University of Kent
School of Engineering and Digital Arts

A Novel Online Any-Angle Path Planning Algorithm

Paul Oprea

A thesis submitted for the degree of
Doctor of Philosophy in Electronic Engineering
2017

Abstract

Any-angle path planning algorithms are a popular topic of research in the fields of robotics and video games with a key focus in finding true shortest paths. Most online grid-constrained path-planning algorithms find sub-optimal solutions that present as unrealistic paths, a shortcoming which the any-angle class of algorithms attempt to address. While they do provide improvements in finding shorter paths, it generally comes in the form of a trade-off, by sacrificing runtime performance. The lack of a robust solution, that does not compromise on any of the desirable properties – online, reduced search-space, low runtime, short paths – of an any-angle path-planning algorithm, is a prime motivator for the current research.

A novel any-angle algorithm for 2-dimensional uniform-cost octile grids is introduced that operates purely online and reduces the search-space and runtime without sacrificing path-length. The methodology presents an atypical any-angle path-planning algorithm which employs a best first-search that races individual paths towards a target with a free-space assumption. The paths exhibit bug-like properties in that they either move towards a target or wall-follow, but are allowed to terminate early. Wall-following determines points on the boundary that are candidate heading changes in the path. At each step, the path is analysed and pruned in order to maintain its tautness at all times. Together with a purely heuristic cost based on the assumption of free-space between heading changes, the algorithm drives the search towards expanding the most promising path first. Once a path has reached the goal, it checks the free-space assumption between its heading changes and updates its cost accordingly. The shortest-path is determined when the cost estimate of any remaining paths is longer than the solution path.

The proposed algorithm is shown experimentally to be competitive on a number of performance metrics with state-of-the-art any-angle algorithms. It also presents desirable properties that allow it to have a reduced search-space and make it suitable for providing multiple solutions.

Acknowledgements

First and foremost, I would like to express my profound gratitude to my supervisor, Dr Konstantinos Sirlantzis, for his continuous support though my research degree, and for his guidance, motivation and patience.

My sincere thanks goes to Professor Farzin Deravi, and Dr Sanaul Hoque, who were also on my supervisory committee, for their encouragement and support.

I would like to thank Dr Michael Gilham, my colleague and advocate for his friendship and counsel.

I am grateful to my fellow colleagues Will Wycherley, Sotiris Chatzidimitriadis and Yankun Yang and to the interns whom I had the pleasure of working with over the past years.

Furthermore, I would like to thank Mr Chris Barron, School Administration Manager, the Administrative Team, and Technical Support Team of the School of Engineering for their assistance and encouragement.

Not least of all, I wish to thank Krisha and my parents for their patience, support and fortitude in motivating me during the course of my research.

Publications

S. Chatzidimitriadis, P. Oprea, M. Gillham, and K. Sirlantzis, Evaluation of 3d obstacle avoidance algorithm for smart powered wheelchairs," in *2017 Seventh International Conference on Emerging Security Technologies* (EST), pp. 157–162, Sept 2017.

V. Canoz, M. Gillham, P. Oprea, P. Chaumont, A. Bodin, P. Laux, M. Lebigre, G. Howells, and K. Sirlantzis, Embedded hardware for closing the gap between research and industry in the assistive powered wheelchair market," in *2016 IEEE/SICE International Symposium on System Integration (SII)*,pp. 108–113, Dec 2016.

This is to certify that I am responsible for the work submitted in this thesis, that the original work is my own except as specified in acknowledgements or in footnotes, and that neither the thesis nor the original work contained therein has been submitted to this or any other institution for a degree.

Paul Oprea

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

This chapter discusses the motivation of developing path-planning algorithms, and the basis on which this work has been implemented - in particular, the concepts of navigation, path-planning and map discretisation.

1.1 Introduction and overview

Artificial intelligence, in its various forms, offers solutions for solving problems traditionally associated with human or animal cognition, such as natural language processing, vision, playing games or bipedal walking, to name a few. While the impact of these advancements on society as a whole are subject to debate, the driving forces behind them are pushing smart technologies towards wide-spread adoption and acceptance. Addressing the numerous challenges of having intelligent agents such as autonomous robots and smart wheelchairs interacting and integrating seamlessly with natural and human-made environments implies an increase in the complexity of hardware and software technologies. To that end, scientists and engineers take inspiration from various aspects of nature, in everything from artificial neural networks [1] and genetic algorithms [2] to using slime mould [3] as a tool for redrawing more efficient routes around congested cities.

The interdisciplinary field of robotics has garnered attention due to numerous technological advances over the recent decades. As technology gradually permeates all aspects of human society and the reliance on automation and artificial intelligence solutions becomes commonplace, the distinction between human and machine gets blurred. A commonality of robotics is that of producing behaviours resembling those found in nature. Fields of study such as bio-inspired robots and soft-robotics take lessons from nature and incorporate observed designs, ideas and behaviours to address specific engineering challenges. A key interest among these challenges is the problem of path-planning. As an essential component of navigation in developing autonomous agents and video games, it has received abundant attention.

Path-planning algorithms are generally fast. However, constraints imposed by specific environments can apply restrictions to the time available to reach an optimal solution. For games with a large number of agents, for example, the number of active agents that can perform path planning simultaneously is impacted and can prove too resource intensive. Performing fast searches is important [4] for a number of reasons: the problem of path-planning is only one component of the navigation hierarchy that integrates into a overarching system with other components competing for resources (e.g. high-end video games) or, for robotics applications, sharing limited resources that impact on system performance and time-management and are dependent of portable energy sources (e.g. planetary rover with solar panel charging).

The current work focuses on one specific aspect of the path-planning problem, namely online any-angle path-planning in known 2D environments. In the context of path-planning, a purely online algorithm does not require any preprocessing of the search-space. In essence, there is no further information presented to the algorithm at run-time with the exception of the occupancy map of the environment. Path-panning on known 2D environments is applicable to both ground based robots with range sensing capabilities and AI gaming characters. As such, there is interest in improving on the state-of-the-art.

The algorithms presented perform single source path-planning on 2D uniform cost octile grids and operates on a number of assumptions, namely that the treating the search agent as a point object with no holonomic constrains, meaning that it can travel in any direction, unrestricted.

The thesis introduces a novel best-first search algorithm for finding any-angle paths on grid-constrained graphs. The proposed algorithm is shown experimentally to be competitive on a number of performance metrics with current state-of-the-art any-angle algorithms. It also presents desirable properties that allow it to have a reduced search-space and make it suitable for providing any-time solutions and multiple paths.

For the purpose of this research, certain assumptions can be made regarding the hypothetical agent solving the path-finding problem in question:

- it is treated as a point object with zero width and no kinematic constraints
- it can observe its immediate environment
- it is perfectly holonomic, being able to move in any direction
- it only requires freedom of movement in a 2D plane
- it can orient itself in the direction of a desired goal and move in said direction
- has perfect knowledge of the map topology and/or perfect memory of the environment it has previously explored

- it can transverse the environment only when no obstacle blocks its direction of travel.

1.2 Motivation

Let us consider two simple, everyday path-planning scenarios that a person would face under the umbrella of real-world navigation problems. The first problem implies the trivial task of moving inside the home, e.g. going from living-room to kitchen. For the second problem we consider driving from one city to another. Conceptually, both of these tasks imply solving a path-planning problem. In practice, however, getting from Paris to London by car or train is, in essence, a different challenge than walking from the living-room sofa to a cupboard in the kitchen, for example. To solve the former problem, one can adopt a simple level of abstraction, as the transport infrastructure confines the movement of a vehicle through the pre-existing road network (e.g. restriction to lanes, direction of roads etc.). As such, the road system may be interpreted as a graph, having a city as a vertex (i.e. node) and roads as edges connecting cities together. A real-world GPS-based driving assistant application (e.g. SatNav, Google Maps) would require higher level information to adequately represent the road network as a weighted, directed graph. Such a representation is necessary to reflect the direction of travel (e.g. dual-carriage vs. one-way roads), or traffic conditions and restrictions (e.g. number of lanes, road works, accidents and/or congestion, speed restrictions, toll charge).

A graph can be used to abstract the indoor environment in the latter scenario as well, for example, with a room represented by a vertex and a door representing an edge in the graph. For such a problem domain, the level of abstraction is too great as it looses information regarding the free-space within a room.

Two very well established approaches to path-finding are the deterministic solution (graph search with heuristics) and the reactive one, based on environment sensor sampling. Deterministic approaches offer the optimum solution but are penalised by map scalability, while the reactive ones are scale independent but do not guarantee a minimum cost path [5], [6]. However, the two aforementioned approaches generally operate on different problem domains, i.e. deterministic solutions require complete information about the search-space before converging to the optimal solution, while reactive algorithms generally have little or no prior knowledge about their environment. The problems are referred to in the literature as path-finding in known environments (deterministic) vs. unknown environments (reactive).

Generally, with the possible exception of some bug algorithms (e.g. Tangent-Bug), the behaviours of path-finding algorithms do not resemble behaviours that a human would adopt. Humans are highly capable of searching and solving the shortest path problems given an accurate means of calculating a desired metric (e.g. distance, time, energy expenditure, heading changes). It is trivially obvious that a human would also use a heuristic estimation if given incomplete information, or when relying on egocentric/external cues (e.g. biological clock, time of day, tiredness).

Some voices in the AI community argue that the emphasis on shortest path solutions may be misguided, as one would likely desire to introduce (or account for) some level of noise and inefficiency when emulating the real-world. Following a discussion on the power of inadmissible heuristics, Christer Ericson's AI blog article make the point as quoted:

"...much too much effort is spent in games in finding the shortest paths! [...] In our everyday lives we rarely, if ever, take a shortest path. Instead, we often optimize for search effort, taking a path we're familiar with (which we've chunked or otherwise memorized so as to require no search). Well, the same applies to games and the A* algorithm."[7]

Human navigation is the concept in which humans visualise routes in their head to get from start to destination, using a variety of navigational strategies. The simplest of this form would be to identify a reference point, and adjust route path as the landmark gets closer or further away through sensory feedback. This has been considered to be an approximation, due to human error – the inability to accurately generate a cognitive map. There are several parameters which humans take into account - such as shortest distance, least time, first noticed route as explored by [8].

Research from the Imagery Lab at the Harvard Medical School examines human navigation [9]. Generally, people use measures of distance or time of travel and absolute directional terms, i.e. cardinal points, in order to visualize the best route when navigating. These strategies are considered part of an allocentric navigation strategy, which is characterised by an object-to-object representational system. Information about the location of an object or its parts is encoded with respect to other objects. Complementary to this strategy, in an egocentric navigation strategy an individual relies on more local landmarks and personal directions (left/right) to visualize a route when navigating.

A complete layout for a cognitive engine based on human or animal spatial cognition remains an open question. Nonetheless, some of the ideas presented in this work attempt to crystallize, in part, some of the principles that could underline the strategies for reasoning on the geometric representations of spatial layouts. Inspiration can be drawn from an introspective look on spatial reasoning and navigation behaviours, as well as other path-planning methodologies available in the literature. In order to set the foundations for an any-angle path-planning algorithm, we consider a number of thought experiments aimed at identifying reasoning strategies of a human-like agent navigating towards a destination. Looking at the problem through the prism of human behaviour, a number of questions arise. Most notably, we wish to ask: How does a human attempt to solve a shortest path problem in a realistic environment? If a cognitive plan can be identified from such thought experiments, what insights could be used to outline a path-planning methodology? How well would such an algorithm fare against existing algorithms?

1.3 Path planning and navigation

The problem of searching for shortest paths finds applications in diverse areas [10], such as package routing in data networks (e.g. RIP - Routing Information Protocol, OSPF - Open Shortest Path First), route planning and guidance (e.g. Google Maps), traffic congestion management, video games, to name a few. While conceptually broad, the focus of this work is on finding shortest paths in 2-dimensional environments, as it applies to the field of robotic navigation.

Navigation is the process of ascertaining one's position, planning a route and following the route towards a desired destination. Looking briefly at navigation in animals, natural selection has provided biological agents with diverse solutions to tackle various challenges of navigation. Throughout the evolutionary process within the animal kingdom, the problem of navigation has been of paramount importance for the purposes of survival, in the vital challenges of finding food and water, avoiding predators, returning to a nest.

An abundant variety of sensing organs has evolved in the animal kingdom that tackle these trials, in forms such as olfaction for detection of chemical markers (e.g. recognition of siblings, detection of potential mates, territorial boundary marking, insect chemical trails), echolocation (e.g. dolphins, bats), detection of electric fields (e.g. sharks, platypus), magneto-reception for detecting the Earth's magnetic field (e.g. homing pigeons), infra-red sensing (e.g. snakes, vampire bats), as well as presenting diverse means of locomotion to seek out sources of energy or avoid predation and other harmful environments, and proprioception for perceiving movement and spatial orientation within the body. Human navigation has had its own revolution, as we moved away from relying solely on our senses and found solutions within technological fields, from compasses and star maps, to radar navigation and GPS systems in modern times.

In the fields of robotics and video game development, navigating has also presented a challenge. For the purpose of robotic navigation, the problem can be broken down into several sub-problems: localisation (knowing where one is), obstacle avoidance (detecting and avoiding objects in the immediate environment), mapping (storing information in memory about the environment), path planning (constructing a plan from one's current location to a destination) and exploration (discovering one's environment). Among the key aspects from this list, this work will focus on the problem of path-planning, which has garnered attention due to a wide set of challenges and a complex problem domain. Generally speaking, path planning consists of finding a path between a given start location and a given goal location if such a path exists. This task implies a level of knowledge about one's relative or absolute location. For an intelligent agent navigating an environment, decision-making rules would seek to minimise the energy expenditure required in planning and navigation.

1.4 Map discretisation and notations

An intelligent agent (robotic platform) operates in a continuous environment, typically a Cartesian plane as is represented by $(x, y) \in \mathbb{R}^2$. We assume a holonomic ground robot in a 2-dimensional flat environment. The set of valid poses the robot can find itself in is known as its free space. Invalid states correspond to obstacles or poses the robot cannot occupy. This allows partitioning robot configuration into two classes: free space - C_{free} and occupied space - $\mathbb{R}^2 - C_{free}$. Given this configuration, the problem of path-planning can be described as identifying the set of valid states belonging to C_{free} that get the agent from one configuration to another. An intelligent robot would also require a means of perception (i.e. sensors) to sample the environment. With local measurements of C_{free}, it would require the ability to localise (determine where it is in the environment), and constructing a representation of C_{free} - mapping, based on sensor inputs. Performing simultaneous localisation and mapping (i.e. SLAM [11] [12], [13], [14]) implies determining its pose and C_{free} without knowing either, by sampling the environment. For the path planning task, however, the existence of a map is assumed implicitly. Given a known continuous environment, an effective discrete representation is required for storing, manipulating and querying the free space. This involves map discretisation, or spatial decomposition, by which the continuous environment is discretely sampled to represent space itself, rather than having to discriminate or identify individual obstacles [15].

Discretising a path allows a continuous path to be realised in a 2-D graphical format. Grid paths are created from simple iterated geometric shapes. Within a two-dimensional representation the three most popular types of grid paths are square, triangular and hexagonal. The most common being the square type. They are considered the easier grid shape to use for two

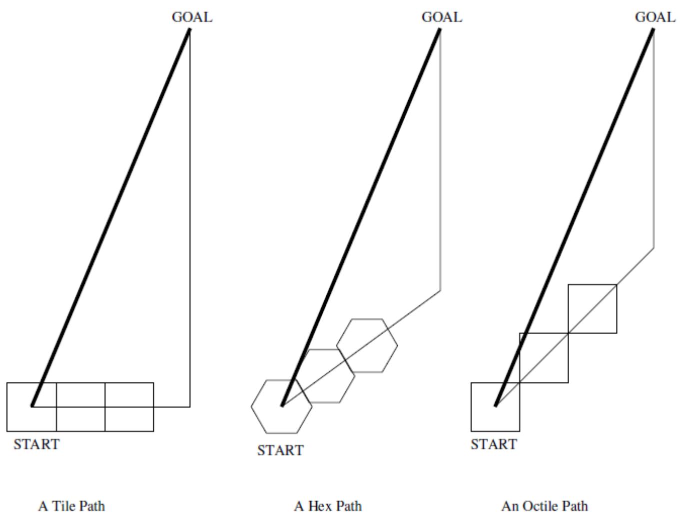


Figure 1.1: Path approximations on grid geometries – true shortest path (thick line) vs grid-constrained path (thin line with heading change) [16]

reasons, one being the ability to map coordinates into a Cartesian format, and the axes of the of which will hence be orthogonal. Hexagonal paths present a decreased path-length distortion in comparison to square grids but are not as simple to manipulate. The path-length distortion for paths on a square grid is increased in comparison with a continuous path – the shortest path on the graph is not the shortest path [4]. Figure 1.1 shows approximated paths using square tiles, hex tiles, and octal tiles in comparison to actual optimal path (i.e. linear distance).

A structure $G = (V, E)$ describes a graph that comprises of vertices or nodes, belonging to a set V, and of edges that belong to set E, such that an element of E is defined by an arc $\{v_1, v_2\} \subset E$, with the two component vertices $v_1 \in V$ and $v_2 \in V$. The graph structure $G = (V, E)$ is comprised of discrete samples of the continuous environment $\mathbb{R}^2$ such that the totality of free regions in G bound by edges in E describe C_{free} (the free space), while the non-free regions in G bound by edges in E describe $\mathbb{R}^2 - C_{free}$ (the occupied space). A graph can be traversed by travelling from node to node along edges.

Many types of graphs can be adopted. For example, a directed graph is one that limits the direction of travel along an edge. If no such restrictions apply, the graph is considered undirected. In a weighted graph, values are associated with edges. The "weights" reflect the cost of traversing the graph through the respective edge.

A grid induces a graph where each node corresponds to a cell and an edge connects nodes of cells that neighbour each other. Four-point connectivity will only have edges to the north, south, east, and west, whereas eight-point connectivity will have edges to all cells surrounding the current cell. For our purposes, the implementations presented in this work assume an undirected, un-weighted 8-connectivity grid graph (Figure 1.3).

The algorithms presented in this work operate on an un-weighted 8-connectivity grid graph $G(V, E)$ with vertices indexed as a 1-dimensional array, and described by the relationships:

$$s_{x,y} \equiv s_i \tag{1.1}$$

$$x = i \mod \mathbf{W} \tag{1.2}$$

$$y = \lfloor \frac{i}{\mathbf{W}} \rfloor \tag{1.3}$$

$$i = \mathbf{W} * y + x \tag{1.4}$$

where, $s_i \in V$, x and y represent the Cartesian coordinates of the node s and $\mathbf{W}$ represents the discrete width of the map (number of tiles per row for a square-grid tessellation).

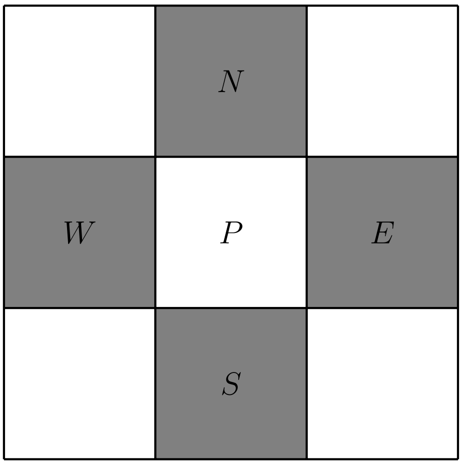


Figure 1.2: Von Neumann neighbourhood (4-connectivity)

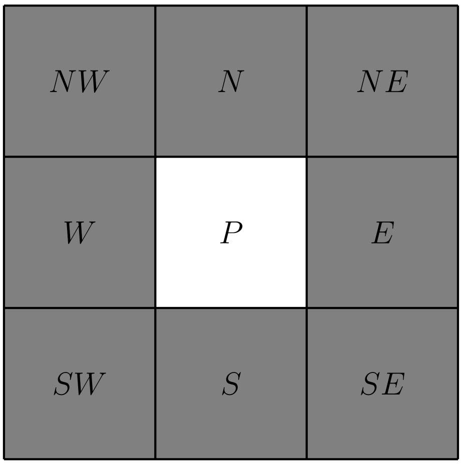


Figure 1.3: Moore neighbourhood (8-connectivity)

1.4.1 Heuristics

Generally, path-planning algorithms require a means of estimating the distance between two locations. For this purpose, one employs heuristic estimations based on knowledge and/or assumptions about the environment. For addressing the shortest path problem two properties of heuristic functions are considered.

Admissible heuristics

An admissible heuristic cannot overestimate the cost of reaching a goal. For a heuristic $h(s)$ to be admissible it must respect the inequality:

$$h(s) \leq h^*(s),\ \forall s \in V \tag{1.5}$$

where $h(s)$ is the estimated cost of travelling from s to the goal and $h^*(s)$ is the actual cost of travel.

Consistent heuristics

A heuristic function is said to be consistent, if its estimate is always less than or equal to the estimated distance from any neighbouring node to the goal, plus the step cost of reaching that neighbour. A consistent heuristic is also considered admissible, meaning that it never overestimates the cost of reaching the goal. Formally, for every node s and each successor s' of s, the estimated cost of reaching the goal from s is no greater than the step cost of getting to s' plus the estimated cost of reaching the goal from s':

$$h(s) \leq g(s, s') + h(s') \tag{1.6}$$

where $h(s_{goal}) = 0$, h is the heuristic function, s is a node in the graph, s' is any descendant of s ($parent(s') = s$), $g(s, s')$ is the cost of reaching node s' from s, and s_{goal} is the goal node.

Distance metrics

The choice of heuristic is also dependent of the freedom of movement [17] on the graph: For square grids that allow 4 directions of movement (e.g. sliding puzzle games) with neighbours belonging to the Von Neumann neighbourhood illustrated in Figure 1.2, the Manhattan distance is sufficient:

$$D_{Manhattan}(p_1, p_2) := |x_2 - x_1| + |y_2 - y_1| \tag{1.7}$$

On an octile square grid that allows 8 directions of movement (e.g. chess) with neighbours belonging to the Moore neighbourhood illustrated in Figure 1.3, the heuristic of choice is the Chebyshev distance (or Diagonal distance):

$$D_{Chebyshev}(p_1, p_2) := \max(|x_2 - x_1|, |y_2 - y_1|) \tag{1.8}$$

A special case of the Chebyshev distance is the Octile distance. For square grids that allows 8 directions of movement and have a diagonal step cost of $D_2 = \sqrt{2}$ and an orthogonal step cost of $D = 1$ the Octile distance is used instead:

$$D_{Octile}(p_1, p_2) := D\max(|x_2-x_1|, |y_2-y_1|)+(D2-D)\min(|x_2-x_1|, |y_2-y_1|) \tag{1.9}$$

The consistent heuristic for square grids which allow any direction of movement (e.g. any-angle paths) is the Euclidean distance:

$$D_{Euclidian}(p_1, p_2) := \sqrt{(x_{start} - x_{end})^2 + (y_{start} - y_{end})^2} \tag{1.10}$$

On Euclidean graphs the straight-line Euclidean distance, is both admissible and consistent [4].

1.5 Contribution

The contribution presented in this work addresses the problem of online any-angle path-planning on 2D octile grids with uniform cost.

The methodology presents an atypical any-angle path-planning algorithm which uses a best first-search strategy to race individual paths towards a target with a free-space assumption. The paths exhibit bug-like properties in that they either move towards a target or wall-follow, but are allowed to terminate early. The algorithm operates purely online with no pre-processing of the map, and is competitive with state-of-the-art alternative algorithms.

Wall-following determines points on the obstacle's boundary that are candidate heading changes in the path. The path is analysed and pruned in order to maintain its tautness at each step. Together with a purely heuristic cost based on the assumption of free-space between heading changes, the algorithm drives the search towards expanding the most promising path first. Once a path has reached the goal, it checks the free-space assumption between its heading changes and updates its cost accordingly. The shortest path is determined when the cost estimate of any remaining paths is greater than the solution path.

Paths propagate towards the goal through free space in a straight line using a novel adaptation of Breshenman's line algorithm for 1D-indexed grids. The algorithm performs a line-of-sight search between two points and, if an obstacle is encountered, returns the intersections points situated on the boards of the obstacle. These points are then fed to the wall-following algorithm.

For the purpose of emulating a wall-following behaviour on 2D octile grids, we introduce a novel and elegant contour tracing algorithm, which is used by paths to explore the free-space around an object.

The algorithm presents some desirable properties that allow it to have a reduced search-space, and as a best-first search algorithm it allows for any-time solutions. It can also be adapted to provide multiple solutions with an increase in run-time.

1.6 Chapter Summary

The chapter introduces the field of path-planning within the broader concept of navigation. We describe the problem domain and the motivation for having undertaken the current research, and present an overview of map discretisation for 2-dimensional ground-planes into occupancy grids.

Common heuristic functions used in path-planning on occupancy grids are introduced. They allow informed path-search algorithms to judge the preferable course of action when searching for a connected route through the free-space between a start point to a destination point.

Finally, we introduce our contribution in the form of a best-first search algorithm that propagates individual bug-like paths towards a goal with a free-space assumption and optimises the paths that have made it to the target.

Chapter 2

Path planning methodologies

This chapter introduces path-finding methodologies from the literature that are relevant to the research presented in this work. Three classes of path-finding solutions are of interest, namely, bug algorithms, grid-constrained algorithms and any-angle algorithms.

2.1 Introduction

The current chapter introduces some path-finding methodologies from the literature that are relevant to the research presented in this work. The focus is on three classes of algorithms: bug algorithms, grid-constrained path-finding algorithms and any-angle path-finding algorithms.

A number of existing methodologies for path-planning are based on variations of the A* algorithm. Because of the success of A*, it has been widely adopted by developers in the fields of robotics and video games. When applied to path planning on grid maps, however, the solution can result in unrealistic looking paths, with paths being restricted to orthogonal or

45° orientations. Additionally, shortest path solutions only have heading changes at the corners of obstacles, but A* can have arbitrary heading changes other than those at the corners of obstacles. The A* algorithm is based on Dijkstra's algorithm for graph transversal, but because A* was originally designed for weighted graph transversal, and is arguably one of the most popular general-purpose graph search algorithms when there's a way to estimate the distance to the goal [18]. The majority of path-finding algorithms are variations on Dijkstra's graph search algorithm and A* (e.g. HPA [19], DHPA* & SHPA* [20], Best-first search [21], D* [22], IDA* [23]).

Generally, path-planning algorithms use heuristic cost functions to determine the order in which the algorithm visits nodes in the search-space. A* uses a knowledge-plus-heuristics cost function composed from the sum of a past path-cost function (distance from the starting node to the current node) and a future path-cost function (a heuristic estimate of the distance from the current node to the goal). These solutions fall under the category of grid-constrained algorithms and thus have shortcomings such as unnecessary heading changes resulting in unrealistic looking paths.

A good path-planning algorithm aims to have a number of properties:

Correctness - if a solution is found, there exists a path $\pi \in C_{free}$ that connects the start and goal nodes.

Completeness - the algorithm can correctly answer whether or not a solution exists.

Optimality - if a solution is found, the identified path is the shortest path from start to goal, given the constrains of the algorithm;

Until recently, the existence of an on-line, optimal, any-angle path finding algorithm was an open-ended question. Courtesy of Harabor et al. the affirmative answer came in the form of Anya, an online, optimal, any-angle algorithm [24], [25]. Anya does not require any pre-processing of the map and performs searches an order of magnitude faster relative to A* on grid-maps. A generalised version of Anya, called Polyanya [26], extends the original functionality of Anya to navigation meshes.

The main focus of our work is on purely-online algorithms which does not require any pre-processing of the topology prior to a search. While algorithms that perform off-line pre-processing of the map beforehand have good on-line performance ([27], [28], [29]), they have certain undesired properties which limits their uses and effectiveness. Before any search, the search graph must be pre-computed during an off-line pre-processing step. Another limitation is that, if the search space changes at any point, the search graph is invalidated and must be reconstructed off-line, which can be prohibitively resource intensive. For example, Sub-2, an any-angle variant of Subgoal Graphs (2-level), while dominating purely online any-angle algorithms (e.g. Theta* family, Field A*, Block A*), can require up to 35 seconds of preprocessing time [29].

Using graph algorithms on uniform grids may not scale very well, when map topologies contain relatively few obstacles and a large amount of free space. Some, like Jump-Point-Search, have attempted to address these problems [30].

With the exception of Anya [24] and Polyanya [26], to the author's knowledge, no other online, optimal, any-angle algorithms exists at the time of this writing.

2.2 Bug algorithms

Bug algorithms are a class of path-finding methodologies for navigating unknown environments ([31], [32], [33], [34], [35], [36], [34], [37], [38], [39], [40], [6], [41], [42]). Generally, bug algorithms are employed by robotic agents, making them dependent on the hardware setup of the robot [43]. Bugs are a class of reactive simple automata that perform goal seeking behaviours in the presence of non-drivable areas. These algorithms operate by alternating between two behaviours: wall-following (tracing the boundary of an obstacle until a condition is met) and motion-to-goal (travelling towards a target until it is reached or the bug encounters an obstacle) [44]. Bug algorithms are among the earliest and simplest planners and have the benefit of having provable guarantees.

Sensors are a quintessential component of a robot running a bug algorithm. Sensors such as tactile, range, imaging cameras are used to to detect its immediate environment [45]. Odometry information or other external signals (e.g. RFID, GPS, landmarks) are also required to establish the direction of travel, and/or estimate the distance to a goal. The robot also requires the capacity for memory (or possibly marking its position with a token), in order to determine if a location had been previously visited [46]. These agents can be susceptible to errors because of imperfect sensor information, cumulative errors in odometry due to wheel slippage or slow data rates etc. Bug algorithms in general operate under the assumption of perfect information and error-free sensor data.

Operating without a map, fundamentally limits a robot's behaviour, as it cannot see the "big picture" and therefore takes paths that are locally but not globally optimal [44].

The following sections summarise some of the more popular bug algorithms that present relevant parallels (Chapter 3) to some behaviours of the novel methodology introduced in this work.

2.2.1 Bug-1 Algorithm

Developed by Lumelsky and Stepanov, the Bug-1 algorithm [31] involves a mobile robot navigation strategy in unknown environments. The robot's behaviour relies of tactile sensors to detect obstacles. An example scenario is provided by Figure 2.1. It starts from a given start position and moves towards a goal, unless it encounters an obstacle. The point of intersection is memorised and labelled, after which it proceeds to trace the obstacle on the left-hand side. During the wall-following procedure it determines and labels the leave point by calculating the distances between the current position and that of the target. The leave point is the point on the obstacle's boundary that is closest to the goal. When the robot revisits the point of intersection, it tests if the target can be reached by checking if the robot can move towards the target at the memorised leave point. If this check fails, the target is unreachable. Otherwise, the robot chooses the wall-following direction of minimal distance between the intersection and leave points. After reaching the leave point a second time, it reverts back to moving towards the target. This cycle repeats until a solution is found or the algorithm determines that the target is unreachable.

In essence, Bug-1 searches each obstacle's boundary for a point closest to the goal. If the robot determines that the target is reachable, it can infer that by leaving at the memorised leave point, it will never re-encounter the obstacle.

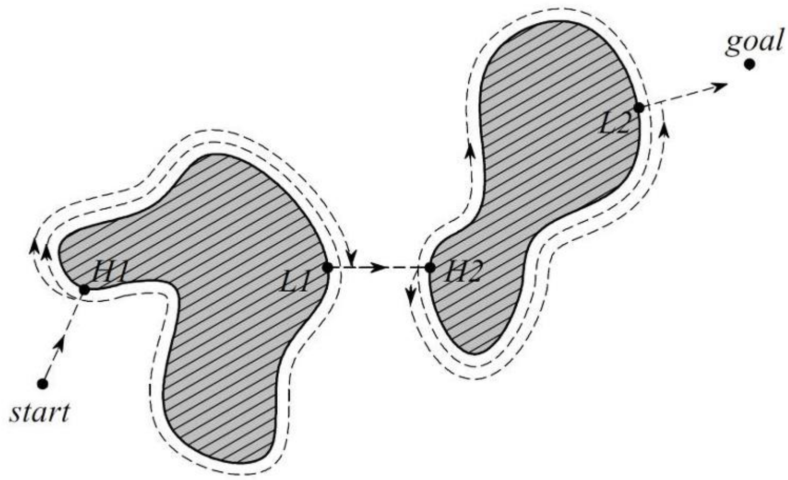


Figure 2.1: Bug-1 Algorithm [47]

2.2.2 Bug-2 Algorithm

Lumelsky and Stepanov present a second, less conservative bug algorithm in the form of Bug-2. As an augmentation to Bug-1, the Bug-2 algorithm [31] is a greedy algorithm that can leave an obstacle's boundary earlier than Bug1. It does so by making use of the M-line, an imaginary line that connects the start and target points. A mobile robot running the Bug-2 algorithm initiates a move towards the goal, following the M-line, until it either reaches the target, in which case it terminates, or, it encounters an obstacle. If the latter happens, the point of intersection is memorised and labelled, after which it proceeds to trace the obstacle on the left-hand side. Wall following continues until it finds the initial M-line again. Otherwise, if the robot makes it back around to the intersection point without encountering the M-line, it infers that the target is unreachable. If the point identified on the M-line, however, is closer to the target than any other point (i.e. current distance to target is less then previous shortest distance - initially $d = Distance(P_{intersection}, P_{target})$) and the robot is able to move towards the target, the position is labelled as a leave point and memorised, after which it cycles back and resumes following the M-line. Otherwise, if the robot cannot move towards the target, it continues the wall-following procedure and d is updated to reflect the new shortest distance: $d = Distance(P_{current}, P_{target})$.

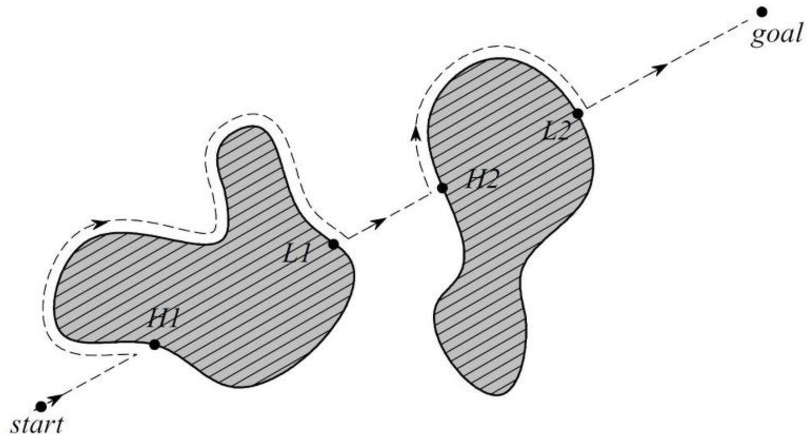


Figure 2.2: Bug-2 Algorithm [47]

Bug-2 is a greedy algorithm that performs better on topologies with simple obstacles. In contrast, Bug-1 performs an exhaustive search, always tracing the full contour of an obstacle before deciding on the leave point. Both Bug-1 and Bug-2 can outperform each other depending on the topology of the environment [15].

2.2.3 Tangent Bug Algorithm

An alternative approach to Bug-2 was proposed by Kamon, Rivlin and Rimon. Their algorithm, TangentBug [36] uses 360 degree distance sensors to build a local tangent graph of a robot's immediate surroundings and uses it to minimize the path length by movement towards a "locally optimal direction". The local minimum is defined as the smallest value within a set of points, which may or may not be a global minimum. In the case of the Tangent Bug, this would be a obstacle in the path, as detected by the sensors.

The underlying concept of the TangentBug algorithm (Figure 2.3) is described as follows: A 'motion–to–goal' behavioural pattern is followed as long as the path, unless an obstacle is detected by the sensors. If a local minimum (obstacle) is detected by the sensors, the algorithm would then switch to a line following behaviour. During this period, the minimal distance between current position at the sensed boundary, and goal node is calculated repeatedly. Once this value is less than the distance between any further obstacles (or distance to goal if no further obstacles), the algorithm would return to the motion–to–goal behaviour.

