the result as part of a post-processing step that improves both its cost and aesthetic appeal. String pulling has two disadvantages: (i) it requires an additional computation beyond just finding a path (ii) it can only yield approximately shortest paths.

A number of algorithms improve on string-pulling by integrating post-processing into node expansion during search. Field D* [Ferguson and Stentz, 2005] is one such algorithm. It uses a simple approach based on linear interpolation that smooths paths one grid cell at a time. Theta* [Nash et al., 2007] and its variants [Nash et al., 2009, 2010; Muñoz and Rodríguez-Moreno, 2012] are similar ideas that can smooth much longer path segments in a single operation. This algorithm relies on line-of-sight checks from each successor of a node to its grandparent. A successful check introduces a new shortcut that bypasses the current node. Another approach, Block A* [Yap et al., 2011], avoids line-of-sight checks entirely by precomputing a database of exact costs between pairs of points in a localised area. Each of Field D*, Theta* and Block A* improve on string pulling in terms of solution quality and, in many cases, also in terms of running time. A main disadvantage of all these algorithms is that they provide no optimality guarantees.

Accelerated A* [Šišlák et al., 2009a,b] is a recent technique for the Any-angle Pathfinding Problem which is conjectured to be optimal. Similar in spirit to Theta* this algorithm differs in two ways: (i) it performs line-of-sight checks to a set of previously expanded nodes (not just a single ancestor); (ii) it implements a block-based expansion strategy that allows it to consider sets of nodes at a time. A primary advantage of Accelerated A* is that it uses less memory than competing algorithms (e.g. Theta*). Another advantage is that computed solutions are shorter than those produced by competing algorithms. In a range of grid-based domains the authors show that Accelerated A* always finds the optimal path. Unfortunately no theoretical guarantees are provided. A main disadvantage of Accelerated A* is its running time. During each node expansion there can be a large number of line-of-sight checks between the current node and a set of potential alternative parent nodes. For large problems with long optimal solutions this set can comprise most nodes appearing on the closed list.

2.9.2 Euclidean Shortest Path Algorithms

The Euclidean Shortest Path Problem is a well known and well researched topic in the areas of Computational Geometry and Computer Graphics. It can be seen as a generalisation of the Any-angle Pathfinding Problem. It asks for a shortest path in a plane but does not impose any restrictions on obstacle shape or obstacle placement (cf. grid aligned polygons made up of unit squares).

Visibility Graphs [Lozano-Pérez and Wesley, 1979] (discussed in Section 2.2) are a well-known and popular technique for solving the Euclidean Shortest Path Problem. Searching in such graphs requires $O(n^2 \log_2 n)$ time. There are two main

disadvantages: (i) computing the graph requires an offline preprocessing step; (ii) the graph is static and must be repaired if the environment changes; (iii) storing the graph can introduce an $O(n^2)$ memory overhead (each node in the graph can be adjacent to every other node). Tangent Graphs [Liu and Arimoto, 1992] are a particularly efficient variant but their space requirements remain worst-case quadratic.

Other exact approaches are based on the Continuous Dijkstra [Mitchell et al., 1987; Mitchell, 1997] paradigm. The most efficient of these algorithms [Hershberger and Suri, 1999] involves a precomputation requiring $O(n \log_2 n)$ space and $O(n \log_2 n)$ time. The result is a Shortest Path Map (Section 2.2); a planar subdivision of the environment that can be used to find a Euclidean shortest path in just $O(\log_2 n)$; but only for queries originating at a fixed source. Like Visibility Graphs, this approach also introduces additional memory overheads (storing the subdivision) and the preprocessing step must be re-executed each time the environment or the start location changes.

2.10 Summary and Discussion

Single-agent Pathfinding is a common topic in many areas of Computer Science. It is studied in the context of Algorithmics, Artificial Intelligence, Computational Geometry, Computer Graphics, Game Development and Robotics. A common formulation involves an agent navigating amidst obstacles in a two or three dimensional operating environment. Researchers often simplify the problem by discretising the environment into a fixed set of connected points called a search graph. Such discretisation approaches include:

- Grid Maps.
- Navigation Meshes.
- Roadmaps.
- Shortest Path Maps.
- Visibility Graphs.

There exist several popular strategies for finding a path in a search graph: (i) Blind Search; (ii) Local Search; (iii) Informed Search. Each has its own advantages and disadvantages. Blind Search methods, for example, unfold a search graph into a tree and explore the nodes in a fixed order. Under certain conditions these algorithms are guaranteed to find a path (including the optimal path) though they are typically not very efficient. In some cases, for example in exponential domains, blind search methods require an amount of time, or space, that is exponential in the length of the returned solution. Local Search algorithms, by comparison, use simple rules-of-thumb to find a path. They are fast (in the sense that they can consider many nodes very quickly) and they are memory efficient but they are not guaranteed to find the optimal path or indeed any path. Informed Search algorithms, by comparison, are

systematic pathfinding strategies that employ heuristic lower-bound functions that estimate the cost-to-go until the target is reached. Such algorithms rank nodes by merit and always explore the most promising node first. Informed Search methods are popular because they are complete, optimal and optimally efficient.

Despite very attractive theoretical characteristics there exist many domains of practical interest where even Informed Search strategies cannot find optimal paths within a specified time limit or using only a fixed amount of memory. A variety of techniques have been developed to improve the efficiency of search in these cases. These efforts can be broadly categorised as follows:

- Spatial Abstraction.
- Improved Heuristics.
- Symmetry Breaking Techniques.
- Dominance Detection Methods.

Within the Artificial Intelligence community pathfinding with spatial abstraction usually involves creating small approximations of a given input graph. Though fast and memory efficient, such approaches do not find optimal paths. A related strategy involves improving the accuracy of heuristic functions that guide search. Such methods usually guarantee optimality but often at the cost of substantive memory overheads.

Within the Algorithmics community a different line of research has led to the development of specialised abstraction hierarchies that, unlike their counterparts in AI, can preserve solution optimality. Developed for routing on road networks, these approaches are very fast but suffer from two drawbacks: (i) they introduce substantive memory overheads (compared to the amount of space needed to store the map) and (ii) they exploit properties specific to road-maps (e.g. there are typically very few optimal paths between arbitrary pairs of nodes on a road map) which do not necessarily hold in other settings.

In some domains of practical interest, such as grid maps, it has been suggested that optimal pathfinding is made difficult by the existence of many equivalent optimal paths. To deal with such cases researchers in Artificial Intelligence have developed a variety of Dominance Detection techniques. Typically preprocessing-based, such algorithms focus on the identification of areas on a map that do not need to be considered during search. Ignoring these areas can speed up informed search strategies such as A* by many factors; albeit at the cost of some memory overhead.

Equivalences in search domains can be due to paths having direct symmetries in the environment (i.e. automorphisms between states) or they can be the result of interleaving equivalent and interchangeable actions (i.e. isomorphisms between sequences of states). A large body of work dealing with both types of symmetries can be found in the literature of AI Planning, Constraint Programming and Model Checking. Such works are usually developed in the context of transitions systems

having compact representations and a small set of applicable operators. By comparison, pathfinding domains have large explicit representations and transitions that are state-specific. Thus there is little work applying such ideas to pathfinding.

Until now our discussion has focused on graph-theoretic approaches that rely on discretisation to simplify the operating environment. In the areas of Computational Geometry and Computer Graphics there exists a line of research devoted to solving the Euclidean Shortest Path Problem. This is a variation on pathfinding that focuses on exact solutions to navigation problems rather than graph-based solutions. A variety of efficient techniques exist in this context but all are preprocessing based and require memory overheads. Among the most successful of these, Continuous Dijkstra, requires a fixed starting state; a very strong and often unreasonable constraint. Any-Angle Pathfinding an instance of the Euclidean Shortest Path Problem which is restricted to grid-based environments. The problem appears in the context of Computer Games and Robotics. A number of methods from AI and Game Development exist that can solve the problem but all are suboptimal. One recent algorithm, Accelerated A*, is conjectured to be optimal but no proof is given.

Rectangular Symmetry Reduction

In this chapter we present Rectangular Symmetry Reduction (RSR): a graph pruning algorithm for undirected uniform-cost grid maps which is fast, memory efficient, optimality preserving and which can, in some cases, eliminate entirely the need to search. The central idea that we will explore involves the identification and elimination of *path symmetries* from the search space. Symmetry appears in many domains; for example in planning [Fox and Long, 1999], constraint programming [Walsh, 2007] and combinatorial optimisation [Fukunaga, 2008]. Unless it is handled properly, symmetry almost always increases the size of the search space and forces search algorithms to waste time considering variations of already known solutions.

To deal with path symmetries RSR decomposes an arbitrary uniform-cost grid map into a set of empty rectangles, removes from each such rectangle all interior nodes and retains only nodes from along the perimeter. A series of macro edges are then added between selected pairs of perimeter nodes to facilitate provably optimal traversal through each rectangle. We first develop RSR in the context of 4-connected grid maps; a common setting in video games and one which appears often in the AI literature. We then generalise RSR to 8-connected grid maps where the increase in branching factor makes effective symmetry elimination more challenging. We also develop two new pruning strategies which can significantly reduce the number of nodes that need to be explored during search. The first enhancement is applied offline and allows us to discard, in many cases, nodes from the perimeter of an empty rectangle. The second enhancement is applied online and allows us to speed up node expansion by selectively evaluating either all neighbours associated with a node or only a small subset.

We evaluate RSR on a range of uniform-cost grid maps from the academic literature and find that it can improve the running time of A* search by several factors. In certain cases the speedup can be as much as one order of magnitude. We then compare RSR to Swamps [Pochter et al., 2010], a recent state-of-the-art pruning technique. We find that the two algorithms have complementary strengths and that they could be easily combined. We also identify a range of benchmark problems on which RSR dominates convincingly.

The contributions described in this chapter have been presented previously in [Harabor and Botea, 2010; Harabor et al., 2011; Harabor, 2011].

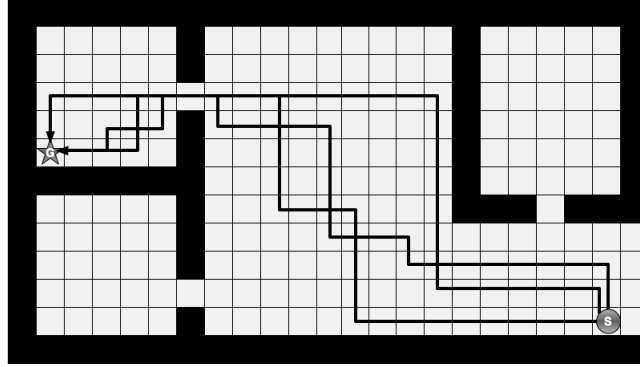


Figure 3.1: A pathfinding instance on a 4-connected grid map. There are many optimal paths between the start and target location but all are permutations of one another. We highlight three such paths.

3.1 Path Symmetries

Pathfinding in modern video games often involves exploring environments such as cities, sewers or dungeons. Often represented using a grid map, these locales tend to be topographically simple (e.g. empty rooms, connected by corridors) but can be surprisingly difficult to search. This is because between any pair of locations in a grid map there usually exist many possible paths. Sometimes these paths represent alternative ways of reaching one location from the other. More often they are symmetric in the sense that the only difference between them is the order in which individual moves appear. Definition 2 introduces the concept of symmetry more formally. As an illustrative example consider Figure 3.1. Suppose we run an optimal algorithm such as A* to solve this problem. Each time we expand a node n we need to expand all nodes on any shortest path to n , including all permutations each of which is also a shortest path to n . Depending on how we break ties, we might expand a majority of all nodes before reaching the goal.

Despite a wealth of research on the topic of symmetry breaking (see Chapter 2.7) few works deal with the problem in the context of pathfinding search. This lack of attention may stem from definitional issues. For example, researchers commonly define a path as a sequence of connected edges which together represent a walk in a search graph. The problem is that edges are strictly ordered whereas the actions they represent could be pairwise commutative [Haslum and Geffner, 2000]; i.e. we could apply the actions in any order and still reach the same state. To help identify such situations we introduce a slightly different notion of a path:

Definition 2 A grid path $\pi = \langle \vec{v}_1, \dots, \vec{v}_k \rangle$ is an ordered sequence of vectors, where each vector \vec{v}_i represents a single step from one node of the grid to an adjacent neighbouring node.

Put more simply Definition 2 says that a path is just a sequence of single steps, or moves, and that each move is a direction on the grid such as North, South, East

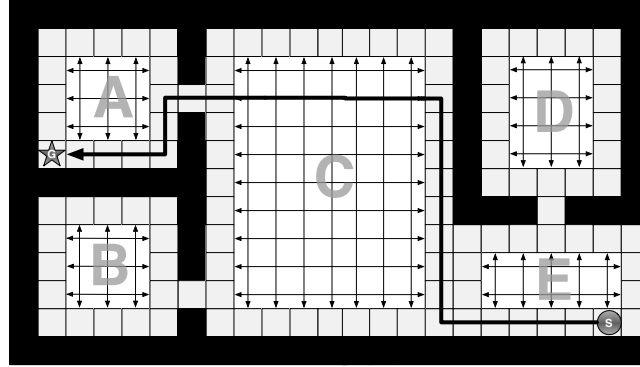


Figure 3.2: RSR decomposes a grid map into a set of empty rectangles. Nodes interior to each rectangle are pruned leaving a set of perimeter nodes. Macro edges are then added to facilitate optimal travel from one side of the perimeter to the other.

or West. Using Definition 2 we can now distinguish between paths that are merely equivalent (i.e. of the same cost) and those that are symmetric:

Definition 3 Two grid paths π and π' are symmetric iff: (i) they share the same start and end node (ii) they have the same cost and (iii) one can be derived from the other by swapping the order of the constituent vectors.

3.2 Symmetry Breaking In 4-Connected Grid Maps

As a first step, we will describe a simple technique for breaking path symmetries in the context of 4-connected grid maps. This is a domain which appears regularly in the academic literature [Yap, 2002; Wang and Botea, 2008; Pochter et al., 2010] and is often found in the pathfinding systems of modern computer games. Some examples include Square Enix’s *Heroes of Mana* (released in 2007 for the Nintendo DS), Astraware’s *My Little Tank* (2008, iPhone) and Atari’s *Dragon Ball Z: Legacy of Goku* (2002, Gameboy Advance). We propose the following strategy:

1. Decompose the grid map into a set of empty rectangles, each of which contains no obstacles.
2. Prune all nodes from the interior but not the perimeter of each rectangle.
3. Add a series of *macro edges* that connect each node on the perimeter of a rectangle with a node from the directly opposite side¹. The cost of each edge is equal to the Manhattan distance² between its two endpoints.

The decomposition of maps into empty rectangles involves a greedy flood-filling strategy which we discuss in Section 3.2.3. Note however that rectangle sizes are not

¹We can avoid storing macro edges by generating them on-the-fly during search.

²Defined in Chapter 2.6

fixed and can vary across a map depending on the placement of obstacles. Trivial rectangles which contain no interior nodes (for example those with a width w or height $h \leq 2$) are left unmodified by steps 2 and 3. Figure 3.2 shows an example of this process. For each non-trivial rectangle we prune $(w - 2) \times (h - 2)$ interior nodes and, in the process, eliminate a large number of symmetric paths between nodes on the perimeter. We claim that this approach preserves optimality when traversing across any arbitrary rectangle.

Lemma 1 *Let R be an arbitrary rectangle that is free of obstacles and m, n be two locations on its perimeter. Then m and n can be connected optimally through a path that mentions only nodes on the perimeter of R and possibly involves a macro edge.*

Proof:

There are two distinct cases to consider. Case 1 is when m and n are placed on the same side of the perimeter, or on two orthogonal sides. To obtain an optimal path we can simply travel along the perimeter from m to n . Case 2 is when m and n are placed on opposite sides of the perimeter. To obtain an optimal path we can simply follow the macro edge at m and navigate directly to a node m' located on the same side of the perimeter as n . Then, go from m' to n along the perimeter. The resultant path is optimal as its cost is equal to the Manhattan distance between m and n . \square

A direct corollary to Lemma 1 is that we can prune from consideration all nodes from the interior of R and limit ourselves to only searching nodes appearing along its perimeter. The only remaining consideration is how to deal with interior nodes that happen to be the start or goal location for the search at hand. We address this case in the following section.

3.2.1 Online Insertion

Often an interior node pruned as a result of offline symmetry elimination is required as a start or goal location for an agent. We handle such cases by inserting back into the graph interior start and goal nodes for the duration of the search. We use the following procedure (highlighted in Figure 3.3):

1. If the start and goal are in the same rectangle no insertion is required. Since it is guaranteed that there are no obstacles between the two locations, an optimal path is trivially available. This case will be ignored in the rest of our discussion.
2. If the start and goal are not in the same rectangle, connect each of them to the closest neighbours on each side of the perimeter of the empty rectangle.

We claim that this procedure retains optimality when searching from the start (or goal) location to all nodes on the perimeter of its rectangle.

Lemma 2 *Let R be an empty rectangular rectangle. For any nodes m, n , with m a re-inserted interior node and n a node on the perimeter, it is always possible to find an optimal cost path which mentions no interior nodes except for m .*

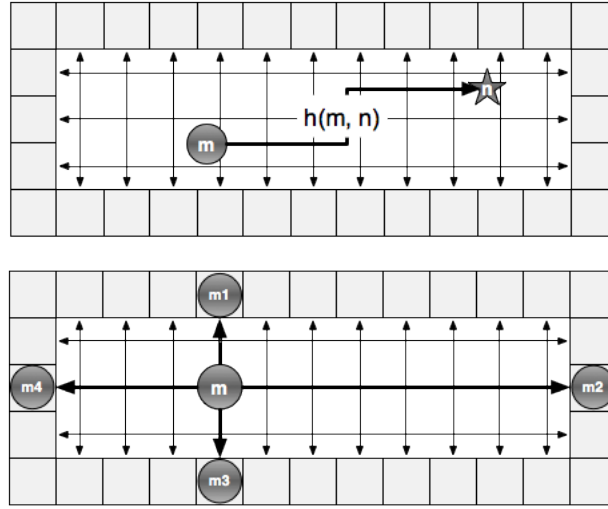


Figure 3.3: (Top) When m and n are in the empty rectangle no insertion (or search) is necessary. (Bottom) m is a previously pruned interior node. We insert m into the graph and connect it to neighbours on each side of the empty rectangle.

Proof: We insert m into the graph and connect it to m'_1, m'_2, m'_3, m'_4 , the closest neighbours on each side of the perimeter. The weight of each edge incident with m is equal to the straight-line distance between m and each m'_i . To find an optimal path to n we travel from m to the node m'_i which is on the same side as n on the perimeter. From there we travel along the perimeter of R until we reach n . \square

Once the search has finished we remove the start and goal from the graph. The time required in each case (insertion and deletion) is constant.

3.2.2 Optimality

We claim that A^* , when applied to a 4-connected grid map pruned by RSR, will always return an optimal solution if one exists.