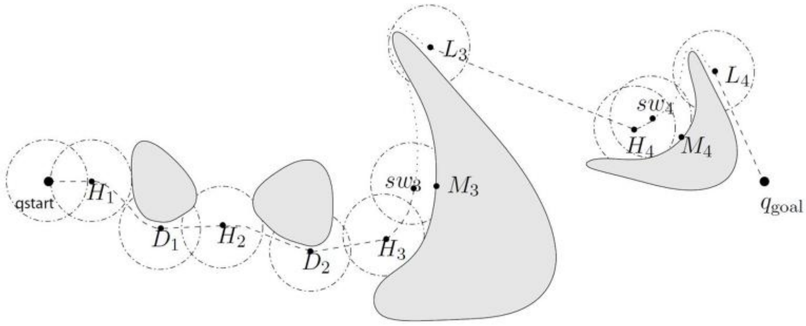


Figure 2.3: Tangent Bug Algorithm [36]

2.2.4 MBPP - Multi-Bug Path Planning

A 2016 paper [48] presents a multi-bug path planning algorithm reminiscent of ABUG [6]. The "Multi Bug Path Planning" (MBPP) algorithm presented in [48] operates under a free-space assumption, with an initial bug moving from a start node towards the goal node. If the bug encounters an obstacle, it generates a new bug, and they proceed along the obstacle walls in opposite directions. The same bug generation is employed for any new obstacle encountered, until the target node is reached by all live bugs. MBPP thus evaluates the resulting paths and chooses the best route. Each bug follows the wall of the obstacle until it encounters the M-line (imaginary line between the start and goal nodes). It then leaves the obstacle edge and reverts to moving towards the goal. During the wall following procedure a line-of-sight check is continuously being performed between the current position on the wall and the last state in the path. A visual representation of MBPP solving a spiral scenario can be seen in Figure 2.4. The experimental results presented by Bhanu Chander et al. [48] show an improvement in comparison to paths generated by A* with Post-Smoothing. To the author's knowledge, no large-scale experimental results have been presented as of this writing (such as maps from the Moving AI Lab database).

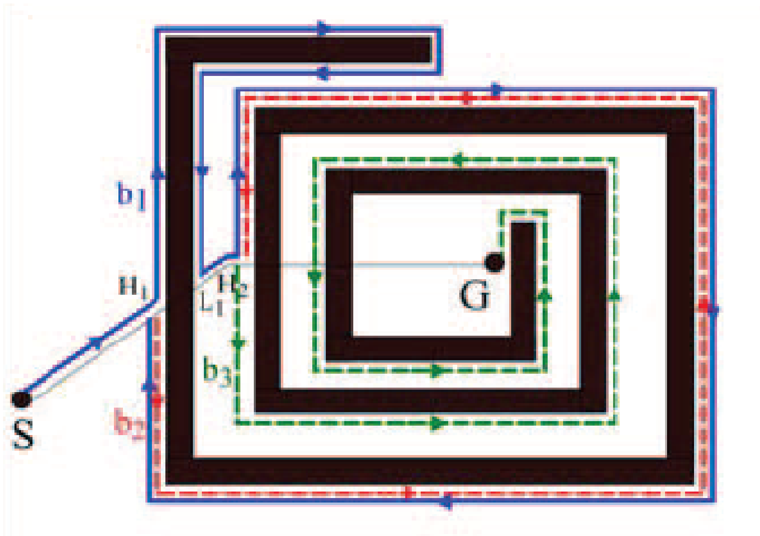


Figure 2.4: Multi Bug Path Planning on Spiral example [48]

The paper describes MBPP as an offline bug algorithm for known environments. The principle of operation of MBPP along with other algorithms (e.g. TangentBug [36], Theta* [4], HCTNav [5]) share some similarities with the methodology presented in the current work, which are discussed in detail in Chapter 4. Our proposed methodology races paths towards the goal (in a motion-to-goal behaviour similar to that of TangentBug), splitting them when an obstacle is encountered, which is reminiscent of the bugs generated by MBPP and the path-splitting behaviour of HCTNav. Both MBPP and Theta* perform continuous line-of-sight checks while exploring the search space. However, our methodology differs from these algorithms in that it only performs line-of-sight sparingly, in two manners: when optimizing a path that has already reached the goal or when moving towards the goal through free space. In addition, unlike MBPP, our algorithm does not require for all paths to terminate before the solution is reached.

2.3 Grid constrained algorithms

For 8-connected regular grids, a grid-constrained path is a sequence of cell corners where consecutive pairs of cell corners must belong to the same grid cell. Formally, given a node $s_i \in G(V, E)$ belonging to a grid-constrained path, the node belongs to the Moore neighbourhood of the parent node: $s_i \in M(parent(s_i))$. For example, grid-constrained A* with an consistent heuristic finds the shortest path composed of edges bound to the grid, but not a true-shortest path, such as would be identified by A* on visibility graphs.

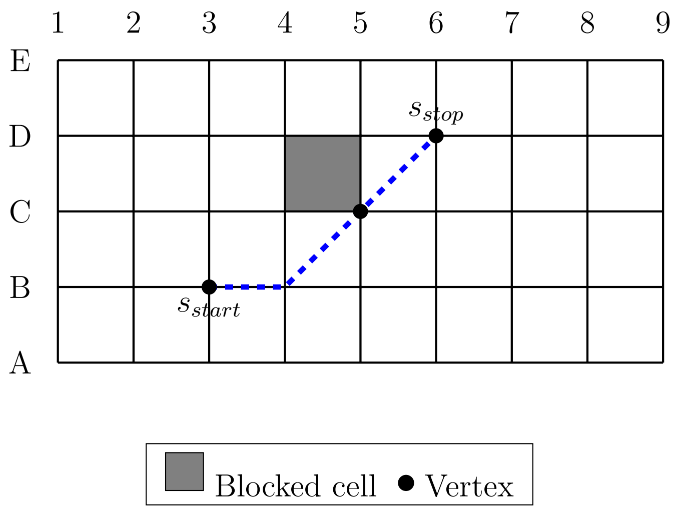


Figure 2.5: Grid-constrained path found by A*: heading change occurs at B_4; $\{s_{start}, B_4, C_5, s_{stop}\}$ is not taut around D_4 obstacle

The grid-constrained algorithms described in this work operate on 8-neighbour grids.

2.3.1 Dijkstra's algorithm

Dijkstra's algorithm, published in 1959 and named after its creator, Dutch computer scientist Edsger Dijkstra, can be applied on a weighted graph [49]. Dijkstra's algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.

Most grid-constrained path-planning algorithms are variations on Dijkstra's algorithm. The algorithm maintains two structures: – a closed list, in essence, a set which contains all the nodes that have been expanded ('explored') – an open list, which is a set that contains the nodes of the search space that are not in the closed list ('unexplored').

For each node in the search space, Dijkstra's algorithm maintains two values: a g score which represents the length of the shortest path from the start node to the currently expanded node, s, and a reference to the parent of node s, $parent(s)$, so that the start node can be traced back after the end node has been reached. The parents of any $s \in S$ form a search tree that root at the start node. The cost function of each node is simply the distance travelled from the start node:

$$g(s) = g(parent(s)) + D(parent(s), s) = \sum_{i=start}^{n} D(s_i, parent(s_i)) \quad (2.1)$$

where g represents the cost of reaching a specific node n represents the index of node s and D represents the distance between two nodes.

Dijkstra's algorithm works by visiting nodes in the graph starting with the starting node s_{start}. Similar to a ripple in a pond, it propagates from the starting node and expands each node until it reaches the goal. A visited node is only ever expanded once. The algorithm repeatedly examines the closest node not yet examined, adding its vertices to the set of vertices to be examined. It expands outwards from the starting point until it reaches the goal. Dijkstra's algorithm is guaranteed to find a shortest path from the starting point to the goal, for any graph with non-negative step costs.

For a finite search space, Dijkstra's algorithm terminates in one of two cases. It either detects that the end node has been reached, in which case it reconstructs the path by following the parents of each node from s_{goal} to s_{start}, or it exhaustively expands all connected nodes in the search space until the open list becomes empty. The only requirement for the algorithm to be correct, complete and optimal is that the lengths of all edges in the graph are non-negative, and thus it can be used to search any Euclidean graph.

2.3.2 A* algorithm

Developed by Peter Hart, Nils Nilsson and Bertram Raphael, the A^* algorithm [18] is the result of pioneering research on “Shakey the robot” developed at the Stanford Research Institute. Designed as a informed search variation on $Dijkstra's$ algorithm, A^* reduces the search runtime without impacting the length of the resulting path. It does this by employing a focused search so that the goal can be found with a shorter runtime and generally with fewer node expansions. In order to accomplish this, the algorithm makes use of a knowledge-plus-heuristics cost function that records a score for each node that it explores. This score is comprised of two components: the cost of travel (known) and an estimation of the cost of reaching the goal (heuristic cost). The cost function f associated with a node $s \in G(V, E)$ is computed by:

$$f(s) = g(s) + h(s) \tag{2.2}$$

where g represents the cost of travel from the start node to s, and h represents an estimate of distance from s to the goal node. The value of g is described by Equation 2.1. In contrast to A^*, $Dijkstra's$ algorithm only preserves information about the cost of travelling from the start node to s, that is to say, a node's g value, making $Dijkstra's$ algorithm an uniformed search-algorithm.

The choice of selecting a heuristic function for A^* can impact the behaviour of the algorithm [17]. When considering the heuristic function in Equation 2.2, the algorithm's behaviour can be altered if the h score function is set to 0. As such, only the g is taken into consideration, and in essence, A^* denatures into $Dijkstra's$ algorithm, which is still guaranteed to find the shortest grid-constrained path. On the other hand, if h is always lower or equal to the cost of moving from s to the goal, which implied having a consistent heuristic, A^* is guaranteed to find a shortest path. The lower the h score, however, the more node need to be expanded, in turn increasing the runtime of the algorithm. A^* can also be made to run faster by providing a non-admissible heuristic (i.e. one that violates 1.5). If one relaxes the h score so that it sometimes overestimates the cost of moving from the current node to the goal, A^* is no longer guaranteed to find a shortest path, but this can prove a reasonable trade-off as it can reduce the algorithm's runtime. In special cases, if $h(s)$ is exactly equal to the cost of moving from s to the goal, then A^* will only follow the best path and never deviate, making it very fast. The choice of heuristic can be a useful tool in trading run-time performance for longer paths.

The only requirement for A^* to be complete, correct and optimal is that the lengths of all the edges in the graph are non-negative and that the h-values are admissible. On Euclidean graphs the straight-line Euclidean distance, is both admissible and consistent [50]. In our implementation, A^* is implemented with the completeness, correctness and optimality properties and uses the Manhattan distance heuristic.

The pseudo-code for the A^* algorithm is presented in Algorithm 1. The methodology employed by the A^* algorithm is similar to that of Dijkstra, except that the cost function that it computes during the expansion process is different: $Dijkstra's$ algorithm, chooses the next node based on the smallest g score, while A^* chooses the node with the smallest f score.

Algorithm 1 A*

function MAIN(s_{start}, s_{goal})
    $g(s_{start}) \leftarrow 0$
    $opem \leftarrow \emptyset$
    $opem$.INSERT($s_{start}, g(s_{start}) + h(s_{start})$)
    $closed \leftarrow \emptyset$
    while $open \neq \emptyset$ **do**
        $s \leftarrow open$.POP()
        if $s = s_{goal}$ **then**
            return *"path found"*
        $closed \leftarrow closed \cup s$
        foreach $s' \in nghbr_{vis}(s)$ **do**
            if $s \notin closed$ **then**
                if $s \notin open$ **then**
                    $parent(s') \leftarrow NULL$
                    $g(s') \leftarrow \infty$
                UPDATEVERTEX(s, s')
    return *"no path found"*

function UPDATEVERTEX(s, s')
    $g_{old} \leftarrow g(s')$
    COMPUTECOST(s, s')
    if $g(s') < g_{old}$ **then**
        if $s' \in open$ **then**
            $open$.REMOVE(s')
        $open$.INSERT($s', g(s') + h(s')$)

function COMPUTECOST(s, s')
    if $g(s) + c(s, s') < g(s')$ **then**
        $parent(s') \leftarrow s$
        $g(s') \leftarrow g(s) + c(s, s')$

The A^* algorithm is arguably one of the most popular online path-planning and graph traversal solutions, due in part to its simplicity but also because of its optimality guaranty.

As long as the h-values provide accurate estimates of the lengths of shortest paths from a vertex to the goal vertex, the expansion equation computed by A^* is more informed than the one computed by $Dijkstra's$ algorithm.

Thus, an A^* search is able to avoid examining a larger number of paths in the graph than a Dijkstra search, which often results in shorter run-times. As A^* is a essentially a graph algorithm, it can run on any graph representation. When run on visibility graphs A^* indeed finds true shortest paths, as true shortest paths have heading changes only at the corners of blocked grid cells. However, A^* on visibility graphs can be slow, as shown experimentally by [50]. Running A^* on visibility graphs requires pre-computing of a map's visibility graph, a procedure that can be exceedingly expensive and which is done off-line, in a preprocessing stage [50]. On octile graphs, however, the paths found by A^* are artificially constrained to be formed by edges of the octile graph. Paths found by A^* on octile graphs are not true shortest paths and are unrealistic looking since they either deviate substantially from the true shortest paths or have excessive heading changes, which provides the motivation for smoothing them.

Because of its simplicity and versatility along with its optimality guarantee, the A^* algorithm has garnered widespread adoption. A number of existing methodologies for path-planning are based on variations of the A^* algorithm.

When applied to path planning on grid maps (octile graphs), however, the solutions found can result in unrealistic looking paths due to unnecessary heading changes. For evaluation purposes, the A^* variant operating on octile graphs has been chosen. Optimizations to the A^* algorithm have attempted to address A^*'s limitations. In terms of performance, a notable mention, the Jump-Point-Search algorithm [51] works by reducing symmetries in the in the search space. It accomplishes this by means of graph pruning, thus being able to "jump over" nodes in a straight line without expanding them. Although it can potentially reduce the running time by an order of magnitude, it still produces unrealistic looking paths. Theta* and Light-Theta* are two any-angle algorithms that build on top of the A^* algorithm in an attempt to improve on the issue of unrealistic looking paths. The algorithms accomplish this by propagating information along grid edges. The most relevant of the above mentioned algorithms are described in more detail in the following sections.

2.4 Any angle algorithms

Any-angle path-planning algorithms are a class of path-finding algorithms that search for paths between two nodes in the free-space by propagating information along graph edges, similar to grid-constrained algorithms, but without restricting the paths to be formed by graph edges. A consequence is that a given node can have as a parent any other node as long as there is a line-of-sight between them: $\forall s \in G(V,E)$ and $\forall s' \in G(V,E)$: $parent(s) = s'$ if $LineOfSight(s, s')$ is true. Two vertices are defined as having line-of-sight if the segment connecting them does not pass through the interior of any occupied grid cells nor between any orthogonally or diagonally adjacent pair of occupied grid cells (i.e. any pair of occupied cells that share one or two vertices).

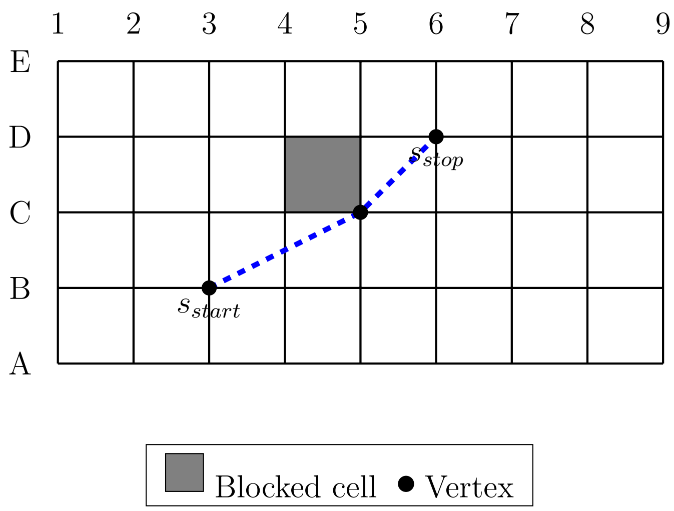


Figure 2.6: Any-angle path: heading change occurs only at C_5

Generally, any-angle solutions place vertices at the corners of the grid cells, rather than their centres.

Line-of-sight

Performing line-of-sight checks between nodes in a grid-graph enables any-angle algorithms to partially uncouple the path's topology from that of the grid-constrained exploration of the free-space. The path is partially uncoupled because the vertices forming an any-angle solution are still bound to the graph. Performing a line-of-sight on a 2D regular grid is equivalent to drawing a line plot on a raster display. For this purpose, an extremely popular line drawing algorithm is introduced, which has been used extensively in computer graphics.

Bresenham's line algorithm [52], developed at IBM in 1962 by Jack Elton Bresenham, is an incremental error algorithm and one of the earliest algorithms developed in the field of computer graphics. It can be used to determine the points of a raster which most closely approximate a straight line between two given points. The effectiveness of Bresenham's algorithm, is that it can be implemented using cheap operations [53] for modern architectures: integer addition, subtraction and bit shifting. The line-of-sight algorithm (Algorithm 2) is used by a number of algorithms such as A^* $with\ post-smoothing$ and the $Theta^*$ family.

The line-of-sight algorithm presented in Algorithm 2 is a functionally equivalent re-factored variant of Bresenham's line algorithm presented by Nash et al. in [4]. The main distinction is that the implementation of Algorithm 2 operates on a grid-map with vertices indexed on a 1-dimensional array (as described by Equation 1.1), and it restricts a path from crossing between diagonally adjacent occupied cells for all path-planning algorithms.

In order to control for potential variations in performance metrics due to different implementations of the line-of-sight algorithm in the literature, all path-planning algorithms presented in this work which require line-of-sight checks make use of the same implementation presented in Algorithm (2). The one exception is the novel path-planning methodology presented in Chapter 3, which requires a more complex variant of the line-of-sight algorithm, but which is directly derived from Algorithm (2). This variation of Bresenham's line algorithm is described and discussed separately in more detail in Section 3.2. The line-of-sight is led by a driving axis going from s_{start} to s_{stop}. The driving axis always advances by $step_{major}$, determined based on the maxim displacement between the projections of the two nodes on the orthogonal axes. No displacement on one of the axes implies orthogonal movement which routes execution onto the "if" branch at Line 19. The "else" branch at Line 26 handles non-orthogonal movements, by accumulating the error in slope. The line advances on the minor axis (non-driving axis) if the error exceeds the slope, and the error is reset. Sometimes, moving on the minor axis results in no error (Line 32). This occurs in situations where the path passes through the vertex exactly, without affecting neighbouring cells, thus, the vertex needs to be checked to verify it is not a double corner (Line 34). The main while loop terminates early for any grid-cells that would block the line-of-sight, otherwise the algorithm returns true, in which case nodes s_{start} and s_{stop} have a direct line-of-sight with each other.

Algorithm 2 Line of Sight - 1D indexed grid variant

function LineOfSight(s_{start}, s_{stop})
  $\Delta_x \leftarrow (s_{stop} \bmod \mathbf{W}) - (s_{start} \bmod \mathbf{W})$ ▷ **W** - map width
  $\Delta_y \leftarrow (s_{stop}/\mathbf{W}) - (s_{start}/\mathbf{W})$
  $step_x \leftarrow (\Delta_x < 0 \implies -1) \wedge (\Delta_x \geq 0 \implies 1)$
  $step_y \leftarrow (\Delta_y < 0 \implies -\mathbf{W}) \wedge (\Delta_y \geq 0 \implies \mathbf{W})$ ▷ horizontal offset for quadrants I and IV relative to s_{stop}
  $step'_x \leftarrow (\Delta_x < 0 \implies -1) \wedge (\Delta_x \geq 0 \implies 0)$ ▷ vertical offset for quadrants I and II relative to s_{stop}
  $step'_y \leftarrow (\Delta_y < 0 \implies -\mathbf{W}) \wedge (\Delta_y \geq 0 \implies 0)$
  $s_{next} \leftarrow s_{start}$ ▷ current node with offset in line of sight expansion
  $s'_{next} \leftarrow s_{next} + step'_y + step'_x$
  $\Delta_x \leftarrow |\Delta_x|$
  $\Delta_y \leftarrow |\Delta_y|$ ▷ driving axis always increments
  $step_{major} \leftarrow (\Delta_x \geq \Delta_y \implies step_x) \wedge (\Delta_x < \Delta_y \implies step_y)$
    ▷ secondary axis increments with slope progression
  $step_{minor} \leftarrow (\Delta_x \geq \Delta_y \implies step_y) \wedge (\Delta_x < \Delta_y \implies step_x)$
  $\Delta_{max} \leftarrow \max\{\Delta_x, \Delta_y\}$
  $\Delta_{min} \leftarrow \min\{\Delta_x, \Delta_y\}$
  $error \leftarrow 0$ ▷ secondary axis increments with slope progression
  while $s_{next} \neq s_{stop}$ **do**
    $is\_free \leftarrow$ isFree(s'_{next})
    if $\Delta_{min} = 0$ **then** ▷ only moving horizontally or vertically
      if isDoubleCorner(s'_{next}) **then**
        return $false$
      if $\neg$ is_free **then**
        ▷ can't pass between 2 blocked cell tiles
        $nbr \leftarrow s'_{next} - step_{minor}$
        if $\neg$ isFree(nbr) $\vee$ $\neg$isFree($s'_{next} - step_{major}$) **then**
          ▷ previous cell not free, and neighbour not free
          return $false$
    else ▷ non-orthogonal line of sight expansion
      $error \leftarrow error + \Delta_{min}$
      if $error \geq \Delta_{max}$ **then** ▷ moving on minor axis
        if $\neg is\_free$ **then**
          return $false$
        $error \leftarrow error - \Delta_{max}$
        if $error = 0$ **then** ▷ moving diagonally
          $s_{diagonal} \leftarrow s'_{next} + (step_{minor} + step_{major} + \mathbf{W} + 1)/2$
▷ vertex defined by diagonally adjacent cells (see Equation 3.13)

Line of Sight - 1D indexed grid variant: Continued

```
                if isDoubleCorner($s_{diagonal}$) then
                    return $false$
            $s_{next} \leftarrow s_{next} + step_{minor}$
            $s'_{next} \leftarrow s'_{next} + step_{minor}$
            if $error = 0$ then
                $is\_free \leftarrow$ isFree($s'_{next}$)
        if $error \neq 0$ then                                  ▷ on driving axis
            if $\neg is\_free$ then
                return $false$
        $s_{next} \leftarrow s_{next} + step_{major}$            ▷ moving on driving axis
        $s'_{next} \leftarrow s'_{next} + step_{major}$
    return $true$                        ▷ $s_{start}$ and $s_{stop}$ have line of sight
```

2.4.1 A* with post-smoothing

To mitigate the shortcomings of the grid-constrained A^* algorithm, Thorpe et al. [54] introduce a smoothing technique to reduce the number of nodes and shorten the path solution.

Grid-constrained algorithms have some shortcomings when searching on octile grids. The drawback of A^* is that it can only ever move either in straight lines or at 45° degree angles and a node's parent can only be an immediate neighbour. This means that a node can only belong to its parent's Moore neighbourhood 1.3, meaning that any node is orthogonally or diagonally adjacent to its parent. This behaviour of A^* produces unrealistic looking paths. In the context of gaming, it can diminish the user experience and in the context of robotics, it would increase energy consumption because of following longer paths or preventing conservation of momentum. A simple way to address this issue is by path smoothing. A^* can be transformed into an any-angle path-finding algorithm by applying a post-smoothing procedure after the grid-constrained, shortest-path solution has been reached by A^*. Algorithm 2 describes how the post-processing step can be applied to a path returned by the A^* algorithm to "smooth" and therefore shorten the path. The post-smoothing procedure involves performing a a line-of-sight check (e.g. using the Bresenham line algorithm 2) between pairs of nodes in the path. If two nodes in the path have a line-of-sight, the nodes that connect them can be pruned from the path. In other words, for a path P, the parent of a node $s_i \in P$ can be any node $s_j \in P$ with $j < i - 1$ as long as $LineOfSight(s_i, s_j)$ is true.

Algorithm 2 A* Post-Smoothing [4]

function PostSmoothPath($[s_0, \ldots, s_n]$) ▷ $s_0 \equiv s_{start}$; $s_n \equiv s_{goal}$
  $k \leftarrow 0$
  $t_k \leftarrow s_0$
  foreach $i \in 1, \ldots, n-1$ **do**
    if $\neg$ LineOfSight(t_k, s_{i+1}) **then**
      $k \leftarrow k + 1$
      $t_k \leftarrow s_i$
  $k \leftarrow k + 1$
  $t_k \leftarrow s_n$
  return $[t_0, \ldots, t_k]$

2.4.2 Theta* Algorithm

The *Basic Theta** algorithm is an any-angle path-planning variation of A^* that produces near-optimal solutions [55] with a running time comparable to that of A^* on 8-directional grids. However, one disadvantage is that it can often find non-taut paths that make unnecessary turns, It also operates slower that other algorithms because it performs line-of-sight checks on-top of a full A^* search.

*Theta** [56] operates in a similar fashion to A^* which it interleaves with path smoothing. When expanding a vertex, it checks for a successor with a direct line-of-sight to the parent of the vertex. If such is the case, it bypasses the vertex, and instead connects the successor to the parent and, similar to A^*, assigns a cost score (distance travelled), to the node accordingly. The main difference between A^* and Theta* algorithms is the change in the cost computing function. The implementation of the Theta* variant of the cost function is presented in Algorithm 3. Theta*, given a node s and it's immediate neighbour s', when s' has line-of-sight to the parent of node s, the parent becomes a direct parent for node s' (see Line 4 in Algorithm 3).

Algorithm 3 Theta* [4]

function ComputeCost(s, s')
    if LineOfSight($parent(s)$, s') **then**
        if $g(parent(s)) + c(parent(s), s') < g(s')$ **then**
            $parent(s') \leftarrow parent(s)$
            $g(s') \leftarrow g(parent(s)) + c(parent(s), s')$
    else
        if $g(s) + c(s, s') < g(s')$ **then**
            $parent(s') \leftarrow s$
            $g(s') \leftarrow g(s) + c(s, s')$

*BasicTheta** is not optimal, meaning that it is not guaranteed to find true shortest paths. As explained in [4], this is due to the fact that the parent of a vertex, s, has to be either a visible neighbour or the parent of a visible neighbour of s. *BasicTheta** can also have unnecessary heading changes that do not correspond to the corners of blocked cells, a property that holds true for all shortest paths.

In a variation of the algorithm, named *StrictTheta** [57], Shunhao Oh et al. demonstrate that by restricting the search space of Theta* to taut paths, the algorithm can, in most cases, find shorter paths. They accomplish this by introducing a tautness check between a vertex, its parent and its grandparent. Non-taut paths incur an additional penalty in their cost. A second variation of *StrictTheta**, *RecursiveStrictTheta** [57], extends the tautness check beyond the grandparent, recursively checking for tautness until the first ancestor with a line-of-sight to the explored to the vertex. The two *StrictTheta** variants show a good improvement over Theta* in finding taut paths, that are closer to optimal. As mentioned in their work, Shunhao Oh et al. [57] make an argument for online path-planning algorithms: “In practice, slight sub-optimalities in the found path is often not an issue, but non-taut paths would contribute to the perceived irrationality of the agent, as the agent takes paths with clearly better alternatives. A good grid-based any-angle path-finding algorithm is fast, can compute near-optimal paths, and is online."

2.4.3 Lazy Theta* Algorithm

A variation on the Basic Theta* algorithm, Lazy Theta* aims to reduce the number of visibility checks. Basic Theta* is overly ambitious in performing line-of-sight checks even when it doesn't have to. When expanding a vertex s, it performs a visibility check for each unexpanded neighbour of s regardless of those vertices ever being expanded.

Nash et al. argue for the reduction of line of sight checks by delaying them until necessary. Lazy Theta* delays visibility checks by assuming that s has line-of-sight from $parent(s)$. When the algorithm expands s, it checks for a line-of-sight between s and its parent. If not, it updates the its g-score by using the g-score of its predecessors and proceeds to expand s. Whenever a line of sight test fails, a costly clean-up step is required to undo the effect of an incorrect assumption. Lazy Theta* attempts to refine Theta* by finding similar paths despite performing fewer line of sight tests. The paths returned by Lazy Theta* are not always the same as those returned by Theta* since the edge relaxation occurs at a different point in the iteration [58].

An additional variant, Lazy Theta* with Optimizations can find longer paths with a decrease in runtime. It does so by using weighted h-values with weights greater than one (similar to Weighted A* [59]): $h(s) = w * c(s, s_{goal})$. This variation can reduce runtime without a significant increase in path lengths while performing two orders of magnitude fewer line-of-sight checks and more than one order of magnitude fewer vertex expansions [4]. The pseudo-code for Lazy Theta* is presented in Algorithm 4.

Algorithm 4 Lazy Theta* [4]

```
function Main(s_start, s_goal)
    g(s_start) ← 0
    parent(s_start) ← s_start
    opem ← ∅
    opem.Insert(s_start, g(s_start) + h(s_start))
    closed ← ∅
    while open ≠ ∅ do
        s ← open.Pop()
        SetVertex(s)
        if s = s_goal then
            return "path found"
        closed ← closed ⋃ s
        foreach s′ ∈ nghbr_vis(s) do
            if s ∉ closed then
                if s ∉ open then
                    parent(s′) ← NULL
                    g(s′) ← ∞
                UpdateVertex(s, s′)
    return "no path found"

function SetVertex(s)
    if ¬ LineOfSight(parent(s), s) then
        parent(s) ← argmin_{s′∈nghbr_vis(s)∩closed}(g(s′) + c(s′, s))
        g(s) ← min_{s′∈nghbr_vis(s)∩closed}(g(s′) + c(s′, s))

function ComputeCost(s, s′)
    if g(parent(s)) + c(parent(s), s′) < g(s′) then
        parent(s′) ← parent(s)
        g(s′) ← g(parent(s)) + c(parent(s), s′)
```

2.4.4 Anya

One of the most notable additions to the any-angle path-finding family, is an optimal any-angle path-planning algorithm named Anya [24],[25]. Its optimality guarantee and any-angle property offer it the status of being the first online optimal any-angle algorithm. Unlike most shortest-path algorithms, Anya does not search over individual nodes in the grid. Rather, it constructs 2-dimensional visibility cones consisting of a root (i.e. node in the graph) and an interval (i.e. horizontal bound region delimited by bounds visible from the root). Each visibility region takes the form of a tuple (I, r), where r represents a root (a vertex corresponding to an outer corner of an obstacle) and I represents an interval describing all the points along a row of the grid-map visible from the root in question. To direct the search, Anya estimates, by means of a heuristic, the shortest distance from a root r to the goal node that passes through the interval I. Anya performs the search by expanding rows from r and generating intervals in which, if a turning point is found, it becomes a new root to be considered for expansion. The algorithm terminates when an interval containing the goal node is expanded. The path is reconstructed similar to A* (and related algorithms), by following the parents of interval roots from the goal to the start node.

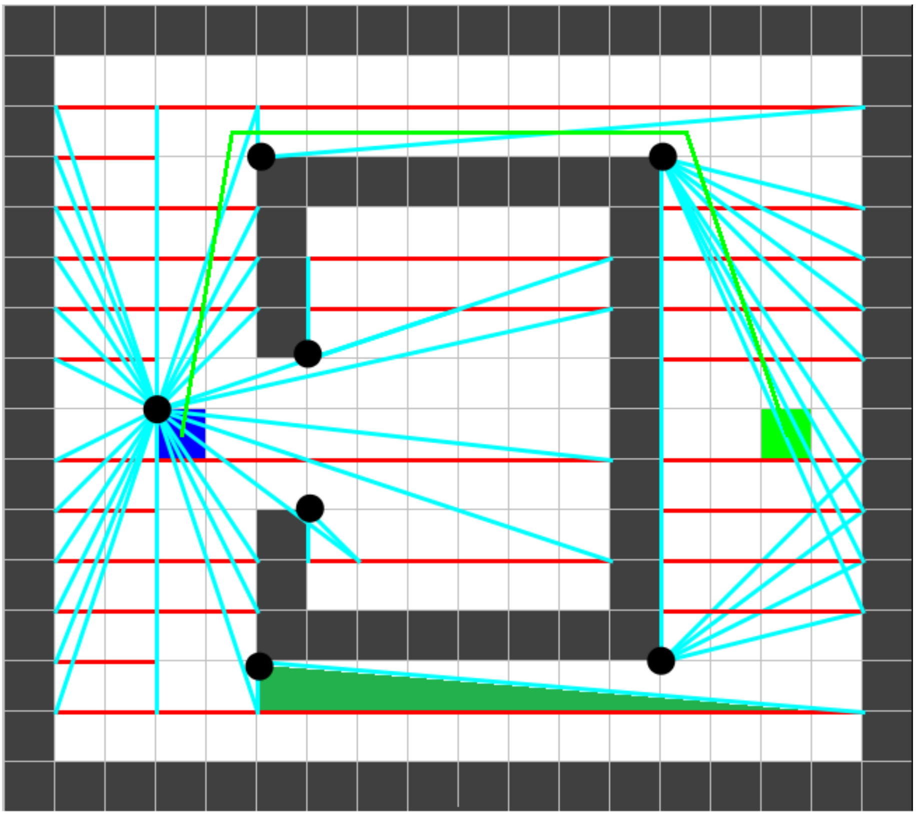


Figure 2.7: Screen-shot of an expansion of the Anya search tree, with red lines representing the interval of a row, the black circles mark interval roots, and the light blue lines the visibility cone of each interval; green line - shortest path solution; blue & green cells - start & stop respectively

In Figure 2.7, an example of a visibility cone is marked by a dark green triangle towards the bottom of the image. The interval of a tuple (I, r) is represented by a red line (the longer bottom cathetus of the marked triangle in our example). Each interval lies between its left and right bounds, delimited by bright blue lines (the short cathetus and the hypotenuse of the green triangle in Figure 2.7). The interval lines intersect each other in the tuple's root, r. In our figure, where the green triangle touches the lower left vertex (marked by the black) a new root r is created with its parent as the start node (dark blue square).

Uras and Koenig observe in their analysis [29] that Anya "is the algorithm with the highest variance in runtime between different types of maps". As Anya expands over intervals rather than grid cells, it can transverse over open spaces with ease. Harabor et al. provide an open-source implementation of Anya, publicly available on the code repository "Bitbucket" at [60].

Uras and Koenig note that for maps such as those found in the "Dragon Age: Origins" database, for example, that have many tight corridors, Anya's performance is degraded, as it requires many more interval expansions before reaching the goal. In their analysis, Anya came out as the slowest algorithm on random maps and on the "Dragon Age: Origins" database. It is to be noted, however, that the implementation used by [29] differs from the one provided by Harabor et al. [60].

Using their own implementation of Anya, Harabor et al. experimentally show Anya to outperform four purely online algorithms [25]. The four algorithms in question are A* [18], Theta* [55], Lazy Theta* [50] and Field D* [22].

In their implementation, which uses a bit-packed map representation (i.e. array of integers in which the binary representation of the integer reflects a cell's occupancy), traversing a row becomes a very cheap operation, requiring bit-shift operations, which avoids checking individual cells for occupancy (e.g. an integer consisting of 32 bits with value 0 representing a free-space interval). Furthermore, the cost of performing lookups for grid map occupancy is minimal as the search can be aided by modern microprocessors capable of caching the occupancy map for faster lookup times. As such, even though Anya may cover a large search space, as seen in 2.8, it can do so with little impact on performance.

The performance increase of the Anya algorithm relative to A* makes Anya a good candidate for evaluation as a fast, online, optimal, any-angle path-planning algorithm. However, in a real-world scenario, it could prove prohibitively costly for a robotic platform (restricted by battery, wear and tear, sensors, etc.) to perform a search in an equivalent manner to the method employed by the Anya algorithm. This is due to large search-space coverage in environments with large open spaces or ones that do not have a solution (Figure 2.8).