Theorem 1 *For every optimal cost path $\pi^*(s, g)$ in a 4-connected grid map there exists an equivalent cost path in the pruned version of the grid map.*

Proof: Follows from Lemma 1 and Lemma 2. For every optimal cost segment of $\pi^*(s, g)$ which traverses through an empty rectangle, from a perimeter node m to a perimeter node n , there is an equivalent segment which mentions only nodes on the perimeter of that rectangle (and possibly one macro-step). \square

A direct corollary of Theorem 1 is that optimal solutions pruned by our symmetry reduction can be easily reconstructed; for example to avoid unnatural looking paths where agents seem to hug walls. Consider a path fragment between m and n , two nodes on the perimeter of an empty rectangle. Assume, without any generality loss,

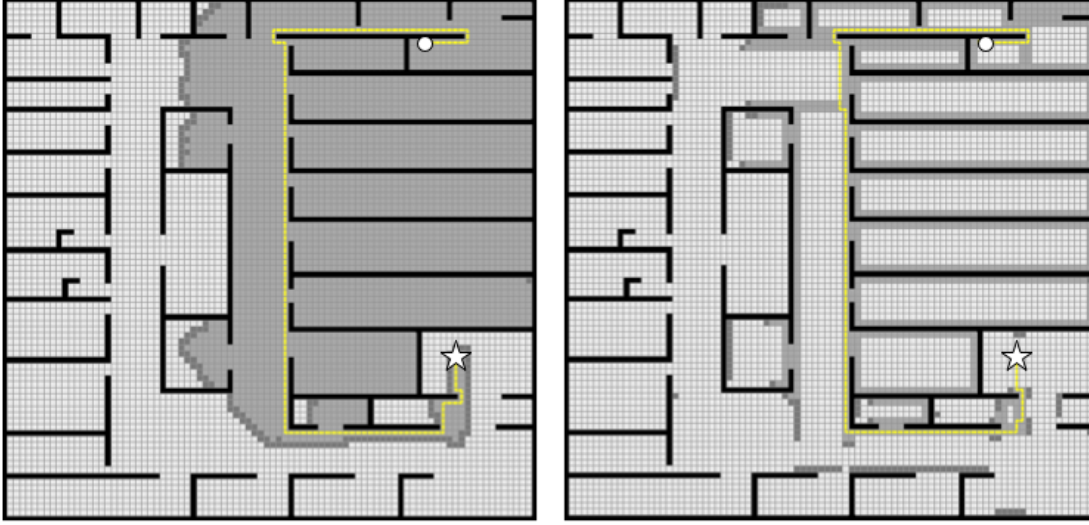


Figure 3.4: (Left) A* solving a problem on an unmodified (86×88) grid map. Expanded nodes are marked dark grey. (Right) A* solving the same problem using our modified grid map. The algorithm only considers nodes along the perimeter of the identified rectangles.

that the path fragment contains r moves to the right and u moves upwards. All optimal path fragments between m and n can be obtained by interleaving r moves to the right and u moves upwards in any order (e.g right-right-up, right-up-right, up-right-right).

In Figure 3.4 we highlight the effectiveness of our symmetry breaking technique using a map, due to Vadim Bulitko, that has characteristics similar to what one might expect in a modern role-playing game³; i.e. there are many rooms with entrances and corridors connecting them. A* running on the original grid map expands almost half the nodes in the state space of the shown example. We then apply our technique to eliminate symmetries and re-run A*. This time A* expands less than 15% of all nodes (more than a three-fold improvement) and returns an optimal solution 3 times faster.

3.2.3 Identifying Empty Rooms

In this section we give a simple but effective flood-fill-based algorithm for decomposing a grid map into empty rectangles. We will try to build large rectangles before small ones and prefer rectangles which contain as many interior nodes as possible:

1. For each traversable tile t , build a maximal size empty rectangle which has t as its upper left corner. Each such rectangle should contain only traversable tiles which have not already been assigned to a rectangle.
2. Using a Max-Heap, sort the list of traversable tiles using the number of interior nodes in the rectangle of each t as its priority.

³In fact, many computer game maps tend to be somewhat bigger than our example but for demonstration purposes it is sufficient.

3. Take from the heap the tile t with highest priority which has not already been assigned to a rectangle.
4. Verify the priority of t ⁴ by building another maximal size empty rectangle (as per Step 1) which has t as its upper left corner and contains no obstacles or tiles already assigned to another rectangle.
5. If the number of interior nodes in the new rectangle is equal to the priority of t we say that the rectangle forms a rectangle and add it to our decomposition. Otherwise, we update the priority of t with the number of interior nodes contained in the new rectangle.
6. Repeat Steps 3 to 5 until the heap is empty and all nodes have been assigned to a rectangle in the decomposition.

The construction of empty rectangles is similar to the computation of *clearance values* in [Harabor and Botea, 2008]. In that work the objective is to calculate the amount of traversable space at any given location on the map. This is achieved by constructing maximum sized squares that originate at each traversable tile on the map. Our rectangle identification procedure can be seen as a variation of this method in which we extend each such square into a maximal size rectangle.

As we will see the performance of A* on our modified grid map is closely related to the total number of nodes we are able to prune. Thus, identifying large rectangles is critical. Although our decomposition technique is not optimal for this purpose it is simple to understand and implement and produces good results in practice.

3.3 Symmetry Breaking In 8-Connected Grid Maps

We now describe a variation of the rectangle-based symmetry breaking technique from Section 3.2 which is applicable to 8-connected grid maps. The generalisation is more challenging than it might look at a first glance. On a 4-connected map no perimeter node requires more than one macro-edge (to the closest node on the opposite side of the perimeter) to retain optimality. Thus, it is easy to maintain a low branching factor. In the 8-connected case many more macro edges are needed to preserve optimality.

A straightforward generalisation approach would be to add a macro-edge between any two nodes on the perimeter of a rectangle. We will call the sub-graph resulting from such an operation a *perimeter clique*. Although the perimeter clique approach guarantees optimality, it has the disadvantage of creating a large branching factor and slowing down search (the number of necessary macro edges for a rectangle is quadratic in the number of perimeter nodes). We will therefore introduce an alternative strategy, that creates fewer macro-edges, by defining a *dominance* relation between macro-edges.

⁴Some of the tiles in the associated rectangle may have been assigned to another rectangle since we added t to the heap.

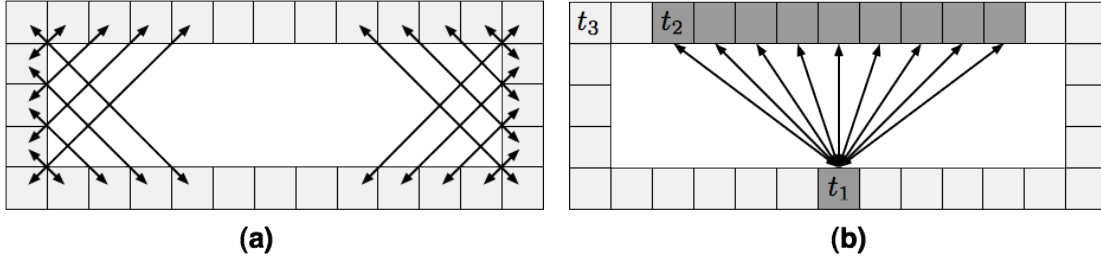


Figure 3.5: (Left) Macro edges between nodes on orthogonal sides of an empty rectangle. (Right) Each node on the perimeter is connected to a set of nodes on the opposite side.

Definition 4 A macro-edge connecting two arbitrary nodes t_1 and t_2 in a perimeter clique is non-dominated if all other paths between t_1 and t_2 in the perimeter clique have a cost strictly larger than the macro-edge at hand.

By starting with a perimeter clique and applying Definition 4 it is easy to see that the set of non-dominated macro-edges are precisely the ones that we identify below. There are three cases to discuss: connections between nodes on the same rectangle side, connections between orthogonal rectangle sides, and connections between opposite rectangle sides. In the discussion that follows note that the cost of each added macro-edge is equal to the heuristic distance between its two endpoints – as measured using h_{OD} or octile distance (we describe this heuristic in Chapter 2.6).

The first case is simple: adjacent nodes on the same perimeter side are connected just as in the original grid. In the second case, two nodes on orthogonal sides of the perimeter of a rectangle R are connected *iff* the shortest path between them is a diagonal (45-degree) line; this is illustrated in Figure 3.5 (left). Notice that in both cases we introduce no more than two macro edges per node. In the third case, we generate for each perimeter node a “fan” of neighbours from the opposite side R . Figure 3.5 (right), illustrates this idea. Starting from a node such as t_1 we step to the closest neighbour from the opposite side of R and extend the fan by progressing away from the middle in both directions adding each node we encounter. The last node on either side of the fan is placed diagonally, at 45 degrees, from t_1 (such as t_2) or located in the corner of the perimeter (whichever we encounter first). There is no need to add further nodes, such as t_3 , as these can be reached optimally from t_1 via the path $\{t_1, t_2, \dots, t_3\}$.

In the rest of this chapter we will use macro-edge to refer to a non-dominated macro-edge. We show next that the non-dominated macro-edges computed using our strategy are both necessary and sufficient to ensure optimal traversal between any two perimeter nodes.

Proposition 1 All non-dominated macro-edges are necessary to ensure optimal paths in a perimeter clique.

Proof: By definition, a non-dominated macro-edge e is the only way to travel optimally between its end nodes in a perimeter clique. Therefore, dropping e would

result in losing path optimality in the perimeter clique. \square

Lemma 3 *Let R be an empty rectangle in an 8-connected grid map. Let m and n be two perimeter locations. Then, m and n can be connected optimally through a path that contains only non-dominated macro-edges.*

Proof: We split the proof over the 3 cases discussed earlier: 1) m and n are on the same side of the perimeter; 2) m and n are on orthogonal sides of the perimeter; and 3) m and n are on opposite sides of the perimeter.

In the first case we can simply walk along the perimeter from m to n ; the optimality of this path is immediate. In the second and third case we argue as follows: the two nodes can be connected through an optimal path that has one diagonal macro-edge (either at the beginning of the path or at the end) and zero or more straight macro-edges. See again the example of travelling from t_1 to t_3 in Figure 3.5 (Right). \square

Node Insertion: Sometimes a node from the interior of an empty rectangle is required as a start or goal location for an agent. To handle such situations we give an online procedure that temporarily re-inserts nodes back into the map for the duration of a search. It proceeds as follows: If the start and goal are interior nodes in the same room no insertion is necessary; an optimal path is trivially available. On the other hand, if the start and goal are not in the same rectangle, add four “fans” (collections) of macro edges. Each fan connects the start (goal) node to a set of nodes on one side of the rectangle’s perimeter. Fans are built as shown earlier.

Given the simple geometry of rectangles, it is possible to identify in constant time the set of nodes which the start or goal must be connected to. Further, these neighbours could be generated on the fly.

Lemma 4 *Let R be an empty rectangle in an 8-connected grid map. For any nodes m , n , with m a re-inserted interior node and n a node on the perimeter, it is always possible to find an optimal cost path which mentions no interior nodes except for m .*

Proof: Our re-insertion procedure connects the start or goal to a set of nodes on each side of R . The procedure in each case is the same as the one given when connecting two nodes on opposite sides of R . To prove optimality we can simply run the argument given for Step 3 of Lemma 3 for each node on the perimeter of R , in each case substituting m for the newly inserted node. \square

We claim that eliminating symmetries as outlined earlier preserves the completeness and the solution optimality:

Theorem 2 *For every optimal path π on an original grid, there exists an optimal path π' on the modified graph with the property that π and π' have the same cost.*

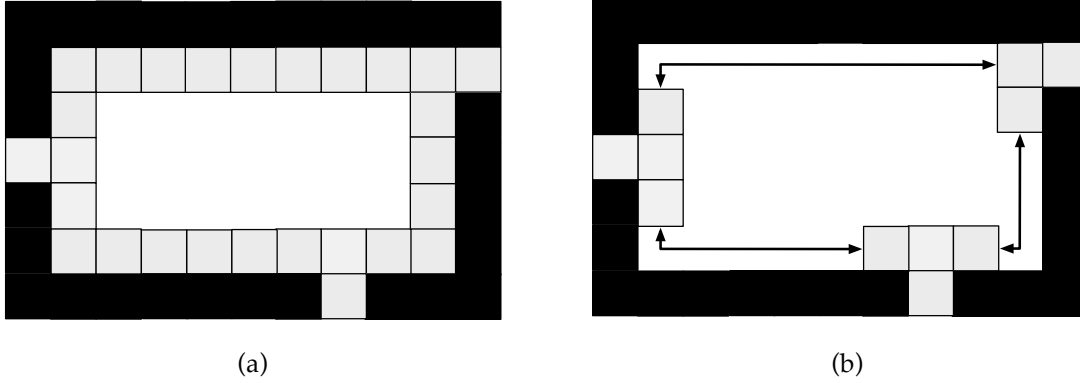


Figure 3.6: (a) From each empty rectangle we prune all nodes which have no neighbours in any adjacent rectangle. (b) Remaining perimeter nodes are then connected directly.

Proof: Consider an optimal path π on the original map and a rectangle R that is crossed by π . Let m and n be the two perimeter points along π . According to Lemma 3, there is a way to connect m and n optimally in the modified graph. Thus, we can replace the original segment $[m \dots n]$ in π with the cost-wise equivalent segment that corresponds to the modified graph. The case when m (or n) is the start or goal node is addressed similarly using Lemma 4. By performing such a path segment replacement for all rectangles intersected by π , we obtain a path π' that satisfies the desired properties. \square

3.4 Further Enhancements

Consider an empty rectangle of width w and height h where $w > h$. After adding all non-dominated macro edges, each node from the perimeter will have between h to $2h - 1$ neighbours from the opposite side of the rectangle, up to 2 neighbours from the same side of the rectangle and up to 5 other neighbours from adjacent rectangles. Such a high branching factor is undesirable as individual node expansion operations take longer.

In this section we study two branching factor reduction methods. The first is an offline technique that prunes nodes from the perimeter of each rectangle. The second is an online pruning strategy which we apply during individual node expansion operations. We discuss both methods in the context of 8-connected grid maps however they are equally applicable to 4-connected maps.

3.4.1 Perimeter Reduction

We observe that in both 4-connected and 8-connected maps there are many cases where nodes on the perimeter of an empty rectangle have no neighbours from any adjacent rectangle. Such nodes are adjacent to obstacles and cannot lead into or out

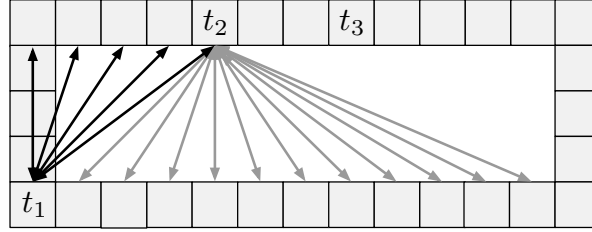


Figure 3.7: Assume that t_1 is the parent of t_2 . When t_2 is expanded, there is no need to generate its secondary neighbors (nodes connected to t_2 by grey edges). These nodes can be reached directly from t_1 by a shorter or equal-cost path that does not involve t_2 .

of any empty rectangle. Figure 3.6(a) shows an example. We propose to prune all such perimeter nodes. To preserve optimality, we will connect the neighbours of each pruned node directly to each other. The weight of each new edge is set appropriately to the octile distance between the two neighbours⁵. Figure 3.6(b) shows the result of this procedure. As we will see such perimeter reduction can have a dramatic effect on the average performance of A* on certain types of maps.

Lemma 5 *Perimeter reduction preserves path optimality.*

Proof: Each time we prune a node from the perimeter we add a new edge between all its neighbours with weight equal to the distance between each pair of neighbours. Thus, if a path exists between a pair of nodes before the application of perimeter reduction it is guaranteed to exist afterward. Further, the cost of this path is unchanged. \square

3.4.2 Online Node Pruning

Given a perimeter node n , let us partition its macro neighbors (connected to n by macro-edges) on the perimeter into *primary neighbours* and *secondary neighbours*. Secondary neighbours are those which are located on the opposite side of the perimeter to as compared to n (excluding any corner nodes). Primary neighbours are all the rest.

When expanding an arbitrary node from the perimeter of a rectangle we observe that it is not necessary to consider any secondary neighbours if both the node and its predecessor belong to the same rectangle. Figure 3.7 shows an example of such a situation; any path to a secondary neighbour is strictly dominated by an alternative path through the predecessor. We apply this observation as follows: During node expansion, determine which rectangle the parent of the current node belongs to. If the current node has no parent or the parent belongs to a different rectangle, then

⁵This operation shares some similarity to Contraction Hierarchies; a well known and very fast algorithm for finding shortest paths in road networks. Like RSR, this work creates an abstract (but multi-level) representation of the graph in which single edges are replaced by equivalent-cost “shortcuts” (i.e macro-edges). Geisberger et al. [2008]

process (i.e., generate) all primary and secondary neighbours. Otherwise, process only primary neighbours.

Lemma 6 *Online node pruning preserves path optimality.*

Proof: Let m be a node on the perimeter of a rectangle. Assume that its parent p belongs to the same rectangle. Let n be a secondary successor of m . Recall that n and m are on opposite sides of the rectangle. We argue below that passing through m cannot possibly improve the best path between p and n . Therefore, there is no need to consider (m, n) macro-edges when m and p belong to the same rectangle.

There are 4 cases when a node m and its parent p belong to the same rectangle. In case 1, p is a re-inserted node from the interior of the rectangle. Obviously, the path segment p, m, n is sub-optimal, as we zigzag from p to m on one side of the rectangle and then to n on the opposite side of the rectangle. In cases 2, 3, and 4, p and m are on opposite sides, on orthogonal sides or on the same side of the rectangle. As in case 1, it is possible to check in each case that taking a detour through m does not improve the shortest path from p to n . \square

3.5 Memory Requirements and Dynamic Environments

Memory Requirements: In the most straightforward implementation, RSR requires storing the id of the parent rectangle for each node in the original grid. This equates to an $O(|V|)$ memory overhead, where V is the set of nodes in the underlying graph. Due to the simple geometric nature of empty rectangles, the set of macro edges associated with each perimeter node can be compactly identified on-the-fly and in constant time by computing the extent of the “fan” of neighbours from the opposite side of the rectangle at hand. Enumerating these nodes during expansion is a linear time operation.

Dynamic Environments: In many application areas, most notably computer games, the assumption of a static environment is sometimes unreasonable. For example: obstacles may appear on the grid or existing obstacles may be destroyed as the game progresses. In such cases the underlying graph representing the world must be updated. If a new obstacle appears, or an existing one is destroyed, we can simply invalidate the affected rectangles and recompute new ones.

3.6 Experimental Setup

We evaluate the performance of RSR on three benchmarks taken from the freely available pathfinding library Hierarchical Open Graph (HOG)⁶: **Adaptive Depth** is a set of 12 maps of size 100×100 in which approximately $\frac{1}{3}$ of each map is divided into rectangular rooms of varying size and a large open area interspersed with large

⁶<http://www.googlecode.com/p/hog2>

randomly placed obstacles. **Baldur’s Gate** is a set of 120 maps taken from BioWare’s popular role-playing game *Baldur’s Gate II: Shadows of Amn*. Often appearing as a standard benchmark in the literature [Björnsson and Halldórsson, 2006; Harabor and Botea, 2010; Pochter et al., 2010] these maps range in size from 50×50 to 320×320 and have a distinctive 45-degree orientation.⁷ **Rooms** is a set of 300 maps of size 256×256 which are divided into symmetric rows of small rectangular areas (7×7), connected by randomly placed entrances. This benchmark has previously appeared in [Sturtevant et al., 2009; Pochter et al., 2010; Goldenberg et al., 2010]. As discussed later, we also use a variant of each benchmark where every map is scaled up by a factor of 3. In effect, our input data contains 864 maps in total, with sizes up to 960×960 .

Since our work is applicable to both 4 and 8 connected grid maps we used two copies of each map: one in which diagonal transitions are allowed and another in which they are not. For each map we generated 100 valid problem instances, checking that every instance could be solved both with and without the use of diagonal transitions. Our test machine had a 2.93GHz Intel Core 2 Duo processor, 4GB RAM and ran OSX 10.6.2. Our implementation of A* is based on one provided in HOG, which we adapted to facilitate our online node pruning enhancement.

3.7 Results

To evaluate RSR we use a generic implementation of A* and discuss performance in terms of search time speedup. That is, the relative improvement to the average time A* needs to solve an instance when running on a pruned vs. unpruned grid. For example, a speedup of 2.0 is twice as fast (higher is better). Note that on approximately 2% of all instances the start and goal are located in the same rectangle and RSR computes the optimal solution without search. We exclude these instances from our results even though RSR solves them in constant time. Their exclusion makes for a clearer comparison between RSR and contemporary approaches from the pathfinding literature.

Pre-processing Times: Table 3.1 presents a summary of average pre-processing times for each of our three (non-scaled) benchmarks. We also give the average number of nodes and edges as an indication of map size. We notice that RSR takes very little time to pre-process all input maps. We did not encounter any that took longer than a second, and most required significantly less than that.

3.7.1 4-Connected Grid Maps

We begin with an evaluation of RSR on 4-connected grid maps. For comparison, we will develop several intermediate variations of the algorithm. These allow us to assess the individual impact of each of the ideas described in this chapter. 4ERR

⁷Such an orientation makes effective symmetry breaking using RSR more challenging but we include the benchmark nonetheless for a more thorough evaluation.