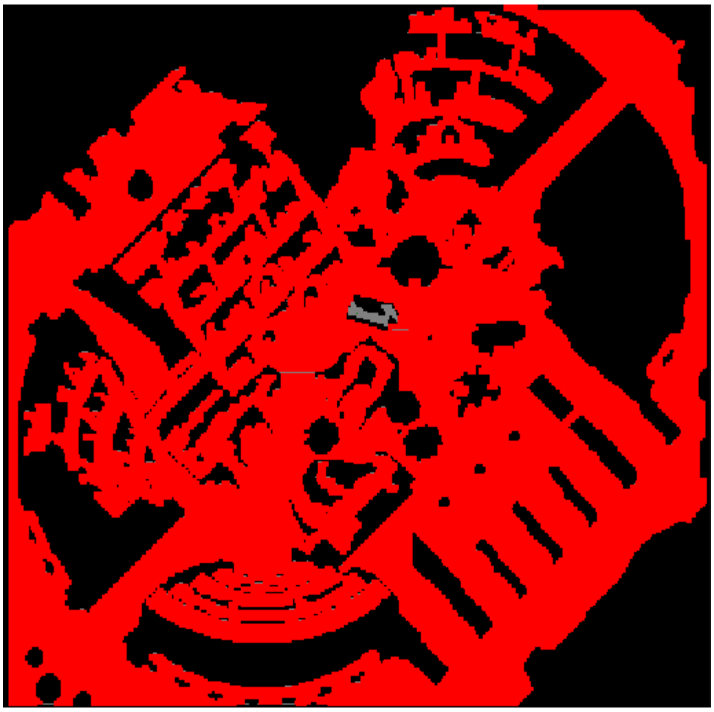


Figure 2.8: Screen-shot of Anya performing an exhaustive search (explored cells shown in red) for a scenario with no solution (map AR0700SR - Baldur's Gate – modified to isolate goal)

2.5 Additional algorithms

2.5.1 Dubins Curves

This method was first proven by Lester Dubins in 1957. The Dubins path refers to the shortest curve which connects two points a in a two-dimensional Euclidean plane. It showed that the shortest path would consist of straight line segments (S) and/or maximum curvature (C) [61].

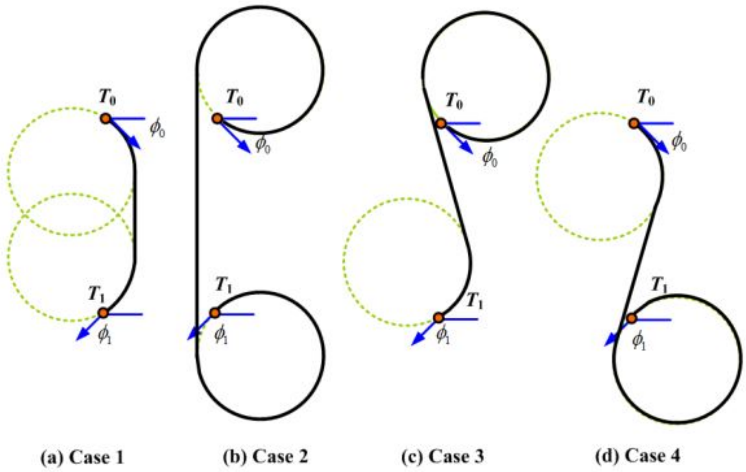


Figure 2.9: The family CSC paths of four combinations. Case (a) RSR, Case (b) LSL, Case (c) LSR, Case (d) RSL [62]

There are considered to be two families of curves, which are a combination of the straight and curved lines: CCC and CSC. The CCC family contains curves in the formation of LRL and RLR, where L and R denote left turn arc and right turn arc respectively. The family CSC contains four combinations: LSL, LSR, RSR, RSL, where S denotes the straight segment [62], this is shown in 2.9.

2.5.2 Artificial Potential Fields

Artifical potential field (APF) is a well-known path planning algorithm methodology. It has the core concept to replicate the characteristics of electostatic potential. In particular, due to its 'mathematical elegance and simplicity' it has been particularly favoured due to its effectiveness in real-time obstacle avoidance [63]. Under the influence of the repulsive potential fields and the attractive potential field, the robot goes from a high to low (i.e. the global minimum) potential field, along the negative gradient, and would also be repellent to any obstacles (i.e the local maximum). APF is known to be suitable for online and offline path generation due to its reactive nature [64]. The APF methodology is depicted in 2.10.

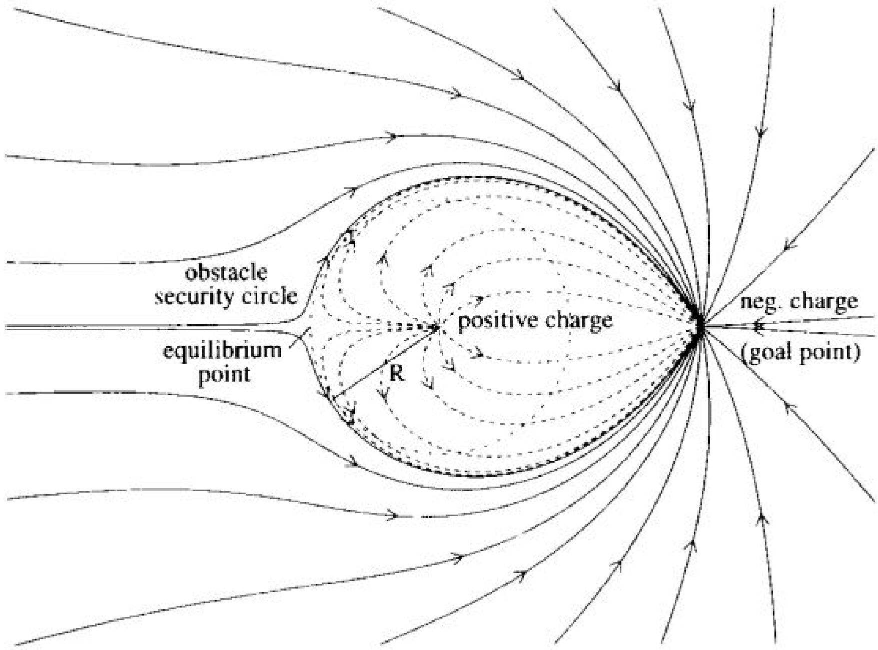


Figure 2.10: Electric potential field, showing some of the gradient lines. This diagram features a positive charge at the security circle, and a negative charge at goal. [64]

There are several disadvantages with using the APF methodology, though there have been improvements to the original algorithm in order to counteract some of these issues incurred, [65], [66].

1. While the robotic system is further away from the goal point, the attractive force is great. Therefore, this may lead to the robot to come too close to any obstacles.
2. Goals Non-Reachable with Obstacle Nearby (GNRON), [67].

3. When the potential field between the repulsive and attractive forces are almost equal or equal in nature, the potential force of the system is zero, and hence it would cause a trap at the local minima, or oscillate [68],[69].

2.6 Chapter Summary

This chapter introduced path-finding methodologies in detail. These methodologies have a number of properties - correctness, completeness and optimality. There are various classes of path planning algorithms, including – Bug algorithms, Grid constrained algorithms and any angle algorithms.

Bug algorithms are a class of path-finding methodologies for navigating unknown environments and operate by alternating between two behaviours – wall–following and motion-to-goal. Grid–constrained algorithms such as grid–constrained $A*$ find the shortest path composed of edges bound by the grid. Additional methodologies include Dubins curves and Artificial potential fields.

Finally, any angle algorithms are a class, most relevant to this work, which search for paths between two nodes in the free-space by propagating information along graph edges, similar to grid–constrained algorithms, but without restricting the paths which are formed by graph edges.

Chapter 3

Ray Path Finder: Path construction

This chapter introduces a number of algorithms developed with the purpose of aiding searches for the path-finding methodology described in Chapter 4 - An Any-angle path planner. Algorithms for performing line-of-sight and contour tracing are introduced and a number of different concepts, such as path direction, sidedness, pruning and redundancy, are explored in depth.

3.1 Introduction

The novel path-planning algorithm introduced in Chapters 3 & 4 operates under the free-space assumption, in the sense that it optimistically assumes that there are no obstacles between a start and goal node, nor between any two nodes that it identifies as part of a path. The principle of operation of the overarching path-planning method is simple to understand. The search is initiated by performing a line-of-sight towards the goal, until an obstacle is encountered. Afterwards, the search is propagated through two diverging paths travelling in opposite directions along the edges of the encountered obstacle until it is determined that the paths can leave the boundary and resume moving in a straight line towards the goal. Once a path has reached a goal, the algorithm checks if the path that arrived to the target has line-of-sight between its nodes, and if not, repeats the same procedure mentioned earlier. The underlining algorithm that is conducive to the shortest path solution is described in greater detail in Chapter 4.

We introduce the notion of "path state" as a means of keeping track of the different stages a path goes through during its life-cycle. A path can have 6 possible states, as depicted in Figure 3.1.

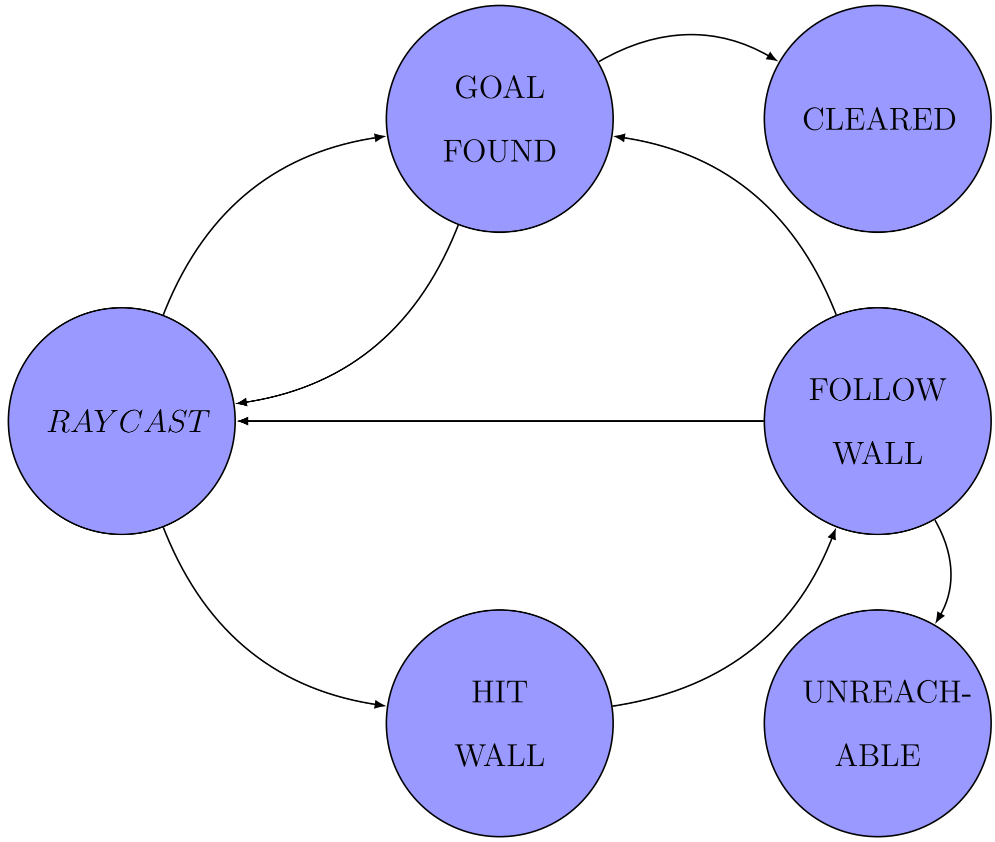


Figure 3.1: Path states with possible transitions

Each path being explored can be thought of as an independent state-machine (not unlike a bug algorithm) which stores the state specific to the desired behaviour a path should adopt, given its immediate environment e.g. travelling in a straight line if there are no obstructions in the direction of travel, following an obstacle's wall after having intersected its boundary, terminating if unreachable.

All classes of paths that terminate early (e.g. redundant, looping, locked-in, locked-out) are bundled under the umbrella term of unreachable. Throughout the life-cycle of each path, the state machine is updated accordingly, depending on the desired behaviour. Figure 3.1 illustrates the possible transitions of paths from one state to another.

A number of functions are necessary for the path-finding algorithm to exhibit the behaviour mentioned. All of the algorithms operate on a 2D grid-map with uniform cost representing a binary space-state (1-for occupied; 0-for unoccupied). Firstly, moving towards a target involves performing a line-of-sight and identifying the point of intersection if encountering an obstacle (Section 3.2). Secondly, following the obstacle's wall implies tracing the contour of the obstacle (Section 3.3). The algorithms that assume the role of constructing a path identify corners at points of interest around an obstacle's edges (Section 3.4), while those for maintaining a path's tautness imply pruning nodes from the path (Section 3.6). A path also maintains a sense of direction (Section 3.5) that helps it decide when to leave the obstacle boundary so as to avoid deadlock. Lastly, the search aims to ensure termination and avoid redundancy in paths (Sections 3.7 & 3.8). The following sections detail these features.

3.2 Line of sight with intersection

A path is defined as clear or unblocked if any two subsequent vertices within the path have line-of-sight between them. Any two vertices of a grid graph are defined as having line-of-sight if the segment connecting them does not pass through the interior of any occupied grid cells nor between any orthogonally or diagonally adjacent pair of occupied grid cells (i.e. any pair of occupied cells that share one or two vertices). We refer to a vertex shared by exactly two diagonally adjacent occupied cells as a double corner (Figure 3.2).

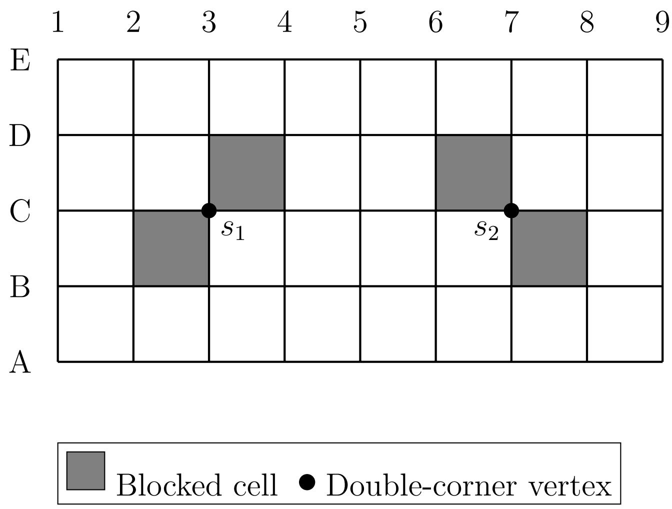


Figure 3.2: Line of Sight: s_1 and s_2 – examples of double corner vertices

A vertex shared by exactly three occupied cells is referred to as an inner corner, while a vertex described by exactly three unoccupied cells and one occupied cell is referred to as an outer corner (Figure 3.3). A true shortest-path solution would only be comprised of outer corners.

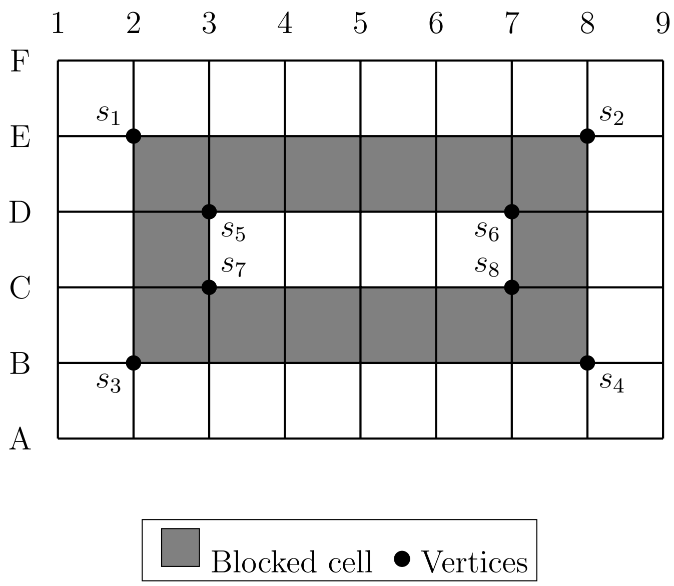


Figure 3.3: Line of Sight: s_1, s_2, s_3 & s_4– examples of outer corner vertices; s_5, s_6, s_7 & s_8– examples of inner corner vertices;

While, for simplicity, other any-angle algorithms (e.g. A* Post Smoothing, Theta*, Lazy-Theta*) do not explicitly disallow paths through double corners by default, RPF enforces this limitation for both "wall following" and "line-of-sight" behaviours. This restriction is consistent with the maps from the "Moving AI Lab" database [70] that we use for experimental evaluation.

Performing a line-of-sight on a 2D regular grid is equivalent to drawing a line plot on a raster display. A variation of Bresenham line drawing algorithm (Algorithm 2) is used for this purpose.

As described in Algorithm 2, Theta* performs a line-of-sight check after a node has been expanded, in order to update its parent if a line of sight exists between it and its neighbour's parent, and as such, the line of sight algorithm requires returning a simple yes-or-no answer. The Bresenham algorithm employed by Theta* [71] and Lazy Theta* [50] allows a straight line to pass between diagonally adjacent blocked grid cells [4], for the purpose of simplicity. For these algorithms, cutting corners through walls is prevented by ignoring double-corner vertices when expanding a node's neighbours, before the line-of-sight check is to be executed. This is unfortunately not the case for the RPF algorithm, as it does not use the Bresenham algorithm as a "string-pulling" technique, but rather incorporates the line-of-sight into the search-space expansion procedure. As such, it is more similar in principle to the "ray-casting" technique used in computer graphics for determining intersections with objects for rendering purposes. RPF's behaviour is different from the Theta* family of pathfinding algorithms, as it uses Bresenham's algorithm to directly perform node expansion.

The variation on Bresenham's algorithm utilised by RPF requires two additional properties. It must actively not allow straight lines to pass between diagonally adjacent occupied grid cells, and, if no line-of-sight exists between the two nodes, it must provide the point of intersection with the obstacle in its path. This implies returning the the indices of two cells, the occupied and unoccupied cells that define the intersection. For instance, the examples illustrated in Figures 3.4 and 3.5 do indeed have a line-of-sight between the two nodes, as at least one edge is shared with a free cell and there exist no double-corners between s_{start} and s_{stop}. For figure 3.5, the cells D_2 through D_8 are unoccupied, while for 3.4, the cells from C_2 through C_5 and cells D_5 through D_8 are unoccupied and also the pair (C_5, D_5) prevent a double corner.

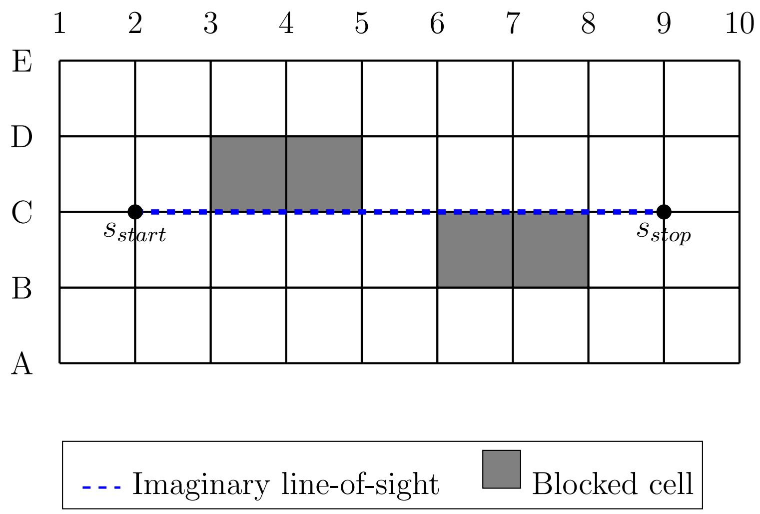


Figure 3.4: Line of Sight: Valid line-of-sight Example 1

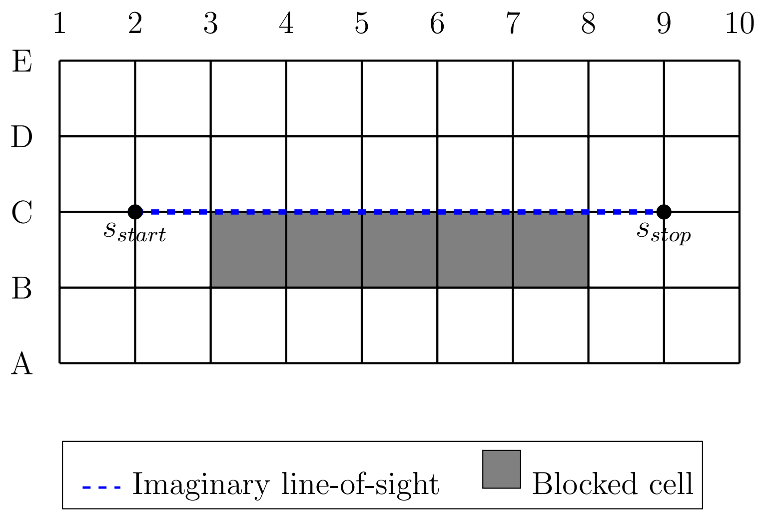


Figure 3.5: Line of Sight: Valid line-of-sight Example 2

For the configuration in Figure 3.4, the s_{start} vertex at C_2 has a line of sight to vertex s_{stop} at C_9. If the grid cell corresponding to vertex C_5 were occupied, it would break the line of sight as vertex C_5 would become a double corner. Similarly, if the grid cell corresponding to vertex D_5 were occupied, it would again break line of sight as vertex C_6 would become a double corner.

The complete line-of-sight method is presented in Algorithm 5. To exemplify the behaviour of the RPF variant of the line-of-sight algorithm, let us first consider the simple scenario described in Figure 3.6, with vertex C_2 as the the starting node (labelled as s_{start}) and vertex C_8 as the goal node (labelled as s_{stop}). The obstacle is represented by the blocked grid cells corresponding to vertices D_5 and C_5 respectively. The line-of-sight between s_{start} and s_{stop} is broken by the obstacle, as the line segment crosses between the two orthogonally adjacent occupied cells forming the obstacle. As the grid cells of s_{start} and s_{stop} are horizontal in our example, the difference in their y coordinate components is 0, which is assigned to Δ_{min} at Line 15. For this configuration, the algorithm only follows the orthogonal exploration branch at Line 22. The occupancy check first fails with C_5 at Line 40. The occupancy of the grid cell at D_5 decides if an obstacle was encountered. To obtain the index of D_5, one subtracts the step value for the minor axis (i.e. the y axis for horizontal travel) from the current cell index (Line 41).

In Figure 3.6, the start node at C_2 does not have line-of-sight to vertex s_{stop} at C_8, as it intersects an obstacle at C_5. In Algorithm 5, the occupancy tests at Lines 40 and 43 fail, while Line 44 passes, executing Lines 48 – 50. Thus $s_{pre\_intersect}$ becomes C_4 and $s_{intersect}$ becomes C_5 and the algorithm exits.

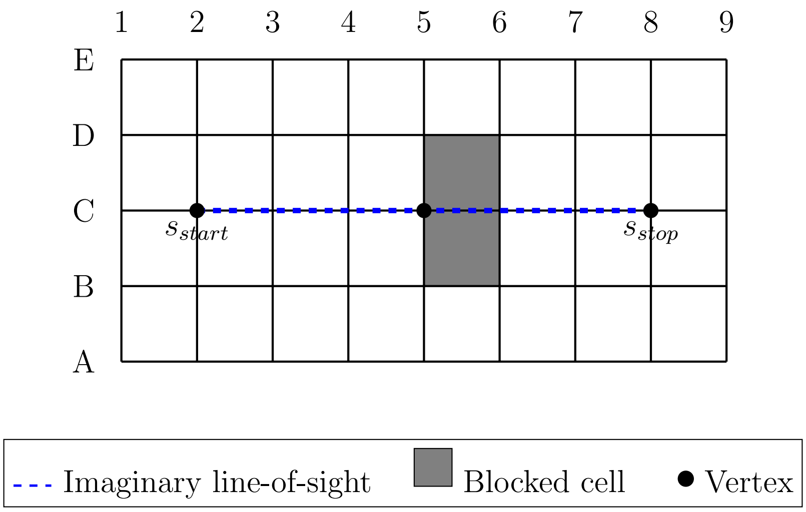


Figure 3.6: Example of line-of-sight intersecting obstacle

Observing the configuration in Figure 3.7, the s_{start} vertex at C_2 does not have line of sight to vertex s_{stop} at C_8, as it intersects an obstacle at C_5, similar to Figure 3.6. In this scenario, C_4 is occupied, meaning that the occupancy test at Line 44 fails, resulting in $s_{pre\_intersect} = D_4$ and $s_{intersect} = D_5$ through the execution of Lines 45 – 47.

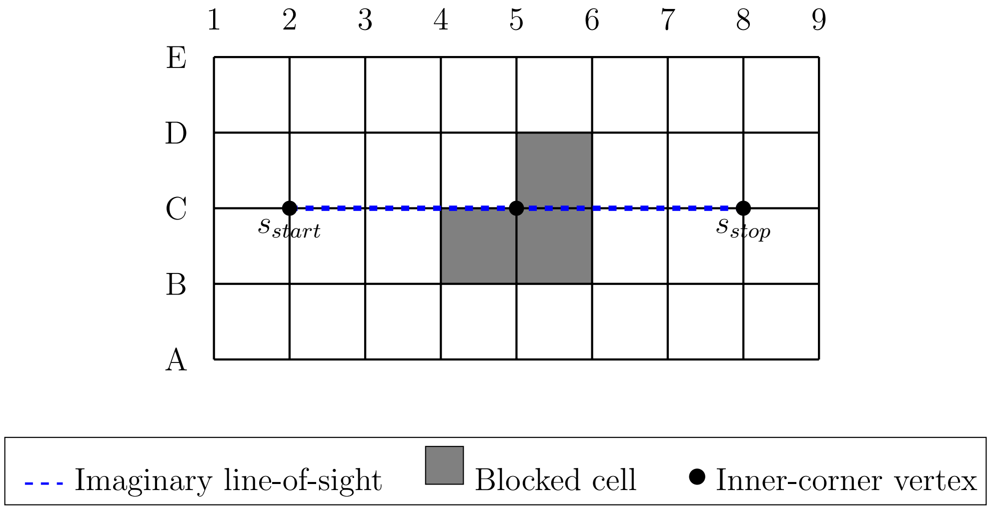


Figure 3.7: Line of Sight: Intersecting inner-corner

Double corner special cases are handled separately. Figures 3.8 & 3.9 are both handled by Line 24 in Algorithm 5, the difference being that $step_{major} < 0$ ($step_{minor} \equiv 0$ for orthogonal movement).

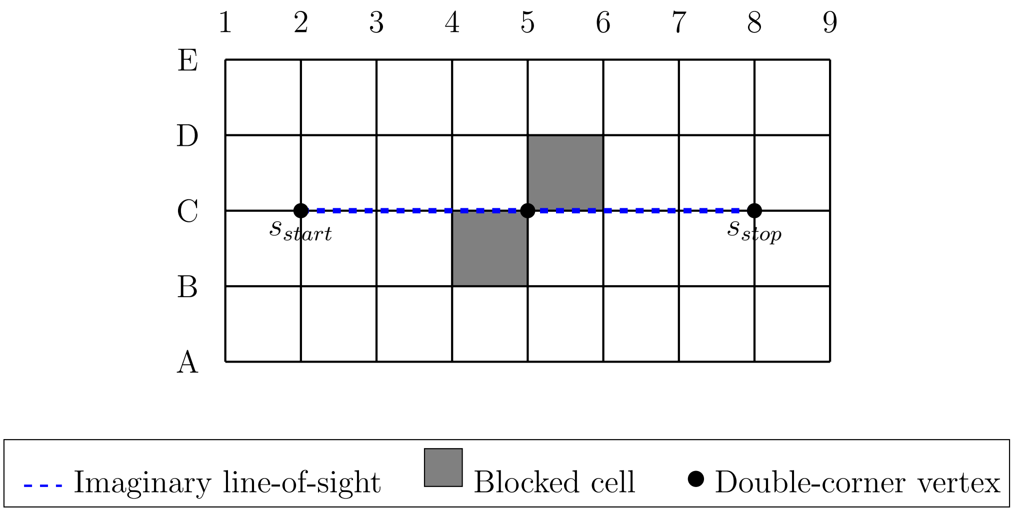


Figure 3.8: Line of Sight: Double Corner vertex with Free Cell

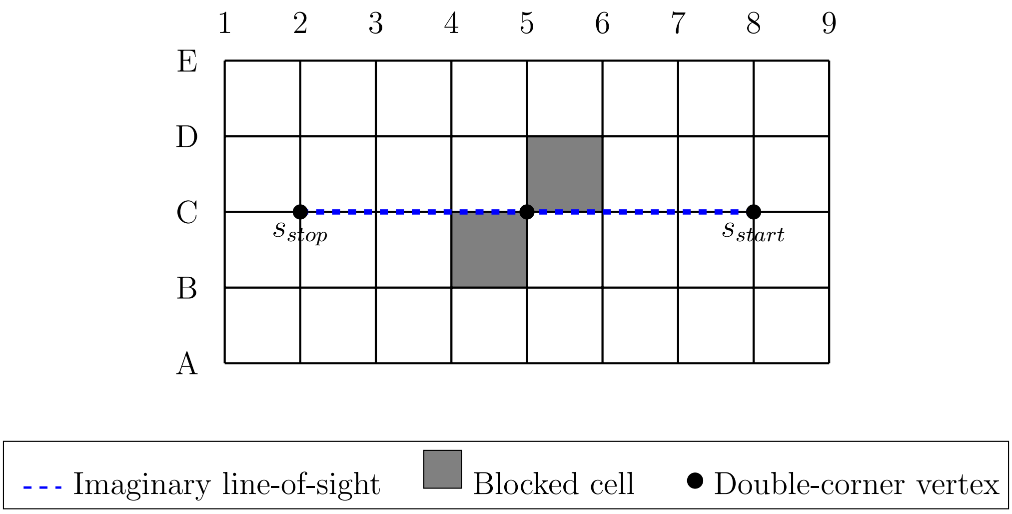


Figure 3.9: Line of Sight: Double Corner vertex with Free Cell – Negative step

Figures 3.10 & 3.11 present a double corner with an occupied cell and are both handled by Line 32 in Algorithm 5. The difference between them is that $step_{major} < 0$ for Figure 3.11 ($step_{minor} \equiv 0$ for orthogonal movement).

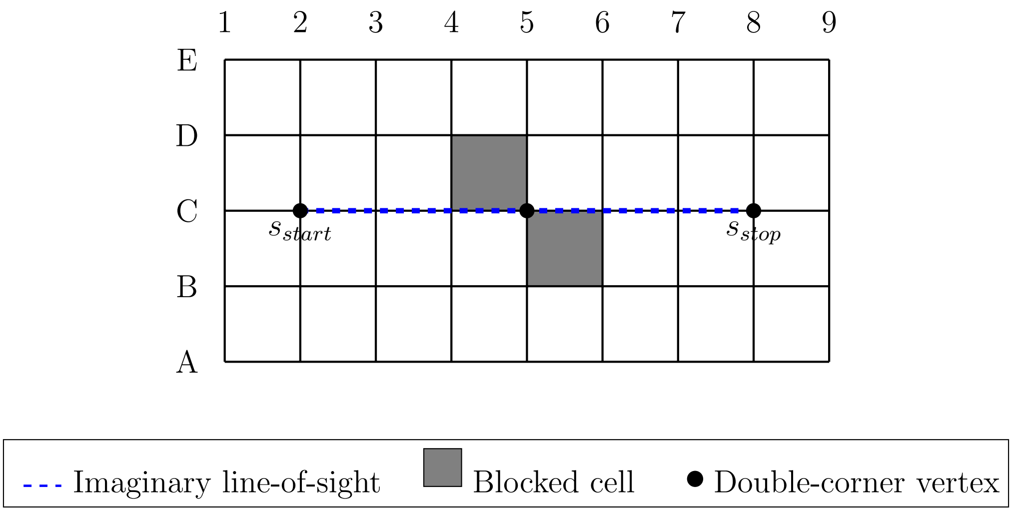


Figure 3.10: Line of Sight: Double Corner occupied vertex

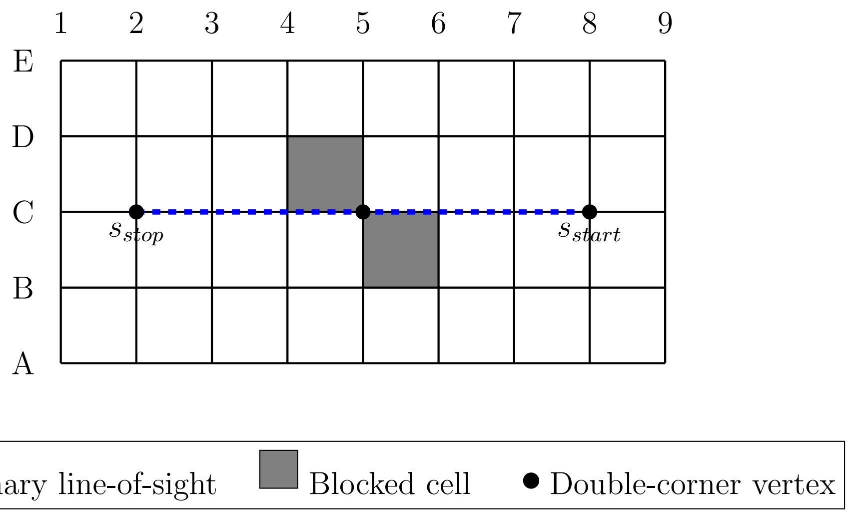


Figure 3.11: Line of Sight: Double Corner occupied vertex – Negative Step

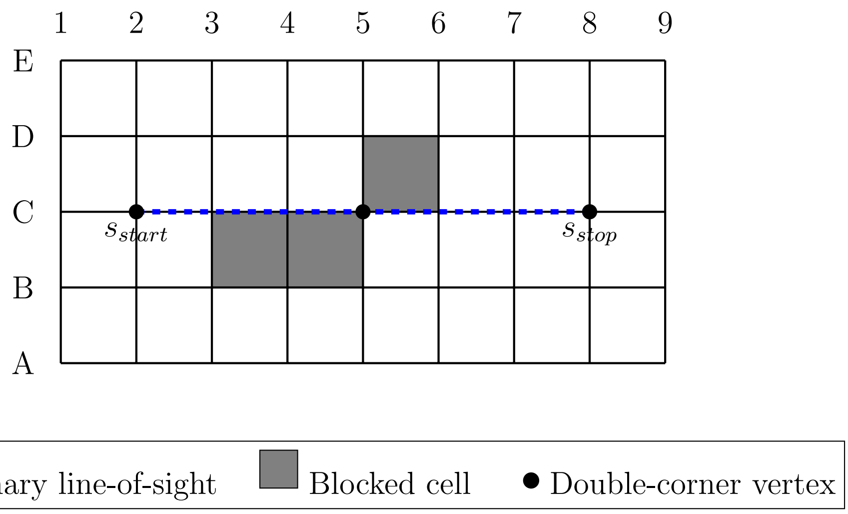


Figure 3.12: Line of Sight: Double Corner Vertex with free cell and Occupied previous step

The scenarios presented are identical for nodes in a vertical configuration. The algorithms imposes two restrictions on the global start and goal vertices of a search scenario, namely that their corresponding grid-cell (vertex at upper left corner of the observed cell) must be unoccupied (consistent with the map database used for evaluation [70]). An additional restriction is imposed such that neither start nor goal can be double-corners.

The s_{start} and s_{stop} vertices passed to the $LineOfSight$ function should not be confused with the global start and goal vertices passed to the Ray Path Finder algorithm. For the *LineOfSight* function, the parameters s_{start} and s_{stop} can take the values of either the global start and global goal (i.e. initial inputs of the find-path problem) vertices or they can take values of vertices belonging to a path. It is to be noted that the algorithm is only used to perform line-of-sight checks between vertices of a valid path. This implies that any vertices that are passed to the *LineOfSight* function are valid corners on the grid map, and as such, certain inconsistent scenarios will never occur, and are not addressed by the algorithm. A vertex is only ever added to a path if it is a valid corner on the corresponding 2D grid map (3.4), meaning that neither s_{start} nor s_{stop} can be double corners. Also, for any vertex belonging to a path, the 4 grid cells describing (sharing) the vertex have a 3-unoccupied/1-occupied configuration (see Figure 3.30), with the exception of the global start and goal vertices which can be surrounded by an arbitrary number of unoccupied cells (between 1 and 4).