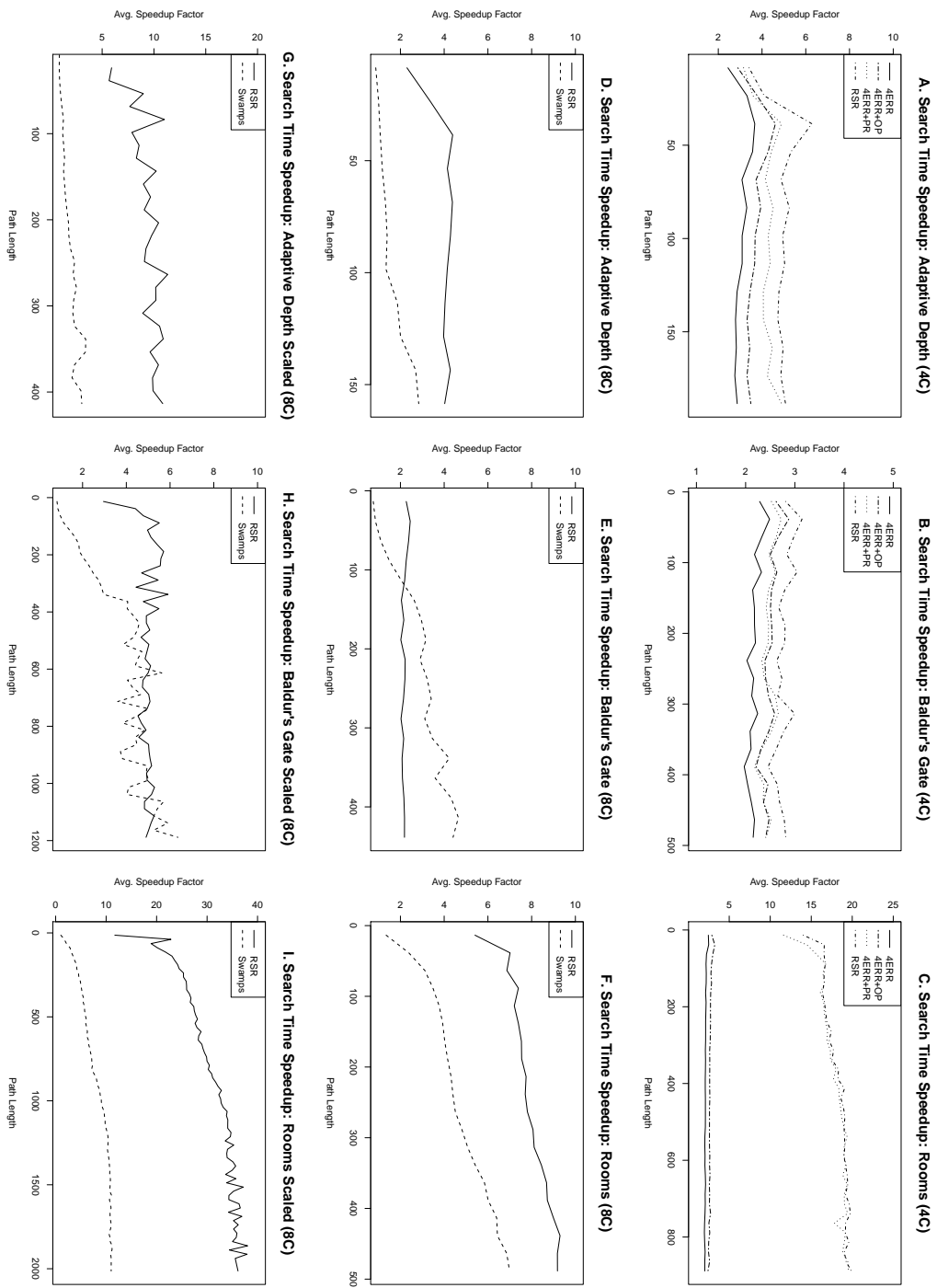


Figure 3.8: Average search-time performance. Results are given in terms of relative improvement vs. A* (i.e. speedup).

Benchmark	Avg. Nodes	Avg. Edges	Preproc RSR	Preproc Swamps
Adaptive Depth	8765	32773	0.10	5.06
Baldur’s Gate	4507	16557	0.65	3.15
Rooms	51437	166417	0.39	16.9

Table 3.1: Input map size and average pre-processing times (seconds).

(4-Connected Empty Rectangular Rooms) refers to the simple symmetry breaking approach from Section 3.2. 4ERR+OP is the combination of 4ERR with online node pruning only and 4ERR+PR is the combination of 4ERR with perimeter reduction only. Figure 3.8 (A to C) presents the main result of the chapter.

Note that RSR (\equiv ERR+OP+PR) shows a convincing speed improvement over 4ERR and all its variants across all input maps. This allows us to conclude that that RSR is the better choice on 4-connected maps. When analysing the impact of each enhancement, we note that 4ERR+PR yields the biggest improvement on all three benchmarks, speeding up A* by up to 20 times. 4ERR+OP compares well with 4ERR+PR on both Adaptive Depth and Baldur’s Gate but is of little benefit on Rooms where perimeter pruning has already reduced the branching factor.

The large performance variation from one benchmark to another can be attributed to how effectively we can decompose the map. A good decomposition forms large rectangles with few perimeter nodes after pruning. This is the case for Rooms. A poor decomposition builds small rectangles with many transitional perimeter nodes that cannot be pruned. This is the case for Baldur’s Gate.

3.7.2 Comparison to Swamps:

Next, we compare and contrast the performance of RSR with the Swamps algorithm [Pochter et al., 2010]. To evaluate Swamps we used the authors’ source code, including their own implementation of A*. We then ran all experiments using their recommended running parameters: a swamp seed radius of 6 and “no change limit” of 2. Figure 3.8 (D to F) gives search time speedup results for both RSR and Swamps running on the 8-connected variants of our three benchmark problem sets. On Adaptive Depth and Rooms, where the terrain can be naturally decomposed into rectangles, RSR achieves higher speedups and is shown consistently better than Swamps. On Baldur’s Gate, where the 45-degree orientation of maps reduces the effectiveness of rectangle decomposition, Swamps-based pruning is more effective.

Next, we scaled every map in each benchmark by a factor of 3 and randomly generated a new set of 100 problem instances per map. Scaling has the effect of producing larger open areas and allows us to measure the impact of this variable on search time speedup. We present our findings in Figure 3.8 (G to I). We observe that while the maximum speedup achieved by both algorithms has increased, the gain for Swamps is very small while RSR shows dramatic improvement. Infact, if we limit our attention to problems of similar cost to those seen on the original maps we notice

Algorithm	Extra Memory	Baldur's Gate	Rooms
PH-e	$2 V $	3.16	11.9
PH-e	$8 V $	3.07	17.54
RSR	$ V $	2.8	18.2

Table 3.2: Avg. A* search time speedup: RSR vs PH-e. RSR figures are across all maps on each benchmark. PH-e figures are for a small subset selected by its authors (1 of 120 from Baldur's Gate and 5 of 300 from Rooms).

that the performance of Swamps actually decreases.

The observed performance characteristics are not unexpected: Swamps prune out *areas that can be avoided* without introducing a detour while rectangle-based symmetry reduction allows for a faster exploration of *areas that need to be searched*. Since it appears that the two algorithms are orthogonal, a natural extension of this work would be to combine the two: first, apply 4(or 8)ERR+PR (as appropriate) to a grid in order to eliminate as many interior nodes as possible; then, apply a Swamps-based decomposition to the resultant graph.

3.7.3 Comparison to Portal Heuristic:

We now compare RSR with PH-e – the enhanced variant of the recent Portal Heuristic algorithm [Goldenberg et al., 2010]. Although we did not have access to a working implementation of this method we will discuss its performance vs. RSR based on published results obtained by the original authors. As in [Goldenberg et al., 2010] we focus on the 4-connected variants of the Baldur's Gate and Rooms benchmarks. Table 3.2 summarises the main result.

PH-e performs well when it can decompose the map into areas of similar size with few transitional nodes connecting them. RSR performs well when it can decompose the map into large rectangles with few perimeter nodes. On Rooms, both decomposition approaches are highly effective. On Baldur's Gate both are comparatively less effective. Notice however that PH-e requires up to 7 times more memory than RSR to achieve similar results. PH-e narrows the scope of search to a corridor of nodes inside which the optimal path can be found. This means that, like Swamps, PH-e and RSR are entirely orthogonal and two can be easily combined. One possible combination involves using PH-e to more accurately guide search on a map pruned by RSR. Another possibility involves using symmetry elimination to speed up speed up pathfinding between successive pairs of portals during PH-e's refinement phase.

3.8 Discussion

In this chapter we study the problem of pathfinding in 4 and 8-connected grid maps; two domains which appear regularly in modern video games and academic literature. We show that in such settings optimal search can be very difficult because

between any pair of nodes there exist many alternative paths which are symmetric: that is the paths are identical save for the order in which individual moves occur. We presented RSR: a novel optimal offline method for breaking such symmetries which is simple to understand and requires no significant extra memory. Our method involves decomposing a map into empty rectangular rooms, pruning all nodes appearing in the interior and replacing them with a set of *macro edges* that facilitate optimal traversal from the perimeter of any room to the perimeter of any other. We also give an online node insertion technique that extends these guarantees to arbitrary pairs of locations appearing in the original unmodified map.

We evaluate the performance of our algorithm by running A* on a wide range of realistic game maps including one well known set from the game *Baldur's Gate II*. In many cases we are able to prune over 50% of all nodes on a given map and improve the average search time performance of A* by anywhere from several factors to over one order of magnitude. We then compare the performance of RSR to Swamp-based pruning [Pochter et al., 2009, 2011] and the Portal Heuristic [Goldenberg et al., 2010]—two algorithms considered state-of-the-art at the time of their publication and which are contemporaries to RSR.

We show that Swamp-based pruning and RSR have complementary strengths. In particular we find that Swamps are more useful on maps with small open areas while RSR becomes more effective as larger open areas are available on a map. We also identify a broad range of instances where RSR dominates convincingly and is clearly the better choice. When we compare RSR to the enhanced Portal Heuristic we find that our method exhibits similar or improved performance but requires up to 7 times less memory. We find that both Swamps and the Portal Heuristic are complementary ideas to RSR and we suggest how they could be easily combined.

The effectiveness of RSR is strongly dependent on the topography of individual maps: in the presence of large rooms or wide open areas (both commonly seen in video games⁸) we can often compute optimal paths much faster than searching on the original map. On less favourable map topographies we achieve more modest improvements. However, since our method is orthogonal to existing search techniques, it could be integrated as part of a larger framework involving specialised heuristics or other speedup techniques; for example as described in [Botea et al., 2004; Björnsson et al., 2005; Björnsson and Halldórsson, 2006].

⁸For example, Blizzard's popular multi-player game *World of Warcraft*

Jump Point Search

In Chapter 3 we showed that in regular domains, such as grid maps, symmetry can significantly impact the performance of pathfinding search. To address the problem we developed Rectangular Symmetry Reduction: an offline preprocessing strategy which decomposes the map in order to identify and break symmetries. In this chapter we develop Jump Point Search (JPS): an online pruning strategy that deals with symmetry by selectively expanding only certain nodes on a grid map which we call *jump points*. Moving from one jump point to the next involves travelling in a fixed direction while repeatedly applying a set of simple neighbour pruning rules until either an obstacle or a jump point is reached. Because we do not expand any intermediate nodes our strategy can have a dramatic positive effect on search performance.

We make the following contributions: (i) a detailed description of the jump points algorithm; (ii) a theoretical result which shows that searching with jump points preserves optimality; (iii) an extensive empirical analysis using a range of synthetic and real-world benchmarks from the pathfinding literature. We find that jump points can improve the search time performance of standard A* by an order of magnitude and more. We also report significant improvement over Swamps [Pochter et al., 2010] (a recent optimality preserving pruning technique) and search time performance that is competitive with HPA* [Botea et al., 2004] (a well known sub-optimal pathfinding algorithm).

JPS is unique in the pathfinding literature in that it has very few disadvantages: it is simple, yet highly effective; it preserves optimality, yet requires no extra memory; it is fast, yet requires no preprocessing. Further, our method is orthogonal to and easily combined with many other speedup techniques from the literature. We are unaware of any other algorithm which has all these features.

The contributions in this chapter have appeared previously in [Harabor and Grastien, 2011; Harabor and Grastien, 2012].

4.1 Notation and Terminology

We work with undirected uniform-cost grid maps, as described in Chapter 2.2.1. Each node has ≤ 8 neighbours and is either traversable or not. Each straight (i.e. horizontal or vertical) move, from a traversable node to one of its neighbours, has a cost of 1; diagonal moves cost $\sqrt{2}$. Moves involving non-traversable (obstacle) nodes are disallowed. The notation \vec{d} refers to one of the eight allowable movement directions (up, down, left, right etc.). We write $y = x + k\vec{d}$ when node y can be reached by taking k unit moves from node x in direction \vec{d} . When \vec{d} is a diagonal move, we denote the two straight moves at 45-degrees to \vec{d} as \vec{d}_1 and \vec{d}_2 .

A path $\pi = \langle n_0, n_1, \dots, n_k \rangle$ is a cycle-free ordered walk starting at node n_0 and ending at n_k . We will sometimes use the setminus operator in the context of a path: for example $\pi \setminus x$. This means that the subtracted node x does not appear on (i.e. is not mentioned by) the path. We will also use the function len to refer the length (or cost) of a path and the function $dist$ to refer to the distance between two nodes on the grid: e.g. $len(\pi)$ or $dist(n_0, n_k)$ respectively.

4.2 Jump Points By Example

The intuition for Jump Point Search is simple: in the process of searching for an optimal path on a grid map we will aim to selectively expand only certain “important” nodes and ignore all the rest. We term these important nodes *jump points* and give an example of the basic idea in Figure 4.1(a).

When expanding x we may notice that there is little point to evaluating any neighbour reachable by a dashed line as such a move is always dominated by (i.e. no better than) an alternative path which mentions node $p(x)$ (i.e. the parent of node x) but does not mention node x itself. We will make this idea more precise in Section 4.3 but for now it is sufficient to observe that the only non-dominated neighbour of x lies immediately to the right. Rather than generating this neighbour and adding it to the open list, as in the classical A* algorithm, we propose to simply step to the right and continue moving in this direction until we encounter a node such as y ; which has at least one other non-dominated neighbour (here z). If we find a node such as y (a jump point) we generate it as a successor of x and assign it a g -value (or cost-so-far) of $g(y) = g(x) + dist(x, y)$. Alternatively, if we reach an obstacle we conclude that further search in this direction is fruitless and generate nothing.

Figure 4.1(a) is an example of a jump that is based only on straight moves. In the remainder of this chapter we will develop a macro-step operator which is also applicable in the case where the x is reached from $p(x)$ by a diagonal move. This latter procedure is illustrated in Figure 4.1(b). It is similar in spirit to the previous example; the main difference is that before each diagonal step we must ensure that the node being skipped cannot be a possible turning point for any optimal path. We explain this process in more detail in Section 4.3 when we encapsulate the process of jumping from one node to another in two simple pruning rules. One rule is applicable to straight moves, the other diagonal moves. The purpose of each rule is

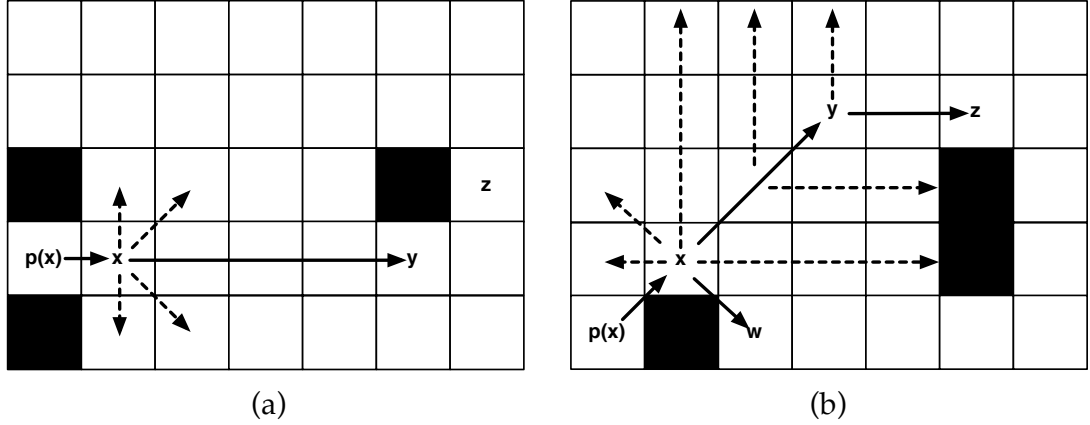


Figure 4.1: Examples of straight (a) and a diagonal (b) jump operations. In both cases the current search is expanding node x with parent node $p(x)$. Strong straight lines leading away from x indicate successor nodes that are identified as jump points. Dashed lines indicate a sequence of interim node evaluations that do not yield any jump point successors.

to decide whether a node should be generated or skipped. With these rules in hand we will proceed to give a detailed description of the Jump Point Search algorithm. We also give a corresponding theoretical result that shows that the process of “jumping over” nodes has no effect on the optimality of search.

4.3 Neighbour Pruning Rules

In this section we develop two simple rules for recursively pruning the set of nodes immediately adjacent to some node x from the grid. The objective is to identify from each set of such neighbours, i.e. $neighbours(x)$, any nodes n that do not need to be evaluated in order to reach the goal optimally. We achieve this by comparing the cost of two paths: π , which begins with node $p(x)$ visits x and ends with n and another path π' which also begins at node $p(x)$ and ends with n but does not mention x . Additionally, each node mentioned by either π or π' must belong to $neighbours(x)$. There are two cases to consider, depending on whether the transition to x from its parent $p(x)$ involves a straight move or a diagonal move. Note that if x is the start node $p(x)$ is null; neither case is applicable and nothing is pruned.

1. **Straight Moves:** When x is reached from $p(x)$ by a straight move we prune any neighbouring node $n \in neighbours(x)$ which satisfies the following dominance constraint:

$$len(\langle p(x), \dots, n \rangle \setminus x) \leq len(\langle p(x), x, n \rangle) \quad (4.1)$$

Figure 4.2(a) shows an example of this pruning rule applied. Here $p(x) = 4$ and we prune all neighbours except $n = 5$.

2. **Diagonal Moves:** When x is reached from $p(x)$ by a diagonal move we prune any neighbouring node $n \in neighbours(x)$ which satisfies the following domi-

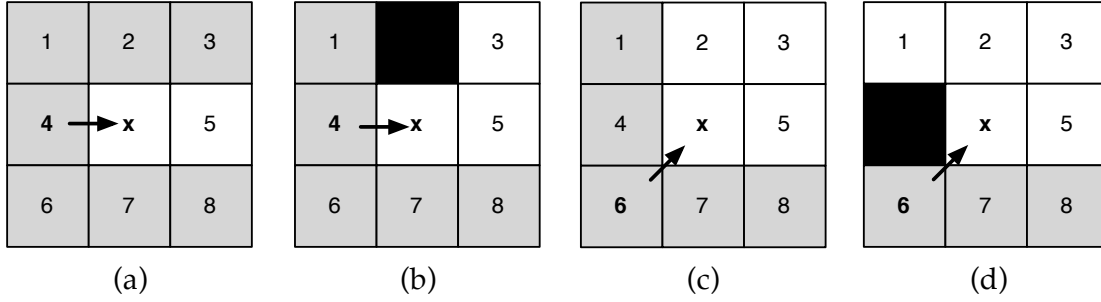


Figure 4.2: We show several cases where a node x is reached from its parent $p(x)$ by either a straight or diagonal move. When x is expanded we can prune from consideration all nodes marked grey.

nance constraint:

$$\text{len}(\langle p(x), \dots, n \rangle \setminus x) < \text{len}(\langle p(x), x, n \rangle) \quad (4.2)$$

This case is similar to the pruning rule for straight moves; the only difference is that the path which excludes x must be strictly dominant. Figure 4.2(c) shows an example. Here $p(x) = 6$ and we prune all neighbours except $n = 2$, $n = 3$ and $n = 5$.

4.3.1 Forced and Natural Neighbours

Assuming $\text{neighbours}(x)$ contains no obstacles, we will refer to the nodes that remain after the application of straight or diagonal pruning (as appropriate) as the *natural neighbours* of x . These correspond to the non-gray nodes in Figures 4.2(a) and 4.2(c). When $\text{neighbours}(x)$ contains an obstacle, we may not be able to prune all non-natural neighbours. If this occurs we say that the evaluation of each such neighbour is *forced*.