Figures 3.13, 3.14, 3.15 & 3.16 illustrate the 4 possible cases of encountering a double corner while travelling diagonally and which are handled by Lines 60 - 62 in Algorithm 5.

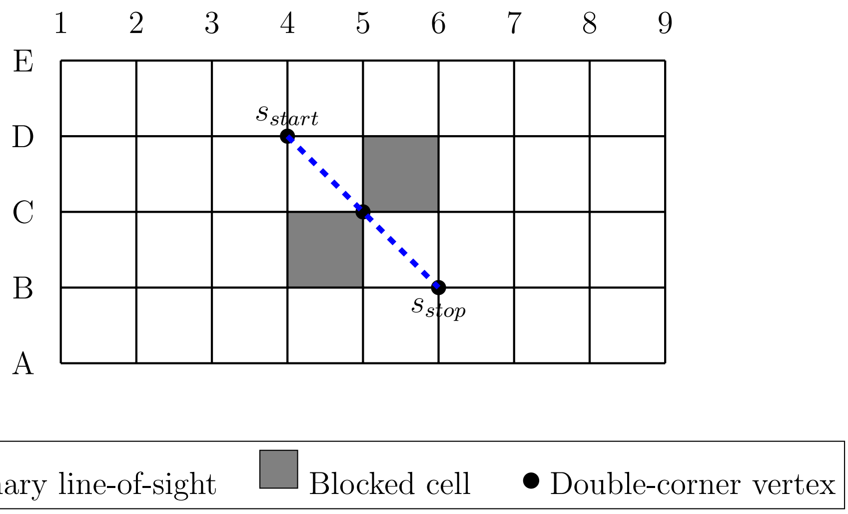


Figure 3.13: Line of Sight: Double corner with free vertex – diagonal step

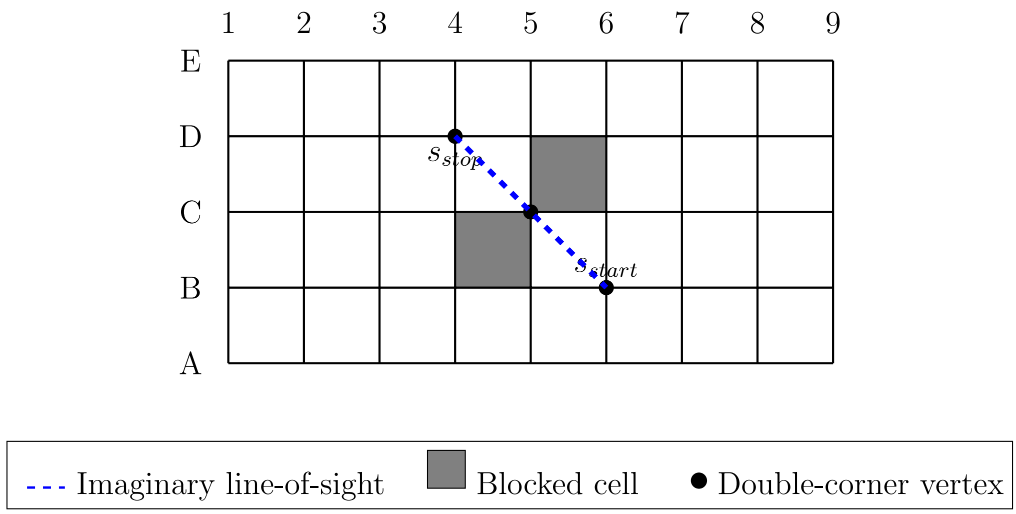


Figure 3.14: Line of Sight: Double corner with free vertex – negative diagonal step

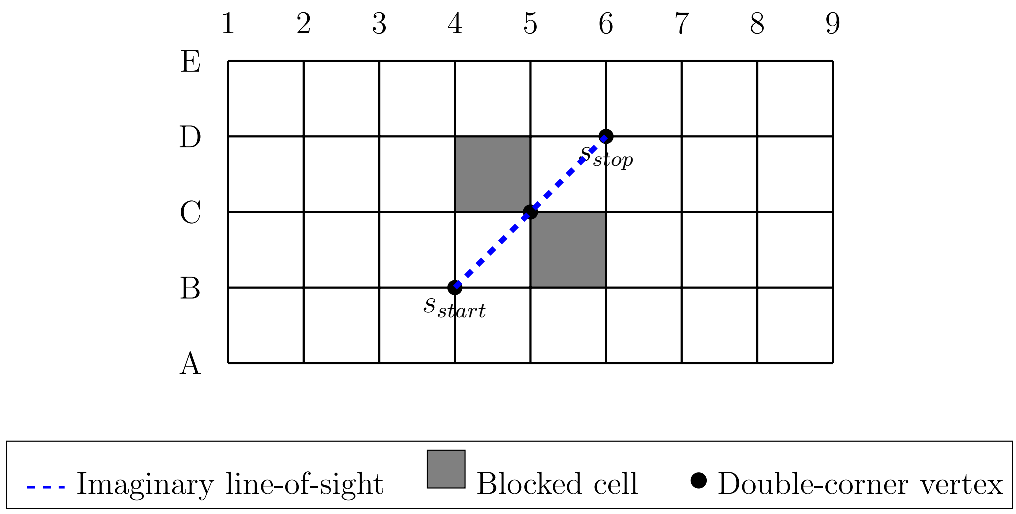


Figure 3.15: Line of Sight: Double corner vertex with occupied cell – diagonal step

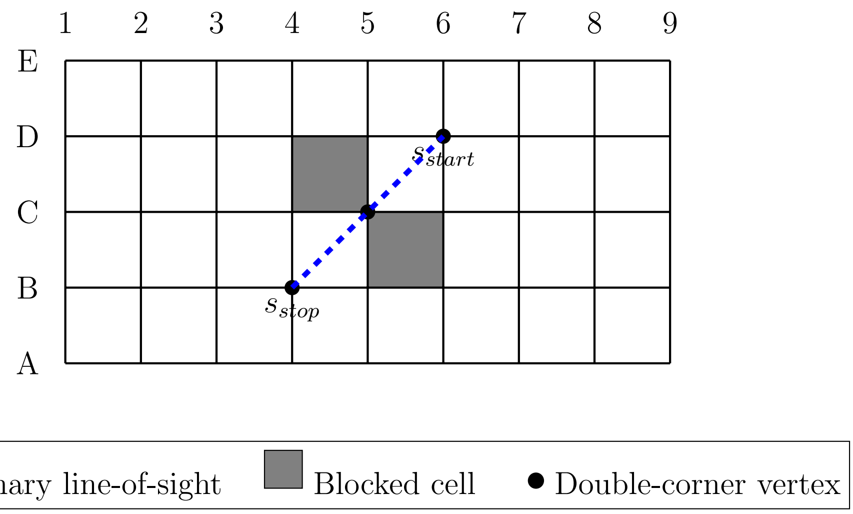


Figure 3.16: Line of Sight: Double corner vertex with occupied cell – negative diagonal step

A special case for the line-of-sight algorithm that requires to be handled separately is illustrated in Figure 3.17. The exception arises when the cell corresponding to a corner vertex is occupied and thus breaks the line of sight to a target. As an example, let us consider the vertex s_{start} at D_4, which corresponds to an occupied grid cell, and any of the 3 vertices $s_{stop1}(B_5)$, $s_{stop2}(B_6)$ or $s_{stop3}(C_6)$ that are obscured from view. For this configuration, $s_{pre\_intersect}$ is initialised to the occupied D_4 node (Line 18 in Algorithm 5) and the loop terminates early at Line 56. The condition at Line 78 identifies this scenario and $s_{pre\_intersect}$ is allowed to step backward (Line 79) and retrieve the E_3 cell as a pre-intersection point. This is allowed because s_{start} is a corner vertex, which implies that it must have 3 unoccupied cells describing it (i.e. tiles E_3, E_4 and D_3).

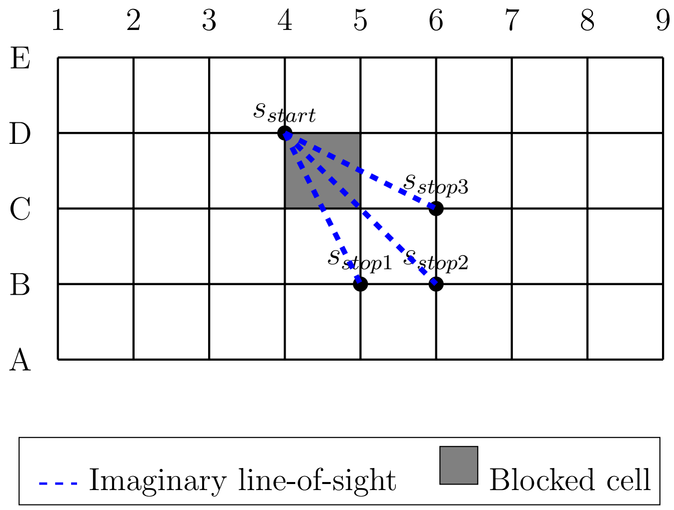


Figure 3.17: Line of Sight: Corner vertex with occupied cell and targets in quadrant four

The red highlighted tiles in Figure 3.18 exemplify the nodes that Algorithm 5 expands while moving at a 45° angle from s_{start} at tile 391 to s_{stop} at tile 205. Although the path (marked in green) passes directly through the vertices of the tiles 360, 329, 298, 267 and 236, the tiles have to be checked for occupancy, to ensure they are not double corners.

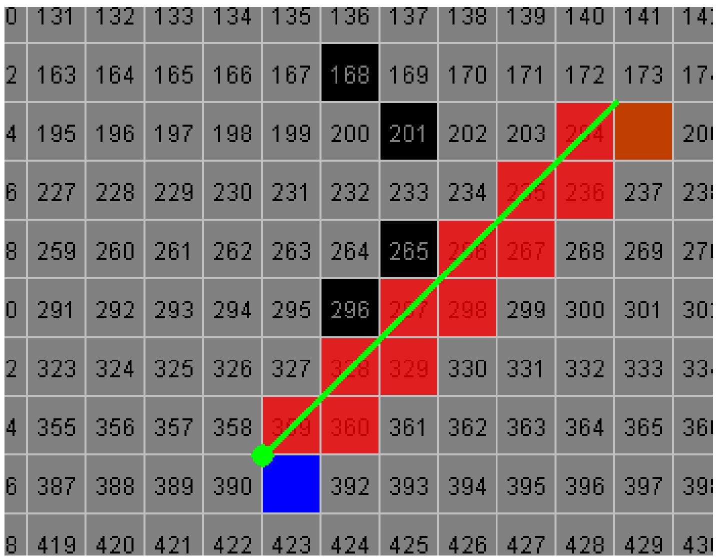


Figure 3.18: Expanded cells for diagonal Line of Sight (Screenshot from RPF application)

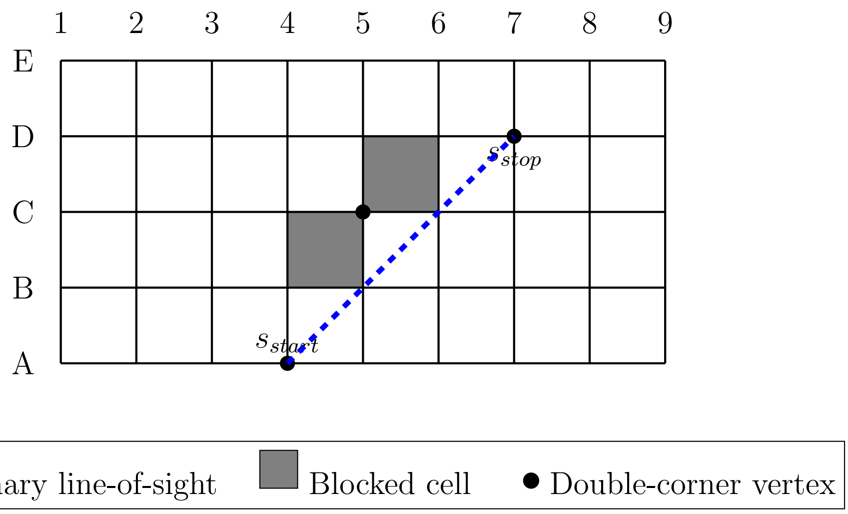


Figure 3.19: Line of Sight: Through double corner free cell – no intersection

The screen capture in Figure 3.20 illustrates the expanded cells for a valid line-of-sight with a single blocked obstacle at cell 201. Notice that the only extra cell that needs to be expanded is the one above the blocked cell. As 169, the neighbouring tile of the occupied tile 201 (expanded at Line 41) is free (Line 43), the algorithm continues as normal until reaching s_{stop} at tile 205.

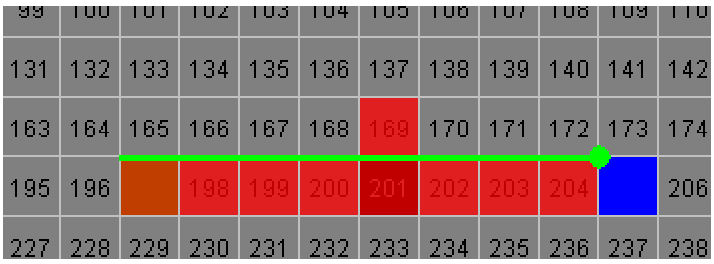


Figure 3.20: Expanded cells for horizontal Line of Sight with obstacle (Screenshot from RPF application)

The screen captures in Figures 3.21 & 3.22 illustrate the special cases of the line-of-sight being blocked by double corners while moving horizontally (Line 23). The red dot marks the vertex of the free cell $s_{pre\_intersect}$, and the purple dot marks the vertex of the occupied cell $s_{intersect}$. For Figure 3.21, Line 23 is true, as tile 201 is a double corner and, being an occupied cell, the else branch at Line 35 stores $s_{pre\_intersect} = 200$ and $s_{intersect} = 201$ and exits the loop. For Figure 3.22, the algorithm follows the same path and yields $s_{pre\_intersect} = 202$ and $s_{intersect} = 201$.

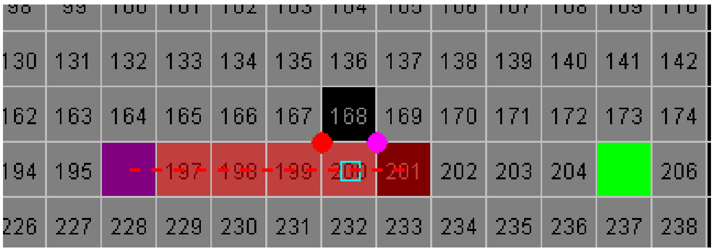


Figure 3.21: Double corner intersection (red–vertex of pre-intersection, purple–vertex of intersection) (Screenshot from RPF application)

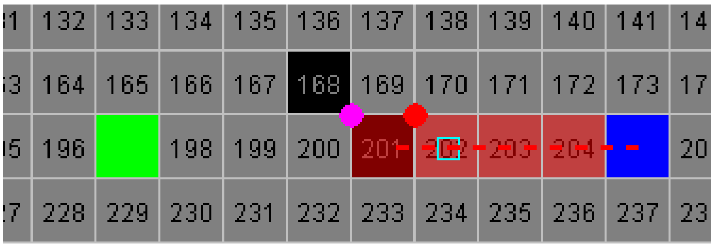


Figure 3.22: Double corner intersection – with negative step (red–vertex of pre-intersection, purple–vertex of intersection) (Screenshot from RPF application)

Algorithm 5 Line of Sight - Ray Path Finder

function $\textsc{LineOfSight}(s_{start}, s_{stop}, \mathbf{W})$
    $\Delta_x \leftarrow (s_{stop} \bmod \mathbf{W}) - (s_{start} \bmod \mathbf{W})$
    $\Delta_y \leftarrow (s_{stop} \backslash \mathbf{W}) - (s_{start} \backslash \mathbf{W})$
    $step_x \leftarrow (\Delta_x < 0 \implies -1) \wedge (\Delta_x \geq 0 \implies 1)$
    $step_y \leftarrow (\Delta_y < 0 \implies -\mathbf{W}) \wedge (\Delta_y \geq 0 \implies \mathbf{W})$
        $\triangleright$ horizontal offset for quadrants I and IV relative to s_{stop}
    $step'_x \leftarrow (\Delta_x < 0 \implies -1) \wedge (\Delta_x \geq 0 \implies 0)$
        $\triangleright$ vertical offset for quadrants I and II relative to s_{stop}
    $step'_y \leftarrow (\Delta_y < 0 \implies -\mathbf{W}) \wedge (\Delta_y \geq 0 \implies 0)$
        $\triangleright$ current node in line of sight exploration
    $s_{next} \leftarrow s_{start}$
        $\triangleright$ current node with offset in line of sight expansion
    $s'_{next} \leftarrow s_{next} + step'_y + step'_x$
    $\Delta_x \leftarrow |\Delta_x|$
    $\Delta_y \leftarrow |\Delta_y|$
        $\triangleright$ driving axis always increments
    $step_{major} \leftarrow (\Delta_x \geq \Delta_y \implies step_x) \wedge (\Delta_x < \Delta_y \implies step_y)$
        $\triangleright$ secondary axis increments with slope progression
    $step_{minor} \leftarrow (\Delta_x \geq \Delta_y \implies step_y) \wedge (\Delta_x < \Delta_y \implies step_x)$
    $(\Delta_{max} \geq \max\{\Delta_x, \Delta_y\}$
    $(\Delta_{min} \geq \min\{\Delta_x, \Delta_y\}$
    $error \leftarrow 0$
    $is\_edge \leftarrow false$ $\triangleright$ track orthogonal move on edge of blocked tile
    $s_{pre\_intersect} \leftarrow s_{start}$ $\triangleright$ index of cell tile prior to intersection
    $s_{intersect} \leftarrow 0$ $\triangleright$ index of cell tile at intersection
    while $s_{next} \neq s_{stop}$ **do**
        $is\_free \leftarrow \textsc{isFree}(s'_{next})$
        if $\Delta_{min} = 0$ **then** $\triangleright$ only moving horizontally or vertically
            if $\textsc{isDoubleCorner}(s'_{next})$ **then**
                if is_free **then**
                    if is_edge **then**
                        $s_{pre\_intersect} \leftarrow s'_{next} - step_{major} - step_{minor}$
                        $s_{intersect} \leftarrow s'_{next} - step_{minor}$
                    else
                        $s_{pre\_intersect} \leftarrow s'_{next}$
                        $s_{intersect} \leftarrow s'_{next} + step_{major}$
                else
                    if is_edge **then**
                        $s_{pre\_intersect} \leftarrow s'_{next} - step_{minor}$
                        $s_{intersect} \leftarrow s'_{next} + step_{major} - step_{minor}$
                    else
                        $s_{pre\_intersect} \leftarrow s'_{next} - step_{major}$
                        $s_{intersect} \leftarrow s'_{next}$
                break

Line of Sight - Ray Path Finder: Continued

$is\_edge \leftarrow false$
if $\neg\ is\_free$ **then**
$nbr \leftarrow s'_{next} - step_{minor}$
$is\_edge \leftarrow true$
if $\neg$ IsFree(nbr) **then**
▷ can't pass between 2 blocked tiles
if $\neg$IsFree($s'_{next} - step_{major}$) **then**
▷ if previous cell not free, then neighbour's previous cell is free
$s_{pre\_intersect} \leftarrow nbr - step_{major}$
$s_{intersect} \leftarrow nbr$
$break$
$s_{pre\_intersect} \leftarrow s'_{next} - step_{major}$
$s_{intersect} \leftarrow s'_{next}$
$break$
else
▷ non-orthogonal line of sight expansion
$error \leftarrow error + \Delta_{min}$
if $error \geq \Delta_{max}$ **then** ▷ moving on minor axis
if $\neg is\_free$ **then**
$s_{intersect} \leftarrow s'_{next}$
$break$
$error \leftarrow error - \Delta_{max}$
if $error = 0$ **then** ▷ moving diagonally
▷ vertex defined by diagonally adjacent cells (see Equation 3.13)
$s_{diagonal} \leftarrow s'_{next} + (step_{minor} + step_{major} + \mathbf{W} + 1)/2$
if IsDoubleCorner($s_{diagonal}$) **then**
$s_{pre\_intersect} \leftarrow s'_{next}$
$s_{intersect} \leftarrow s'_{next} + step_{minor}$
$break$
$s_{pre\_intersect} \leftarrow s'_{next}$
$s_{next} \leftarrow s_{next} + step_{minor}$
$s'_{next} \leftarrow s'_{next} + step_{minor}$
if $error = 0$ **then**
$is\_free \leftarrow$ IsFree(s'_{next})
if $error \neq 0$ **then** ▷ on driving axis
if $\neg is\_free$ **then**
$s_{intersect} \leftarrow s'_{next}$
$break$
else
$s_{pre\_intersect} \leftarrow s'_{next}$
$s_{next} \leftarrow s_{next} + step_{major}$ ▷ moving on driving axis
$s'_{next} \leftarrow s'_{next} + step_{major}$

Line of Sight - Ray Path Finder: Continued

if $s_{intersect} > 0$ **then**
  if $s_{pre\_intersect} = s_{start} \wedge \neg$ ISFREE(s_{start}) **then**
    ▷ line of sight blocked by vertex's own tile, i.e. goal in quadrant IV
    $s_{pre\_intersect} \leftarrow s_{pre\_intersect} - step_{major}$
  return $(s_{pre\_intersect}, s_{intersect})$
return $NULL$ ▷ s_{start} and s_{stop} have line of sight

3.3 Contour tracing

Wall following behaviours have been adopted in path-finding solutions such as bug algorithms and maze solving strategies. The proposed path-planning algorithm also makes use of a wall following behaviour. The implementation detailed in this work operates on 2-dimensional grid-maps, represented by a 2D occupancy matrix that encodes a cell as block or unblocked.

On a grid-map representation, wall following is functionally equivalent to contour-tracing, a widely used segmentation technique in image analysis. For this reason, we introduce a simple contour-tracing algorithm. We assume as known two grid-cells (one occupied and one unoccupied) that represent the starting position on an obstacle's boundary from which to perform the contour tracing. The proposed technique is introduced independently of the RPF and, as such, we assume the tracing performs a full transversal of the contour. In practice, the RPF algorithm handles the stopping criteria as it rarely performs a full trace of the obstacle (generally not required if target is outside its convex hull) but can also allow the trace to extend beyond closing the contour (i.e. the trace is allowed to continue and the stopping criteria is left to the discretion of the higher-level main function in the RPF algorithm – Algorithm 9 discussed in 4.3). For a generic standalone implementation of the proposed contour trace algorithm, a sufficient stopping criteria would be revisiting the two initial input cells.

Given a cell P and its Moore neighbourhood $M(P)$ (Figure 3.23), we consider the cell above P to have index 0 in M, as it coincides with the cardinal $North$ direction (i.e. $M_0(P) \equiv P_{North}$). All other cells in the Moore neighbourhood have indices incrementing in clockwise and counter-clockwise directions based of the side that is considered. The indices for the left and right sided Moore neighbourhoods are symmetric along the $North - South$ axis and have the following relationship:

$$index_{side} = (8 - index_{opposite\_side}) \bmod 8 \tag{3.1}$$

where $index_{opposite\_side} \in \{0, \ldots, 7\} \implies index_{side} \in \{0, \ldots, 7\}$.

$-\mathbf{W} - 1$	$-\mathbf{W}$	$-\mathbf{W} + 1$
-1	P	1
$\mathbf{W} - 1$	$\mathbf{W}$	$\mathbf{W} + 1$

Figure 3.23: Indices relative to cell P in Moore-neighbourhood on 1-D indexed grid

Algorithm 4 Ray Path Finder: Contour tracing

```
function TryNextNeighbour(s_wall, side)
    direction ← (direction + 1) mod 8                         ▷ next direction
    s_try ← s_wall + getStep(direction, side)
    if s_try = ∅ then
        return ∅                                              ▷ path has gone off map
    if IsFree(s_try) then
        s_free ← s_try                                        ▷ move to newest free node
    else
        s_wall ← s_try                                        ▷ move to newest occupied node
        direction ← getIndex(s_free − s_wall, side)

function getStep(direction, side)
    if side = RIGHT then
        direction = (8 − direction) mod 8
          ▷ W – width of the map
    STEP = {−W, −W + 1, 1, W + 1, W, W − 1, −1, −W − 1}
    return STEP[direction]

function getIndex(step, side)                                 ▷ function is the reverse of
GetStep()
    switch step do
        case −W :                                             ▷ North
            index ← 0
        case −W + 1 :                                         ▷ North-East
            index ← 1
        case 1 :                                              ▷ East
            index ← 2
        case W + 1 :                                          ▷ South-East
            index ← 3
        case W :                                              ▷ South
            index ← 4
        case W − 1 :                                          ▷ South-West
            index ← 5
        case −1 :                                             ▷ West
            index ← 6
        case −W − 1 :                                         ▷ North-West
            index ← 7
    if side = RIGHT then
        index = (8 − index) mod 8
    return index
```

The proposed contour-tracing technique is a variation on the Moore tracing algorithm [72]. Similar to the Moore neighbourhood algorithm (see Figure 3.24), the proposed solution iterates over the neighbours in the Moore neighbourhood of an occupied cell, until in encounters another occupied cell, after which it moves to the newly found cell and repeats the procedure until it closes the contour of the object, i.e. returning to its initial position.

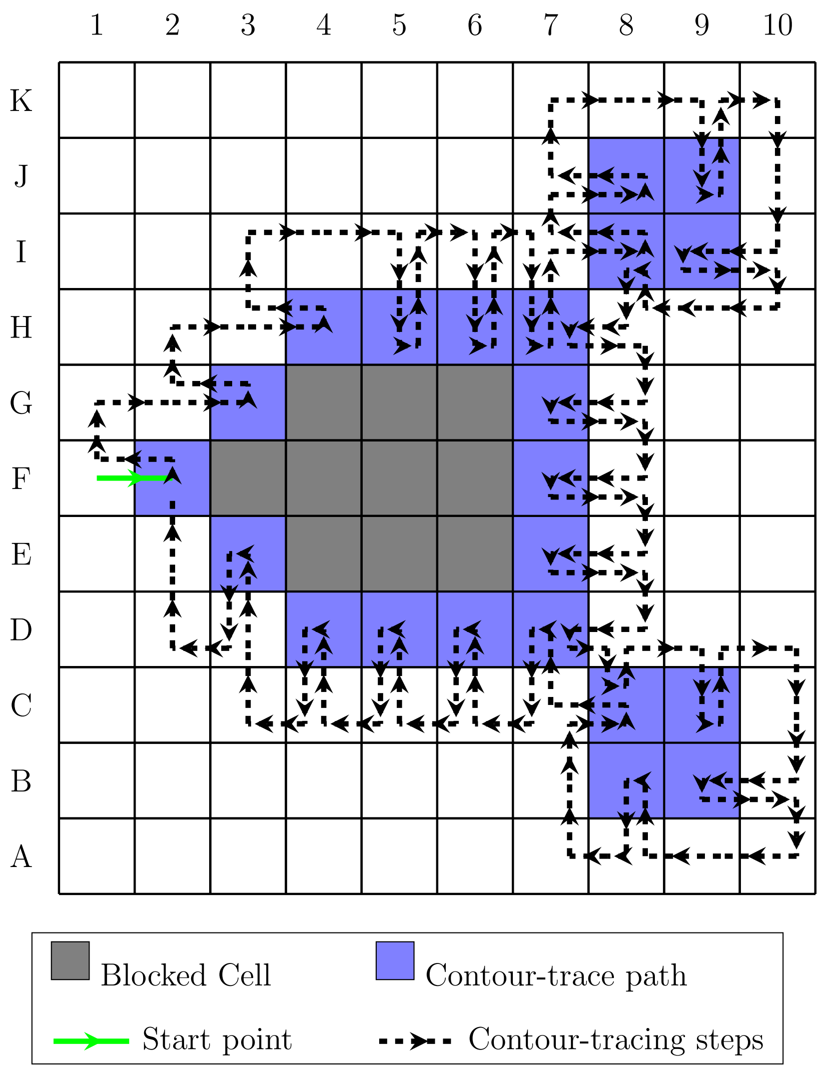


Figure 3.24: Contour tracing result of the Moore Neighbourhood Tracing algorithm

A contour tracing algorithm published by Seo et al. in 2016 [73] presents a similar functional behaviour to our proposed method (see Figure 3.25) but with a different underlining principle and a more complex algorithm. The former algorithm has a higher complexity as it needs to treat individual patterns separately. While the contour traces generated by both algorithms have the fewest number of operations on unoccupied cells, the algorithm developed by Seo et al. has a higher number of operations on blocked cells. This difference occurs for certain cases in which their algorithm revisits blocked cells.

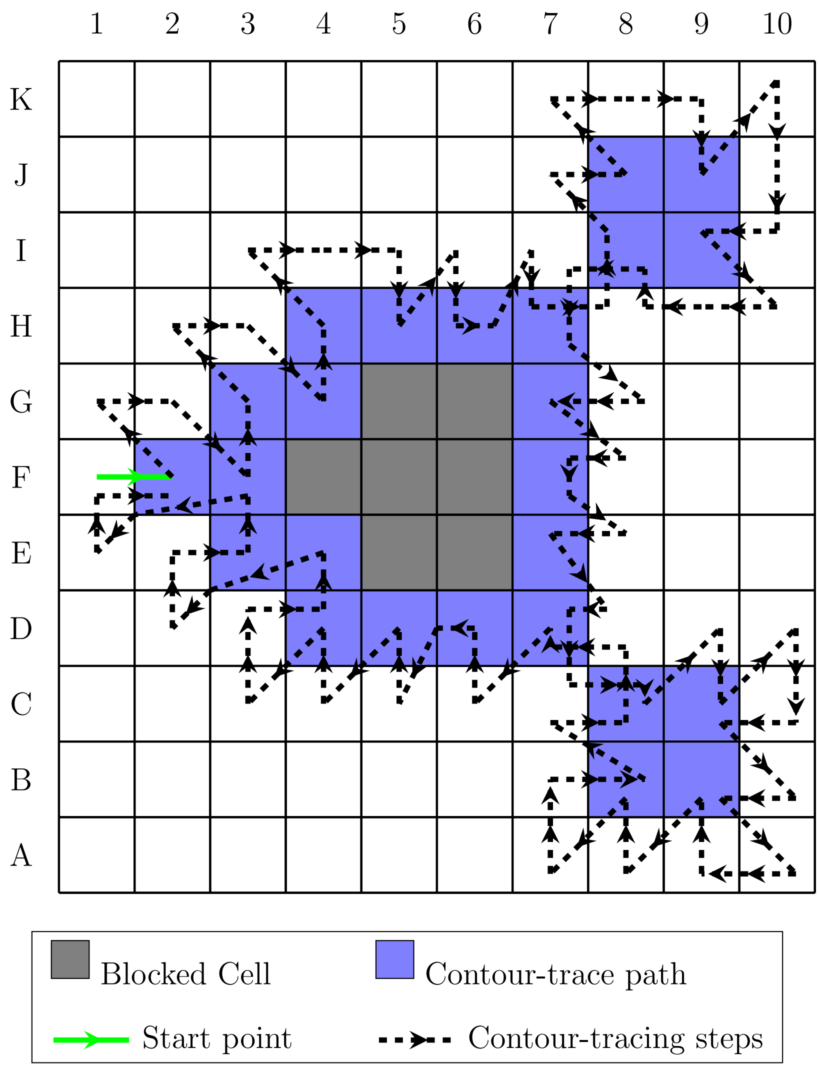


Figure 3.25: Contour tracing result using methodology presented by Seo et al.

Figure 3.26 illustrates a comparative example of the proposed algorithm's contour tracing behaviour relative to the Moore-neighbourhood algorithm (Figure 3.24) and the one proposed by Seo et al. (Figure 3.25).

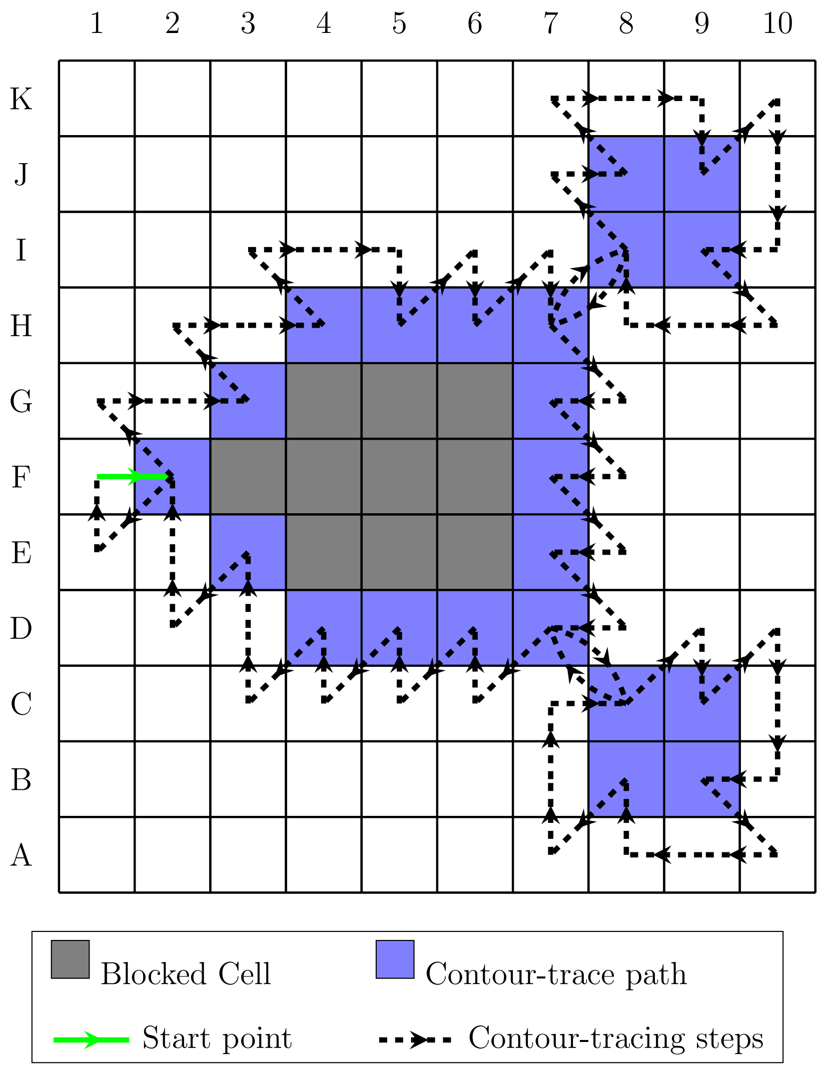


Figure 3.26: Result of left-sided contour tracing using proposed algorithm

A notable observation is that, as opposed to the contour tracing algorithm by Seo et al., the proposed method does not trace inner corner tiles, as this is beyond the needs of the path-planning algorithm. Inner corner tiles are blocked cells which share a vertex with two other blocked cells and one free cell, with the free cell being diagonally opposite to the inner corner. The tiles at G_4, F_2 and E_4 (Figure 3.26) are examples of inner corners ignored by the proposed Algorithm 4. The proposed algorithm can, however, be easily extended to account for inner corners as well. This would be accomplished by performing one additional occupancy check after the blocked cell on the contour has been explored. This step could be extended further, which would allow the algorithm to account for inner cavities as well, if the cavities are 1 cell distance from the outer contour of the object. If the additional step does not detect an inner corner, but a free cell, it can either be the entrance to a cavity (e.g. if E_4 were a free cell it would represent a cavity in the object) or, as in the case of H_8, a cell belonging to the outer edge of the object. As such, care must be taken in order to prevent infinite loops. To avoid this situation, the algorithm could be allowed to terminate normally, i.e. tracing the outer edge of the object, while remembering any diagonally free cells along with their corner pair. If, in the trace of the contour, the free cells are encountered again, such as would be the case for H_8, they are ignored. Otherwise, the remaining cells would indicate the existence of cavities inside the object which can be explored similarly, until the starting block cell is re-entered.