Definition 5 A node $n \in \text{neighbours}(x)$ is forced if:

1. n is not a natural neighbour of x
2. $\text{len}(\langle p(x), x, n \rangle) < \text{len}(\langle p(x), \dots, n \rangle \setminus x)$

In Figure 4.2(b) we show an example of a straight move where the evaluation of $n = 3$ is forced. Figure 4.2(d) shows an similar example involving a diagonal move; here the evaluation of $n = 1$ is forced.

4.4 Algorithmic Description

In this section we give a complete description of the Jump Point Search algorithm. We begin by first making precise the concept of a jump point.

Definition 6 Node y is the jump point from node x , heading in direction \vec{d} , if y minimizes the value k such that $y = x + k\vec{d}$ and one of the following conditions holds:

1. Node y is the goal node.
2. Node y has at least one neighbour whose evaluation is forced according to Definition 5.
3. \vec{d} is a diagonal move and there exists a node $z = y + k_i\vec{d}_i$ which lies $k_i \in \mathbf{N}$ steps in direction $\vec{d}_i \in \{\vec{d}_1, \vec{d}_2\}$ such that z is a jump point from y by condition 1 or condition 2.

Figure 4.1(b) shows an example of a jump point which is identified by way of condition 3. Here we start at x and travel diagonally until encountering node y . From y , node z can be reached with $k_i = 2$ horizontal moves. Thus z is a jump point successor of y (by condition 2) and this in turn identifies y as a jump point successor of x .

The process by which individual jump point successors are identified is given in Algorithm 1. We start with the pruned set of neighbours immediately adjacent to the current node x (line 2). Then, instead of adding each neighbour n to the set of successors for x , we try to “jump” to a node that is further away but which lies in the same relative direction to x as n (lines 3:5). For example, if the edge (x, n) constitutes a straight move travelling *right* from x , we look for a jump point among the nodes immediately to the right of x . If we find such a node, we add it to the set of successors instead of n . If we fail to find a jump point, we add nothing. The process continues until the set of neighbours is exhausted and we return the set of successors for x (line 6).

In order to identify individual jump point successors we will apply Algorithm 2. It requires an initial node x , a direction of travel \vec{d} , and the identities of the start node s and the goal node g . In rough overview, the algorithm attempts to establish whether x has any jump point successors by stepping in the direction \vec{d} (line 1) and testing if the node n at that location satisfies Definition 6. When this is the case, n is designated a jump point and returned (lines 5, 7 and 11). When n is not a jump point the algorithm recurses and steps again in direction \vec{d} but this time n is the new initial node (line 12). The recursion terminates when an obstacle is encountered and no further steps can be taken (line 3). Note that before each diagonal step the algorithm

Algorithm 1 Identify Successors

Require: x : current node, s : start, g : goal

- 1: $\text{successors}(x) \leftarrow \emptyset$
 - 2: $\text{neighbours}(x) \leftarrow \text{prune}(x, \text{neighbours}(x))$
 - 3: **for all** $n \in \text{neighbours}(x)$ **do**
 - 4: $n \leftarrow \text{jump}(x, \text{direction}(x, n), s, g)$
 - 5: add n to $\text{successors}(x)$
 - 6: **end for**
 - 7: **return** $\text{successors}(x)$
-

Algorithm 2 Function *jump*

Require: x : initial node, \vec{d} : direction, s : start, g : goal

```

1:  $n \leftarrow \text{step}(x, \vec{d})$ 
2: if  $n$  is an obstacle or is outside the grid then
3:   return null
4: end if
5: if  $n = g$  then
6:   return  $n$ 
7: end if
8: if  $\exists n' \in \text{neighbours}(n)$  s.t.  $n'$  is forced then
9:   return  $n$ 
10: end if
11: if  $\vec{d}$  is diagonal then
12:   for all  $i \in \{1, 2\}$  do
13:     if  $\text{jump}(n, \vec{d}_i, s, g)$  is not null then
14:       return  $n$ 
15:     end if
16:   end for
17: end if
18: return  $\text{jump}(n, \vec{d}, s, g)$ 

```

must first fail to detect any straight jump points (lines 9:11). This check corresponds to the third condition of Definition 6 and is essential for preserving optimality.

4.5 Optimality

In this section we prove that for each optimal cost path in a grid map there exists an equivalent cost path which can be found by only expanding jump point nodes during search (Theorem 3). Our result is derived by identifying for each optimal path a symmetric alternative which we split into contiguous segments. We then prove that each *turning point* along this path is also a jump point.

Definition 7 A turning point is any node n_i along a path where the direction of travel from the previous node n_{i-1} to n_i is different to the direction of travel from n_i to the subsequent node n_{i+1} .

Figure 4.3 depicts the three possible kinds of turning points which we may encounter on an optimal path. A diagonal-to-diagonal turning point at node n_k (Figure 4.3(a)) involves a diagonal step from its parent n_{k-1} followed by a second diagonal step, this time in a different direction, from n_k to its successor n_{k+1} . Similarly, a straight-to-diagonal (or diagonal-to-straight) turning point involves a straight (diagonal) step from n_{k-1} to reach n_k followed by a diagonal (straight) step to reach its successor n_{k+1} (Figure 4.3(b) and 4.3(c) respectively). Other types of turning points,

such as straight-to-straight, are trivially sub-optimal and not considered here (they are pruned by the rules we developed earlier; see again Figure 4.2).

We are now ready to develop an equivalence relation between jump points and the turning points that appear along certain optimal cost symmetric paths which we term *diagonal-first*.

Definition 8 A path π is *diagonal-first* if and only if it contains no straight-to-diagonal turning point $\langle n_{k-1}, n_k, n_{k+1} \rangle$ which could be replaced by a diagonal-to-straight turning point $\langle n_{k-1}, n'_k, n_{k+1} \rangle$ s.t. the cost of π remains unchanged.

Given an arbitrary optimal cost path π , we can always derive a symmetric diagonal-first path π' by applying Algorithm 3 to π . Note that this is merely as a conceptual device.

Lemma 7 Each turning point along an optimal diagonal-first path π' is also a jump point.

Proof: Let n_k be an arbitrary turning point node along π' . We will consider three cases, each one corresponding to one of the three possible kinds of optimal turning points illustrated in Figure 4.3.

1. **Diagonal-to-Diagonal:** Since π' is optimal, there must be an obstacle adjacent to both n_k and n_{k-1} which is forcing a detour. We know this because if there were no obstacle we would have $\text{dist}(n_{k-1}, n_{k+1}) < \text{dist}(n_{k-1}, n_k) + \text{dist}(n_k, n_{k+1})$ which contradicts the fact that π' is optimal. We conclude that n_{k+1} is a forced neighbour of n_k . This is sufficient to satisfy the second condition of Definition 5, making n_k a jump point.
2. **Straight-to-Diagonal:** In this case there must be an obstacle adjacent to n_k . If this were not true n_k could be replaced by a Diagonal-to-Straight turning point which contradicts the fact that π' is diagonal-first. Since π' is guaranteed to be diagonal-first we derive the fact that n_{k+1} is a forced neighbour of n_k . This satisfies the second condition of Definition 5 and we conclude n_k is a jump point.

Algorithm 3 Compute Diagonal First Path

Require: π : an arbitrary optimal cost path

- 1: **select** an adjacent pair of edges appearing along π where (n_{k-1}, n_k) is a straight move and (n_k, n_{k+1}) is a diagonal move.
 - 2: **replace** (n_{k-1}, n_k) and (n_k, n_{k+1}) with two new edges: (n_{k-1}, n'_k) , which is a diagonal move and (n'_k, n_{k+1}) which is a straight move. The operation is successful if (n_{k-1}, n'_k) and (n'_k, n_{k+1}) are both valid moves; i.e. node n'_k is not an obstacle.
 - 3: **repeat** lines 1 and 2, selecting and replacing adjacent edges, until no further changes can be made to π .
 - 4: **return** π
-

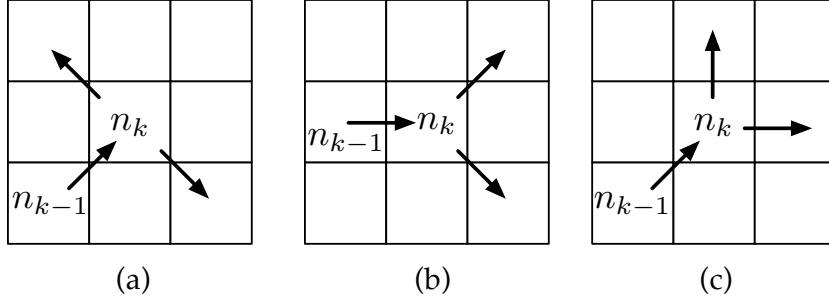


Figure 4.3: Types of optimal turning points. (a) Diagonal-to-Diagonal (b) Straight-to-Diagonal (c) Diagonal-to-Straight.

3. **Diagonal-to-Straight:** There are two possibilities in this case, depending on whether the goal is reachable by a series of straight steps from n_k or whether π' has additional turning points. If the goal is reachable by straight steps then n_k has a jump point successor which satisfies the third condition of Definition 5 and we conclude n_k is also a jump point. If n_k is followed by another turning point, n_l , then that turning point must be Straight-to-Diagonal and, by the argument for that case, also a jump point. We again conclude that n_k has a jump point successor which satisfies the third condition of Definition 5. Thus, n_k is also a jump point.

□

Theorem 3 *Searching with jump point pruning always returns an optimal solution.*

Proof: Let π be an arbitrarily chosen optimal path between two nodes on a grid and π' a diagonal-first symmetric equivalent which is derived by applying Algorithm 3 to π . We will show that every turning point mentioned by π' is expanded optimally when searching with jump point pruning. We argue as follows:

Divide π' into a series of adjacent segments s.t. $\pi' = \pi'_0 + \pi'_1 + \dots + \pi'_n$. Each $\pi'_i = \langle n_0, n_1, \dots, n_{k-1}, n_k \rangle$ is a subpath along which all moves involve travelling in the same direction (e.g. only “up” or “down” etc). Notice that with the exception of the start and goal, every node at the beginning and end of a segment is also a turning point.

Since each π'_i consists only of moves in a single direction (straight or diagonal) we can use Algorithm 2 to jump from $n_0 \in \pi'_i$, the node at beginning of each segment to $n_k \in \pi'_i$, the node at the end, without necessarily stopping to expand every node in between. Intermediate expansions may occur but the fact that we reach n_k optimally from n_0 is guaranteed. It remains to show only that both n_0 and n_k are identified as jump points and thus necessarily expanded. By Lemma 7 each turning point along π' is also a jump point, so every turning point node must be expanded during search. Only the start and goal remain. The start node is necessarily expanded at the beginning of each search while the goal node is a jump point by definition. Thus both are

expanded. □

4.6 Weighted Grid Maps

The jump point identification procedure given in Section 4.4 breaks path symmetries found in uniform-cost grid maps. It assumes that straight steps always have a cost of 1 and diagonal steps always have a cost of $\sqrt{2}$. Though such settings are common in a variety of different application areas, including robotics and computer games, it is sometimes desirable to represent the map as a weighted grid. Such a change allows us model more realistic environments and represent the fact that some areas on a map can be easier (or more difficult) to traverse than others. In this section we will discuss a simple modification to Jump Point Search which allows it to identify symmetries in such settings.

Since we are unaware of any standard approach for representing weighted grids we propose the following simple cost model: We assume the input map is an undirected grid where each tile is either traversable or not. Each traversable tile is associated with a particular terrain type and each terrain type has a corresponding numeric value ≥ 1 called the terrain cost. The cost of moving one step in the grid is equal to the uniform-cost distance between the two tiles at hand (i.e. 1 or $\sqrt{2}$) multiplied by the average of the terrain costs of all tiles intersected during the transition. In the case of a straight step, the multiplier is the average of the terrain costs of the source and destination tiles. In the case of diagonal steps, the multiplier is averaged over the terrain costs of the three tiles the agent must intersect while moving.

We now propose a very simple modification Jump Point Search; simply replace Definition 5 with the following stronger alternative:

Definition 9 *A node $n \in \text{neighbours}(x)$ is forced if:*

1. *n is not a natural neighbour of x*
2. *$\text{len}(\langle p(x), x, n \rangle) < \text{len}(\langle p(x), \dots, n \rangle \setminus x)$*
3. *$\text{terrain}(x) \neq \text{terrain}(n)$*

Clauses 1 and 2 are identical to Definition 5; they guarantee that JPS will continue to break uniform-cost symmetries when Algorithm 2 is recursing in a region where all tiles have the same terrain type. Clause 3 is newly introduced; its purpose is to terminate the recursion whenever Algorithm 2 encounters a node n that has a different terrain type to the current node x . This approach is equivalent to dividing the map into a set of uniform-cost regions and applying JPS to each region; stopping the recursion each time we find a node that is forced according to Definition 5 or whenever we encounter a node that allows that search to transition from one weighted region to another. Since we only ever jump over nodes that have the same terrain type as the current node we are guaranteed to break only uniform-cost symmetries. It is thus easy to see that this strategy is optimality preserving.

4.7 Time and Space Requirements

Jump Point Search performs on-the-fly symmetry breaking by recursively travelling along the map in a given direction and checking each node it encounters for the presence of forced neighbours. Given an initial node x , Algorithm 2 steps recursively in a fixed direction direction \vec{d} until one of three stopping conditions are satisfied: (i) the procedure encounters the goal node (line 6); (ii) the procedure encounters a node with a forced neighbour (line 9); (iii) the procedure encounters an obstacle or the edge of the map (line 3).

In the case of a straight jump (i.e. one where \vec{d} is one of the four cardinal directions) Algorithm 2 will recurse at most until it reaches the edge of the map. In the process it will check up to $(h - 1)$ or $(w - 1)$ nodes (h being the height of the map and w its width) depending on whether the direction of travel is horizontal or vertical. In the case of a diagonal jump Algorithm 2 can once again recurse at most until it reaches the edge of the map. However it must perform two straight jumps before recursing diagonally. Assuming there are no obstacles that will result in early termination, it is easy to see that the procedure can visit every node on the map, checking each one for forced neighbours. Each forced neighbour check requires a constant amount of memory and takes a constant amount of time. Moreover, since JPS does not cache the results of these operations, the same node can be checked multiple times during the course of solving single pathfinding problem.

We will see in Section 4.9 JPS is efficient despite needing to perform many forced neighbour checks. The reason for this efficiency is due to the fact that our search space (i.e. the grid at hand) is defined explicitly and can be stored in memory. Because the grid is readily available each forced neighbour check is equivalent to a small handful of memory operations. By comparison, expanding a node is much more expensive. In addition to reading values to and from memory we also need to invoke the heuristic function h and push and pop values to and from the open list (using a binary heap open list operation takes $O(\log_2 n)$ time). We explore these issues in more depth in Chapter 5 wherein we suggest a number of enhancements to JPS that make forced-neighbour checks cheaper still.

4.8 Experimental Setup

We implemented Jump Point Search using the freely available pathfinding library Hierarchical Open Graph¹ (HOG). We evaluate its performance on four benchmarks provided by HOG:

- **Adaptive Depth** is a set of 12 maps of size 100×100 in which approximately $\frac{1}{3}$ of each map is divided into rectangular rooms of varying size and a large open area interspersed with large randomly placed obstacles. For this benchmark we randomly generated 100 valid problems per map for a total of 1200 instances.

¹<http://www.googlecode.com/p/hog2>

	Adaptive Depth	Baldur's Gate II	Dragon Age: Origins	Rooms
JPS	20.37	215.36	35.95	13.41
Swamps	1.89	2.44	2.99	4.70
HPA*	4.14	9.37	9.63	5.11

Table 4.1: Average A* node expansion speedup.

- **Baldur's Gate** is a set of 120 maps taken from BioWare's popular role-playing game *Baldur's Gate II: Shadows of Amn*; it appears regularly as a standard benchmark in the literature [Björnsson and Halldórsson, 2006; Harabor and Botea, 2010; Pochter et al., 2010]. We use the variation due to Nathan Sturtevant where all maps have been scaled to size 512×512 to more accurately represent modern pathfinding environments. Maps and all instances are available from <http://movingai.com>
- **Dragon Age** is another realistic benchmark; this time taken from BioWare's recent role-playing game *Dragon Age: Origins*. It consists of 156 maps ranging in size from 30×21 to 1104×1260 . For this benchmark we used a large set of randomly generated instances, again due to Nathan Sturtevant and available from <http://movingai.com>.
- **Rooms** is a set of 300 maps of size 256×256 which are divided into symmetric rows of small rectangular areas (each of size 8×8) that are connected by entrances randomly placed along their perimeter. This benchmark has previously appeared in [Pochter et al., 2010]. For this benchmark we randomly generated 100 valid problems per map for a total of 30000 instances.

Our test machine is a 2.93GHz Intel Core 2 Duo processor with 4GB RAM running OSX 10.6.4.

4.9 Results

To evaluate Jump Point Search (JPS) we use a generic implementation of A* which we adapted to facilitate online neighbour pruning and jump point identification. We discuss performance in terms of *speedup*: i.e. relative improvement to the time taken to solve a given problem, and the number of nodes expanded, when searching with and without graph pruning applied to the grid. Using this metric a search time speedup of 2.0 is twice as fast while a node expansion speedup of 2.0 indicates half the number of nodes were expanded. In each case higher is better. Figure 4.4 shows average search time speedups across each of our four benchmarks. Table 4.1 shows average node expansion speedups; the best results for each column are in bold.

4.9.1 Comparison with Swamps

We begin by comparing JPS to Swamps [Pochter et al., 2010]: an optimality preserving pruning technique for speeding up pathfinding. We used the authors’ source code, and their implementation of A*, and ran all experiments using their recommended running parameters: a swamp seed radius of 6 and “no change limit” of 2.

As per Figure 4.4, JPS shows a convincing improvement to average search times across all benchmarks. The largest differences are observed on Baldur’s Gate and Dragon Age where JPS reaches the goal between 25-30 times sooner than A* while Swamps attains only a 3-5 times speedup. Similar trends are observed when looking at Table 4.1, where the improvement to the total number of nodes expanded is even more pronounced.

Based on these results we conclude that while Swamps are effective for identifying areas of the map not relevant to reaching the goal, those areas which remain still require significant effort to search. JPS uses a much stronger yet orthogonal strategy to prove that many nodes expanded by Swamps can be ignored. As the two ideas appear complementary we posit that they could be easily combined: first, apply a Swamps-based decomposition to prune areas not relevant to the current search. Then, use JPS to search the remaining portions of the map.

4.9.2 Comparison with HPA*

Next, we compare JPS to the HPA* algorithm [Botea et al., 2004]. Though sub-optimal, HPA* is very fast and widely applied in video games. To evaluate HPA* we measure the total cost of insertion and hierarchical search. We did not refine any abstract paths, assuming instead that a database of pre-computed intra-cluster paths was available. Such a configuration represents the fastest generic implementation of HPA* but requires additional memory overheads. While searching we used a single-level abstraction hierarchy and a fixed cluster size of 10×10 . These settings are recommended by the original authors who note that larger clusters and more levels are often of little benefit².

As per Figure 4.4, JPS is shown to be highly competitive with HPA*. On Adaptive Depth, Baldur’s Gate and Rooms, JPS has a clear advantage – improving on HPA* search times by several factors in some cases. On the Dragon Age benchmark jump points have a small advantage for problems of cost < 500 but for longer instances there is very little between the two algorithms. Table 4.1 provides further insight: although searching with jump points expands significantly fewer nodes than HPA*, each such operation takes longer.

We conclude that JPS is a competitive substitute for HPA* and, for a wide variety of problems, can help to find solutions significantly faster. HPA* can still be advantageous, particularly if a memory overhead is acceptable and optimality is not

²Recent experiments by Sturtevant and Geisberger [2010] have shown that a second level of abstraction can yield additional benefits for pathfinding search. Note however that this work uses an abstraction mechanism that is different to HPA*.