For the purposes of path-planning, the initial strategy is adopted as the RPF algorithm does not allow diagonal crossing. As such, we can safely ignore inner corners and cavities. The proposed contour tracing algorithm presented in Algorithm 4 operates on a 1D indexed grid-map, and moves along grid-cells (which can be though of as pixels in binary images) rather than vertices. The solution proposed in this work traces the contour of an object with the fewest number of steps necessary to touch the entirety of the contour. This is because it keeps track of both the last blocked and unblocked cell in the contour trace, thus inferring the direction by which the blocked cell has been entered and always moving to the immediate next cell in the Moore neighbourhood. The algorithm is very simple to understand and implement. It is efficient because it avoids unnecessary cell re-entries. With the exception of closing the contour of an object when the algorithm reaches the starting pair of free-blocked cells, ($\langle F_1, F_2 \rangle$ in our example), the only time the algorithm revisits a cell is in the case of diagonally adjacent blocked cells, for instance H_7, I_8, D_7 and C_8 in our example. An exception for when a free cell may be revisited would be if the object has a concave corridor more than 1 cell long but only 1 cell wide (i.e. blocked cells belonging to the same object sharing opposing edges of a free cell).

The variable *direction*, with $direction \in \{0,1,2,3,4,5,6,7\}$, represents the index of the position of s_{free} relative to s_{wall} and is calculated as the displacement between cardinal $North$ and the cardinal direction indicated by $\overrightarrow{s_{wall}s_{free}}$.

Figure 3.27 illustrates the process of our proposed contour-tracing algorithm. Each sub-figure representing the tracing of free-space in the Moore neighbourhood of an obstructed cell and the transition to the next occupied cell on the obstacle's contour. To exemplify the behaviour of the algorithm, let us consider the configuration illustrated in 3.27a, where grey cells represent the occupied cells of an object and white cells the free space around it. For our example, we initiate the contour trace starting with the top left-hand corner. The cell marked by **X** with a red cross-hatch pattern represents the initial and current occupied cell, represented by s_{wall} in Algorithm 4. The unoccupied cell with a blue cross-hatch pattern represents the initial and current free cell adjacent to **X** and presented in Algorithm 4 by s_{free}. The contour-tracing is performed following the object's edge on the left-hand side as viewed in the direction of $\overrightarrow{s_{free}s_{wall}}$. The difference between s_{free} and s_{wall} is $-\mathbf{W}-1$, which, in the left-sided Moore neighbourhood, corresponds to index 7. This initial index is stored in *direction* and the algorithm may proceed with contour tracing. With the first call to the function, *direction* advances by 1 to become 8 and takes the remainder of dividing by 8 (the number of neighbours) to give index 0. It retrieves the step by which to advance, in this case $direction_0 = -\mathbf{W}$ and add it to s_{wall} to retrieve the next neighbour. The resulting cell is stored in s_{try}

and because it corresponds to a free cell in the map it is stored in s_{free} and the function returns. On the next call, *direction* advances from 0 to 1 (represented in sub-figure 3.27b by the blue outlines). As before, the value is stored in s_{free}. On the third call, *direction* becomes 3 with a step of 1. The cell at index $s_{wall} + 1$ is an occupied cell which is then stored in s_{wall}. At this step, the algorithm calculates the new direction between s_{free} and s_{wall}; $s_{free} - s_{wall} = -\mathbf{W}$ which corresponds to index 0. Sub-figure 3.27c shows the new positions of s_{free}(blue cross-hatch with new $direction = 0$) and s_{wall} (red cross-hatch marked by **X**). This procedure repeats, with s_{free} advancing through all the free cells around **X** in sequence, until encountering an occupied cell, in which case s_{wall} advances. Because *direction* keeps track of the direction between each new s_{wall} and the s_{free} node from which it was entered, the algorithm minimises the number of times a free cell is visited. The number of visits to occupied cells is also minimised, as the occupied cell is never re-entered from the same side. In sub-figures 3.27e and 3.27g the occupied cell is revisited, as it is not an outer corner yet two of its edges are part of the object boundary. It is trivially evident that, for any configuration where two occupied cells share only one vertex, the algorithm would revisit at least one of the occupied cells.

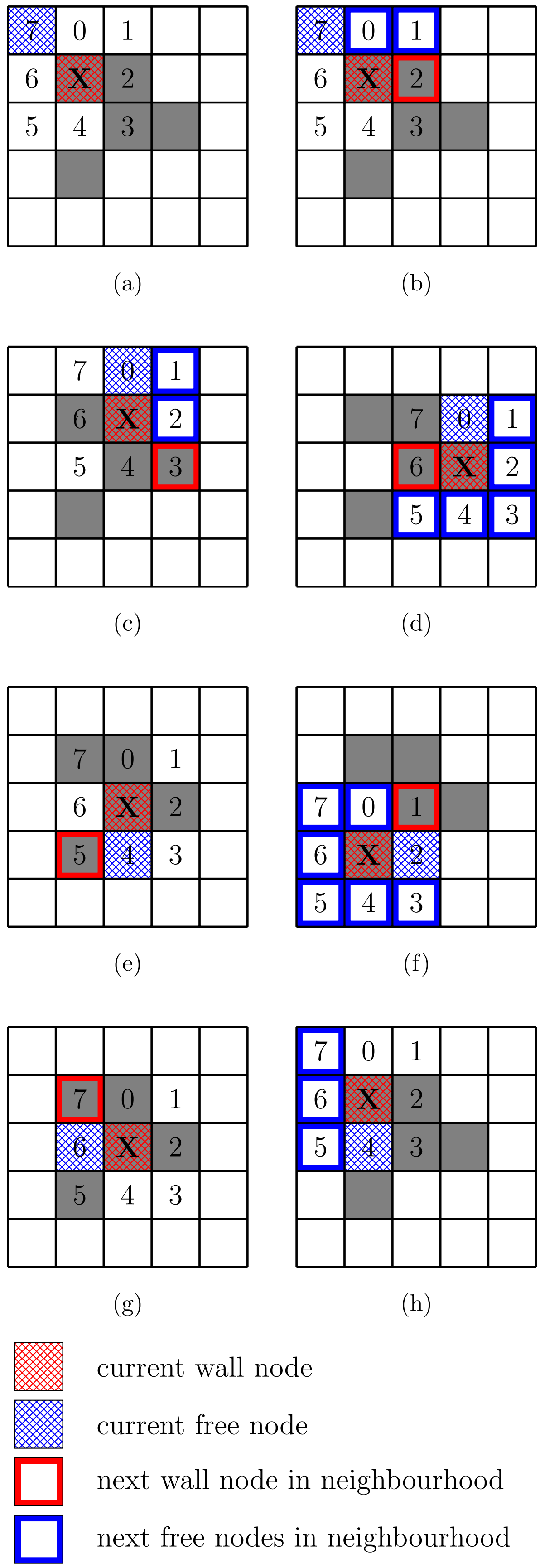


Figure 3.27: Scanning steps for left-sided contour-tracing

Of particular use for RPF's needs is the ability of the contour tracing algorithm to trace an object's edge in both left-sided and right-side directions, which can be achieved very easily. Tracing in the opposite direction requires two minor modifications (Lines 14 & 37) that change a left-sided Moore neighbourhood to a right-sided one. This involves performing a check on the variable *side* that indicates the desired side for tracing and if true, inverting *direction* and *index* by using equation (3.1). Figure 3.28 illustrates the right-sided contour-tracing steps performed by the algorithm. For the right-sided approach, the algorithm rotates around the occupied cell **X** in a counter-clockwise direction, while exploring all of the same cells in reverse order.

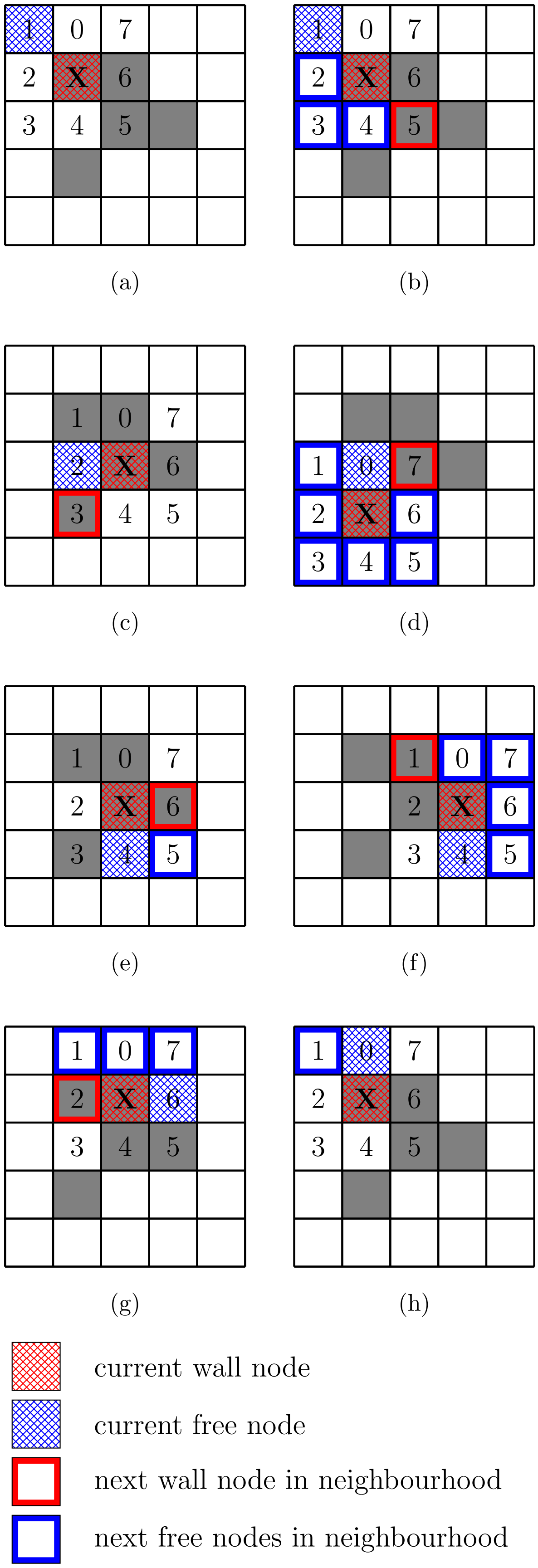


Figure 3.28: Scanning steps for right-sided contour-tracing

3.4 Path corners

Sidedness plays more than one key role with respect to a path's behaviour. Awareness on the part of the algorithm of the notion of path sidedness is required in order to identify path corners and also to preserve path tautness during the path pruning phase. A path's sidedness simply represents the side being considered relative to the path's direction of travel (i.e. first person view), colloquially left and right. The sidedness of a path is determined by the sidedness of its parent path (the path from which it branches off) at the point of intersection with an obstacle. As such, any child path always has opposite sidedness to that of its parent. For the initial input, the sidedness of the "root" path can be chosen at random, and is preserved for the entire life-cycle of the path. When an obstacle is encountered and the path bifurcates along the edge of the obstacle, the child path splits from the parent, following the obstacle on the opposite side.

In evaluating sidedness, the cross product is used to determine the sign of the acute angle defined by three points $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ and $P_3 = (x_3, y_3)$. This angle corresponds to the direction of the cross product of the two coplanar vectors $\overrightarrow{P_1P_2} = \vec{\mathbf{u}} = \langle \Delta x_{\mathbf{u}}, \Delta y_{\mathbf{u}} \rangle$ and $\overrightarrow{P_1P_3} = \vec{\mathbf{v}} = \langle \Delta x_{\mathbf{v}}, \Delta y_{\mathbf{v}} \rangle$. Operating in two-dimensional space, we describe the cross product through P_1, P_2 and P_3:

$$\vec{\mathbf{u}} \times \vec{\mathbf{v}} = \begin{vmatrix} \Delta x_{\mathbf{u}} & \Delta y_{\mathbf{u}} \\ \Delta x_{\mathbf{v}} & \Delta y_{\mathbf{v}} \end{vmatrix} = \Delta x_{\mathbf{u}} \Delta y_{\mathbf{v}} - \Delta x_{\mathbf{v}} \Delta y_{\mathbf{u}}$$

$$\vec{\mathbf{u}} \times \vec{\mathbf{v}} = (x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1) \tag{3.2}$$

If the points are collinear, $\vec{\mathbf{u}} \times \vec{\mathbf{v}} = 0$ and P_3 lies on line $\overrightarrow{P_1P_2}$. Otherwise, the sign of the cross-product depends of the handedness of the coordinate system, i.e. the sign of $\vec{\mathbf{u}} \times \vec{\mathbf{v}}$ tells whether P_3 lies to the left or to the right of $\overrightarrow{P_1P_2}$. For a "right-handed" coordinate system, a positive cross-product implies that P_3 lies on the left side of $\overrightarrow{P_1P_2}$ (Figure 3.29).

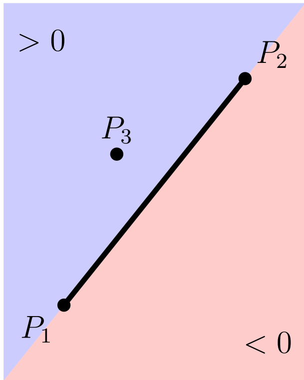


Figure 3.29: Cross product of right-handed coordinate system relative to $\overrightarrow{P_1P_2}$: the cross-product for all points in the blue region has positive values while the cross-product of all the points in the red region has negative values

The cross-product is used by RPF for a number of purposes, in determining which nodes should be considered as heading changes for the path when it alters its direction of travel, in maintaining path tautness by pruning nodes from the path, and in deciding when a path can leave the edge of an obstacle and resume moving towards the goal.

As described in 3.3, unlike operating with path nodes, the contour tracing algorithm operates on grid-cells (tiles) rather than vertices. For this reason, we wish to be able to transition from grid-cells to vertices, during the contour tracing phase.

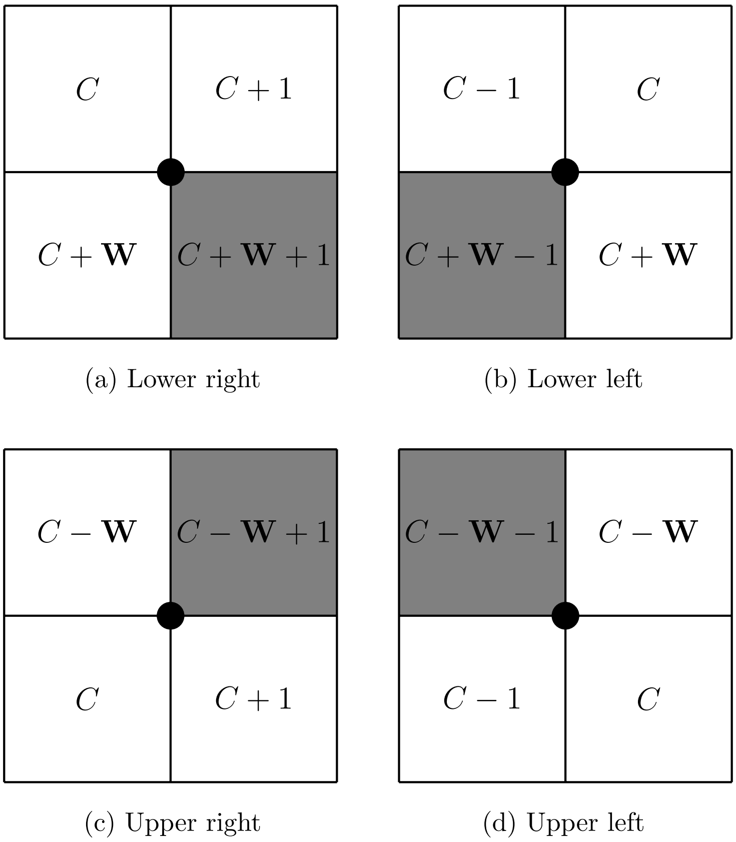


Figure 3.30: Corner vertex cases: C - unoccupied corner node; grey cell - occupied node diagonal to C; **W** - width of the map

Analysing Figure 3.30, we explore the four possible instances for a corner vertex:

$$
\begin{aligned}
\text{Case 3.30a:} \quad & |C + \mathbf{W} + 1 - C| = \mathbf{W} + 1 \\
\text{Case 3.30b:} \quad & |C + \mathbf{W} - 1 - C| = \mathbf{W} - 1 \\
\text{Case 3.30c:} \quad & |C - \mathbf{W} + 1 - C| = \mathbf{W} - 1 \\
\text{Case 3.30d:} \quad & |C - \mathbf{W} - 1 - C| = \mathbf{W} + 1 \\
\text{Let:} \quad & |s_{occ} - s_{free}| \quad = |\Delta s| \\
& \Rightarrow \quad |\Delta s| \quad = \mathbf{W} \pm 1 \\
& \Rightarrow \quad ||\Delta s| - w| \quad = 1
\end{aligned} \tag{3.3}
$$

$$\begin{cases} x_4 = x_1 + \mathbf{W} + 1 & (3.4) \\ x_4 = x_2 + \mathbf{W} & (3.5) \\ x_4 = x_3 + 1 & (3.6) \end{cases}$$

$$\begin{aligned} &\overset{3.4}{\Longrightarrow} 2x_4 = 2x_1 + 2\mathbf{W} + 2 \\ &2x_4 = x_1 + \mathbf{W} + 1 + x_1 + \mathbf{W} + 1 \\ &\overset{3.4}{\Longrightarrow} 2x_4 = x_4 + x_1 + \mathbf{W} + 1 \\ &x_4 = \frac{x_1 + x_4 + \mathbf{W} + 1}{2} \end{aligned} \tag{3.7}$$

$$\begin{aligned} &\xrightarrow{3.5,\ 3.6} 2x_4 = x_2 + \mathbf{W} + x_3 + 1 \\ &x_4 = \frac{x_2 + x_3 + \mathbf{W} + 1}{2} \end{aligned} \tag{3.8}$$

$$\text{Case 3.30a:} \quad \overset{3.7}{\Longrightarrow} \quad x_4 = \frac{2x_1 + x_4 - x_1 + \mathbf{W} + 1}{2}$$

$$x_4 = x_1 + \frac{x_4 - x_1 + \mathbf{W} + 1}{2} \tag{3.9}$$

$$\text{Case 3.30b:} \quad \overset{3.7}{\Longrightarrow} \quad x_4 = \frac{2x_2 + x_3 - x_2 + \mathbf{W} + 1}{2}$$

$$x_4 = x_2 + \frac{x_3 - x_2 + \mathbf{W} + 1}{2} \tag{3.10}$$

$$\text{Case 3.30c:} \quad \overset{3.8}{\Longrightarrow} \quad x_4 = \frac{2x_3 + x_2 - x_3 + \mathbf{W} + 1}{2}$$

$$x_4 = x_3 + \frac{x_2 - x_3 + \mathbf{W} + 1}{2} \tag{3.11}$$

$$\text{Case 3.30d:} \quad \overset{3.8}{\Longrightarrow} \quad x_4 = \frac{2x_4 + x_1 - x_4 + \mathbf{W} + 1}{2}$$

$$x_4 = x_4 + \frac{x_1 - x_4 + \mathbf{W} + 1}{2} \tag{3.12}$$

$$\xrightarrow{3.3,3.9,3.10,3.11,3.12} \quad x_4 = s_{free} + \frac{\Delta s + \mathbf{W} + 1}{2} \tag{3.13}$$

Algorithm 5 Ray Path Finder: Identify Outer-Corner Vertex

function GETVERTEXIFCORNER(s_{free}, s_{occ})
  $\Delta s \leftarrow s_{occ} - s_{free}$ ▷ s_{free}- unoccupied; s_{occ} - occupied
  if $||\Delta s| - \mathbf{W}| = 1$ **then**
    ▷ vertex of bottom right cell (Equation 3.13)
    return $s_{free} + \frac{\Delta s + \mathbf{W} + 1}{2}$
  return -1 ▷ the two grid-cells don't describe a valid corner vertex

Considering Figure 3.31 as reference, we can explore how sidedness helps in identifying heading changes during path exploration. Given s_{start} at (F_2) and s_{goal} at (F_{11}), and the cul-de-sac obstacle with the depression facing the start node, let us consider the two possible paths around the obstacle. We will explore the left-bound path in the first instance.

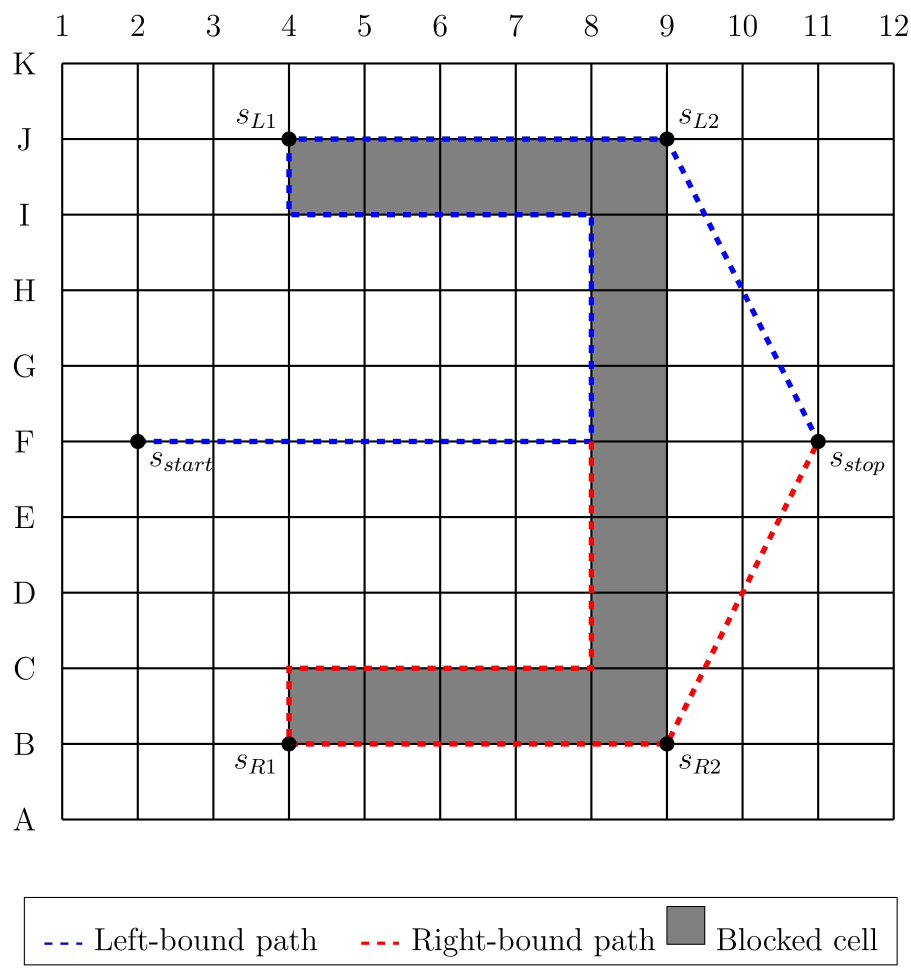


Figure 3.31: Cul-de-sac stage 1: both paths reach goal with equal cost estimation

The path begins from (F_2) and encounters the obstacle at vertex (F_8). The path is split into two, a left-bound path and a right-bound path that trace the obstacle in opposite directions. Using the processes described in Algorithms 4 & 5, the left-bound path proceeds to trace the obstacle boundary towards vertex (G_8), and continuing through (I_8), (I_4) and (J_4). Reaching vertex (J_5), the path can determine that the vertex at (J_4) (marked by s_{L1}) is a potential heading change, as (J_5) lies on the right-hand side of $\overrightarrow{s_{start}s_{L1}}$. As such, a taut path from s_{start} needs to pass through (J_4) in order to reach (J_5). Vertex (J_4) is added to the left-bound path as s_{L1}, and the algorithm continues to trace the obstacle's boundary. Reaching vertex (I_9), the path identifies (J_9) as a potential heading change of the path and adds it to the path as s_{L2}, because (I_9) lies on the right-hand side of $\overrightarrow{s_{L1}s_{L2}}$. As (I_9) lies of the right-hand side of $\overrightarrow{s_{L2}s_{stop}}$, the path is has a potential line-of-sight to the goal node.

For the right-bound path, the process is symmetrical. Vertex (B_4) (marked as s_{R1}) is found to be a potential heading change, as (B_5) lies on the left-hand side of $\overrightarrow{s_{start}s_{R1}}$. A taut path from s_{start} needs to pass through (B_4) in order to reach (B_5). Continuing on the right-side of the obstacle, the path reaches vertex (C_9) and identifies (B_9) as a potential heading change of the path. The vertex is added to the path as s_{R2}, because (C_9) lies on the left-hand side of $\overrightarrow{s_{R1}s_{R2}}$. With both paths having identified their heading changes using cross-product and the path's sidedness, the remaining step is to check for visibility between the pairs of nodes of each path. As all nodes are visible from their parents, the resulting paths are identified as: $s_{start}, s_{L1}, s_{L2}, s_{stop}$ – left-bound path; $s_{start}, s_{R1}, s_{R2}, s_{stop}$ - right-bound path. The two paths are illustrated in Figure 3.32.

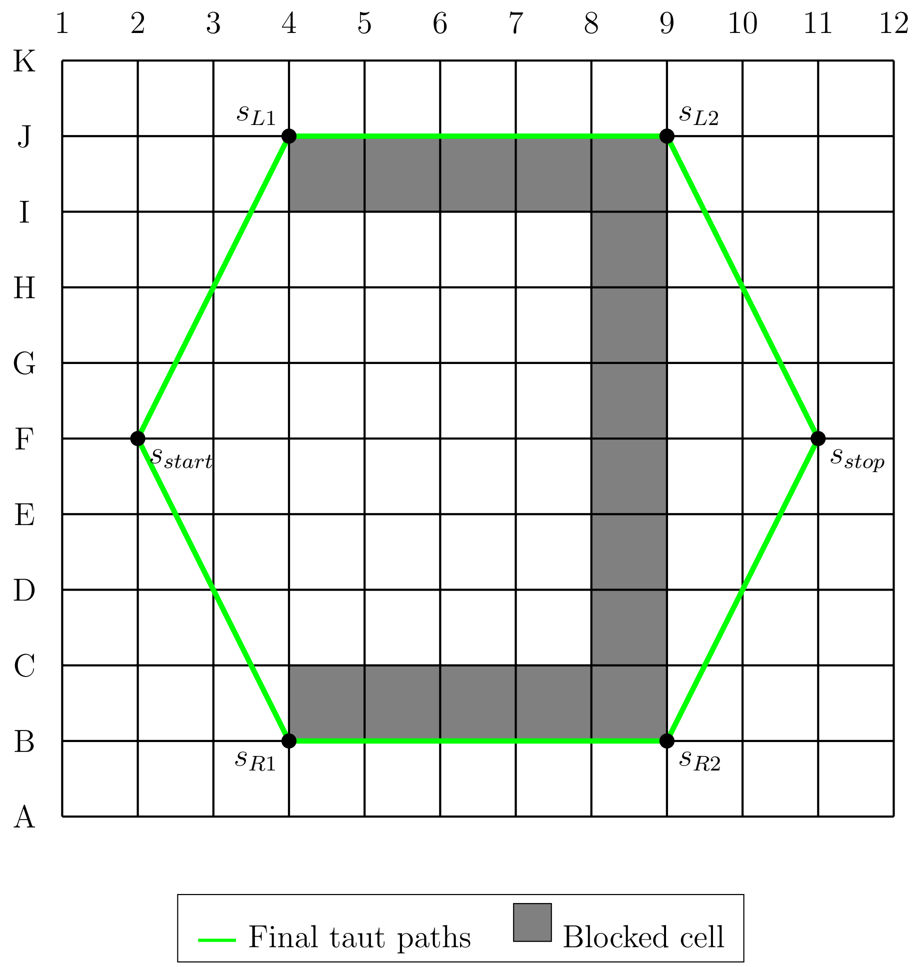


Figure 3.32: Cul-de-sac stage 2: final left-bound and right-bound taut paths of equal length

3.5 Path direction

Paths in RPF require a means of monitoring the direction of travel when wall-following. This is necessary because unlike behaviours found in most bug algorithms, the paths in the RPF algorithm do not circumvent the obstacle in search for a point of shortest distance from which to leave the wall, nor do they make use of the M-line to decide a leave point. Rather, a path leaves an obstacle when it assumes that it is travelling on the convex hull of the boundary and has a potential direct line-of-sight to the target.

The Pledge algorithm [74], named after John Pledge of Exeter, has a similar approach to the above described. This algorithm is designed to evade obstacles, with an arbitrarily chosen path direction at the starting node. Such an algorithm accounts for traps shaped in the upper-case letter “G”. The original Pledge algorithm assumes a fixed arbitrary direction of travel that it uses as the reference to perform wall following while summing the angles of each corner. The condition for the Pledge algorithm to abandon wall-following, is for the sum of the turns to be 0.

In contrast to the Pledge algorithm, the proposed method does not assume a fixed direction of travel. Instead, it keeps track of two directions, that of the path, and that of the goal. For simplicity of explanation, one can visualise two clock hands, with one hand (the minute hand for example) pointing in the direction in which the path is currently travelling, while the other hand (hour hand) always points in the direction of the goal (as would a compass needle for which the target represents North). When moving on the edge of an obstacle, one must keep track of the number of times the minute hand crosses over the hour hand along with the relative difference between the two hands. A path may leave the wall-following behaviour and resume travelling in a straight line towards a target, if the number of turns performed away from the a goal is less then the number of turns performed towards the goal.

Figure 3.33 serves as an example to illustrate the behaviour of the direction monitoring functionality. Considering node s_{start} at (Q_{11}) travelling towards s_{stop} at (Q_{18}), the algorithm explores the right-sided path. It intersects the obstacle at (Q_{15}) and updates the goal direction to indicate s_{stop} (i.e. goal pointed to by 0 in the indexed square). As the next free node on the right-side of the boundary is (P_{14}), the path direction (indicated by downward arrow) gives the number of turns away from the goal. Moving downwards, to (H_{14}), the path direction remains the same but the orientation of s_{goal} changes from *East*–bound to *North – East*–bound, thus the turn count becomes 3. At (E_9) the path is heading *West*–bound, with a turn count of 5. At (O_8), the path crosses over the goal direction (i.e. it goes through 0 to complete a full rotation), heading *East*–bound once more. However, the crossover is taken into account, thus, rather that having a turn count of 1, the full rotation is added for a total turn count of 9.

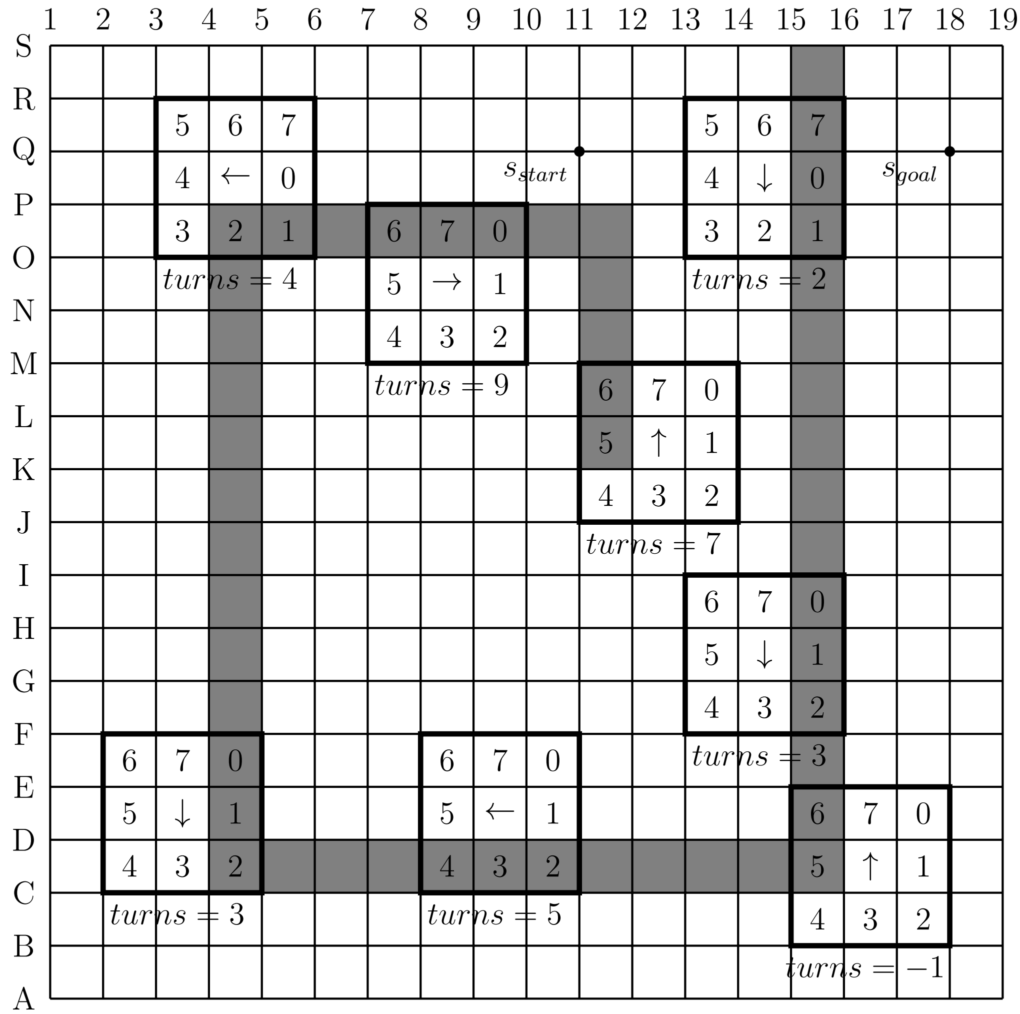


Figure 3.33: Direction Matrix rotations (0 always points towards s_{goal}; arrow shows relative number of turns away from s_{goal} while following wall)

The path continues *South*–bound from (O_{11}) until (K_{11}) with a turn count of 11, after which the path turns towards the goal at (K_{11}) and (K_{12}). As such, the turn count at (L_{12}) becomes 7 as the path crosses over the goal direction in the opposite direction to the path side (i.e. from right to left for the right–bound path in our example). However, because the turn count is greater than 0, the path is not allowed to leave the obstacle boundary and head towards the goal. Keeping track of the path's turn count avoids creating a cyclical path that would intersect the same obstacle and creating a loop. As the path is not allowed to stop following the wall, it arrives at (Q_4) with a turn count of 4 as the node is now horizontal to s_{goal}. Moving *South*–bound towards (E_3) the path is now 3 turns away from the goal. Finally, reaching (D_{16}), the path once more crosses over the goal direction, reaching a turn count of -1. At this stage, the contour tracing subroutine returns, the state changes from $FOLLOW\_WALL$ to $RAYCAST$ and the path is allowed to leave the obstacle boundary and move in a straight line towards the goal.

3.6 Path pruning

Directing the search towards finding the shortest path solution involves presenting the estimated length for the path. The heuristic estimation for the length of each path must preserve the admissibility property (Equation 1.5). As such, RPF can never overestimate the cost of reaching the goal via the path in question, which implies preserving the path's tautness. For this purpose, a path requires the ability to prune any and all nodes that do not respect the triangle inequality (Figure 3.34).

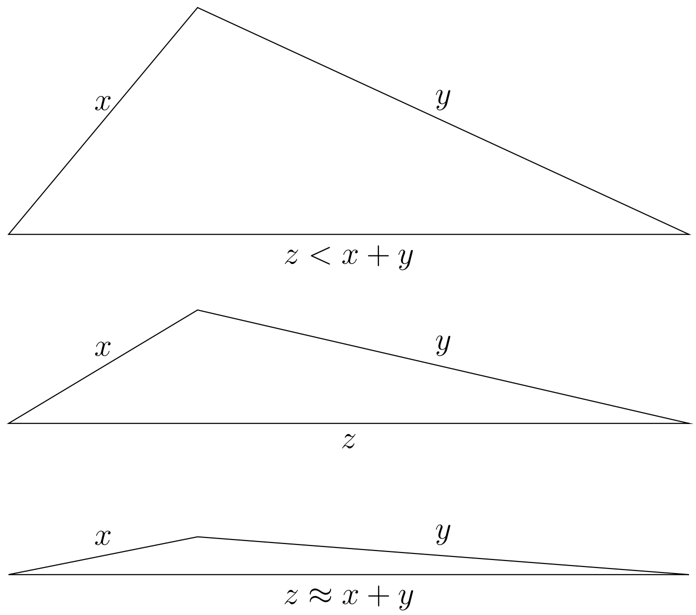


Figure 3.34: Triangle inequality

When a new node is added to the path, if the two nodes prior to it do not respect the triangle inequality, the path becomes suboptimal because the heuristic estimation overestimates the cost of the path. A shortest path must always be taut, and as such can only have heading changes around obstacle corners. The need for pruning with each new corner addition arises from the free-space assumption. The algorithm does not perform line-of-sight checks between vertex pairs in a path, until a path has reached the final target node. Because of this, the algorithm is not aware when expanding towards the goal if there exists a line-of-sight between node paths that break the triangle inequality.