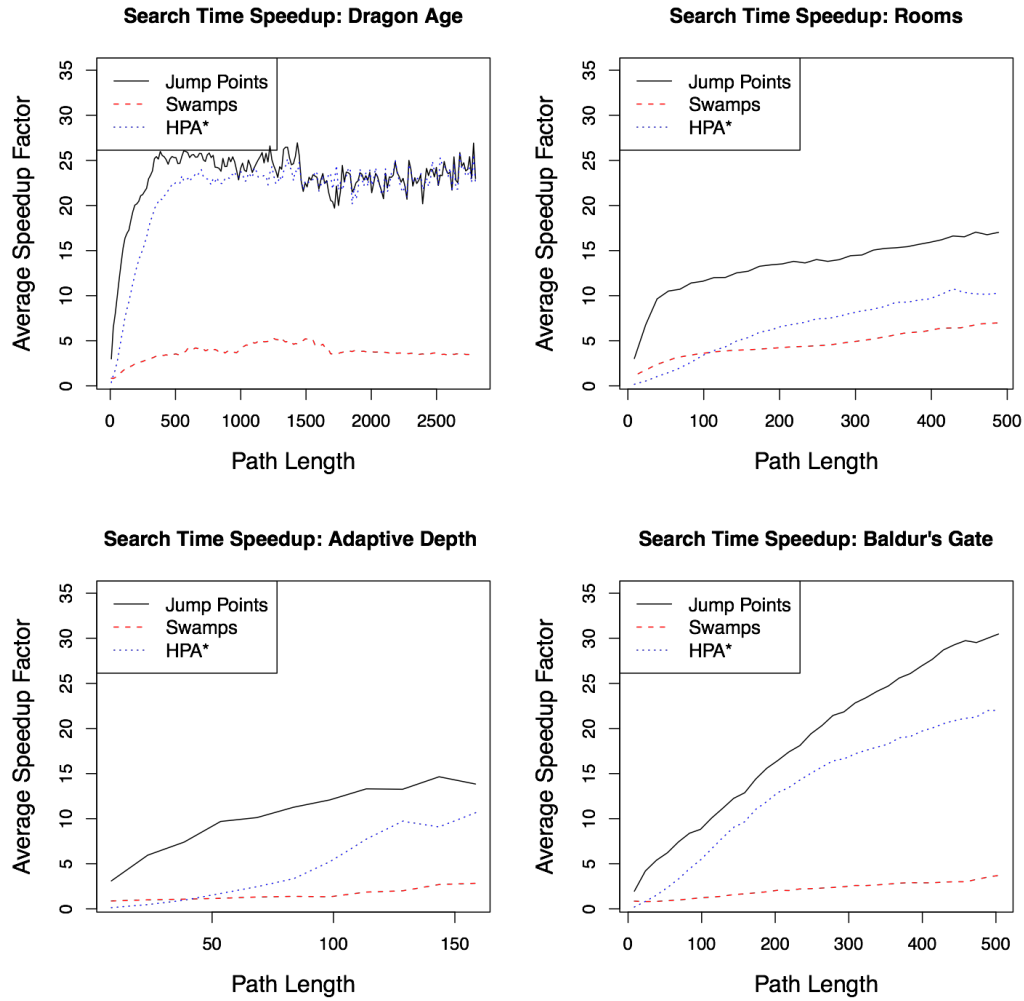


Figure 4.4: Average A* search time speedup on our each of our four benchmarks.

important, but only if the cost of inserting the start and goal nodes into the abstract graph (and possibly refinement, if a path database is not available), can be sufficiently amortized over the time required to find an abstract path. One direction for further work could be to use jump point pruning to speed up HPA*: for example during insertion and refinement.

4.10 Discussion

In this chapter we introduced Jump Point Search: a new online symmetry breaking strategy for speeding up pathfinding on grid maps. Our algorithm identifies and selectively expands only certain nodes from a grid map which we call *jump points*.

Moving between jump points involves only travelling in a fixed direction, either straight or diagonal. We prove that intermediate nodes on a path between two jump points never need to be expanded and “jumping” over them does not affect the optimality of search.

Our method is unique in the pathfinding literature in that it has very few disadvantages: it is simple, yet highly effective; it preserves optimality, yet requires no extra memory; it is fast, yet requires no preprocessing. Further, it is largely orthogonal to and easily combined with other speedup techniques from the literature. We are unaware of any other algorithm which has all these features.

The new algorithm is highly competitive with recent and related works from the literature. When compared to Swamps [Pochter et al., 2010], a recent state-of-the-art optimality preserving pruning technique, we find that jump points are up to an order of magnitude faster. We also show that jump point pruning is competitive with, and in many instances clearly faster than, HPA* [Botea et al., 2004]; a popular though sub-optimal pathfinding technique often employed in performance sensitive applications such as video games.

Improving Jump Point Search

In the previous chapter we described Jump Point Search (JPS): an effective technique for identifying and eliminating path symmetries on-the-fly. JPS can be described as the combination of A* search with two simple neighbour-pruning rules. When applied recursively these rules can improve the performance of optimal grid-based pathfinding by an order of magnitude and more – all without any pre-processing and without the introduction of any memory overheads.

In this chapter we give several online and offline optimisation techniques to further improve the performance of Jump Point Search. In particular:

- We give a new and more efficient procedure for online symmetry breaking. We describe how an efficient bitwise encoding of the grid that allows us to manipulate “blocks” of nodes. We show that using such an encoding it is possible for JPS to effectively break symmetries for many nodes at the same time – rather than considering each node individually.
- We describe a new variant of JPS which uses offline pre-processing to further speed up pathfinding search. Our approach exploits the fact that many jump points are goal-independent. We show that it is possible to pre-compute and store all such jump points for every node on the map.
- We enhance the pruning rules of JPS by showing that many of the nodes which it currently expands are simply intermediary locations that can be safely ignored. These enhanced rules can be easily combined with either of the previously described contributions.

We run a large number of experiments on three benchmark domains taken from real computer games. We find that our enhancements can significantly improve JPS performance by anywhere from several factors (in the online case) to over one order of magnitude (using pre-processing). Each improvement is complete and optimal and requires either no time and memory overhead or only a small amount. Moreover, we demonstrate that our improvements are competitive with and often faster than two very recent state-of-the-art pathfinding algorithms from the literature.

The contributions in this chapter have previously appeared in [Harabor and Grastien, 2014]

5.1 Introduction

The efficiency of Jump Point Search depends on being able to quickly scan many nodes from the underlying grid map in order to identify successor nodes (i.e. jump points). On the one hand such a procedure can typically save many unnecessary node expansions. On the other hand the same operation proceeds in a step-by-step manner and it can scan the same node multiple times during a single search. Consider Table 5.1, where we give a comparative breakdown of how JPS and A* spend their time during search. The results are obtained by running a large set of standard instances on three realistic game benchmarks that appeared in the 2012 Grid-based Path Planning Competition. Observe that JPS spends $\sim 90\%$ of its time generating successors (cf. $\sim 40\%$ for A*) instead of manipulating nodes on the open and closed lists – i.e. searching.

In this chapter we propose a number of ideas that to improve the performance of Jump Point Search. We focus particularly on: (i) more efficient online symmetry breaking that reduces the time spent scanning the grid; (ii) pre-computation strategies for breaking symmetries offline. (iii) more effective online pruning strategies that avoid expanding some jump points; We evaluate our ideas on three realistic grid-based benchmarks and find that our enhancements can improve the performance of Jump Point Search by anywhere from several factors to over one order of magnitude.

	A*		JPS	
	M.Time	G.Time	M.Time	G.Time
D. Age: Origins	58%	42%	14%	86%
D. Age 2	58%	42%	14%	86%
StarCraft	61%	39%	11%	89%

Table 5.1: A comparative breakdown of total search time on three realistic video game benchmarks. M.Time is the time spent manipulating nodes on open or closed. G.Time is the time spent generating successors (i.e. scanning the grid).

5.2 Block-based Symmetry Breaking

In this section we will show how to apply the pruning rules from Jump Point Search to many nodes at a single time. Such block-based operations will allow us to scan the grid much faster and dramatically improve the overall performance of pathfinding search. Our approach requires only that we encode the grid as a matrix of bits where each bit represents a single location and indicates whether the associated node is traversable or not. Bitwise encodings have the advantage that the map is often small enough to be permanently loaded into CPU cache for fast access during search.

For a motivating example, consider the grid presented in Figure 5.1 (this is supposed to be a small chunk of a larger grid). The node currently being explored is $N = \langle 2, 2 \rangle$ and its parent is $P = \langle 1, 1 \rangle$. At this stage, the horizontal and vertical axes must be scanned for jump points before another diagonal move is taken. As it turns

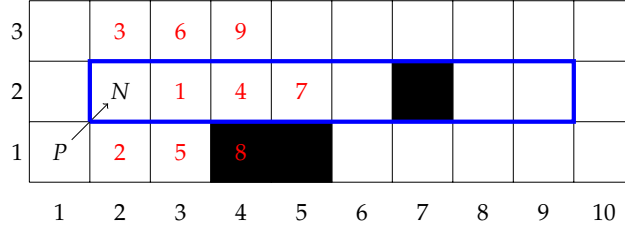


Figure 5.1: A current search state (the grid is assumed larger than the part presented). The red numbers show in which order the traversability of the nodes is tested. The blue rectangle represents the byte that is returned when we scan the grid to read the value of location $N = \langle 2, 2 \rangle$.

out, a jump point will be found at location $\langle 6, 2 \rangle$. When looking for a horizontal jump point on row 2, Jump Point Search will scan the grid more or less in the order given by the numbers in red, depending on the actual implementation. Each one of these nodes will be individually tested and each test involves reading a separate value from the grid.

We will exploit the fact that memory entries are organised into fixed-size lines of contiguous bytes. Thus, when we scan the grid to read the value of location $N = \langle 2, 2 \rangle$ we would like to be returned a byte B_N that contains other bit-values too, such as those for locations up to $\langle 9, 2 \rangle$. In this case we have:

$$B_N = [0, 0, 0, 0, 0, 1, 0, 0] \quad (5.1)$$

Each zero valued bit in B_N represents a traversable node and each set bit represents an obstacle. Note that in practice we read several bytes at one time and shift the returned value until the bit corresponding to location $\langle 2, 2 \rangle$ is in the lowest position. In a similar fashion and using only two further memory operations we can read the values for all nodes immediately above and immediately below those in byte B :

$$B_{\uparrow} = [0, 0, 0, 0, 0, 0, 0, 0] \quad (5.2)$$

$$B_{\downarrow} = [0, 0, 1, 1, 0, 0, 0, 0] \quad (5.3)$$

Note that our implementation uses 32-bit words but for this discussion we will continue to use 8-bit bytes as they are easier to work with.

When searching recursively along a given row or column there are three possible reasons that cause JPS to stop: a forced neighbour is found in an adjacent row, a dead-end is detected in the current row or the target node is detected in the current row. We can easily test for each of these conditions via simple operations on the bytes B_N , B_{\uparrow} and B_{\downarrow} .

Detecting dead-ends:

A dead-end exists in position $B[i]$ of byte B if $B[i] = 0$ and $B[i + 1] = 1$. We can test for this in a variety of ways; CPU architectures such as the Intel x86 family for example

have the native instruction `ffs` (find first set). The same instruction is available as a built-in function of the GCC compiler. When we apply this function to B_N we find a dead-end at position 5.

Detecting forced neighbours:

A potential forced neighbour exists in position i of byte B , or simply $B[i]$, if there is an obstacle at position $B[i - 1]$ and no obstacle at position $B[i]$. We test for this condition with the following bitwise operation (assuming left-to-right travel):

$$\text{forced}(B) = (B \ \& \ !(B \ll 1)) \quad (5.4)$$

When we apply this procedure to B_\downarrow we find a potential jump point at bit position 3; $\text{forced}(B_\uparrow)$ yields no potential jump points. In order to minimise the number of branching instructions (important to prevent CPU stalling) we do not test the individual results of operations: rather we combine their results by way of bitwise disjunction into a single byte B_S (S for stop). For the example in Figure 5.1 we have

$$B_S = \text{forced}(B_\uparrow) \mid \text{forced}(B_\downarrow) \mid B_N \quad (5.5)$$

$$B_S = [0, 0, 0, 1, 0, 1, 0, 0] \quad (5.6)$$

Because $B_S \neq 0$, we know that the search has to stop. Using the `ffs` command we extract the position of the first set bit in B_S (call this bit b_S) and compare it with the position of the first set bit in B_N (call this bit b_N). If $b_S \leq b_N$ we have discovered a jump point at location $B_N[b_S - 1]$; otherwise we have hit a dead-end. Alternatively, if B_S evaluates to zero there is no reason to stop: we jump ahead 7 positions (not 8) and repeat the procedure until termination.

Detecting the target node:

To avoid jumping over the target node we compare its position to the position of the node where the block-based symmetry-breaking procedure terminated – either due to finding a successor or a reaching dead-end. If the target lies between these two locations we generate it as any other jump point successor.

5.2.1 Additional Considerations

The procedure we have described in our running example is applicable in the case where JPS scans the grid left-to-right. The modification for right-to-left scanning is simple: we shift the current node into the most significant position of B_N and replace `ffs` with the analogous instruction `msb`. In the case of up-down travel we have found it helpful to store a redundant copy of the map that is rotated by 90 degrees. Other possibilities also exist that do not incur an overhead: for example the map may be

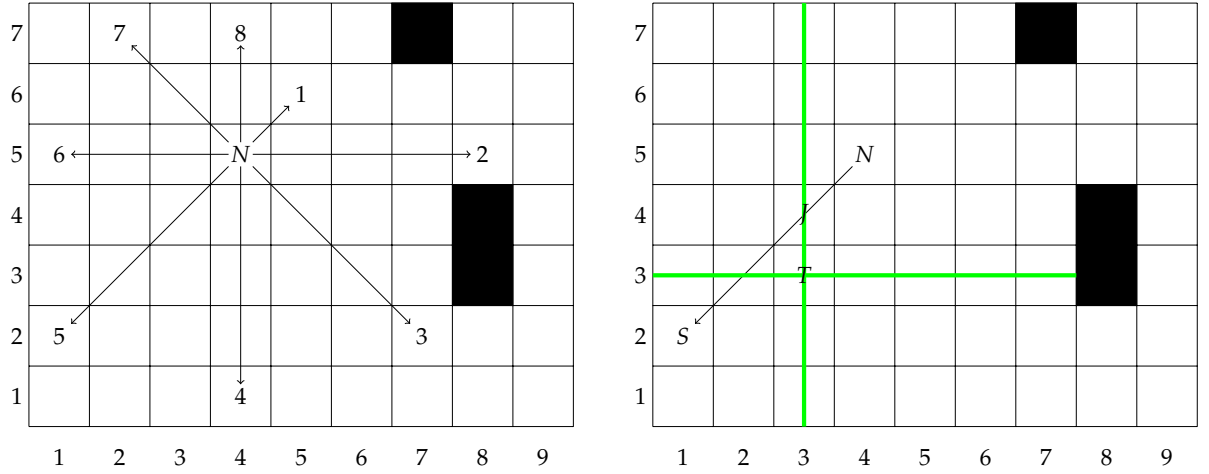


Figure 5.2: (a) A jump point is computed in place of each grid neighbour of node x . (b) When jumping from x to y we may cross the row or column of the target t (here, both). To avoid jumping over t we insert an intermediate successor y' on the row or column of t (whichever is closest to x).

stored as blocks of size $m \times n$ that each contain bits from several rows and columns. Such an approach would also benefit diagonal travel (which in our implementation remains step-by-step) but may reduce the step size during horizontal scanning.

5.3 Preprocessing

In the preceding section we have suggested a strategy for enhancing the online performance of Jump Point Search. In this section we give an offline technique which can improve the algorithm further still. First, recall that Jump Point Search distinguishes between three different kinds of nodes:

- Straight jump points. Reached by travelling in a cardinal direction these nodes have at least one forced neighbour.
- Diagonal jump points. Reached by travelling in a diagonal direction, these nodes have (i) one or more forced neighbours, or (ii) are intermediate turning points from which a straight jump point or the target can be reached.
- The target node. This is a special node which JPS treats as a jump point.

We will precompute for every traversable node on the map the first straight or diagonal jump point that can be reached by travelling away from the node in each of the eight possible cardinal or diagonal directions. During this step we do not identify any jump points that depend on a specific target node. However, as we will show, these can be easily identified at run-time.

The precomputation is illustrated on Figure 5.2(a). The left side of the figure shows the precomputed jump points for node $N = \langle 4, 5 \rangle$. Nodes 1–3 are typical

examples of straight or diagonal jump points. The others, nodes 4–8, would normally be discarded by JPS because they lead to dead-ends; we will remember them anyway but we distinguish them as *sterile jump points* and never generate them (unless they lead to the target).

Consider now Figure 5.2(b), where T is the target node. Travelling South-East away from N , JPS would normally identify J as a diagonal jump point because it is an intermediate turning point on the way to T . However J was not identified as a jump point during preprocessing because T was unknown. Instead, the sterile jump point S is recorded in the precomputed database. We use the location of S to determine whether the jump from N to S crosses the row or column of T (and where) and then test whether T is reachable from that location. This procedure leads us to identify and generate J as a diagonal jump point successor of N . We apply the same intersection test more broadly – to all successors of N . This is sufficient to guarantee both completeness and optimality. We call this revised pre-processing based algorithm JPS+.

5.3.1 Properties

JPS+ requires an offline pre-processing step that has worst-case quadratic time complexity and linear space requirements w.r.t the number of nodes in the grid. The time bound is very loose: it arises only in the case where the map is obstacle free and one diagonal jump can result in every node on the map being scanned. In most cases only a small portion of the total map needs to be scanned. Moreover, if we pre-compute using block-based symmetry breaking the entire procedure completes very quickly. We will show that on our test machine (a midrange desktop circa 2010) pre-processing never takes more than several hundred milliseconds, even when the procedure is applied to large grid maps containing millions of nodes.

5.3.2 Advantages and Disadvantages

The main advantage of JPS+ is speed: instead of scanning the grid for jump points we can simply look up the set of jump point successors of any given location on the grid in constant time. On the other hand, preprocessing has two disadvantages (i) jump points need to be recomputed if the map changes (some local repair seems enough) and (ii) it introduces a substantive memory overhead: we need to keep for each node 8 distinct labels (one for each successor). In our implementation we use two bytes per label. The first 15 bits indicate the number of steps to reach the successor and the final bit distinguishes the successor as sterile or not.

We can use less memory if we store labels for intermediate locations instead of actual jump points: for example using one byte per label we could jump up to 127 steps at one time. The disadvantage of this approach is that more nodes may be expanded during search than strictly necessary. A hybrid algorithm that combines a single-byte database with a recursive jumping procedure is another memory-efficient possibility.

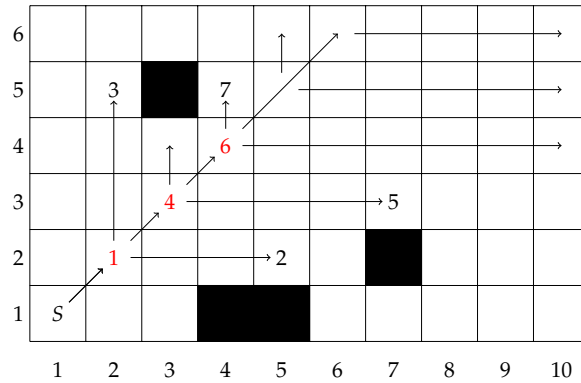


Figure 5.3: We prune all intermediate jump points (here nodes 1, 4 and 6) and instead generate their immediate successors (nodes 2, 3, 5 and 7) as children of the node from where initiated the jump (i.e., S). This allows us to jump from location $\langle 1, 1 \rangle$ to $\langle 6, 6 \rangle$ (and beyond) in a single operation.

5.4 Improved Pruning Rules

We have seen that JPS distinguishes between jump points that have at least one forced neighbour and those that have none. The former can be regarded as the grid equivalent of a visibility point: they are adjacent to at least one obstacle, and they are on at least one optimal path between two neighbours that are not mutually observable. If JPS prunes any of these nodes it is entirely possible that it will not return an optimal path. The second type of jump points are not adjacent to any obstacle. They are simply intermediate locations where the optimal path can change direction in order to reach the first type of jump point or the goal.

We argue that intermediate jump points can be safely pruned without affecting the optimality of Jump Point Search. Each intermediate jump point has at most three successors: the first is a jump point that is reachable horizontally, the second is a jump point that is reachable vertically and the third is the next intermediate jump point that can be reached without changing direction. When we prune an intermediate jump point we store its successors in a list and generate them in its stead. We apply this procedure recursively to any successors which are also intermediate jump points and terminate only when a dead-end is reached. Figure 5.3 shows an example.

To see that our strategy is optimality preserving we need only observe that for each intermediate jump point that is pruned the g -value of any of its successors remains unchanged. We simply generate these nodes earlier without first expanding any intermediary location. Once we have pruned such a node the parent of each newly orphaned successor becomes the starting location from where we initiated the jump. To extract a concrete path we simply walk from one jump point on the final path to the next in a diagonal-first way: i.e. we follow the octile heuristic but take all diagonal steps as early as possible. Such a path is guaranteed to be valid and thus obstacle-free.

Our pruning strategy is applied entirely online; it does not require any special

	StarCraft		Dragon Age: Origins		Dragon Age 2	
	Time (μ s)	Branches	Time (μ s)	Branches	Time (μ s)	Branches
JPS 2011	19.89	3.76	6.36	3.31	4.54	3.25
JPS (B)	1.85	3.76	0.93	3.31	0.85	3.25
JPS (B+P)	7.10	22.72	1.96	8.11	1.54	6.62
JPS+	0.38	3.76	0.21	3.31	0.20	3.25
JPS+ (P)	1.56	22.72	0.52	8.11	0.46	6.62

Table 5.2: A comparison of the average time required to expand (i.e. generate all successors of) one million random starting nodes from each map in our three benchmark sets. We aggregate the figures by benchmark and give results for the average time to expand a single node and the average branching factor. Times are given in microseconds.

data structures and does not store nor compute any additional information. It can be combined with other suggestions discussed in the current work and it allows us to jump further than we otherwise might. The cost is up to two additional open list operations for each node that we prune. Nevertheless, we will show empirically that pruning intermediate nodes from the search tree improves the performance of Jump Point Search.

5.5 Experimental Setup

We implemented Jump Point Search and a number of variants as described in this chapter. All source code is written from scratch in C++. For all our algorithms we have applied a number of simple optimisations that help to effectively stride through memory and reduce the effect of cache misses. This means that (i) we store the input map as a vector of bits, one bit for each node; (ii) we store the map twice, once in row major order and once in column major order; (iii) we pre-allocate memory in 256KB chunks.

We run experiments on a 2010 iMac running OSX 10.6.4. Our machine has a 2.93GHz Intel Core 2 Duo processor with 6MB of L2 cache and 4GB of RAM. For test data we selected three benchmark problem sets taken from real video games:

- **Dragon Age: Origins;** 44,414 instances across 27 grids of sizes ranging 665 to 1.39M nodes.
- **Dragon Age 2;** 68,150 instances across 67 grids of sizes ranging 1369 to 593K nodes.
- **StarCraft;** 29,970 instances across 11 grids of sizes ranging 262K to 786K nodes.

Instances are sampled from across all possible solution costs on each map. All instances have appeared in the 2012 Grid-based Path Planning Competition.

Algo	Database Size (MB)					Prep Time (seconds)				
	Min	Q1	Med	Q3	Max	Min	Q1	Med	Q3	Max
Dragon Age: Origins										
JPS+	0.02	0.08	1.15	5.76	21.65	0.00	0.00	0.02	0.17	0.47
SUB-S	0.00	0.02	0.32	1.64	5.92	0.00	0.00	0.00	0.01	0.04
SUB-TL	0.00	0.02	0.32	1.58	5.84	0.00	0.00	0.02	0.25	1.05
Dragon Age 2										
JPS+	0.03	0.54	1.74	4.16	9.26	0.00	0.01	0.04	0.09	0.20
SUB-S	0.01	0.15	0.49	1.13	2.55	0.00	0.00	0.00	0.01	0.01
SUB-TL	0.01	0.15	0.48	1.12	2.49	0.00	0.01	0.02	0.05	0.76
StarCraft										
JPS+	4.14	4.14	9.24	9.24	12.28	0.18	0.27	0.43	0.73	0.85
SUB-S	1.22	1.33	2.98	3.18	5.25	0.01	0.01	0.03	0.04	0.07
SUB-TL	1.18	1.25	2.79	2.87	4.20	0.27	1.14	2.84	9.39	23.77

Table 5.3: Preprocessing results for JPS+ and the two variants of SUB that we compare against. We present figures for the size of the resultant database (in MB) and the amount of precomputation time needed (in seconds). Columns Q1 and Q3 indicate values for the first and third quartile of each data set.

5.6 Comparison with JPS 2011

We first analyse the impact on JPS search performance for various combinations of our optimisation techniques: JPS (B), which adds block-based jumping, JPS (B + P) which combines block-based jumping with improved pruning, JPS+ which adds pre-processing and JPS+ (P) which combines pre-processing with improved pruning. To avoid confusion we will denote the original algorithm as JPS 2011.

We have previously observed that the bottleneck of the JPS 2011 algorithm is individual node expansion operations. In Table 5.2 we give results for the average time required to expand (i.e. generate all successors of) one million random starting nodes from each input map in each benchmark set. We find that block-based jumping and pre-processing jump points each improve average times by one and two orders of magnitude respectively vs. JPS 2011. Pruning intermediate jump points from the search tree increases the average branching factor by several times but the time-per-expansion is still much better. In Figure 5.4 we give a summary of search performance, in terms of time and node expansions, for all GPPC instances on each of our three benchmark sets. In each case and for each metric we consider relative improvement vs. JPS 2011. We show the spread of results after having assigned all test instances into buckets of similar cost. We observe that any of our alternative approaches is strictly faster than the original. Moreover, JPS (B) and JPS (B + P) have all the same advantages as JPS 2011: they are fast, optimal, online and require, in principle at least, no extra memory vs. JPS 2011 (recall that in practice we store a rotated copy of map twice to improve memory access patterns). A summary of pre-processing requirements is given in Table 5.3. Note that JPS+ and JPS+ (P) have the same requirements and are not listed separately.

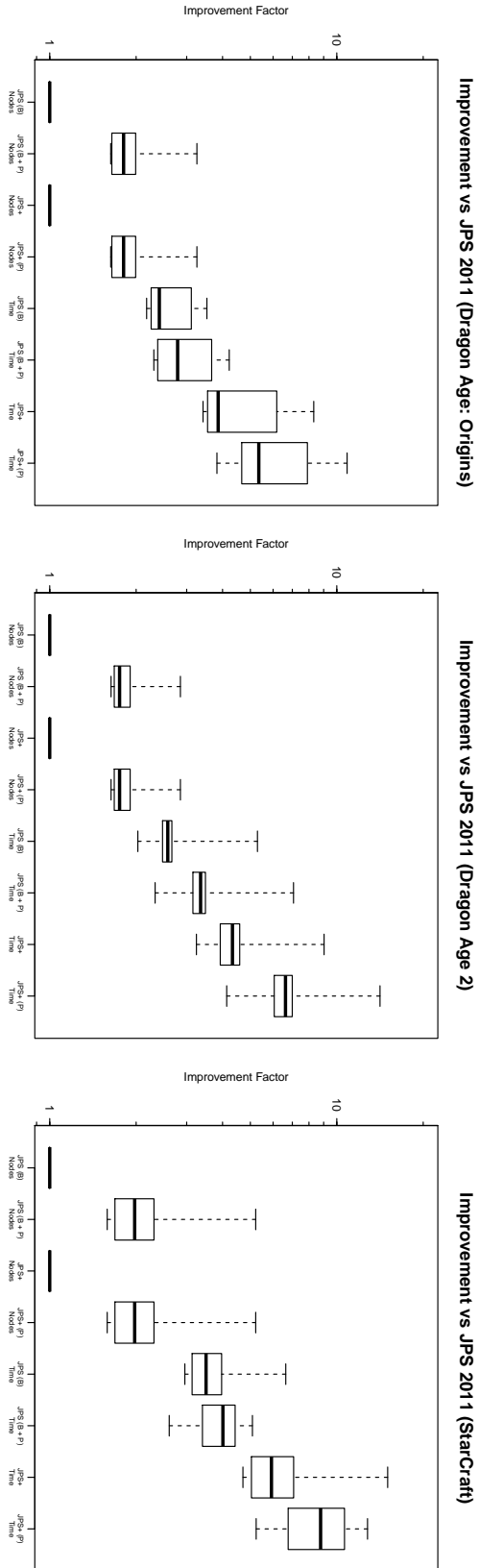


Figure 5.4: We measure the relative performance (or improvement factor) of each of our new JPS variants against the original. We consider two metrics: nodes expanded and search time. An improvement factor of 2 for search time means twice as fast; for node expansions it means half as many nodes expanded. Higher values are always better. Each boxplot shows the distribution of results. We give five figures; from bottom to top in each case: min, 1st quartile, median, 3rd quartile and max.

5.7 Comparison with SUB

SUB Uras et al. [2013] is a recent pathfinding technique from the literature. As one of the joint winners of the 2012 Grid-based Path Planning Competition (GPPC) SUB has been shown to be very fast and is considered the current state of the art. We compare against two variants described by the original authors: SUB-S (S for Simple) and SUB-TL (TL for Two Level). The former is guaranteed optimal while the latter is not. To evaluate SUB-S and SUB-TL we used the authors' original C++ implementation which we obtained from <http://gppc-2012.googlecode.com/svn/trunk/entries/SUB-a/>.

Table 5.3 compares the pre-processing requirements of SUB-S and SUB-TL with JPS+ (JPS+ (P) has identical requirements and is not shown). We observe that both JPS+ and SUB are able to pre-process most maps in well under a second and in most cases using less than 10MB of memory. A small number of notable exceptions arise for both JPS+ and SUB-TL. In Figure 5.5 we compare our four JPS variants with SUB-S and SUB-TL across all game map instances from the 2012 GPPC. We find that JPS (B) and JPS (B + P) are both competitive with, and often faster than, than SUB-S. Meanwhile, JPS+ (P) appears competitive with SUB-TL for a large set of instances. Across our three benchmarks, DA:O, DA2 and SC, we measured an improvement for JPS (B + P) vs. SUB-S in 92%, 84% and 89% of tested instances respectively. For JPS+ (P) vs. SUB-TL we measured an improvement in 43%, 77% and 68% of tested instances respectively.

5.8 Analysis

The results demonstrate the superiority of the approaches presented in this chapter. In JPS (B) and JPS (B + P) we have improved the performance of JPS 2011 by several factors all while retaining the same advantages inherent to the original algorithm: completeness, optimality and little-to-no memory overhead. Such results are remarkable as JPS 2011 has itself been shown to improve the performance of classical search algorithms such as A* by up to one order of magnitude and more.

We have shown with JPS+ and JPS+ (P) that further improvements are also possible. In our experiments we employ an offline pre-processing step together with a small amount of memory (10MB or less in most cases) in order to identify apriori all jump point successors of each grid node. The main advantage is performance: JPS+ and JPS+ (P) can improve the search times of JPS 2011 by up to one order of magnitude and more. The main disadvantage is that if the map changes the pre-processed database needs to be re-computed. We have shown that each such pre-computation can be performed very fast – usually on the order of tens or hundreds of milliseconds. Moreover the pre-computation can be easily parallelised over several time slices with JPS (B + P) employed as a fallback algorithm in the interim.

We have compared our JPS-based approaches against two variants of SUB: a very fast and very recent preprocessing-based pathfinding technique which was judged to be among the most performant entrants in the 2012 Grid Based Path Planning Competition. The first variant, SUB-S guarantees optimality; the second, SUB-TL,

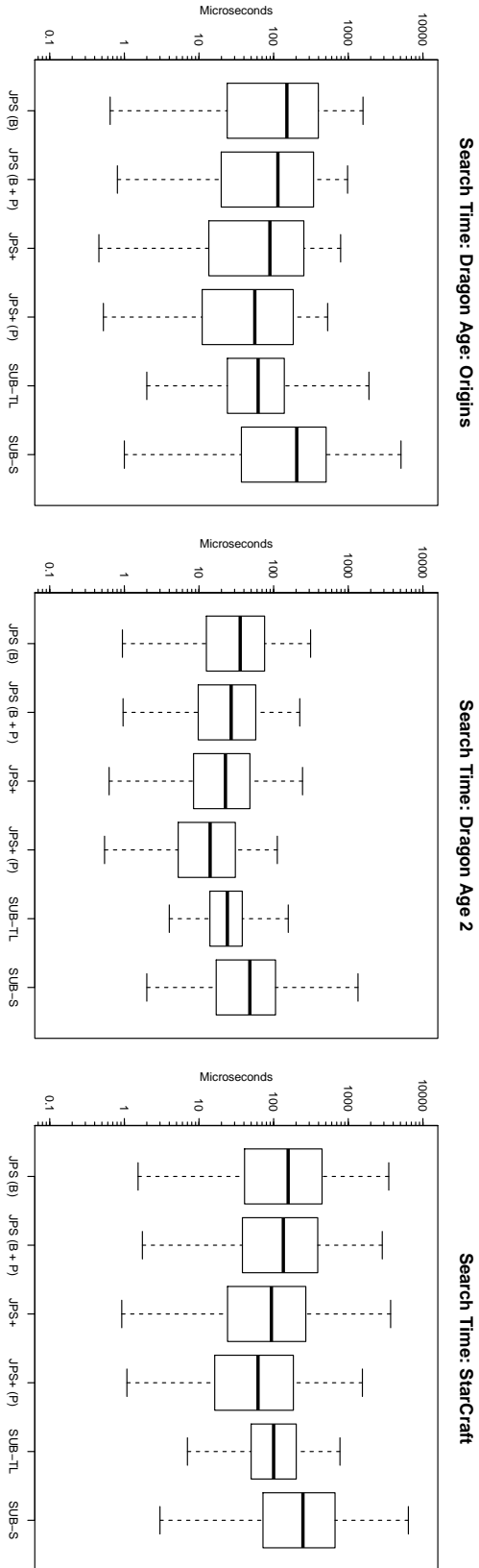


Figure 5.5: We compare the raw search time performance of our improved JPS variants (both online and offline) with two recent and very performant algorithms: simple subgoal graphs (SUB-S) and two-level subgoal graphs with local edge pruning (SUB-TL). All JPS variants and SUB-S are provably optimal. SUB-TL is not. Each boxplot shows the distribution of results. We give five figures, from bottom to top in each case: min, 1st quartile, median, 3rd quartile and max.

does not. We find that across most benchmark instances JPS (B) and JPS (B + P) are not only competitive with but faster than SUB-S. When we compare JPS+ (P) and SUB-TL we find the two algorithms often have complementary strengths: JPS+ (P) always has low pre-processing requirements, always finds the optimal path and is faster on a majority class of tested instances; SUB-TL has low space requirements and quickly finds optimal or near-optimal solutions to a large class of remaining instances.

5.9 Discussion

We study several techniques for improving Jump Point Search (JPS): a recent symmetry-breaking algorithm that facilitates fast pathfinding on grid maps such as those commonly found in computer games. Our first improvement allows us to detect jump points more efficiently by considering sets of nodes at one time (cf. one at a time). Our second improvement is a pre-processing strategy which computes and stores jump point successors for every node on the map. Our third improvement is a pruning strategy that avoids many node expansion operations. We speed up JPS by anywhere from several factors (in the purely online case) to over one order of magnitude (using preprocessing). Experiments on maps drawn from real computer games show that our work is competitive with and often faster than some very recent state-of-the-art grid-based pathfinding techniques.

Any-angle Pathfinding

In Chapter 3, Chapter 4 and Chapter 5 we examined approaches for improving the efficiency of pathfinding search on grid maps – a common setting for computer games and a domain which commonly appears in the AI literature. In addition to minimising search time, a related problem in such settings is computing paths that minimise travel distance and are aesthetically pleasing. For example: characters in a computer game must appear intelligent when navigating and should therefore prefer short realistic-looking paths. However, paths which are computed on a grid map, even optimal paths, necessarily restrict movement to the fixed set of locations defined by the grid.

In this chapter we describe Anya: a new algorithm which addresses this problem by computing *any-angle* paths that do not have such constraints. Anya operates on an input grid map but searches using intervals rather than the fixed points of the grid. From each such interval we select a representative point for which an f -cost is computed. We prove that our approach maintains A* expansion order and, unlike other similar approaches, always returns the shortest possible path. In the process we resolve an open question in the pathfinding community which has been standing since at least 2007 and which has been the subject of studies in the game development community for much longer.

The contributions in this chapter have appeared previously in [Harabor and Grastien, 2013].

6.1 Introduction

Any-angle pathfinding is a navigation problem which appears in robotics and computer video games. It involves finding a shortest path between an arbitrary pair of points on a two-dimensional grid map but asks that movement along the path is not artificially constrained to the points of the grid. Within the game development community a simple and popular solution exists known as *string pulling* [Pinter, 2001; Botea et al., 2004]. The idea is to compute a grid-optimal path in the first instance and smooth the result as part of a post-processing step that improves both its cost and aesthetic appeal. String pulling has two disadvantages: (i) it requires an additional computation beyond just finding a path (ii) it can only yield approximately shortest paths.

Theta* [Nash et al., 2007] and its variant [Nash et al., 2010] improve things by integrating post-processing into node expansion during search. Their idea involves updating a node's parent label following a successful line-of-sight check to a previously expanded ancestor node. Another approach, Block A* [Yap et al., 2011], avoids line-of-sight checks entirely by precomputing a database of exact costs between pairs of points in a localised area. Both Theta* and Block A* improve on the running time and solution quality of string pulling but neither guarantees optimality. Despite this, a number of exact solutions for the problem do exist.

Visibility Graphs [Lozano-Pérez and Wesley, 1979] solve a generalised form of any-angle pathfinding in time $O(n^2 \log_2 n)$. Their primary disadvantage is that storing such a graph requires $O(n^2)$ space as each of the n vertices can be adjacent to (i.e. visible from) every other vertex. Tangent Graphs [Liu and Arimoto, 1992] are a particularly efficient variant of this idea but space requirements remain worst-case quadratic. Other exact approaches are based on the Continuous Dijkstra [Mitchell et al., 1987] paradigm. The most efficient of these algorithms [Hershberger and Suri, 1999] involves a precomputation requiring $O(n \log_2 n)$ space and $O(n \log_2 n)$ time. The result is a planar subdivision of the map that can be used to find a shortest path in just $O(\log_2 n)$ time but only for queries originating at a fixed source.

In this work we discuss a new approach to any-angle pathfinding which addresses shortcomings associated with existing research. Our algorithm, Anya, bears some similarity with Continuous Dijkstra: instead of searching over individual states from the grid we consider contiguous sets of states together as an interval. From each interval we select a representative point that is used to derive an f -value for the set. Intervals are associated with corner points and projected from one row of the grid onto another until the goal is reached. Anya compares favourably with existing research: (i) it always finds a Euclidean shortest path between any two vertices on a grid map (ii) it does not rely on any precomputation (iii) it does not introduce any memory overheads beyond those required by a pathfinding algorithm such as A*.

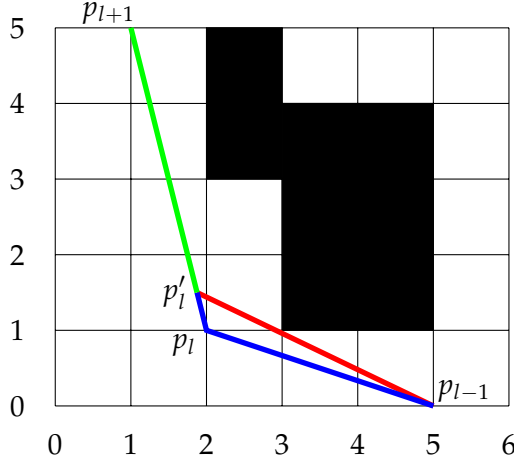


Figure 6.1: Illustration of Lemma 8. A similar observation has been previously made for geodesic paths [Mitchell et al., 1987].

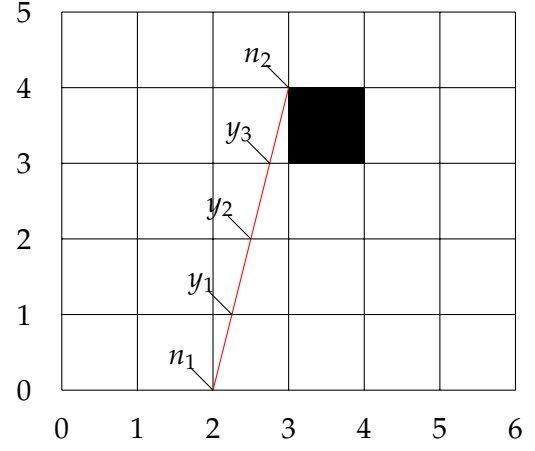


Figure 6.2: When pathfinding from n_1 to n_2 online algorithms such as Theta* only consider the discrete points of the grid and never any points y_i .