Drawing a parallel to Theta*, the pruning procedure in RPF can be interpreted as updating the parent of a node. For Theta*, given a node s and it's immediate neighbour s', when s' has line-of-sight to parent of node s, the node s's parent becomes a direct parent for node s' (see Line 4 in Algorithm 3). This implies that a shortest path can bypass s when travelling from $parent(s)$ to s', or in other words, s can be pruned from the path containing $parent(s)$ and s'. This is also the case for RPF, with two key differences, namely that s and s' do not have to be immediate neighbours, and that RPF does not perform a line-of-sight check, but rather it optimistically assumes that a line of sight exists, postponing the line-of-sight check until the end goal has been reached.

3.6.1 Backward pruning

Back-pruning maintains a path's tautness, by pruning nodes that do not respect the triangle inequality. Pruning nodes allows a path to provide consistent heuristic length estimates. The forward pruning strategy is called at each step when moving along an object's boundary before and after a path has reached the final destination node s_{end}. While not recursively implemented, the back-pruning strategy is reminiscent of the Recursive Strict Theta* algorithm (described in 2.4.2), with the distinction that, for RPF, pruning only ever needs to be applied to corner nodes.

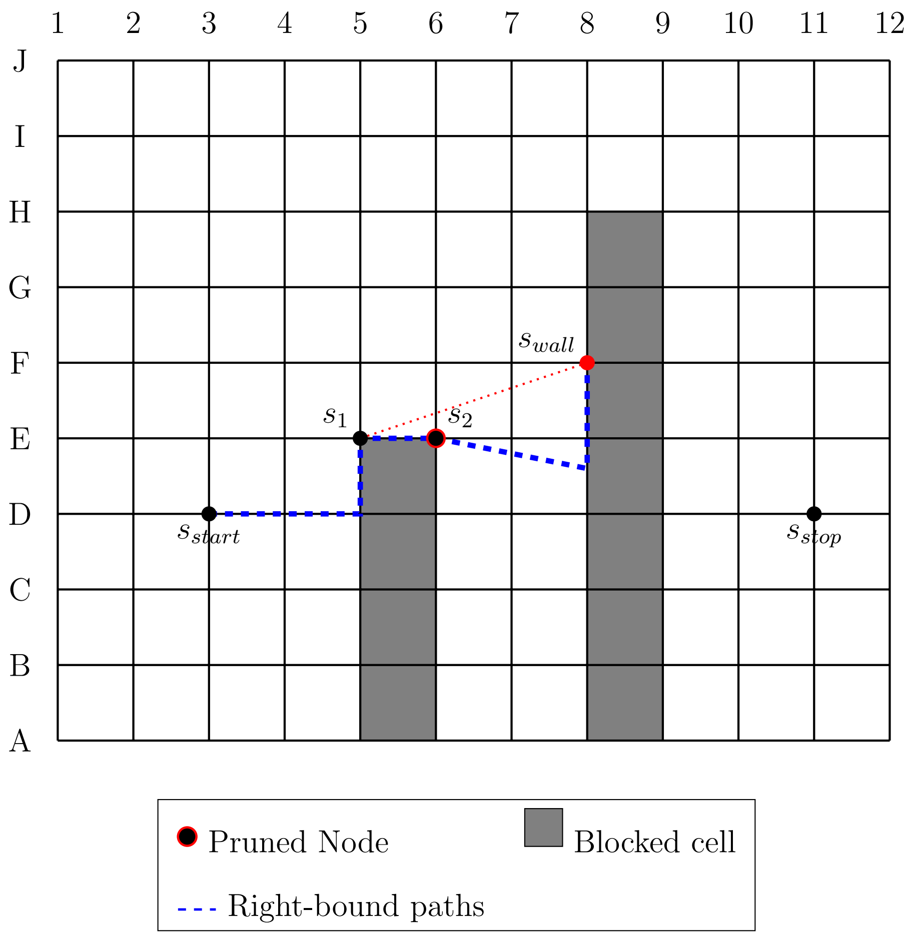


Figure 3.35: Back-pruning scenario: stage 1

To exemplify the backward pruning strategy, let us consider the map configuration illustrated in 3.35. For simplicity, we will only consider the path that always follows an obstacle's edge on the left-hand side. The algorithm initiates the search from s_{start} at (D_3) by ray casting towards s_{stop} at (D_{11}) until in encounters an obstacle at (D_5). As a leftward exploring path, if follows the edge of the obstacle and identifies (E_5) as a corner node. From 3.2, we determine (E_5) to be on the right-hand side of $\overrightarrow{E_6D_3}$, the vector described by the next node on the edge and the last node in the path, in this case s_{start}. We label (E_5) as s_1 and add it to the path. (E_5) now becomes the last node in the path, as we continue with edge following. In the same manner, we discover (E_6) as the next corner, label it as s_2 and add it to the path. As the path's direction of travel has gone bellow 0, as

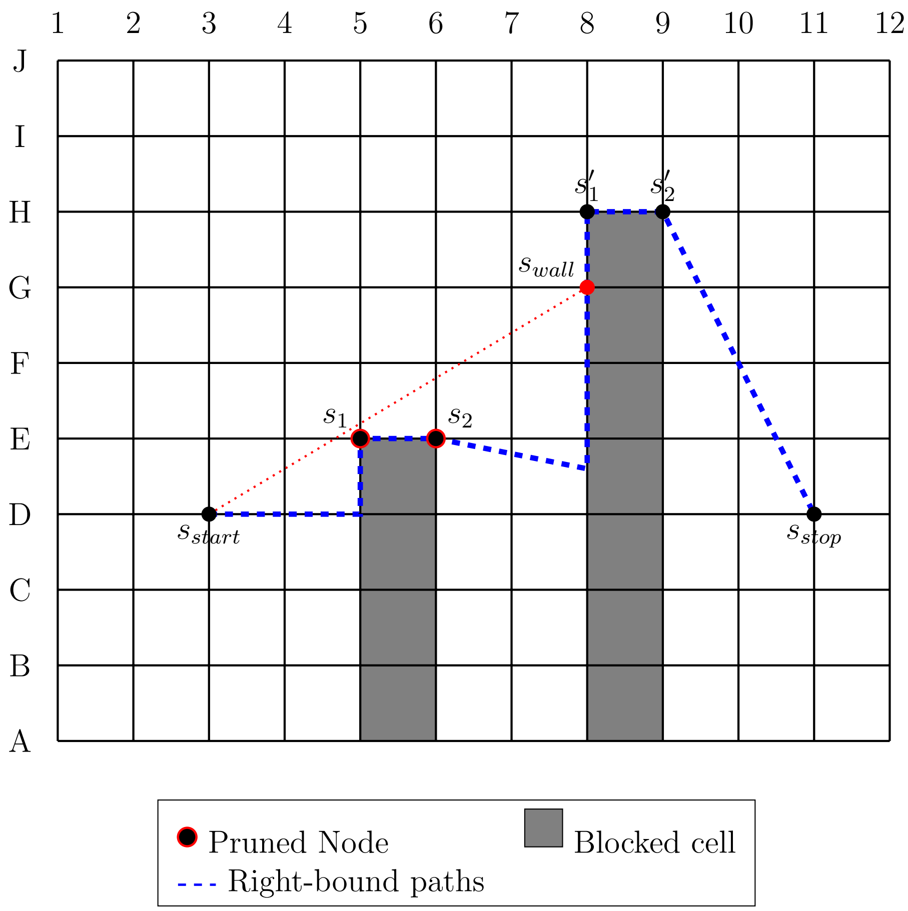


Figure 3.36: Back-pruning scenario: stage 2

described in 3.5, we are now free to leave the edge of the obstacle and once again ray-cast towards s_{stop}. A new obstacle is encountered between D_8 and E_8 and the path proceeds with edge following towards the left. The path encounters the node s_{wall} at (F_8). The last node in the path, s_2, now lies on the right-hand side of $\overrightarrow{s_1 s_{wall}}$, and becomes a candidate for pruning. Node s_2 is removed from the path and s_1 becomes the last (and only) node in the path once again.

When s_{wall} moves to (G_8), as observed in Figure 3.36, s_1 becomes the next candidate for pruning, as it lies on the right-hand side of $\overrightarrow{s_{start} s_{wall}}$. Node s_1 is pruned as the path proceeds with edge following. In the same manner as before, two new nodes are identified, s'_1 and s'_2 and are placed in the path.

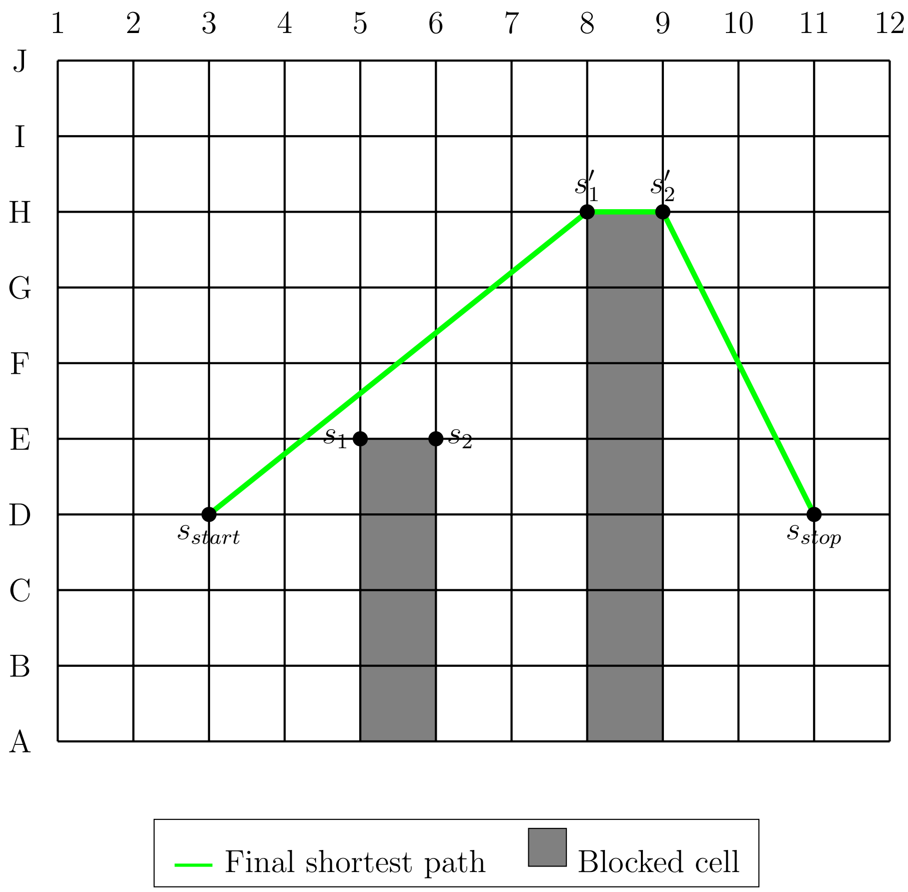


Figure 3.37: Back-pruning scenario: stage 3

Node s'_2 is identified as the new jump-off point and the path leaves the edge of the obstacle. The ray-cast encounters no obstacle between s'_2 and s_{stop}. The path has now reached the end and its state is updated. In the next phase (Figure 3.37), the path performs line of sight checks between subsequent nodes for this simple scenario determines that the path is cleared. Its state is updated, and the it is moved from the open list to the cleared list.

The pseudo-code for the back-pruning strategy can be seen in Algorithm 6. The 1-D indices of the nodes are stored in the path along with the sidedness of the path at the point of their expansion. This can be done efficiently by adopting the convention that all left-bound nodes have negative value and all right-bound nodes have positive value. The choice of sign is not important however, as long as it is consistent throughout, ensuring that left-bound nodes have an opposite sign to right-bound nodes. The algorithm looks at all previous nodes in the path (Line 5). If the node s_i lays on the opposite side of $\overrightarrow{s_{i-1}s_{wall}}$ relative to the sidedness of the path, that node is pruned from the path as the triangle inequality dictates there is a shorter path to s_{wall} that does not pass through s_i. The condition at Line 8 allows for an early termination of the algorithm at the node before the path has switched sides. For example, if going from a left-bound path to a right-bound path, the node s_i will have an opposite sign (i.e. negative for left-bound paths) to the current sign of the path (i.e. positive for right-bound paths). If the condition at Line 8 is true, the algorithm is allowed to terminate early, as no other nodes could be pruned.

To check that the path is taut (Line 10), the sidedness of s_i is tested using Equation 3.2. If the condition is respected, then the triangle inequality is preserved and the algorithm terminates as there is no need to look any further. If the condition fails, however, the node is removed from the path, and the algorithm moves to the previous node in the path.

Algorithm 6 Ray Path Finder: Path Back-Pruning

1: **function** BACKPRUNE(s_{wall})
2:     **if** $path.size < 2$ **then** $\triangleright$ path needs at least 2 nodes in the path
3:         **return**
4:     $n \leftarrow path.$INDEXOF$(s_{interStart})$
5:     **for** $i \leftarrow n$ **down to** 2 **do**
6:         $s_i \leftarrow path.$NODEAT(i) $\triangleright$ $s_i \equiv s_{interStart}$
7:         $s_{i-1} \leftarrow path.$NODEAT$(i-1)$ $\triangleright$ node before $s_{interStart}$
8:         **if** $sgn(s_i) \neq sgn(path.side)$ **then** $\triangleright$ is s_i from opposite side?
9:             **return** $\triangleright$ path at inflection point; cannot prune beyond it
10:         **if** $sgn(\overrightarrow{s_i s_{wall}} \times \overrightarrow{s_{i-1}s_{wall}}) = sgn(path.side)$ **then** $\triangleright$ checking sidedness (Equation 3.2)
11:             **return** $\triangleright$ path is taut; no need to prune
12:         $path.$REMOVE(s_i) $\triangleright$ s_i on opposite side of $\overrightarrow{s_{i-1}s_{wall}}$; prune it
13:         $s_{interStart} \leftarrow s_{i-1}$ $\triangleright$ s_{i-1} becomes new interior starting node
14:     UPDATELENGTH$(path)$

3.6.2 Forward pruning

Similar to the back-pruning strategy, forward-pruning is employed to maintain a path's tautness, by pruning nodes that do not respect the triangle inequality. The forward pruning strategy is only used after a path has reached the final destination node s_{end}. To exemplify the forward pruning strategy, let us consider the map configuration illustrated in 3.38. For the purpose of describing the procedure, we will only consider the path that always follows an obstacle's edge on the left-hand side, similar to the example in 3.6.1.

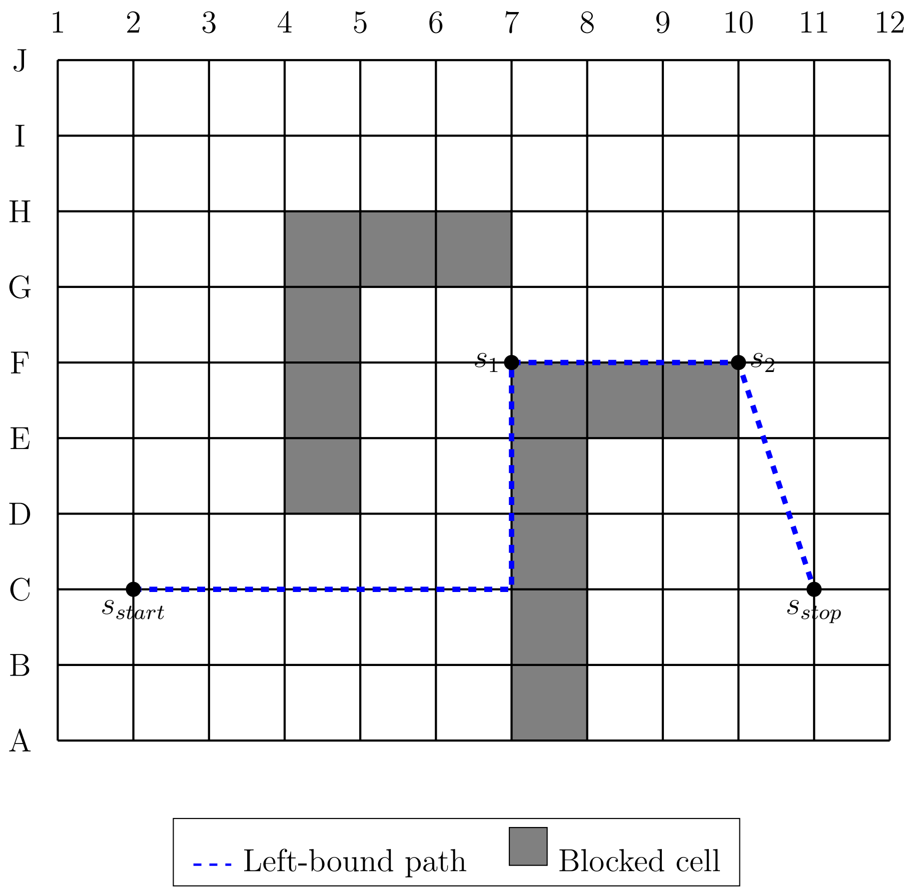


Figure 3.38: Forward-pruning scenario: Initial stage

The algorithm initiates the search from S_{start} at (C_2) towards S_{stop} at (C_{11}) by performing a a line of sight check. The ray-cast intersects an obstacle at (C_7) and updates its state to $FOLLOW\_WALL$. It proceeds by following the edge of the obstacle and identifies (F_7) as a corner node, as (F_7) is on the right-hand side of $\overrightarrow{F_8C_1}$, the vector described by the next node on the edge and S_{start}, the last node in the path. We label (F_7) as s_1 and add it to the path. (F_7) now becomes the last node in the path. Continuing with edge following, we discover the next corner at (F_{10}), (F_{10}) is on the right-hand side of $\overrightarrow{E_{10}F_7}$, the vector described by the next node on the edge and S_1, the last node in the path. We label (F_{10}) as s_2 and add it to the path. As the path's direction of travel has gone bellow 0, as described in 3.5, we are now free to leave the edge of the obstacle and once again ray-cast towards s_{stop}. As s_2 has a line of sight to s_{stop}, the path's state is updated to reflect that it has reached the goal.

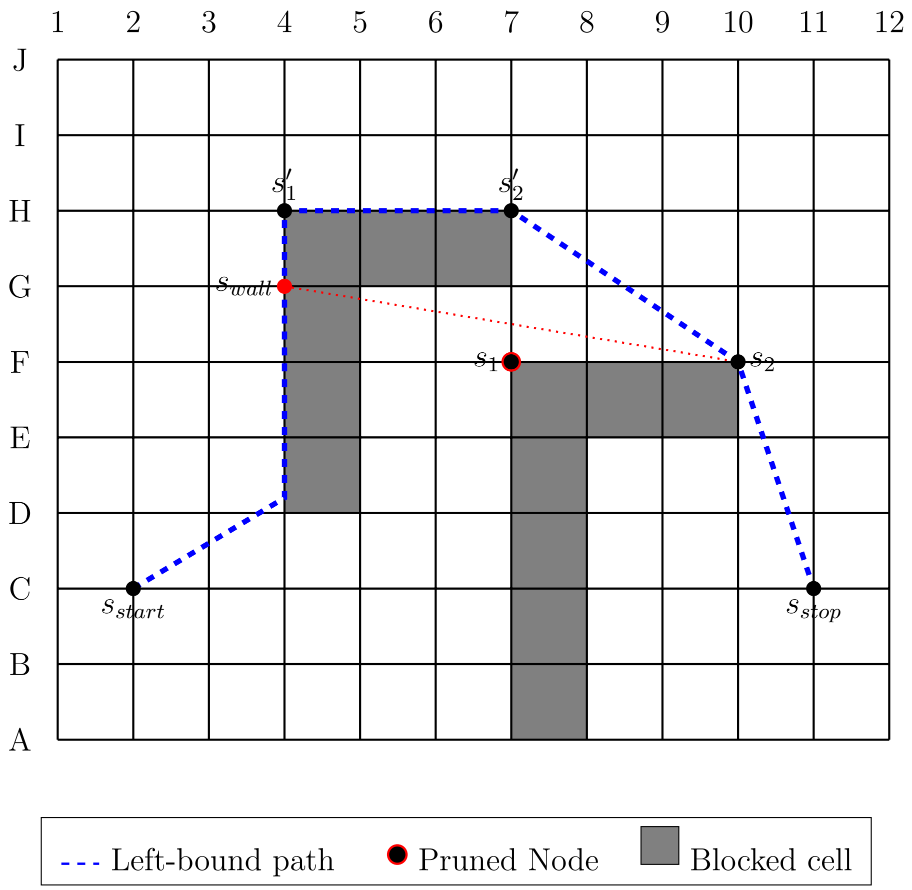


Figure 3.39: Node s_1 on right side of left-bound $\overrightarrow{s_{wall}s_2}$ is pruned from path

In the next phase (Figure 3.39), the path performs line of sight checks between subsequent nodes beginning with the pair (s_{start}, s_1). A new obstacle is encountered between D_4 and E_4 and the path changes state to proceed with edge following towards the left. We encounter the node s_{wall} at (G_4) which is of interest. The next node after s_{start}, s_1, lies on the right-hand side of $\overrightarrow{s_{wall}s_2}$, the vector described the current node on the edge and s_2, the node after s_1 which is currently considered for pruning. Node s_1 is removed from the path and s_2 becomes the new goal as the next node after s_{start}. As s_{wall} progresses on the edge of the obstacle, two new nodes are identified. The first one at (H_4), labelled s'_1, becomes the new start (described as $s_{interStart}$ in Algorithm 9) and is inserted before the new goal s_2. Similarly, the next node on the edge, s'_2 at (H_7) becomes the new start and is inserted before the new goal s_2. After s_2, the path's direction of travel relative to the new goal, goes bellow 0 and the path's state is updated to $RAYCAST$. The line of sight check between s'_2 and s_2 is successful and the path's state is once again updated to $GOAL\_FOUND$. The path once more performs line of sight checks between subsequent nodes and determines that the path is clear (Figure 3.40), it removes the path from the open list and inserts it into the cleared list.

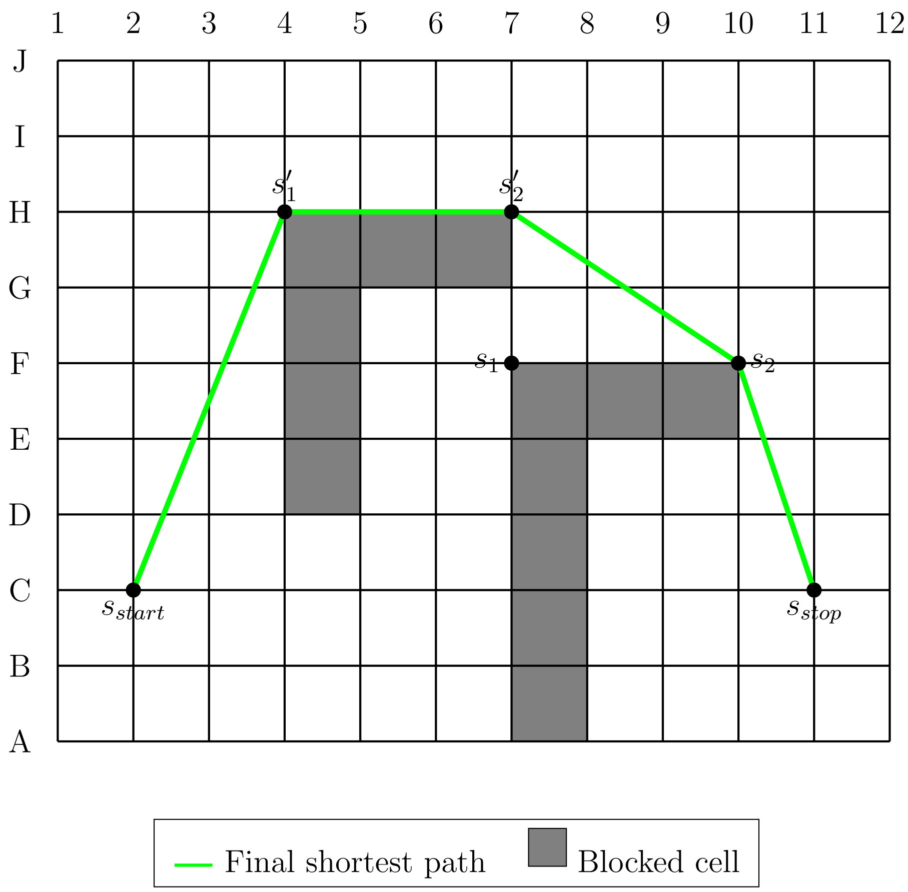


Figure 3.40: Forward-pruning scenario: Final stage

The pseudo-code for the back-pruning strategy can be seen in Algorithm 7. Forward pruning is called at each step when moving along an obstacle's boundary, but only for paths that had previously reached the final destination node s_{end}. The algorithm for forward pruning is similar to that of backward pruning, with a few distinctions. Iteration of the elements happens from the current goal node, $s_{interStart}$, towards the end goal s_{end}. The algorithm looks at the node s_i and the next node in the path, s_{i+1} (Line 7) and performs the cross-product (Equation 3.2) to check if the path is taut (Line 10). The cross-product indicates the sidedness of s_i, relative to $\overrightarrow{s_{wall}, s_{i+1}}$ (as opposed to $\overrightarrow{s_{i-1}, s_{wall}}$ for back-pruning). If the node s_i lays on the opposite side of $\overrightarrow{s_{wall}, s_{i+1}}$, vertex s_i is pruned from the path, otherwise the algorithm terminates, as the path is taut. In the same manner as for back-pruning, the condition at Line 8 allows for early termination of the algorithm when the path has switched sides.

Algorithm 7 Ray Path Finder: Path Forward-Pruning

1: **function** FORWARDPRUNE(s_{wall})
2:   **if** $s_{goal} \notin path$ **then**
3:     **return** ▷ haven't reached final goal; cannot forward prune
4:   $n \leftarrow path$.INDEXOF($s_{interGoal}$)
5:   **for** $i \leftarrow n$ **to** $path.size - 1$ **do**
6:     $s_i \leftarrow path$.NODEAT(i) ▷ $s_i \equiv s_{interGoal}$
7:     $s_{i+1} \leftarrow path$.NODEAT($i+1$) ▷ node after $s_{interGoal}$
8:     **if** $sgn(s_i) \neq sgn(path.side)$ **then** ▷ is s_i from opposite side?
9:       **return** ▷ path at inflection point; cannot prune beyond it
10:     **if** $sgn(\overrightarrow{s_{wall}s_i} \times \overrightarrow{s_{wall}s_{i+1}}) = sgn(path.side)$ **then** ▷ checking sidedness (Equation 3.2)
11:       **return** ▷ path is taut; no need to prune
12:     $path$.REMOVE(s_i) ▷ s_i on opposite side of $\overrightarrow{s_{wall}s_{i+1}}$; prune it
13:     $s_{interStart} \leftarrow s_{i+1}$ ▷ s_{i+1} becomes new interior goal node
14:   UPDATELENGTH($path$)

3.7 Redundant paths

A search-space that contains non-convex obstacles presents an additional challenge for our algorithm. When the search encounters an object it follows the edge of its boundary until the jump-off condition is met, at which point it leaves the edge of the object and resumes ray casting towards a goal. If the path encounters another obstacle when performing the line of sight query, there is no guarantee that the second intersection is with a new obstacle or is in fact with the boundary of the same obstacle encountered previously. The solution to this problem is trivial: Let us consider the example illustrated in figure 3.41 where (D_3) and (D_{11}) represent the start and end point respectively, and the non-convex obstacle indicated by grey cells. We consider the initial path, represented by blue arrows, to be a left-sided path and label it as $path_1$. As in our previous examples, the algorithm initiates its search with a line-of-sight check and encounters an obstacle at (D_5). The path now splits into two paths, $path_1$ which proceeds with following the obstacle on the left-hand side of the boundary and $path_2$ (represented by green arrows) which follows it on the right-hand side. During the boundary following procedure, $path_1$ encounters two corner points, (E_5) and (E_6) which are added to $path_1$ as s_1 and s_2 respectively. As s_2 is a jump-off point with a potential line-of-sight to s_{stop}, $path_1$ abandons following the boundary and proceeds to travel in a straight line towards the goal. At the next intersection with an obstacle, $path_1$ once again splits into two paths. The original path, $path_1$ a left-sided path described by (s_{start}, s_1, s_2) with s_1 and s_2 as left-sided nodes, and the new path, $path_3$ (represented by red arrows) a right-sided path containing the same nodes as $path_1$, i.e. (s_{start}, s_1, s_2) with s_1 and s_2 as left-sided nodes. At this junction, let us consider what happens to the new path, $path_3$. The path proceeds with following the object boundary on the right side until it arrives at (E_6) which it identifies as belonging to the path, as the node s_2 which has an opposite side to the current side of $path_3$. This implies that the path is following the boundary of a non-convex object and it has returned to the jump-off point of it's parent path, i.e. $path_1$. If allowed to continue beyond this point, $path_3$ would eventually return to the initial point of intersection, (D_5), and continue exactly as $path_2$, which would make $path_3$ redundant. From this we may conclude that any path which, after having switched sides, intersects itself from the opposite while following a boundary is redundant, as it would not be taut and because it would imply the existence of an early taut path that would perform the same search as the redundant path.

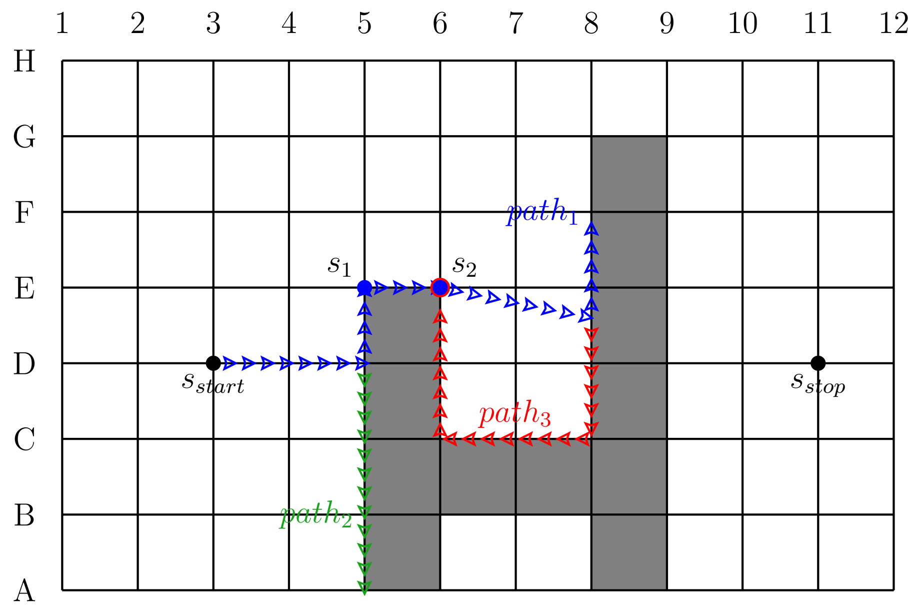


Figure 3.41: Redundant path example: $path_1$ Left-bound path. $path_2$ - Right-bound path (green). $path_3$ left–right-bound path $path_3$ intersects itself at s_2 while following wall in opposite direction to its parent.

Lemma 3.1. *If a path revisits a node that has not been pruned while wall following in the opposite direction it had at the first visit of the node, the path is deemed redundant, as it intersects the same obstacle.*

Proof. If we let the path P continue on the obstacle boundary, beyond the revisited node, it will arrive at the previous intersection point of its parent with the obstacle. At this junction, a path P' had previously split off, given the path bifurcation procedure, and already tracing in the same direction as P. This implies that beyond the intersection, P would retrace the search steps of P', making $P \equiv P'$. Thus, P is a redundant path and can be safely terminated early. □

3.8 Self-intersecting paths

Within a search-space that contains non-convex obstacles, there can be a number of scenarios in which a path may intersect itself. The reasons for these intersections can help in determining what is the appropriate solution in dealing with these scenarios.

G-shaped obstacle

Considering the scenario presented in Figure 3.42, let us follow the right-bound path's expansion. The path moves in a straight line from s_{start} (E_5) towards s_{goal} at (E_{11}) until in encounters an obstacle at E_6. It begins tracing the contour of the obstacle right-bound and discovers two vertex corners, s_1 at (D_6) and s_2 at D_7 and adds them to the path. From s_2 it leave the obstacle's edge towards s_{goal} but encounters the obstacle again. The path reverts back to tracing the object boundary right-bound. As it moves to C_9, the triangle inequality is broken and s_2 is pruned from the path.

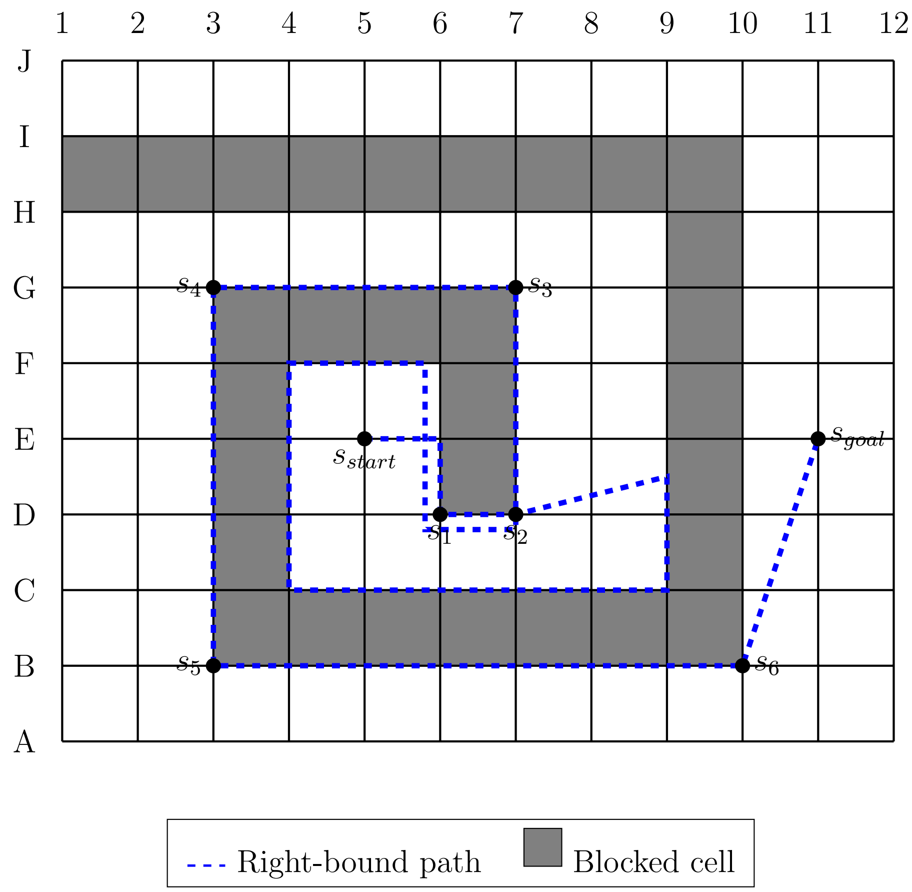


Figure 3.42: Self-intersecting path in “G” shaped obstacle

Moving forward on the boundary, s_1 is also pruned when the path reaches C_6. The path continues to trace the edge through C_4, F_4, F_6 and ends up in s_1 (D_6), which has been previously pruned. The path allows the vertex to be re-added to the path. Similarly, D_7 is re-added to the path, but now the turn counter of the path does not allow the path to leave the edge of the obstacle. It instead continues the contour trace north-bound, towards G_7. It identifies this vertex as a corner vertex and adds it to the path as s_3. Similarly, nodes s_4 at (G_3), s_5 at (B_3) and s_6 at (B_{10}) become part of the path. At vertex (B_{10}), the turn count again becomes negative which allows the path to leave the obstacle's edge and reach the goal from E_{11}.

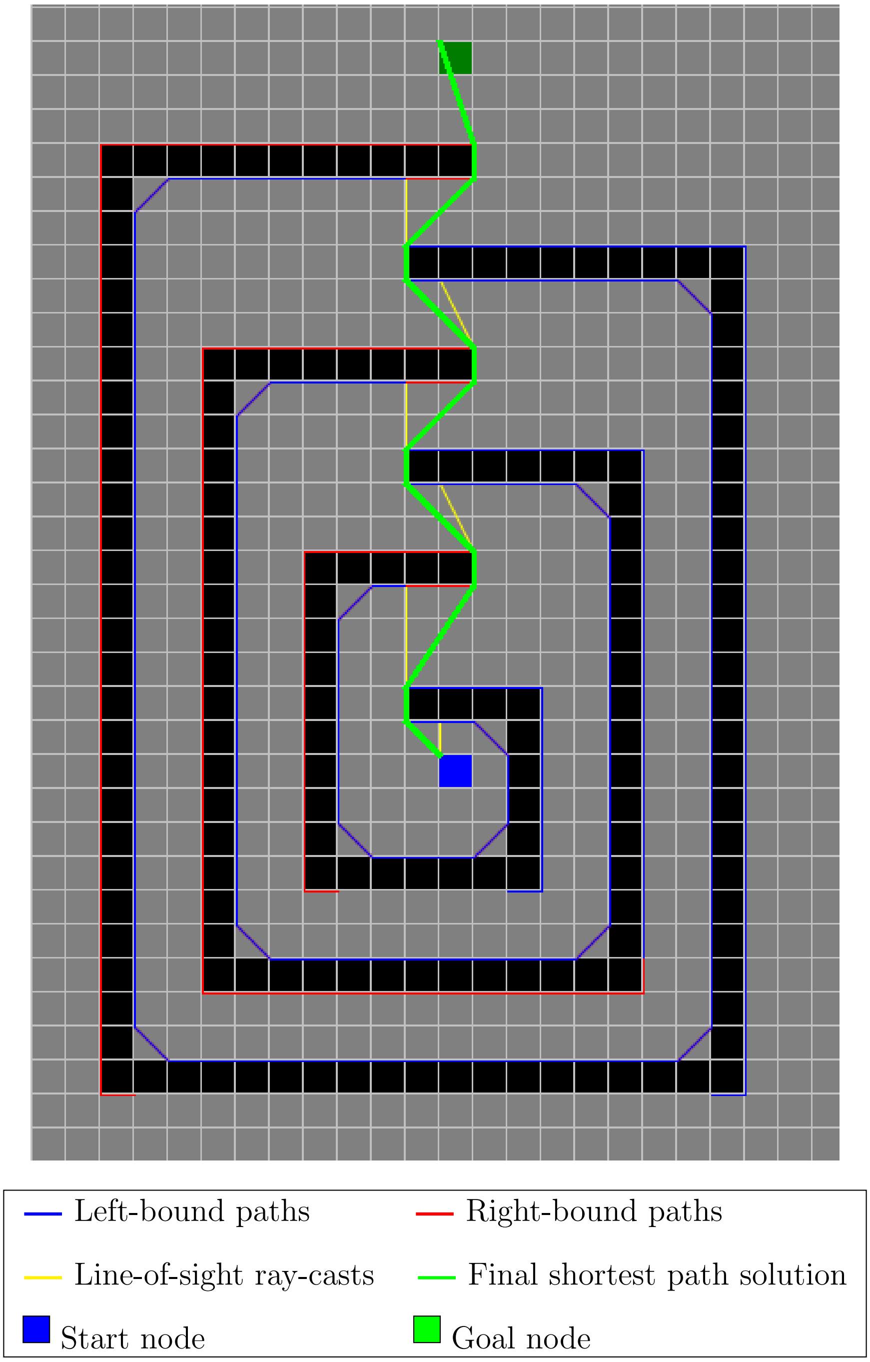


Figure 3.43: Screen-shot of solution in nested G-shaped obstacles (blue – left-bound wall-following; red – right-bound wall-following; yellow – ray-casting)

The screen-shots in Figures 3.43 & 3.44 illustrate two instances of "G-shaped" obstacles scenarios. Figure 3.44 presents a similar topology to Figure 3.43, with the exception of two blocked cells that close off the corridor for the path found in Figure 3.43. In Figure 3.43, the successful path is the one that initiates a left-bound wall-trace after its initial encounter with the smallest of the "G"-shaped obstacles, and afterwards alternates between right-bound and left-bound heading-changes. Closing off the corridor in Figure 3.44 creates the condition for a different path to be successful, namely the path that performs two initial left-bound turns when encountering the same "G"-shaped obstacle twice, similar to the path illustrated in Figure 3.42.

The unsuccessful (abandoned) paths in the figures that trace the walls of the obstacles (marked by the blue and red lines) present with a large coverage relative to length of the final paths identified. The reason for this is related to the method used to calculate the heuristic cost of a path, which computes the sum between the lengths of the segments forming a path, the cost from the last node in the path to the current position and the distance to the final goal. "Overhead" paths that follow the interior of a shape (i.e. curb inward) prune the "corners" that they identify in order to maintain path tautness, but in doing so, underestimate the value returned by the heuristic cost function. This leads the algorithm to mistake these paths as more favourable than they are in actuality, and prioritise their expansion to the detriment of other paths. Underestimated path costs are explored in more detail in Section 4.5 where the limitations of the algorithm are discussed.

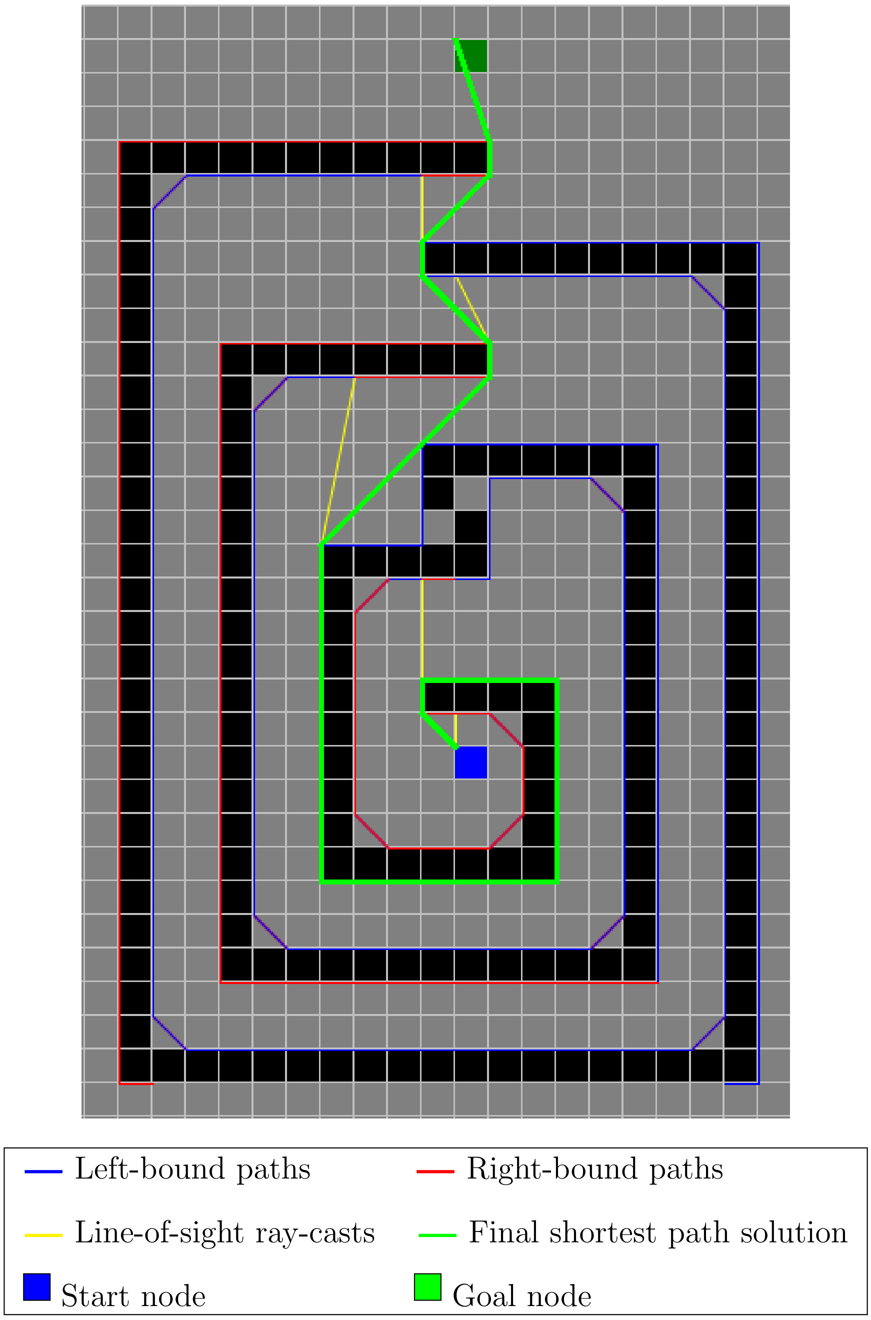


Figure 3.44: Screen-shot of solution in nested G-shaped obstacles with obstructed corridor (blue – left-bound wall-following; red – right-bound wall-following; yellow – ray-casting)

Locked-in start node

Contrasting the "G"-shaped obstacle scenario with the one illustrated in Figure 3.45, we can observe an instance of path intersection that would result in an infinite loop if not terminated.

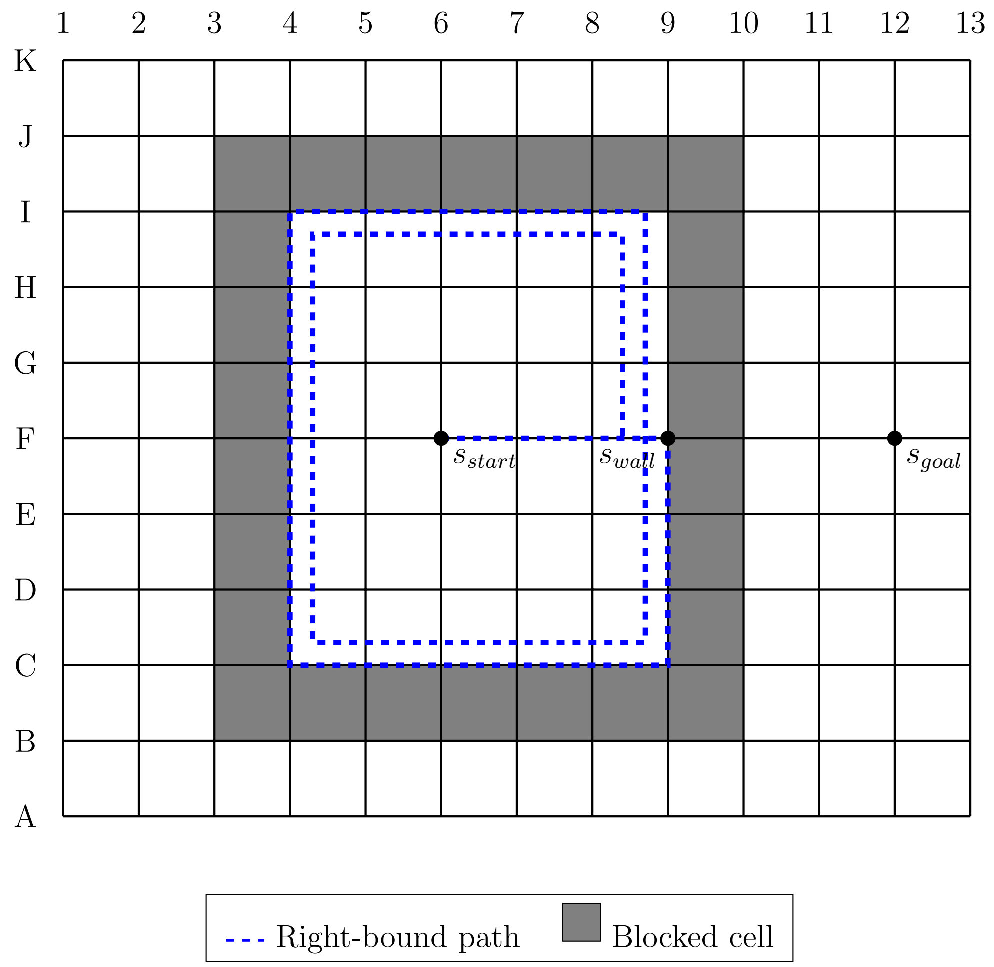


Figure 3.45: Self-intersecting path in locked-in start scenario

The s_{start} node at (F_6) is locked inside an obstacle's shape, with the goal situated outside of the shape, at (F_12). Following the right-bound path towards the goal, the obstacle is encountered at (F_9) (marked by s_{wall}). Wall following around the obstacle's boundary traces the interior contour of the obstacle through C_9, C_4, I_4, I_9 and passes through s_{wall} for a second time, similar to the path in Figure 3.42. Unlike the "G"-shaped scenario, however, the path continues to follow the same wall through C_9, C_4, I_4, I_9 and back to s_{wall}, increasing its turn count by 8 with every pass through s_{wall}. If allowed to continue, the path gets stuck in an endless loop, as no solution can be found. To remedy it, we can keep track of the number of times the path has passed through s_{wall} and terminate it if it passes more than twice. We allow the path to pass twice through s_{wall} to accommodate for the "G"-shaped obstacle in the previous scenario.

Lemma 3.2. *If a path revisits a node while wall following in the same direction it had at the first visit of the node, and the path is tracing along the inner bounds of an obstacle, it is only allowed to revisit the node once, after which the path is deemed unreachable.*

Proof. Given a start node that is locked inside an obstacle, the resulting paths will trace the inner boundary of the obstacle in an endless loop, and thus no solution exists. □

Remark 1. *It should be noted that the revisited node would not belong to the path, as it would have been pruned in order to maintain the path's tautness. The path is allowed to revisit the node once to allow it to escape from a potential "G-shaped" obstacles (e.g. Figure 3.42).*

Locked-in goal node

In the scenario presented in Figure 3.46 a solution does not exist. The end node at F_8, s_{goal}, is isolated within an obstacle with a hollow interior. We will examine the attempt made by the right-bound path to reach the goal. The search begins from node s_{start} at F_2 towards s_{goal} and intersects the obstacle at F_4. The path begins to navigate along the obstacle's edge on the right-hand side. When it reaches B_4 (marked by s_1), it determines the node to be a corner vertex and adds it to the path. The path continues to trace along the obstacle boundary without its turn count going below 0.

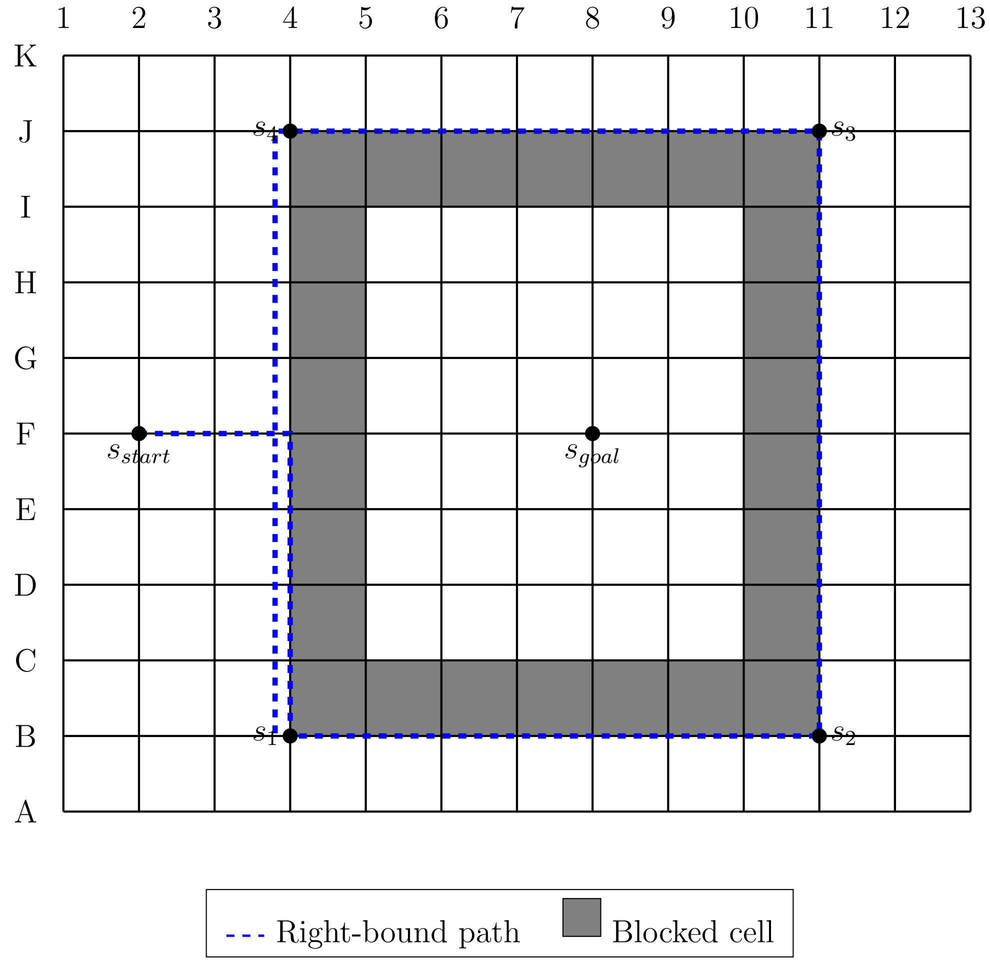


Figure 3.46: Self-intersecting path in locked-in goal scenario

The contour trace finds 3 more corner vertices in B_{11}, J_{11}, and J_4, respectively, and adds them to the path as s_2, s_3 and s_4. From s_4, the path heads south-bound and encounters s_1 once again. Because the path is taut, s_1 has not been pruned from the path. Allowing the trace to continue would result in a an endless loop in which the path would pass repeatedly through the same vertices. The left-bound path fails in a similar fashion, as it would intersect itself in s_4. We can conclude that a path which intersects itself creates a loop, and can be discarded as it would never reach the end goal.

Lemma 3.3. *If a path revisits a node that has not been pruned while wall following in the same direction it had at the first visit of the node, and the path is tracing along the outer bounds of an obstacle, the path is deemed* *unreachable, as it intersects itself to form a loop around the obstacle.*

Proof. If the path is allowed to continue beyond the revisited node, it continues to retrace the outer bounds of the obstacle, revisiting all other nodes identified in the first pass as being part of the path (or convex convex hull of the obstacle). If not terminated, the path would perform this cycle ad infinitum, never arriving at a solution. □

Remark 2. *It is allowed for a path that has reached the goal and is attempting line-of-sight between each pair of nodes to revisit a node s in the same direction if it had been pruned. This can happen if the free-space assumption fails and the shortest path between nodes does pass through s.*

3.9 Chapter Summary

This chapter introduced a number of algorithms developed with the purpose of aiding searches for Ray Path Finder, the proposed path-planning methodology.

The novel path-planning algorithm operates under a free-space assumption. It optimistically assumes there are no obstacles between the start and goal node, nor between any two nodes that it identified as part of the path. It navigates towards a goal in a straight line and if it encounters an obstacle, it follows along the obstacle's boundary in both left and right directions, until it is again able to travel in a straight.

A number of features are necessary for the path–finding algorithm to exhibit the aforementioned behaviour. The notion of path state was introduced, to inform on the condition of each path and to allow it to transition between states or to allow the driving algorithm to discard paths that have reached an impasse. A contour tracing solution was introduced which allows avoiding passing through obstacles by moving around them, along their walls. A line-of-sight algorithm with intersection retrieval is derived in order to move a path in a straight line and determine when an obstacle is intersected. Path–pruning strategies are introduced to maintain a path's tautness during expansion. Methods of identifying redundant and self–intersecting paths are presented, to enable the algorithm to discard such problem-paths. Path direction enables directing the search towards the goal, and allow a path to infer when it should depart from the wall-following behaviour. A key feature, path sidedness allows for identifying nodes at obstacle corners that may become heading changes in a path, and also to prune such nodes if they compromise the path's tautness.

These features are essential for the main search algorithm, in which paths are raced against each other and are selected for expansion in the order of their heuristic lengths.

Chapter 4

Ray Path Finder: An Any-angle path planner

In this chapter we introduce the novel path-planning solution that was developed – Ray Path Finder. We describe how the best-first strategy behind the Ray Path Finder algorithm drives the main search forward across multiple paths and how RPF exploits 2D geometric properties of the topology and makes inferences about its search-space. We discuss the principle behind the algorithm, its properties, and address its current limitations.

4.1 Introduction

For paths identified by grid-constrained path-finding algorithms, such as A*, a node can only have a direct (immediate) neighbour as a parent. Because of this, the paths found are artificially restricted to only move orthogonally or diagonally along grid edges. Often in practice, after a grid-constrained algorithm identifies a solution, smoothing techniques are applied in a post-processing step in order to morph the path into a more realistic looking one, with fewer heading changes. However, post-processing the paths found by traditional edge-constrained find-path algorithms is not always able to improve paths [4]. In contrast, any-angle path-finding algorithms nodes can have as a parent any other node with a direct line-of-sight.

Any-angle path-planning algorithms generally find shorter paths, which have fewer heading changes, and make a robot's behaviour look more natural. For an any-angle algorithm to find the shortest path in a search-space, it must identify a taut path that has the lowest traversal cost between start and goal nodes. In the general sense, a path is considered taut if the path wraps tightly around an obstacle (similar to a Dubins path – Section 2.5.1). For an octile grid, a taut path also implies that all the heading changes in a path are formed by vertices that represent outer corners (i.e. vertices that have one occupied tile and three unoccupied ones as neighbours). A shortest path on octile grids must share this property.

Searching only amongst taut paths is desirable over having to consider all possible non-taut alternatives, which is what the proposed algorithm –Ray Path Finder– aims to do. The RPF algorithm achieves this by operating on a free-space assumption strategy. An initial path is directed to move in a straight line towards the goal. This expansion policy makes use of the variant of Bresenham's line algorithm presented in Section 3.2, which returns the points of intersection with an obstacle that breaks the line-of-sight with the target. At the point of intersection, a new path is generated and splits off from its parent path. The two paths trace the contour of an obstacle using the novel contour tracing algorithm introduced in Section 3.3. The expansion of the paths is guided by a best-first-search strategy which prioritises the path with the lowest heuristic length. Outer-corner points are identified on the contour of an obstacle and added as nodes of the path. When a path is allowed to move in the direction of the target again, it reverts to moving in a straight line. This process is performed for all path until a path reaches the goal. For paths that reach the goal a line-of-sight check is performed between its nodes to verify the free-space assumption. If obstacles are encountered, the path is again split into two and are expanded based on their updated heuristic lengths.

The problem of efficiency in a path-finding algorithm revolves around two key aspects, runtime and path length [75]. Generally, there is a trade-off between minimising the length of a path and minimising the runtime of the algorithm, as the two goals are antithetical. Using a best-first search strategy to prioritize the expansion of the most promising path, RPF wishes to strike a balance between runtime and path length. The best-first search algorithm expands the most promising paths first, allowing for the algorithm to converge to a solution quickly. Subsequently the algorithm attempts to shorten the paths that have reached the goal based on their heuristic length, expanding the most promising one first. When a path has been validated as having line-of-sight between all its subsequent node pairs, the resulting path will have the shortest length.

4.2 A recursive approach

In order to identify some of the issues that an efficient path-planner should attempt to address, let us imagine a naive approach to an RPF variant illustrated in Algorithm 8.

Algorithm 8 Naive Recursive Approach

function RecursiveRaySearch(s_{start}, s_{goal}, $side$)
  $p_{next} \leftarrow$ MoveInStraightLine(s_{start}, s_{goal})
  if $p_{next} = s_{goal}$ **then**
    return $''path\ found''$
  else
    $corner_{left} \leftarrow$ FollowWallUntilCorner($LEFT$)
    $corner_{right} \leftarrow$ FollowWallUntilCorner($RIGHT$)
  if $\exists\ corner_{left}$ **then**
    RecursiveRaySearch($s_{start}, corner_{left}$)
    RecursiveRaySearch($corner_{left}, s_{goal}$)
  if $\exists\ corner_{right}$ **then**
    RecursiveRaySearch($s_{start}, corner_{right}$)
    RecursiveRaySearch($corner_{right}, s_{goal}$)
  return $''no\ path\ found''$

The algorithm can be thought of as a series of paths racing towards a goal, with each path behaving similar to individual bug algorithms. A path travels towards a target until it encounters an obstacle, after which it generates a clone of itself and they both perform wall-following in opposite directions, one tracing the obstacle on the left-hand side and the other on the right. Each bug traces the obstacle boundary until it identifies a "corner", after which two new searches are performed, one attempting to travel from a start node to the corner node and from the corner node to a target node. This process is repeated recursively until either a search fails or a path is found between all node pairs in the recursive stack, in which case a path solution exists between start and goal.

Given a finite distance between start and goal positions, and a goal isolated inside an enclosure with finite bounds, no solution exists in attempting to reach the target (e.g. goal is inside a locked room). In a virtually infinite search space, heuristic path-planning algorithms such as those in the A* family, will exhaustively explore the connected nodes of the free space, which negatively impact runtime. Bug algorithms avoid this drawback, as termination conditions can stop the search of a bug after, for example, the bug travels along the outer perimeter of an obstacle, creating a cycle in its path. Bug algorithms that alternate wall-following directions may allow more than one pass but eventually terminate because of the aforementioned condition. Ideally, a path-planning strategy would be able to take advantage of desirable properties from both bug algorithms and heuristic path-planning algorithms.

The approach presented in Algorithm 8 has many shortcomings and would be very inefficient. From an implementation standpoint, the recursive nature of the search algorithm would employ a large amount of resources for an environment with a complex layout. Also, the algorithm would only be able to handle simple object geometries, as it does not have a clear notion of direction and does not guarantee termination. For example, a path could get stuck in an endless loop inside a “G-shaped” obstacle.

The naive recursive implementation behaves like a depth-first search algorithm [23]. An important drawback of depth-first search algorithms is the risk of non-termination. Also, the algorithm does not make any attempt to maintain path tautness, nor prioritise the most promising paths. Because the search interweaves nodes in the path in a simple manner, if fails to maintain path tautness. The reason for this is simple. Given three consecutive nodes s_1, s_2 and s_3 identified by the search as belonging to a path, and s_1 not having a direct line-of-sight to s_2, the recursive call to find a path from s_1 to s_2 can identify a hypothetical node s'_1 that has a direct line-of-sight to s_3. It such an instance, the taut path (s_1, s'_1, s_3) would not pass through s_2. For an algorithm to be optimal the tautness of each path would need to be preserved throughout the search. Related to this issue, assuming that the algorithm identifies s_2 and afterwards identifies s_3, if s_1 has a line-of-sight to s_3, allowing the algorithm to proceed would pollute the search. Searching for paths between the node pairs (s_1,s_2) and (s_1,s_3) would not be conducive to an optimal path as it would not need to pass through s_2.

Depth-first search solutions can suffer from a termination problem. Let us consider a number of thought experiments. Given an infinite search space, the naive algorithm presented here would exhibit this fault. A simple thought experiment makes this evident. Consider a scenario in which a wall blocks the line-of-sight between the start and stop nodes, and which is bound at one end but extends infinitely in the opposite direction. For such a configuration, Algorithm 8 could potentially run forever if attempting to follow the wall in the unbound direction. The solution to the termination problem is such that, rather than allowing a path to search exhaustively, multiple paths can search alternatively in different directions, prioritising based on the current best guess (heuristic estimation) of a path's cost. In essence, the solution is to manage the paths using a best-first search expansion strategy.

4.3 The Ray Path Finder Algorithm

The concept of Best-first search is one of the most studied topics in path-planning and has been around for decades. Strategies for Best-first search have been discussed in [21], [76], [77], [78] to name a few. In its most general form, Best-first search is an informed graph search algorithm which uses a heuristic cost function to expand the most promising nodes first. Best-first search has the advantage that, if it reaches a dead-end node, the algorithm continues to expand other nodes [79]. The A* algorithm is also, in essence, a Best-first search algorithm, but it distinguishes itself by taking into account the cost-distance already travelled in addition to its heuristic estimate towards the goal [21].

Ray Path Finder, the proposed algorithm, employs a best-first search strategy, but rather than applying it to grid nodes, it applies it to individual paths. RPF selects the path which looks to be the most promising with respect to its assumed length and explores it first. In essence, the algorithm races multiple bug-like paths towards the final goal, until an obstacle in encountered or the goal is reached. If an obstruction is detected, the path splits in two, and they begin to trace the obstacle's boundary in opposite directions, until they are allowed to move in the direction of the final goal.

The order in which the racing paths are expanded is based on their presumed length and distance to the goal node, while operating under a free-space assumption. A path is populated with nodes only found on obstacle boundaries and which break the triangle inequality and, as such, would undermine the free-space assumption between nodes. Until a path reaches the final goal, the nodes are optimistically assumed to have line-of-sight, with a heuristic cost function described by Equation 4.1:

$$H(\pi) = \sum_{i=1}^{n-1} D(s_i, s_{i+1}) + D(s_n, s_{wall}) + D(s_{wall}, s_{stop}) \tag{4.1}$$

where π is the path, $s_{stop} \notin \pi$ and $n = index\,Of\ (s_{interStop})$.

After a path has reached the goal by moving around obstacle boundaries, the algorithm re-evaluates the path's free-space assumption by performing line-of-sight checks between its pairs of consecutive nodes. The heuristic cost function for paths that have reached the goal is described by Equation 4.2:

$$H(\pi) = \sum_{i=1}^{m-1} D(s_i, s_{i+1}) + D(s_m, s_{wall}) + D(s_{wall}, s_n) + \sum_{j=n}^{p-1} D(s_j, s_{j+1}) \tag{4.2}$$

where π is the path, $s_{stop} \in \pi$, $m = index\,Of(s_{interStart})$,
$n = index\,Of(s_{interStop})$ and $p = index\,Of\ (s_{stop})$.

The pseudo-code for Ray Path Finder is presented in Algorithm 9. Let us consider Figures 4.1, 4.2 & 4.3 as examples to illustrate the behaviour of the algorithm.

In the initialisation phase, the main function of the algorithm creates a path with a randomly chosen initial side, either left or right. The choice of side is inconsequential, as any child path will have an opposite side to its parent, thus having exhaustive coverage. For simplicity, the initial path is selected as being left-bound and its state is set to $RAYCAST$, under the assumption that the path can travel in a straight line towards the goal (Line 6).

The main function acts as a best-first search algorithm, prioritising the path that is so far assumed to be the shortest. It does this by employing a priority queue which orders the paths in order of their assumed lengths, from shortest to longest (Line 7). As the paths are expanded and gradually populated with nodes, their heuristic costs are updated, which modifies their order in the queue.

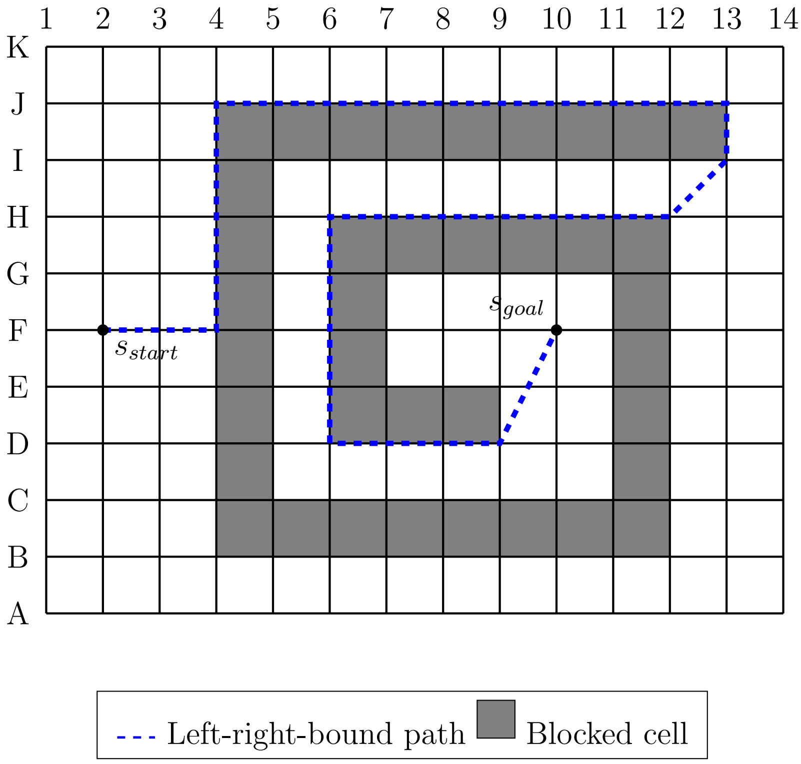


Figure 4.1: Spiral path example: Alternating directions (left-right-bound first, right-bound second)

The initial path that was created is inserted into the queue, after which the main loop is entered (Line 9). The while loop polls the priority queue, extracting the most promising path, until all paths are exhausted. The selected path is asked to perform an action based on the state in which it finds itself, in our case, $RAYCAST$. Referencing Figure 4.1, the path moves in a straight line from F_2 until it encounters an obstacle at F_4, at which point its state changes to HIT_WALL (Line 43) and the new state is returned to be handled by the main function, by the switch case statement. At this stage, a new path is created, which splits off in an opposite direction (i.e. right-bound path in Figure 4.2), and which is also placed in the queue (Line 27). After the paths have been split, their state is changed to $FOLLOW\_WALL$, and the loop reiterates. In F_4, the cost of both paths (left-bound path in Figure 4.1 and right-bound path in Figure 4.2) is equal and will remain as such, with the priority queue expanding them alternatively until they reach J_{12} and B_{12}, respectively, at which point the left-bound path, being the shorter one, will gain an advantage. While both paths can reach the goal and their constituent nodes have line-of-sight between them, the right bound path's heuristic cost estimate will prove longer than that of the left-bound path and, as such, the path will be taken out of the priority queue (Line 17).