6.2 Preliminaries

A *grid* is a planar subdivision consisting of $W \times H$ square cells. Each cell is an open set of *interior* points which are all *traversable* or all *non-traversable*. The vertices associated with each cell are called the *discrete* points of the grid. Each such point p has a unique coordinate (x, y) where $x \in \{0, \dots, W\}$ and $y \in \{0, \dots, H\}$. Edges in the grid can be interpreted as open intervals of *intermediate* two points; each one representing a transition between two discrete points.

A discrete or intermediate point is *traversable* if it is adjacent to at least one traversable cell. Otherwise it is *non-traversable*. A discrete point which is common to exactly four adjacent cells is called an *intersection*. Any intersection where three of the adjacent cells are traversable and one is not is called a *corner*. We will say that two points are *visible* from one another if there exists a straight-line path connecting them which does not pass through any non-traversable point.

An *any-angle path* π is a sequence of points $\langle p_1, \dots, p_k \rangle$ where each p_i is visible from p_{i-1} and p_{i+1} . The *cost* of π is the cumulative distance between every successive pair of points $d(p_1, p_2) + \dots + d(p_{k-1}, p_k)$, where $d((x, y), (x', y')) = \sqrt{(x - x')^2 + (y - y')^2}$ is a uniform Euclidean distance metric. We will say $p_i \in \pi$ is a *turning point* if the segments (p_{i-1}, p_i) and (p_i, p_{i+1}) form an angle.

Lemma 8 *Any turning point in the optimal any-angle path between points p_1 and p_k is also a corner point.*

Proof: Assume an optimal any-angle path $\pi = \langle p_1, \dots, p_k \rangle$ that includes a turning point p_l ($l \notin \{1, k\}$) which is not a corner. We will prove that π is suboptimal which, by contradiction, will prove the lemma. If p_{l+1} is visible from p_{l-1} , then $\pi \setminus p_l$ is a

path which is strictly shorter than π . Hence π is suboptimal. If p_{l+1} is not visible from p_{l-1} , then let p'_l be a point from the segment $\langle p_l, p_{l+1} \rangle$ that (i) is visible from p_{l-1} and (ii) is different from p_l ; because p_l is not a corner, such a p'_l exists. Moreover, the subpath $\langle p_{l-1}, p'_l, p_{l+1} \rangle$ is strictly shorter than $\langle p_{l-1}, p_l, p_{l+1} \rangle$. Hence π is suboptimal. This case is illustrated in Figure 6.1 where the point p'_l is chosen as close as possible from the corner c . \square

Lemma 8 is interesting because it allows us to define any-angle paths as a sequence of straight lines which intersect only at discrete corner points.

6.3 Principle of Anya

Consider the any-angle instance depicted in Figure 6.2. The start point is $n_1 = (2, 0)$ and the target point is $n_2 = (3, 4)$. The best online algorithm for solving such problems, to date, is Theta* [Nash et al., 2007]; a method which computes an any-angle path by only considering the set of discrete points from the grid. Each time such a point is reached Theta* “pulls the string”. Thus when node n_2 is generated its g -value is not the cost of the grid-constrained path from n_1 to n_2 but rather the cost of the direct path $\langle n_1, n_2 \rangle$.

The problem with this approach is that the solution-cost estimate (or f -value), from a parent node to each of its successors, may not be monotonically increasing. The monotone condition is necessary to guarantee that an optimal solution, if one exists, is always found. For instance: Theta* can generate n_2 from the intermediate point $p = (3, 3)$. When p is expanded we have $f(p) = d(n_1, p) + h(p, n_2) = 4.16$. To satisfy the monotone condition we require that $f(n_2) \geq 4.16$. However Theta* computes $f(n_2) = d(n_1, n_2) + h(n_2, n_2) = 4.12$. Clearly p should be expanded after n_2 but in this case the opposite occurs. In order to avoid this mistake we would need to consider, in addition to the set of discrete points from the grid, all the points y_i shown in Figure 6.2. The problem is that the number of such points can be very large: each edge of the grid, together with its discrete endpoints, forms a $[0, 1]$ interval that can be intersected by the optimal path at any point $0 \leq \frac{w}{h} \leq 1$; here w (resp. h) is an integer in $\{0, \dots, W\}$ (resp. $\{0, \dots, H\}$). This is a set whose members are reducible to a Farey Sequence. For any given n (in our case $n = \max(W, H)$) the cardinality of the corresponding set of elements is known to be quadratic in n [Graham et al., 1989](Ch. 9). We are therefore motivated to consider an alternative approach: instead of evaluating each y_i node individually we will evaluate together all the nodes from the corresponding interval in which each y_i appears.

6.4 Algorithm

Definition 10 A grid interval I is a set of contiguous pairwise visible points from any row of the grid. Each interval is defined in terms of its endpoints a and b . With the possible exception of a and b , each interval contains only intermediate and discrete non-corner points.

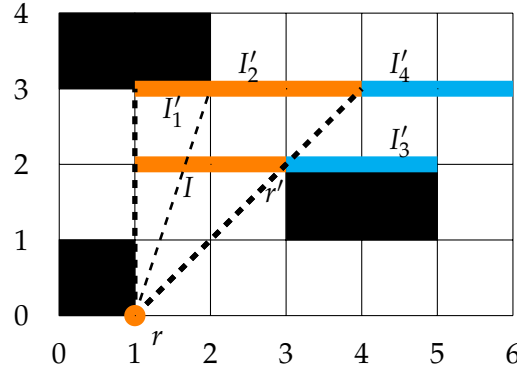


Figure 6.3: (I, r) has four successors: (I'_1, r) and (I'_2, r) which are observable and (I'_3, r') and (I'_4, r') which are not. Notice that intervals of traversable points exist left of I but the local path through I to each such point is not taut.

Identifying intervals is simple: any row of the grid can be naturally divided into maximally contiguous sets of traversable and non-traversable points. Each traversable set forms a tentative interval which we split, repeatedly, until the only corner points are found at a or b .

A significant advantage of Anya is that we construct intervals on-the-fly. This allows us to start answering queries immediately and for any discrete start-target pair. Similar algorithms (e.g. Continuous Dijkstra) require a preprocessing step before any queries can be answered and then only from a single fixed start point.

Definition 11 A search node (I, r) is a tuple where I is an interval and $r \notin I$ is a root point chosen s.t. each $p \in I$ is visible from r . The identity of r is always the most recent turning point on a path from the start point s to any $p \in I$. To represent the start node, set $I = [s]$ and assume r is a point located off the plane and visible only from s . The cost from r to s in this case is zero.

The successors of a search node n are identified by computing intervals over sets of traversable points from the same row of the grid as n and from rows immediately adjacent. We want to guarantee that each point in such a set can be reached from the root of n via a local path which is *taut*. Taut simply means that if we “pull” on the endpoints of the path we cannot make it any shorter.

Definition 12 (I', r') is a successor of (I, r) if each $p' \in I'$ is reached by a taut path $\langle r, p, p' \rangle$ that begins at r and passes through some $p \in I$, and r' is the last common point of these paths. Additionally, the subpath $\langle p, p' \rangle$ must not intersect any interval $J \neq I'$.

We begin with the set of traversable points that are visible from r through I and divide this set into $0 \leq k$ adjacent closed grid intervals. We will say that each such interval is *observable* and generate for each a corresponding successor node (I', r') with root $r' = r$.

Not all successors are observable. For example, the taut path from r can intersect I at an endpoint b which is also a corner point. In this case we reach a set of traversable points that are either adjacent to I or adjacent to the set of observable successors. Each such point is visible from $p = b$ but not from r . From this set of non-visible points we build a single half-open interval $I' = [a', b']$ s.t. I' is open at the endpoint closest to b . We will say I' is *non-observable* and generate a corresponding successor (I', r') with root $r' = b$. Figure 6.3 shows examples of both observable and non-observable successors.

To evaluate a search node $n = (I, r)$ we will select a point $p \in I$ which has a minimum f -value with respect to a target point t . We compute:

$$f(p) = g(r) + d(r, p) + h(p, t) \quad (6.1)$$

where $g(r)$ is the cost of the optimal path from the start point to the root, $d(r, p)$ is the straight line distance from r to p and $h(p, t)$ is an admissible heuristic function that lower-bounds the cost of reaching t from p ; here, we choose $h(p, t) = d(p, t)$.

Although each interval can contain a large number of points it is easy to identify the one with minimum f -value, thanks to the following two lemmas.

Lemma 9 *Let t and r be two points and let I be an interval such that the row of I is between the rows of t and r . Then the point p of I that minimizes the f value is the closest point of I to the intersection of (t, r) with the row of I .*

Proof: If (t, r) intersects I in p_i , then the minimum value of $d(r, p) + h(p, t)$ is $d(r, t)$ which is obtained by choosing $p = p_i$ (by the triangle inequality). Otherwise choose p as the end point of I on the side where (r, t) intersects the row of I . \square

If the precondition of Lemma 9 is not satisfied, it is possible to replace t by its mirrored version t' through I as their distance $d(p, t)$ is the same while t' does satisfy the precondition.

Lemma 10 *The mirrored point t' of target t through interval I is such that $d(p, t) = d(p, t')$ for all $p \in I$.*

Lemma 10 is a trivial geometrical result. Both lemmas are illustrated on Figure 6.4.

The algorithm terminates when we expand a node (I, r) s.t. $t \in I$. By Lemma 9 and 10 the f -value of this interval is guaranteed to be minimum with respect to t . To extract a path we simply follow parent pointers until we reach the node containing the start point. The root points associated with the search nodes we encounter are the turning points on the optimal any-angle path from s to t .

6.5 Correctness and Optimality

To prove correctness and optimality, we show that the optimal path appears in the search space and that the f function defined in the previous section guarantees that

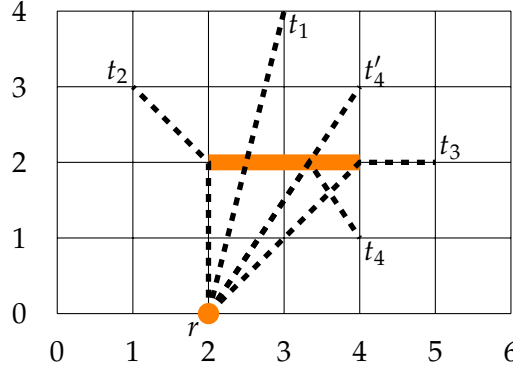


Figure 6.4: An illustration of Lemmas 9 and 10. The points t_1 and t'_4 correspond to the case where the line intersects the interval; t_2 and t_3 where it does not; t_4 where the mirrored target t'_4 must be used.

the search node corresponding to the optimal path will be expanded before any node associated with a sub-optimal path to the target.

Theorem 4 *For any point p that appears on a row, there exists a node in the search tree that corresponds to the optimal path from s to p if such a path exists.*

Proof: By induction. Consider an optimal path $\pi_k = \langle p_1, \dots, p_k \rangle$ where $s = p_1$ and p_{k-1} is either a point one row apart from p_k (similar to a y_i point mentioned in Figure 6.2) or a corner point on the same row. By induction, there is a node (I, r) in the search tree that represents the optimal path to $p_{k-1} \in I$. Now following Definition 12, there is a node (I', r') that is a successor of (I, r) such that $p_k \in I'$ while $r' = r$ if p_k is visible from r and $r' = p_{k-1}$ otherwise, and the node represents the optimal path. \square

We now assume that the search space is explored by expanding the queued node with smallest f value as defined in Equation (6.1).

Theorem 5 *The first expanded node that contains the target t corresponds to the optimal path to t .*

Proof: First we notice that the f value of a node is indeed the minimal value of all the nodes in the interval, which means that f is an under estimate of the actual cost to the target. Second we notice that, given a search node (I, r) and its successor (I', r') , for each point $p' \in I'$, the f value of p' is not smaller than the f value of some point $p \in I$ ($p = r'$ if $r' \neq r$; p is the intersection of I and (r, p') otherwise); the f function is therefore monotonically increasing. Finally, the f function of a search node (I, r) is the cost of the path if $t \in I$. Hence the f function of the nodes representing a sub-optimal path to t will eventually exceed the optimal path distance, while the f function of the nodes representing the optimal path will always remain under this value. \square

6.6 Discussion

We study any-angle pathfinding: a problem commonly found in the areas of robotics and computer games. It involves finding a shortest path between two points in a grid but asks that the path is not artificially constrained to the points of the grid. We give a new algorithm for this problem: Anya. Our approach works by representing sets of points from the grid as intervals and considers all points from an interval together at the same time. From each interval we select a representative point which has a minimum f -value. We show that this approach is both complete and optimal. Moreover, it requires no preprocessing and relies on no special data structures during search. Any-angle pathfinding has received significant attention from the AI and Game Development communities but until now it has been an open question whether any optimal online algorithm exists. Anya answers this question in the affirmative.

Conclusion

Pathfinding is a problem that has received significant attention from both researchers and industrial practitioners. Despite numerous studies, many types of pathfinding problems remain challenging to solve. In the case of mobile robots and computer games, where the world is often represented as a grid, two significant research challenges are: (i) grids are dense and contain many nodes which makes grid-optimal pathfinding difficult; (ii) grid-optimal paths are often not Euclidean-optimal which is undesirable from an efficiency perspective and also from an aesthetic perspective.

To address the first of these challenges we have developed Rectangular Symmetry Reduction (RSR) and Jump Point Search (JPS): two novel and performant pathfinding algorithms that speed up grid-optimal search by identifying and eliminating from consideration paths that are symmetric permutations of each other. JPS and RSR can improve the run-time performance of classical algorithms such as A* by anywhere from several factors to one and sometimes two orders of magnitude – all without extensive pre-processing or the introduction of large memory overheads.

To address the second research challenge we have developed Anya: a novel approach for computing Any-angle (i.e. Euclidean-optimal) paths on a grid map. Previous attempts at computing such paths have either been online methods that yield only approximate solutions or exact methods that require an offline pre-computation step. Anya is the first algorithm that, to the best of our knowledge, combines the advantages of both approaches.

Our work represents a significant advancement of the state-of-the-art. With RSR and JPS we have shown that optimal pathfinding on many challenging grid-based benchmarks from the literature, such as maps drawn from real computer games, can be reduced to problems that require very little effort to solve. With Anya we have resolved an open question from the areas of Artificial Intelligence and Game Development: is there an Any-angle pathfinding algorithm which is both online and optimal? The answer is yes.

7.1 Summary

We study optimal pathfinding in discrete and static two-dimensional environments. Our target applications include mobile robot navigation and pathfinding in computer games, both settings where the world is commonly represented as a grid. Our objectives throughout this thesis have been to: (i) find the shortest path; (ii) as quickly as possible; (iii) as economically as possible with respect to available resources such as memory and pre-computation time.

Two significant challenges that arise when trying to find optimal paths on grid domains are: (i) grids are dense and finding a grid-optimal path often means expanding very many nodes; (ii) paths which are grid-optimal are often not Euclidean-optimal which is undesirable from both an efficiency perspective and an aesthetic perspective. We address the first of these challenges in Chapter 3, Chapter 4 and Chapter 5. We address the second of these challenges in Chapter 6.

7.1.1 Contributions to Symmetry Breaking in Pathfinding Search

In Chapter 3 we identify one significant factor that makes pathfinding in grids challenging: the existence of equivalent permutations (i.e. symmetries) for many shortest paths in a grid. We formalise this idea by defining concretely what it means for two paths to be symmetric. We then give a description of and experimental evaluation for an offline symmetry-breaking approach called Rectangular Symmetry Reduction (RSR). We show that RSR can improve the performance of the classical A* algorithm by several factors in most of our benchmark domains and up to one order of magnitude in some other cases. We then compare RSR with two contemporary pathfinding algorithms, Swamps [Pochter et al., 2010] and the Portal Heuristic [Goldenberg et al., 2010]. We find that RSR has complementary strengths with these approaches and we identify many instances where RSR dominates convincingly.

In Chapter 4 we present Jump Point Search (JPS): an alternative symmetry breaking procedure which selectively expands only certain nodes on a grid map called *jump points*. JPS is recursive in nature, guaranteed optimal and applied entirely on-line. It requires no pre-processing and introduces no memory overheads. JPS can improve the performance of standard A* by one order of magnitude and more in each of our benchmark domains. It compares favourably with and convincingly dominates contemporary optimal speedup techniques such as Swamps. It is also competitive with the non-optimal hierarchical pathfinding method HPA*. JPS is unique in the pathfinding literature in that it has very few disadvantages: it is simple, yet highly effective; it preserves optimality, yet requires no extra memory; it is fast, yet requires no pre-processing. Like RSR, JPS is easily combined with many existing speedup techniques from the literature. We are unaware of any other algorithm which has all these features.

In Chapter 5 we present several enhancements that further improve the performance of Jump Point Search. The first enhancement allows JPS to prune from consideration certain types of nodes known as intermediate jump points. The second en-

Algorithm	Preprocessing Time	Memory	Max Speedup
4-Connected Uniform Cost Grid Maps			
4-ERR	< 1s	$O(V)$	2-3x
4-ERR + Online Pruning	< 1s	$O(V)$	3-4x
4-ERR + Perimeter Reduction	< 1s	$O(V)$	3-20x
RSR	< 1s	$O(V)$	3-20x
8-Connected Uniform Cost Grid Maps			
RSR	< 1s	$O(V)$	3-35x
JPS	0	0	15-35x
JPS (Blocks)	0	0*	37-56x
JPS (Blocks + Pruning)	0	0*	39-64x
JPS+	< 1s	$O(V)$	72-99x
JPS+ (Pruning)	< 1s	$O(V)$	108-149x

Table 7.1: A summary overview of the symmetry breaking algorithms presented in this thesis. We compare preprocessing time (average, measured in seconds), memory overhead (worst-case) and maximum speedup (average, relative to A*). Performance RSR and JPS are measured different (but broadly similar and thus comparable) sets of benchmark problems, all of which are taken from the literature. **NB:** * In the case of JPS (Blocks) and JPS (Blocks + Pruning) we store a redundant copy of the map to allow for faster memory operations.

hancement allows JPS to consider “blocks” of nodes at one time when breaking symmetries (cf. individual nodes). The third enhancement is an offline pre-processing technique that speeds up search by identifying jump points apriori. We show in an empirical evaluation that these improvements have a strong positive effect on Jump Point Search, improving an already performant algorithm by anywhere from several factors to more than one order of magnitude. We compare our approach with two variants of SUB [Uras et al., 2013]: a recent and very fast pathfinding approach that also breaks path symmetries but only during an offline pre-processing step. We find that JPS and SUB have complementary strengths and we identify many instances where one or more of our enhanced JPS variants are faster.

Table 7.1 gives a summary comparison for each of the algorithms presented in Chapter 3, Chapter 4 and Chapter 5. The main features in each case is an emphasis on improving the running time of pathfinding search while requiring little (and in some cases no) preprocessing and additional memory, beyond what is necessary to store the search graph. We have shown throughout this thesis that in empirical comparisons RSR, JPS and their variants are competitive with and often better than competing algorithms appearing contemporaneously in the literature.

7.1.2 Contributions to Any-Angle Pathfinding

In Chapter 6 we turn our attention to Any-angle pathfinding. This is a problem that takes as input two points from a grid and asks for a Euclidean-optimal path connecting them. We describe Anya: a novel approach to Any-angle pathfinding which involves considering sets of states together as an interval. From each inter-

val we select a single representative point which we use to compute an f -value for the entire set. During search intervals are projected from one row of the grid to another until the target is reached. We give theoretical results showing Anya always finds a Euclidean-optimal path if one exists. Moreover, Anya does so entirely online and without any pre-processing. We are unaware of another other approach which guarantees these desirable characteristics.

7.2 Future Work

We discuss future work in terms of our two main contributions: Symmetry breaking in pathfinding search and optimal approaches for Any-angle pathfinding.

7.2.1 Symmetry Breaking

One interesting direction for further work involves stronger pruning rules for Jump Point Search on both weighted and unweighted grid maps. In Chapter 4.6 we have given one simple suggestion for adapting the existing algorithm to weighted grids: simply stop whenever any neighbouring tile has a different terrain type than the current node. A more general approach is to explicitly calculate the weighted cost of local paths: from the parent node to each neighbouring node. The advantage is that we do not necessarily need to stop each time the path transitions from one weighted region to another. We need to be careful however to balance the additional time required to explicitly compute the costs of each local path vs. the time savings from performing fewer node expansions.

In the case of unweighted grid maps we believe stronger symmetry breaking rules are possible which will allow us to jump over at least some of the remaining nodes currently expanded by Jump Point Search. For example we might consider pruning a node n if all its successors have the same f -value as n ; i.e. we can try to keep jumping as long as we are heading in the direction of the goal. A stronger variant of this idea is to keep jumping as long as we are heading in the same direction to the one used to reach n (or in a new direction which is a component of the original). For example, suppose we reach node n by jumping in direction \vec{d} . If all the successors of n can be reached by also jumping in direction \vec{d} or in another direction which is a component of \vec{d} then we can consider pruning n and continuing to recurse in each of these directions. It is likely that this procedure will increase in the branching factor at n but we posit that fewer node expansions are required overall because we do not necessarily need to stop each time the path turns. A similar idea is described in [Uras et al., 2013] where the authors also exploit path symmetries to speed up search – though their approach is offline only.

Another interesting direction for future work is to apply Jump Point Search to other types of grids, such as hexagons or tetes [Yap, 2002]. We propose to achieve this by developing a series of pruning rules analogous to those given for square grids. As the branching factor on these domains is lower than square grids, we posit that jump points could be even more effective than we have observed for square grids.

Finally, synthesising JPS with complementary speedup techniques from the literature appears to be a promising line of research. For example JPS could be used to speed up optimal search in an implementation of Swamps [Pochter et al., 2010] or to speed up start and target insertion in an approximate hierarchical pathfinder such as HPA* [Botea et al., 2004].

7.2.2 Any-angle Pathfinding

An obvious direction for further work is a concrete implementation of Anya together with an empirical evaluation. This is a topic of current research.

Another interesting direction is to generalise the theoretical results from Anya to the problem of finding Euclidean-optimal paths in continuous planar environments with polygonal obstacles – rather than the square obstacles of the type found in grid maps. Such a generalisation could be achieved by (i) using a grid to tessellate the environment and (ii) testing if any obstacles intersect the current interval as we move it from one row of the grid to another. Each time we detect an intersection we generate a successor interval whose y-axis is the same as the obstacle rather than the y-axis of the next row.

Bibliography

- ABRAHAM, I.; DELLING, D.; FIAT, A.; GOLDBERG, A. V.; AND WERNECK, R. F., 2013, Revised May 2014. Highway dimension and provably efficient shortest path algorithms. Technical Report MSR-TR-2013-91.
- ABRAHAM, I.; FIAT, A.; GOLDBERG, A. V.; AND WERNECK, R. F., 2010. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '10* (Austin, Texas, 2010), 782–793. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.
- ALKHAZRAJI, Y.; WEHRLE, M.; MATTMÜLLER, R.; AND HELMERT, M., 2012. A stubborn set algorithm for optimal planning. In *ECAI*, 891–892.
- ANDERSON, K., 2010. Additive heuristic for four-connected gridworlds. In *SOCS*.
- ANTSFELD, L.; HARABOR, D. D.; KILBY, P.; AND WALSH, T., 2012. Transit routing on video game maps. In *AIIDE*.
- AURENHAMMER, F., 1991. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23, 3 (Sep. 1991), 345–405.
- BARRAQUAND, J.; KAVRAKI, L.; LATOMBE, J.-C.; MOTWANI, R.; LI, T.-Y.; AND RAGHAVAN, P., 1997. A random sampling scheme for path planning. *Int. J. Rob. Res.*, 16, 6 (Dec. 1997), 759–774.
- BAST, H.; FUNKE, S.; AND MATIJEVIC, D., 2006. Transit ultrafast shortest-path queries with linear-time preprocessing. In *In 9th DIMACS Implementation Challenge*.
- BAST, H.; FUNKE, S.; MATIJEVIC, D.; SANDERS, P.; AND SCHULTES, D., 2007. In transit to constant time shortest-path queries in road networks. In *ALLENEX*.
- BJÖRNSSON, Y.; ENZENBERGER, M.; HOLTE, R. C.; AND SCHAEFFER, J., 2005. Fringe search: Beating A* at pathfinding on game maps. In *CIG'05*, 125–132.
- BJÖRNSSON, Y. AND HALLDÓRSSON, K., 2006. Improved heuristics for optimal pathfinding on game maps. In *AIIDE*, 9–14.
- BOHLIN, R. AND KAVRAKI, L. E., 2000. Path planning using lazy prm. In *ICRA*, 521–528.
- BOLANCA, M., 2009. Achieving fast and optimal pathfinding through the use of macro steps in obstacle free areas. Australian National University Honours Thesis.

- BOTEA, A.; MÜLLER, M.; AND SCHAEFFER, J., 2004. Near optimal hierarchical path-finding. *J. Game Dev.*, 1, 1 (2004), 7–28.
- BULITKO, V.; BJÖRNSSON, Y.; AND LAWRENCE, R., 2010. Case-based subgoalting in real-time heuristic search for video game pathfinding. *J. Artif. Intell. Res. (JAIR)*, 39 (2010), 269–300.
- BURCH, N. AND HOLTE, R. C., 2012. Automatic move pruning revisited. In *SOCS*.
- CAIN, T., 2002. Practical optimizations for A*. (2002), 146–152.
- CAZENAVE, T., 2006. Optimizations of data structures, heuristics and algorithms for path-finding on maps. In *CIG*, 27–33.
- CHAMPANDARD, A., 2009. Modern pathfinding techniques. In *AIGameDev.com*.
- CHOSSET, H.; LYNCH, K. M.; HUTCHINSON, S.; KANTOR, G.; BURGARD, W.; KAVRAKI, L. E.; AND THRUN, S., 2005. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press.
- CRAWFORD, J. M.; GINSBERG, M. L.; LUKS, E. M.; AND ROY, A., 1996. Symmetry-breaking predicates for search problems. In *KR*, 148–159.
- DAVIS, I. L., 2000. Warp speed: Path planning for Star Trek Armada. In *AAAI Spring Symposium (AIIDE)*, 18–21.
- DECHTER, R. AND PEARL, J., 1985. Generalized best-first search strategies and the optimality of A*. *J. ACM*, 32, 3 (1985), 505–536.
- DEMYEN, D. AND BURO, M., 2006. Efficient triangulation-based pathfinding. In *AAAI*, 942–947.
- DIJKSTRA, E., 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 (1959), 269–271.
- EMERSON, E. A. AND SISTLA, A. P., 1996. Symmetry and model checking. *Formal Methods in System Design*, 9, 1/2 (1996), 105–131.
- ESPARZA, J. AND HELJANKO, K., 2008. *Unfoldings*. EATCS Monographs in Theoretical Computer Science. Springer.
- FELNER, A. AND STURTEVANT, N. R., 2009. Abstraction-based heuristics with true distance computations. In *SARA*.
- FERGUSON, D. AND STENTZ, A., 2005. Field D*: An interpolation-based path planner and replanner. In *ISRR*, 239–253.
- FINKEL, R. AND BENTLEY, J., 1974. Quad trees a data structure for retrieval on composite keys. *Acta Informatica*, 4 (1974), 1–9.

-
- FOX, M. AND LONG, D., 1999. The detection and exploitation of symmetry in planning problems. In *IJCAI*, 956–961.
- FOX, M. AND LONG, D., 2002. Extending the exploitation of symmetries in planning. In *AIPS*, 83–91.
- FUKUNAGA, A. S., 2008. Integrating symmetry, dominance, and bound-and-bound in a multiple knapsack solver. In *CPAIOR*, 82–96.
- GEISBERGER, R.; SANDERS, P.; SCHULTES, D.; AND DELLING, D., 2008. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, 319–333.
- GERAERTS, R. AND OVERMARS, M. H., 2005. Creating small roadmaps for solving motion planning problems. In *EEE International Conference on Methods and Models in Automation and Robotics*, 531–536.
- GODEFROID, P., 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, vol. 1032 of *Lecture Notes in Computer Science*. Springer. ISBN 3-540-60761-7.
- GODEFROID, P. AND WOLPER, P., 1991. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *CAV*, 332–342.
- GODEFROID, P. AND WOLPER, P., 1993. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2, 2 (1993), 149–164.
- GOLDBERG, A.; KAPLAN, H.; AND WERNECK, R. F., 2006. Reach for A*: Efficient point-to-point shortest path algorithms. In *ALLENEX*.
- GOLDBERG, A. V. AND HARRELSON, C., 2005. Computing the shortest path: A* search meets graph theory. In *SODA*, 156–165.
- GOLDENBERG, M.; FELNER, A.; STURTEVANT, N.; AND SCHAEFFER, J., 2010. Portal-based true-distance heuristics for path finding. In *SoCS*.
- GRAHAM, R. L.; KNUTH, D. E.; AND PATASHNIK, O., 1989. *Concrete mathematics - a foundation for computer science*. Addison-Wesley.
- HARABOR, D., 2011. Graph pruning and symmetry breaking on grid maps. In *IJCAI Doctoral Consortium*.
- HARABOR, D. AND BOTEÀ, A., 2008. Hierarchical path planning for multi-size agents in heterogeneous environments. In *CIG'08*.
- HARABOR, D. AND BOTEÀ, A., 2010. Breaking path symmetries in 4-connected grid maps. In *AIIDE*, 33–38.
- HARABOR, D. AND GRASTIEN, A., 2012. The jps+ pathfinding system. In *SoCS*.

- HARABOR, D. AND GRASTIEN, A., 2011. Online graph pruning for pathfinding on grid maps. In *AAAI*.
- HARABOR, D. D.; BOTE, A.; AND KILBY, P., 2011. Path symmetries in undirected uniform-cost grids. In *SARA*.
- HARABOR, D. D. AND GRASTIEN, A., 2013. An optimal any-angle pathfinding algorithm. In *ICAPS*.
- HARABOR, D. D. AND GRASTIEN, A., 2014. Improving jump point search. In *ICAPS*.
- HART, P. E.; NILSSON, N. J.; AND RAPHAEL, B., 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4 (1968), 100–107.
- HASLUM, P. AND GEFFNER, H., 2000. Admissible heuristics for optimal planning. In *AIPS*, 140–149.
- HELMERT, M. AND RÖGER, G., 2008. How good is almost perfect? In *AAAI*, 944–949.
- HERNÁNDEZ, C. AND BAIER, J. A., 2011. Fast subgoaling for pathfinding via real-time search. In *ICAPS*.
- HERSHBERGER, J. AND SURI, S., 1999. An optimal algorithm for Euclidean shortest paths in the plane. *SIAM J. Comput.*, 28, 6 (1999), 2215–2256.
- HIGGINS, D., 2002. How to achieve lightning fast A*. (2002), 133–145.
- KALLMANN, M., 2010. Shortest paths with arbitrary clearance from navigation meshes. In *Proceedings of the Eurographics / SIGGRAPH Symposium on Computer Animation (SCA)* (Madrid, Spain, 2010).
- KAVRAKI, L. AND LATOMBE, J. C., 1994. Randomized preprocessing of configuration space for fast path planning. In *IEEE Conference on Robotics and Automation (ICRA'94)*, 2138–2146.
- KORF, R. E., 1990. Real-time heuristic search. *Artif. Intell.*, 42, 2-3 (1990), 189–211.
- KORF, R. E. AND TAYLOR, L. A., 1996. Finding optimal solutions to the twenty-four puzzle. In *AAAI/IAAI, Vol. 2*, 1202–1207.
- LATOMBE, J., 1991. *Robot Motion Planning*. Kluwer Academic Publishers.
- LAVALLE, S. M., 1998. Rapidly-exploring random trees: A new tool for path planning. Technical report.
- LIU, Y.-H. AND ARIMOTO, S., 1992. Path planning using a tangent graph for mobile robots among polygonal and curved obstacles. *Int. J. Robot Research*, 11 (August 1992), 376–382. doi:10.1177/027836499201100409.

-
- LOZANO-PÉREZ, T. AND WESLEY, M. A., 1979. An algorithm for planning collision-free paths among polyhedral obstacles. *Commun. ACM*, 22, 10 (1979), 560–570.
- MAZURKIEWICZ, A. W., 1986. Trace theory. In *Advances in Petri Nets*, 279–324.
- MITCHELL, J. S. B., 1997. *Shortest Paths and Networks*, chap. 24, 445–466. CRC Press, New York.
- MITCHELL, J. S. B.; MOUNT, D. M.; AND PAPADIMITRIOU, C. H., 1987. The discrete geodesic problem. *SIAM J. Comput.*, 16, 4 (1987), 647–668. doi:<http://dx.doi.org/10.1137/0216045>.
- MOORE, E. F., 1959. The shortest path through a maze. (1959).
- MUÑOZ, P. AND RODRÍGUEZ-MORENO, M. D., 2012. Improving efficiency in any-angle path-planning algorithms. In *IEEE Conf. of Intelligent Systems*, 213–218.
- NASH, A.; DANIEL, K.; KOENIG, S.; AND FELNER, A., 2007. Theta*: Any-angle path planning on grids. In *AAAI*, 1177–1183.
- NASH, A.; KOENIG, S.; AND LIKHACHEV, M., 2009. Incremental Phi*: Incremental any-angle path planning on grids. In *IJCAI*, 1824–1830.
- NASH, A.; KOENIG, S.; AND TOVEY, C. A., 2010. Lazy Theta*: Any-angle path planning and path length analysis in 3d. In *AAAI*.
- OVERMAN, W. T. AND CROCKER, S. D., 1982. Verification of concurrent systems: Function and timing. In *PSTV*, 401–409.
- PARK, J.-S.; PENNER, M.; AND PRASANNA, V. K., 2004. Optimizing graph algorithms for improved cache performance. *IEEE Trans. Parallel Distrib. Syst.*, 15, 9 (Sep. 2004), 769–782.
- PEARL, J., 1984. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley, Boston, MA, USA.
- PINTER, M., 2001. Toward more realistic pathfinding. *Game Developer Magazine*, 8, 4 (2001).
- POCHTER, N.; ZOHAR, A.; AND ROSENSCHEIN, J. S., 2009. Using swamps to improve optimal pathfinding. In *AAMAS*.
- POCHTER, N.; ZOHAR, A.; AND ROSENSCHEIN, J. S., 2011. Exploiting problem symmetries in state-based planners. In *AAAI*.
- POCHTER, N.; ZOHAR, A.; ROSENSCHEIN, J. S.; AND FELNER, A., 2010. Search space reduction using swamp hierarchies. In *AAAI*.
- POHL, I., 1977. Practical and theoretical considerations in heuristic search algorithms. In *Machine Intelligence 8* (Eds. E. W. ELCOCK AND D. MICHIE). Ellis Horwood Ltd and John Wiley & Sons.

- RABIN, S., 2000. A* speed optimizations. In *Game Programming Gems*, 272–287. Charles River Media.
- RAYNER, D. C.; BOWLING, M. H.; AND STURTEVANT, N. R., 2011. Euclidean heuristic optimization. In *AAAI*.
- RINTANEN, J., 2003. Symmetry reduction for sat representations of transition systems. In *ICAPS*, 32–41.
- RONEY-DOUGAL, C. M.; GENT, I. P.; KELSEY, T.; AND LINTON, S., 2004. Tractable symmetry breaking using restricted search trees. In *ECAI*, 211–215.
- RUSSELL, S. AND NORVIG, P., 2003. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edn.
- SAMET, H. AND WEBBER, R. E., 1985. Storing a collection of polygons using quadrees. *ACM Trans. Graph.*, 4, 3 (Jul. 1985), 182–222.
- SANDERS, P., 1999. Fast priority queues for cached memory. In *ALLENEX*, 312–327.
- SANDERS, P. AND SCHULTES, D., 2005. Highway hierarchies hasten exact shortest path queries. In *ESA*, 568–579.
- SANDERS, P. AND SCHULTES, D., 2006. Engineering highway hierarchies. In *ESA*, 804–816.
- SANKARANARAYANAN, J.; ALBORZI, H.; AND SAMET, H., 2005. Efficient query processing on spatial networks. In *GIS*, 200–209.
- SCHULTES, D. AND SANDERS, P., 2007. Dynamic highway-node routing. In *WEA*, 66–79.
- SHARON, G.; STURTEVANT, N. R.; AND FELNER, A., 2013. Online detection of dead states in real-time agent-centered search. In *SOCS*.
- ŠIŠLÁK, D.; VOLF, P.; AND PĚCHOUČEK, M., 2009a. Accelerated A* path planning. In *AAMAS* (2), 1133–1134.
- ŠIŠLÁK, D.; VOLF, P.; AND PĚCHOUČEK, M., 2009b. Accelerated A* trajectory planning: Grid-based path planning comparison. In *4th Workshop on Planning and Plan Execution for Real-World Systems*.
- SNOOK, G., 2000. Simplified 3d movement and. pathfinding using navigation meshes. In *Game Programming Gems*, 288–304. Charles River Media.
- STOUT, B., 1996. Smart moves: Intelligent pathfinding. *Game Developer Magazine*, October (1996), 28–35.
- STURTEVANT, N., 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, (2012).

-
- STURTEVANT, N. R., 2007. Memory-efficient abstractions for pathfinding. In *AIIDE*, 31–36.
- STURTEVANT, N. R. AND BULITKO, V., 2011. Learning where you are going and from whence you came: h- and g-cost learning in real-time heuristic search. In *IJCAI*, 365–370.
- STURTEVANT, N. R.; BULITKO, V.; AND BJÖRNSSON, Y., 2010a. On learning in agent-centered search. In *AAMAS*, 333–340.
- STURTEVANT, N. R.; BULITKO, V.; AND BJÖRNSSON, Y., 2010b. On learning in agent-centered search. In *AAMAS*, 333–340.
- STURTEVANT, N. R. AND BURO, M., 2005. Partial pathfinding using map abstraction and refinement. In *AAAI*, 1392–1397.
- STURTEVANT, N. R.; FELNER, A.; BARRER, M.; SCHAEFFER, J.; AND BURCH, N., 2009. Memory-based heuristics for explicit state spaces. In *IJCAI*, 609–614.
- STURTEVANT, N. R. AND GEISBERGER, R., 2010. A comparison of high-level approaches for speeding pathfinding. In *AIIDE*, 76–82.
- SUN, X.; YEOH, W.; CHEN, P.-A.; AND KOENIG, S., 2009. Simple optimization techniques for A*-based search. In *AAMAS* (2), 931–936.
- SURAZHISKY, V.; SURAZHISKY, T.; KIRSANOV, D.; GORTLER, S. J.; AND HOPPE, H., 2005. Fast exact and approximate geodesics on meshes. In *SIGGRAPH*, 553–560. doi: 10.1145/1186822.1073228.
- TAYLOR, L. A. AND KORE, R. E., 1993. Pruning duplicate nodes in depth-first search. In *AAAI*, 756–761.
- TOZOUR, P., 2002. Ai game programming wisdom 2. In *Search Space Representations* (Ed. S. RABIN), 85–113. Charles River Media.
- URAS, T.; KOENIG, S.; AND HERNÁNDEZ, C., 2013. Subgoal graphs in for optimal pathfinding in eight-neighbour grids. In *ICAPS*.
- VALMARI, A., 1989. Stubborn sets for reduced state space generation. In *Applications and Theory of Petri Nets*, 491–515.
- WALSH, T., 2007. Breaking value symmetries. In *CP*, 880–888.
- WANG, K.-H. C. AND BOTEÁ, A., 2008. Fast and memory-efficient multi-agent pathfinding. In *ICAPS*, 380–387.
- WEHRLE, M. AND HELMERT, M., 2012. About partial order reduction in planning and computer aided verification. In *ICAPS*.
- YAP, P., 2002. Grid-based path-finding. In *Canadian AI*, 44–55.

YAP, P.; BURCH, N.; HOLTE, R. C.; AND SCHAEFFER, J., 2011. Block A*: Database-driven search with applications in any-angle path-planning. In *AAAI*.

YOUNG, T., 2001. Optimizing points-of-visibility pathfinding. In *Game Programming Gems 2*, 324–329. Charles River Media.