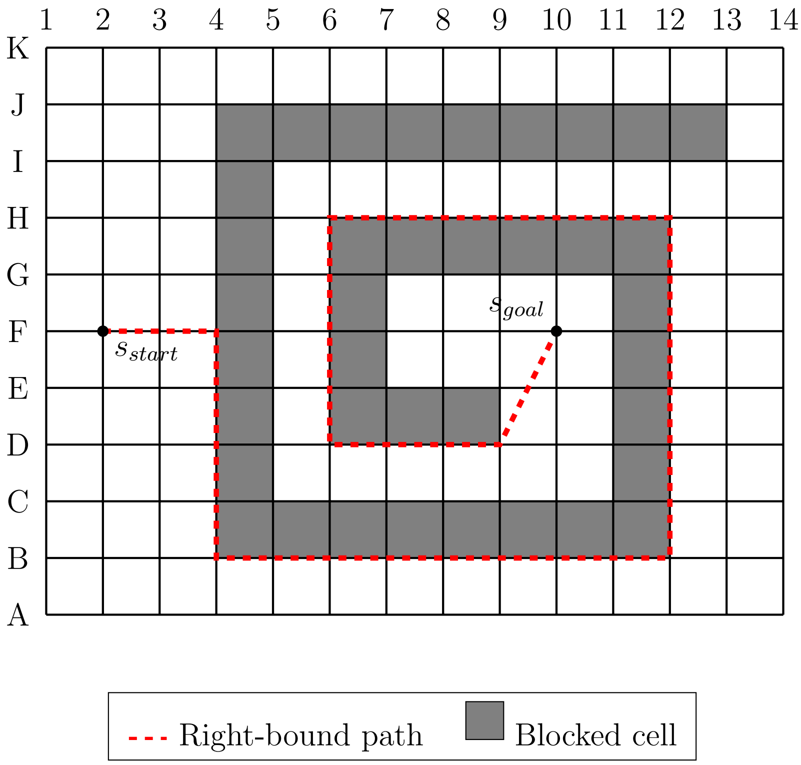


Figure 4.2: Spiral path example: Right-bound path

Switching our focus to the left-bound path (Figure 4.1), The wall following procedure is performed until the path reaches I_{13}, at which stage, the path's state is switched back to $RAYCAST$, and the path moves towards the goal. It, however, encounters an obstacle at H_{12}, at which point a new path is created, with a right-bound direction (Line 27). The original left-bound path, after wall-following through B_{12}, B_4, would intersect itself in J_4, becoming $UNREACHABLE$ and will be discarded by the main loop (Line 24). As such, we continue instead with the new right-bound path (blue line in Figure 4.1), which is conducive to a solution. At H_{12} the path traces the obstacle and identifies H_6, D_6 and D_9 as corners and adds them to the path, after which it reaches the goal and changes state to $GOAL\_FOUND$. Similarly, the right-bound path (red dashed line in Figure 4.2), added to the queue when the obstacle was encountered at F_4. Following the contour of the obstacle, it discovers nodes B_4, B_{12}, H_{12}, H_6, D_6, D_9 and which point it has a clear line-of-sigh to the goal. Although the path would have line-of-sight between all its node, because the left-bound path in Figure 4.1 is shorter, it reaches the goal before the right-bound one, and its final shortest length would be shorter than the heuristic length of the right-bound path, Line 17 would remove the right-bound path before it is cleared. The function invoked for each path in the main which handles the state of each path (Line 33) delegates which subroutines are required by the different states of the path. Consider Figure 4.3 as reference. The figure presents a "G-shaped" obstacle with start node at H_5 and goal node at H_9. The main loop of the algorithm initiates the search by inserting a left-bound path to the queue, and, in the while-loop calls the $HandleNextState$ function on the path with a $RAYCAST$ state. The switch statement is entered and activates the test at Line 36. The line-of-sight function (Algorithm 5) is invoked with parameters $s_{interStart}, s_{goal}$. At this state, the only node in the path is the start node and, as such, $s_{interStart} \equiv s_{start}$. For Figure 4.3, $s_{interStart} = H_5$, and $s_{goal} = H_9$. As the nodes do not have a line-of-sight between them, the else branch is activated and the state is set to HIT_WALL (Line 39), and the function returns control to the main. On the next call, the algorithm enters the branch at Line 48, and updates s_{wall} and s_{free}, the parameters requires for the wall following procedure with the points of intersection with the

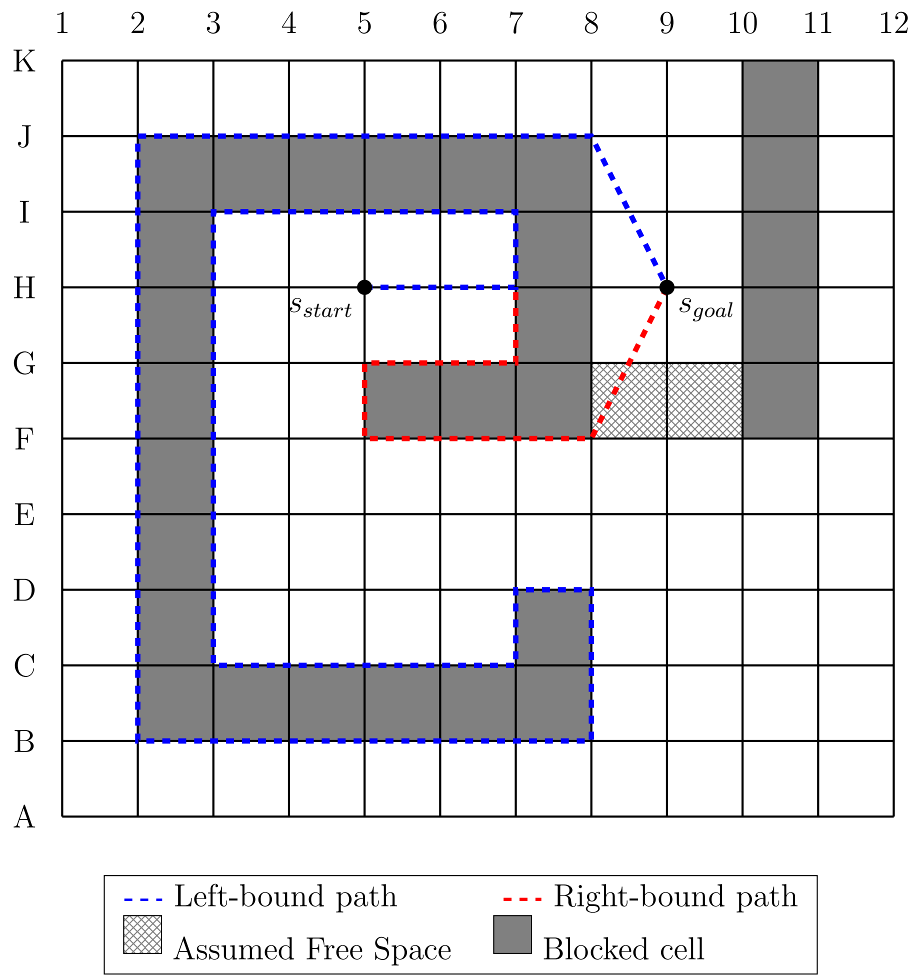


Figure 4.3: Left-bound and right-bound paths for start-node in G-shaped object

obstacle returned by the *LineOfSight* function. The state is updated to *FOLLOW_WALL*, so that, at the next call to the function, Line 47 can invoke the wall following procedure on the path. For Figure 4.3, in the first instance, let us assume that the grey cross-hatched cells at at G_8 and G_9 are free. As such, the right–bound path (marked with a red dashed line) is expanded before the left-bound one, traces the contour of the obstacle, discovering the path nodes G_5, F_5, and finally, F_8, after which its state is changed to *RAYCAST* once more. As there exists a line-of-sight to s_{goal} from F_8, the test at Line 36 succeeds and Line 37 sets the state to *GOAL_FOUND* and the method returns. Because the goal has now been found, the next call to the function activates the branch at Line

40, which iterates over pairs of nodes in the path. For each node pair (s_i, s_{i+1}), the line-of-sight procedure is invoked (Line 42), to check the free-space assumption. If a line-of-sight test fails, the for-loop terminates early, returning the state as HIT_WALL. Because the right-bound path in Figure 4.3 has line-of-sight between all its nodes, all iterations of the for-loop succeed and the state is set to $CLEARED$ (Line 45) and the function returns normally, allowing for the removal of the path from the queue, storing its final length. The longer left-bound path (blue line in Figure 4.3), on its way to s_{goal} will at some point provide a heuristic length that is longer than the stored final length of the right-bound path (Line 17), and will be removed from the queue as it would not be able to provide a shorter solution.

The wall-following strategy at Line 53 traces along an obstacle's boundary, identifying potential heading changes, and adding it to the path. It does this by repeatedly applying the contour-tracing function (Algorithm 4), until a new free node is identified on the boundary. Let us consider the case in which the cell-tiles at G_8 and G_9 in Figure 4.3 are not free. The right–bound path would instead trace the obstacle's wall from F_5 *East*–bound to F_{11}, and *North*–bound onto K_{11}, at which point it would leave the map (Line 56), and the path's state is set as $UNREACHABLE$. Thus, the only other path left in the queue, the left-bound one (blue path in Figure 4.3) is expanded and reaches the goal.

Lines 61 & 64 handle the cases for when a goal or inter-goal lie on the obstacle's edge and are encountered by the contour-tracing algorithm. Line 67 examines a path for redundancy (Section 3.7), making it unreachable if it follows an obstacle's boundary in the opposite direction than the one it had when it first passed through $s_{interStart}$. Line 70 address self-intersecting paths (Section 3.8) which are also unreachable. After the node that passes these tests, the algorithm is allowed to proceed and evaluate it as a possible candidate in the path. If a corner node is identified (Line 73), it needs to be added to the path. Its placement in the path depends on whether the path had previously reached the final goal or not. For a path that had previously reached the goal, and is in the process of verifying the free-space assumption between its nodes, the newly identified corner is placed after the inter-start node (Line 75). In other words, the free-space assumption was wrong and the path does not have line-of-sight between the two consecutive nodes under examination (i.e. $s_{interStart}$ and $s_{interStart+1}$), in which case, the newly identified corner lies on the obstacle boundary that breaks the line-of-sight. For paths that are ray-casting towards the goal (i.e. from $s_{interStart}$ to s_{stop}), the corner is simply appended at the end of the path (Line 77). For both cases, the new corner now becomes the next node from which the search continues (Line 78). After each change in the structure of the path, the path is pruned in order to maintain its tautness (Line 79). This procedure ensures that a path can, at any stage in the search, provide the most optimistic score based on free-space assumption, to ensure an admissible heuristic. This, in turn, allows for the most promising path to be expanded first. Finally, if a path's direction tracker (i.e. turn count – Section 3.5) allows it to abandon following an obstacle's boundary (Line 80), its state is changed to $RAYCAST$ and the path can travel in a straight line towards the goal (Line 36).

Algorithm 9 Ray Path Finder

```
function MAIN(s_start, s_goal)
    pathQueue ← ∅            ▷ paths are ordered from shortest to longest length
    clearedPathQueue ← ∅
    shortestLength ← ∞
    state ← RAYCAST
    path ← PATH(s_start, s_goal, state, LEFT)
    pathQueue.INSERT(path)

    while pathQueue ≠ ∅ do
        path ← pathQueue.PEEK()
        state ← HANDLENEXTSTATE(path, state)

        switch state do
            case GOAL_FOUND :
                UPDATELENGTH(path)
            case FOLLOW_WALL :
                if LENGHT(path) > shortestLength then
                    pathQueue.REMOVE(path)
            case CLEARED :
                if LENGHT(path) < shortestLength then
                    shortestLength ← EUCLEDIANLENGHT(path)
                pathQueue.REMOVE(path)
                clearedPathQueue.INSERT(path)
            case UNREACHABLE :
                pathQueue.REMOVE(path)
            case HIT_WALL :
                pathQueue.INSERT(path.SPLITPATH(path.side.opposite))


    if clearedPathQueue.ISEMPTY() then
        return "no path found"
    else
        return clearedPathQueue.POP() ▷ retrieve shortest path from front of the queue
```

Ray Path Finder: Continued

```
function HandleNextState(path, state)
    switch state do
        case RAYCAST :
            if LineOfSight(s_interStart, s_goal) then
                state ← GOAL_FOUND
            else
                state ← HIT_WALL
        case GOAL_FOUND :
            for i ← 1 to path.size − 1 do
                if ¬ LineOfSight(s_i, s_{i+1}) then
                    state ← HIT_WALL
                    return state
            state ← CLEARED    ▷ Path cleared; potential solution; can be removed from list
        case FOLLOW_WALL :
            path.FollowWall(path.side)
        case HIT_WALL :
            s_wall ← s_intersect
            s_free ← s_pre_intersect
            state ← FOLLOW_WALL
    return state
```

Ray Path Finder: Continued

```
function FollowWall(path)
    do
        s_free ← TryNextNeighbour(s_wall, path.side)
        if s_free = ∅ then
            state ← UNREACHABLE
            return                                          ▷ Path out of bounds
    while ¬ IsFree(s_free)       ▷ Find the next free neighbour on the wall

    if s_free = s_goal then
        state ← GOAL_FOUND
        return                                        ▷ Goal is on obstacle edge
    else if s_free = s_{i+1} then
        state ← GOAL_FOUND
        return                               ▷ Internal goal on obstacle edge
    else if s_free = s_interStart ∧ Side(s_free) ≠ Side(s_interStart) then
        state ← UNREACHABLE
        return ▷ Redundant path: hit side of obstacle and backtracked
    else if s_free ∈ path then
        state ← UNREACHABLE
        return                       ▷ Node already in path. Path is looping
    else if IsCorner(s_free) then
        if s_stop ∈ path then              ▷ Path has already reached the end
            path.InsertAfter(IndexOf(s_interStart), s_free)
        else
            path.AddLast(s_free)
        s_interStart ← s_free
        path.Prune(s_interStart)
        if path.IsOnDirection() then
            state ← RAYCAST
            return                              ▷ Resume going in straight line
```

The extensive line-of-sight checks performed by algorithms such as Theta* can have a cumulative effect of slowing down the search. While Breshenham's line algorithm is not an expensive one, given a large enough search-space, it can still affect performance when compounded by the overhead of the search-algorithm itself. We advocate for a minimal use of Breshenham's line algorithm, and operate with a free-space assumption instead. This results in two use-cases for the line-of-sight algorithm. If the first instance, before a path reaches the final goal, line-of-sight is only performed when travelling in a straight line towards the final goal, and assuming free-space when identifying heading changes on obstacle boundaries. In the second instance, after a path has reached the final goal, line-of-sight is performed between consecutive node pairs in the path. This verifies if the free-space assumption between heading changes that the algorithm has made is correct. Those node pairs in the path for which a line-of-sight checks fails, are handled performing a sub-search using the same wall-following strategy and free-space assumption strategies.

Because the algorithm only performs line-of-sight checks between the last node in the path and the final goal until the path has reached the final goal there is no way to verify the assumption that there are no obstacles between any two nodes of the path. Additional line-of-sight checks may results in non-taut paths that would have to be considered for expansion, but which would not necessarily be conducive to a solution. Such paths would not only be unnecessary but, from an implementation point of view, would also increase resource demands, and negatively impact performance.

The screen-shot of RPF in action, illustrated in Figure 4.4, presents how the algorithm makes use of the line-of-sight algorithm. The yellow lines represent the line-of-sight expansions that the algorithm performs for the illustrated configuration. The only additional line-of-sight casts performed are those between the final nodes of the path that has reached the goal (masked by the green line in Figure 4.4), to assess if the path is clear. As can be observed, in this instance only fifteen line-of-sight expansions are sufficient for Ray Path Finder to arrive to a solution.

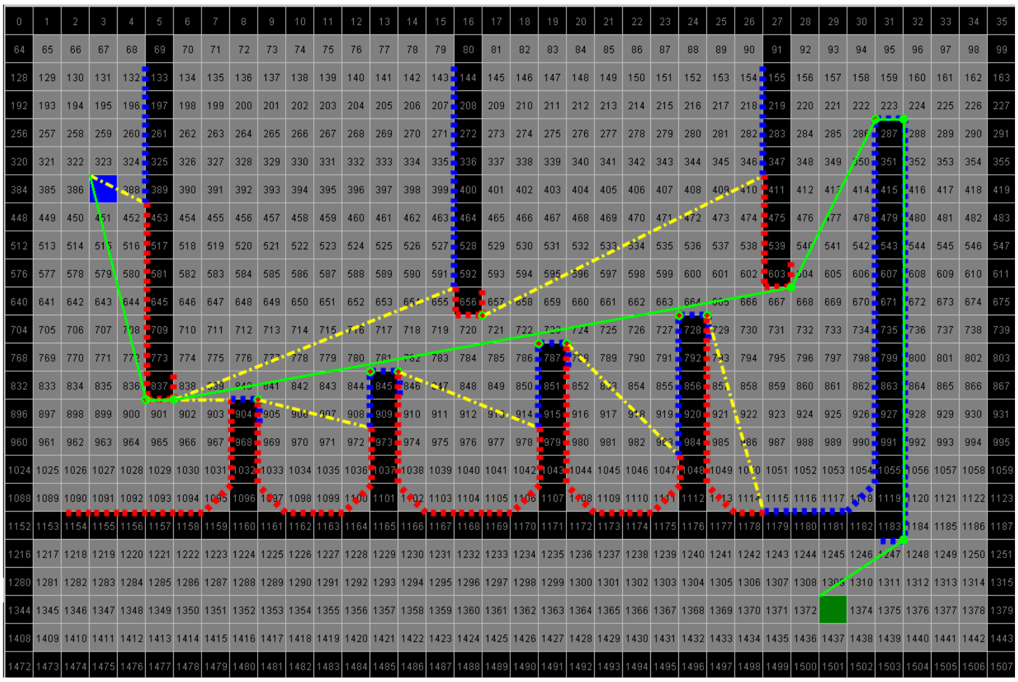


Figure 4.4: Example of minimal line of sight checks

Left-bound paths — Right-bound paths
Line-of-sight casts — Final shortest path solution
Start node — Goal node

4.4 Path updating

For specific topologies, an additional step is required to prevent a path from erroneously leaving the edge. Consider the configuration presented in Figure 4.5. Let us explore the right-bound path that travels from s_{start} at H_5 to s_goal at E_{13}. The path encounters an obstacle at G_8 and traces the edge of the obstacle until if finds the corner node at F_6. Nodes F_6 and F_7 break line of sight with s_{start}. Thus, F_6 is added to the path as s_1. As F_7 is on the left side of $\overrightarrow{s_1, s_{goal}}$ and the number of turns is negative, the path is allowed to leave the edge and travel in straight line towards s_{goal}. It encounters another obstacle at cell F_{11}. From this step, following the left-bound path traditionally yields three more corner paths s_2, s_3 and s_4 at K_5, L_5 and L_{12} respectively. The problem is evident, in that the path does not have a line of sight between s_1 and s_2. Even if that were not the case, a path that would pass through $s_{start}, s_1, s_2, \cdots$ would not be taut. Node s_1 would also not be pruned from the path as it violates no criteria to do so. One possible solution would be to purge s_1 and allow the algorithm to rediscover it when clearing the path, but this could prove inefficient and wasteful. The solution found for this problem is to perform an additional test that checks if the cell shared by the $s_{interStart}$ node (s_1 in our example) breaks line-of-sight with the node on the opposite wall. When tracing the obstacle on the left-side, the path arrives at G_{11} and the (G_6) cell now blocks the "potential" line-of-sight from s_1. To address this, a variation on the $FollowWall$ function in Algorithm 9 is applied, which involves tracing the contour of the obstacle from the $s_{interStart}$ on the opposite side (right-bound: marked in red in Figure 4.5) to the explored path (left-bound: marked with blue). Similar to the original $FollowWall$, the wall-following adds nodes to the path and stops when the updated path (red), under the free-space assumption, is optimistic that the newly discovered corner node has a line-of-sight to the node on the opposite wall. Applying this strategy to the example in 4.5, path follows wall, on the opposite side, from s_1 to G_9, identifies s'_2 as the next valid corner and adds it to the path as a right-bound corner. The normal path tracing resumes until K_8, when the cell at G_8 now blocks the line-of-sight from the new $s_{interStart}$ node (s'_2). Tracing starts from s'_2 until s'_3 is identified as a new corner node with possible line of sight to K_8. At this stage, there are no more issues and the algorithm is allowed to continue normally, identifying s_2, s_3 and s_4 as corners, resulting in the path $\{s_{start}, s_1, s'_2, s'_3, s_2, s_3, s_4, s_{goal}\}$.

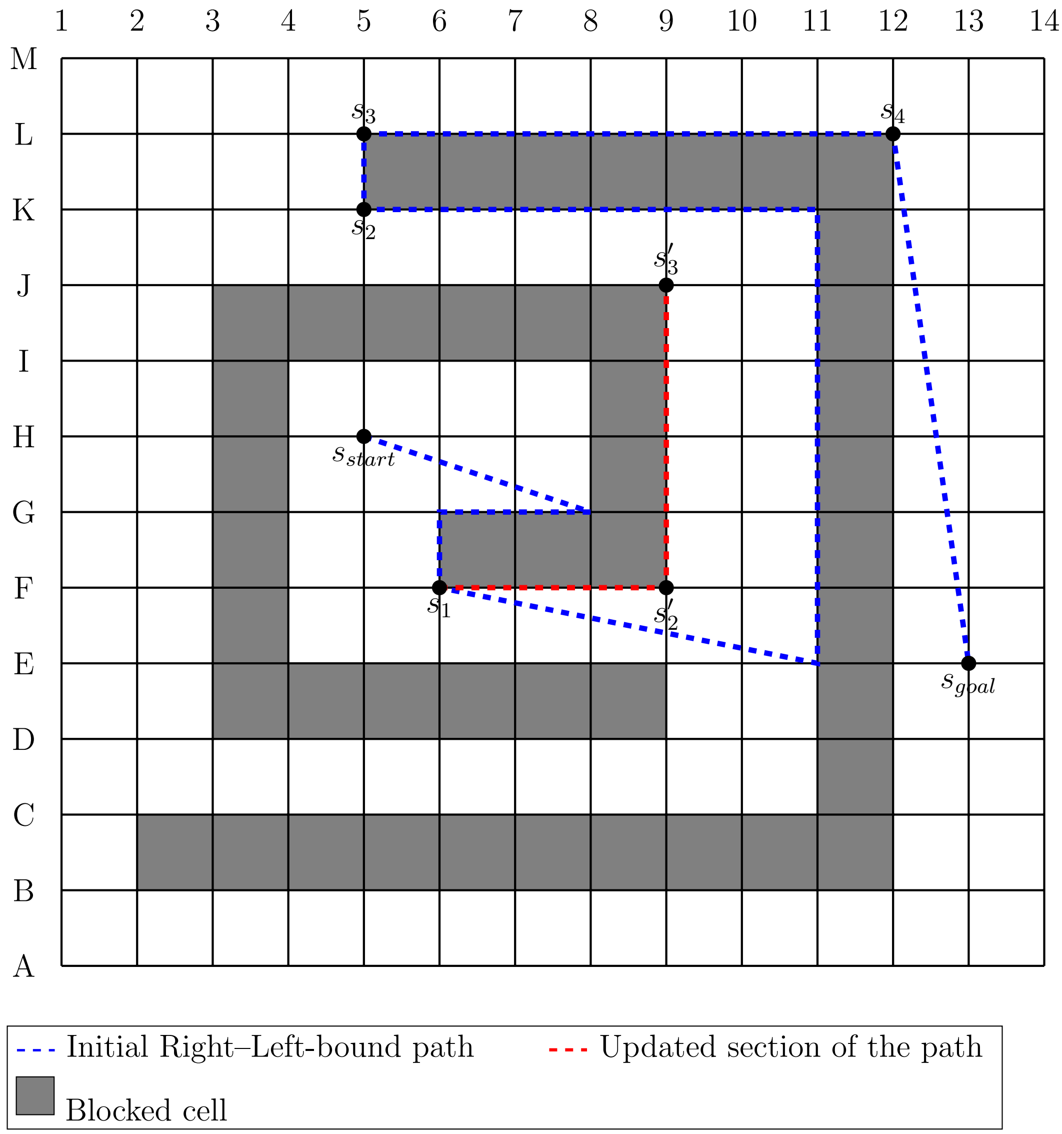


Figure 4.5: Corner updating example: path crosses over at (F_{11}); nodes (F_9) and (J_9) are appended to the path before (K_5) and (L_5)

4.5 Limitations

One important issue that remains unaddressed by the current implementation is that of underestimation of path heuristic lengths for certain topologies. This drawback impacts the performance of RPF, due to the algorithm performing extensive searches along "inward" paths. These circumstances can arise for paths that intersect an obstacle's interior contour when the search-space is bound by it. The class of problematic configurations can be illustrated through a representative example in Figures 4.6 & 4.7. Firstly, with Figure 4.6, we follow the path conducive to a solution and contrast it with the "inward" path Figure 4.7. The nodes are labelled in the order of their discovery. The left-handed path initiates travel from the start node at D_{10} towards the target at J_{10}, but encounters the inner boundary at E_{10}. Tracing the edge, it discovers two corner points, s_1 and s_2 which are appended to the path. After it leaves the wall at s_2, it encounters the obstacle again. The right-bound path (coloured red) is dropped as it is redundant, i.e. it intersects left-sided node s_2 from the right side. The left-bound path resumes wall-tracing, while first back-pruning s_2, and, afterwards, s_1. It discovers node s_3 at E_4 and s_4 at I_4 after which it can leave the wall and finds a line-of-sight to s_{stop}. At this stage, the path in question is $\{s_{start}, s_3, s_4, s_{stop}\}$. The path has reached the goal, and attempts to check if the first node pair (s_{start}, s_3) is clear. Line-of-sight to s_3 is blocked by the cell at E_7. Following the right-handed child path, the nodes s_5 and s_6 are discovered. At this stage, the path becomes $\{s_{start}, s_5, s_6, s_3, s_4, s_{stop}\}$. In the previous to last steps, the left-handed child path rediscovers s_1, attempting to clear (s_{start}, s_5), which it reinserts into the path (according to Lemma 3.2). Lastly, when clearing (s_6, s_3), the left-bound child path discovers node s_7 at the E_5 corner. The final solution is, thus, $\{s_{start}, s_1, s_5, s_6, s_7, s_3, s_4, s_{stop}\}$.

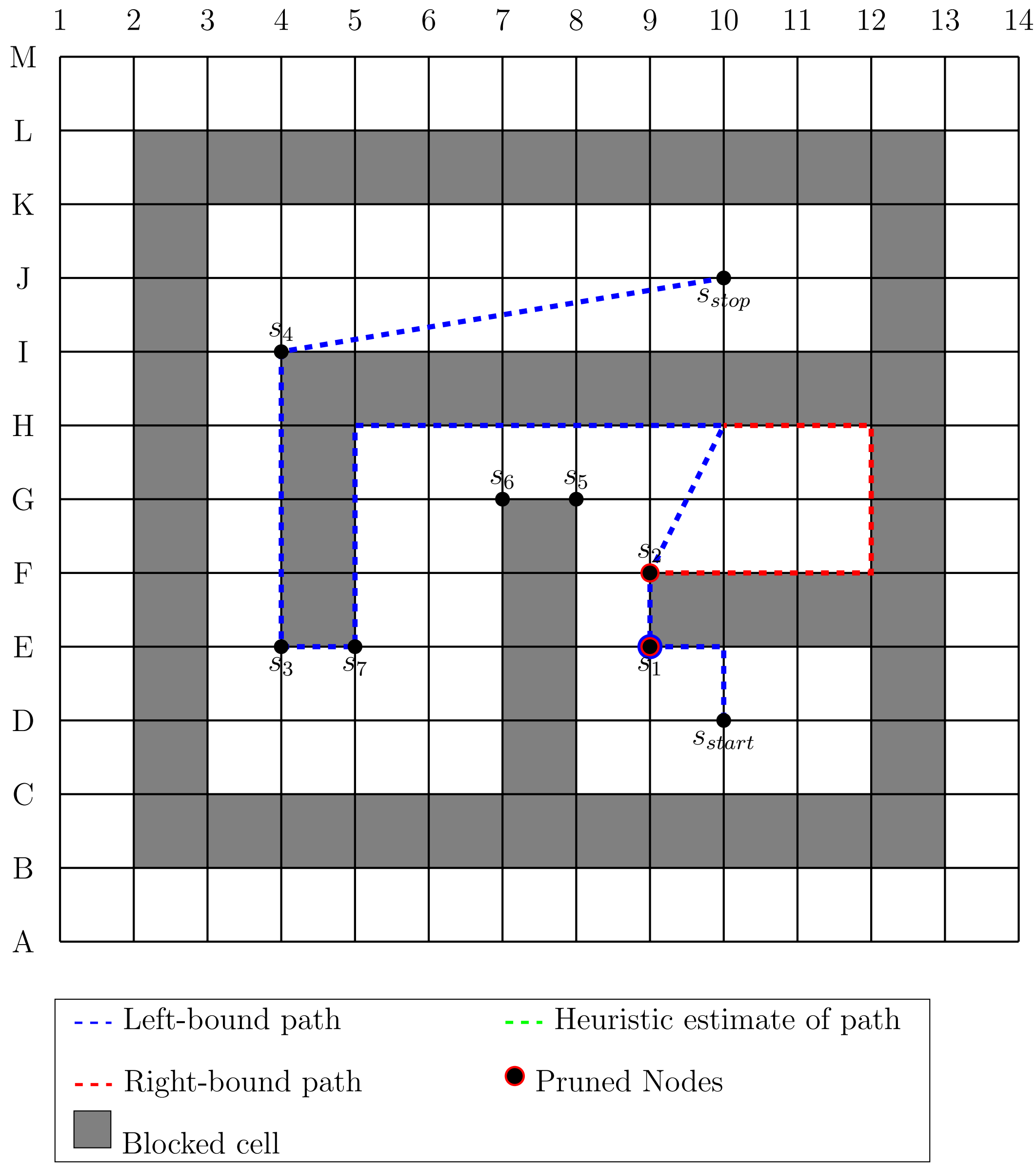


Figure 4.6: Underestimated heuristic length example – left–bound path

If left unchallenged, the left-bound path would reach the goal and the search would terminate with a solution. Unfortunately, the right-bound path does not allow this to happen. Let us explore this problematic scenario and its cause with the aid of Figure 4.7.

Exploring from the intersection at E_{10}, the right-bound path (blue dashed line) splits off from the left and follows the contour from E_{10}, through E_12, C_{12}, C_8 and, after reaching G_8, it identifies the vertex as a corner, adding it to the path as s_1. The next corner, s_2 is found in the next step. The path continues through C_7, C_3 and reaches H_3 at which stage s_2 is pruned from the path. By the time the right-bound path reaches K_8, marked by s_{wall}, both previously identified corner nodes s_1 and s_2 have been pruned from the path. The path does not possess any information regarding the free-space it hasn't explored, and, as such, assumes to have line-of-sight from s_{start} to s_{wall}. From the right-bound path's perspective, the tiles marked by grey cross-hatch are assumed to be hypothetical free-space. Under these assumptions, the next step for s_{wall} would be from K_8 to L_8, which would be identified as a path corner. As such, the hypothetical path $\{s_{start}, L_8, L_2, B_2, B_{13}, E_{13}, s_{stop}\}$ (demarcated by the dotted green line in Figure 4.7) presents the maximum consistent heuristic length estimate. Because the path only computes its heuristic length based on the nodes it contains, it instead greatly underestimates the distance as being $\{s_{start}, s_{wall}, s_{stop}\}$. This has the consequence that the path is greedily prioritized over the left-bound one. The path is eventually terminated, considered to be locked-in, but not before looping over the interior contour a second time. Given more complex topologies that lead to similar situations, numerous locked-in paths could be generated, which would result in a slowing down of the algorithm's performance.

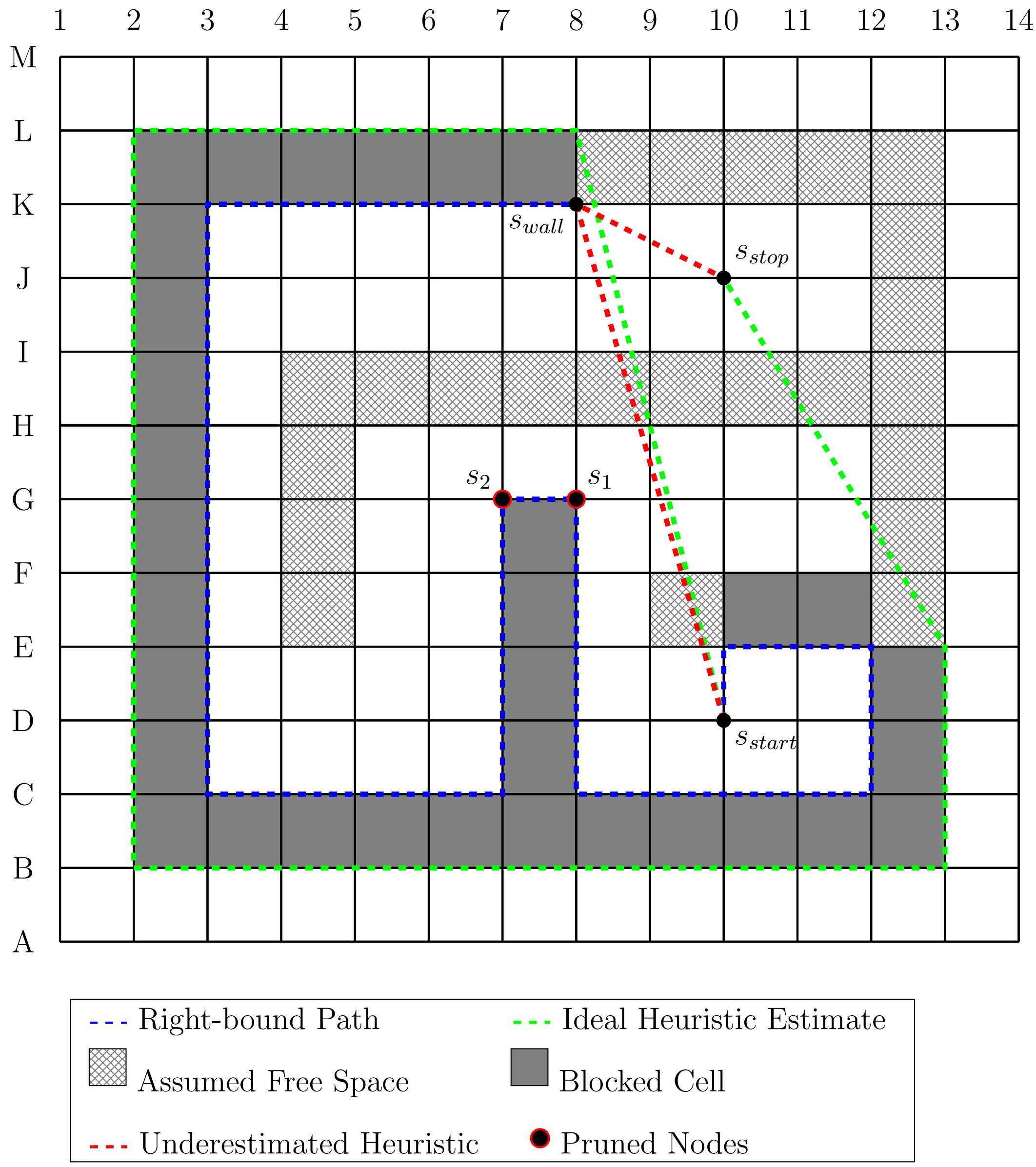


Figure 4.7: Underestimated heuristic length: exploring right-bound path (blue dashed line) from s_{start} to s_{wall}; corner nodes s_1 and s_2 are pruned before path reaches s_{wall}; in s_{wall} path assumes line-of-sight from s_{start} (hypothetical free-space grey cross-hatch tiles); green dashed line – ideal heuristic length estimate of path from s_{start} to s_{wall} passing by s_{wall}

To the author's knowledge, there exists no information on the number of scenarios from the Moving AI database [70], which can present the aforementioned problematic configuration, and performing such an evaluation may not prove feasible. Because of this limitation, the algorithm's performance may be negatively impacted and addressing the issue could potentially boost the performance of the algorithm. Some possible solutions are considered for discussion, but are left for future research. One such solution would be to keep track of an "inner path" that wraps an obstacle's boundary at inner corners (rather than outer corners). For the example in Figure 4.7, the inner path is described by $\{E_{12}, C_{12}, C_3, K_3, K_8,$ which would represent a better approximation for the heuristic length of the path. Such a solution has not been implemented as of this writing. This limitation can compromise the completeness of the algorithm, as the heuristic estimation for the path s_{start}, s_{wall} grossly underestimates its cost, which prioritises these types of paths before others. In practice, scenarios like these can result in long search times. A time-out functionality was introduced to mitigate this problem, but the increase in search time can negatively impact the performance metrics of RPF (Chapter 5).

Additional in-depth knowledge of the environment can also benefit the algorithm. If a preprocessing step can, for example, uniquely identify individual obstacles in the search space, the algorithm could potentially avoid some bifurcations that would result in redundant paths when re-encountering the edge of the same obstacle. Additionally, if the start and stop nodes belong to a free-space region that is bound by the interior boundary of an obstacle (e.g. outer walls of a house delimiting the interior), similar to Figure 4.7, then, recognising the obstacle as bounding the free-space can be exploited. It is evident that following the edge would not be conducive to an optimal solution, in much the same way that following the exterior walls of a house from inside the house (without ever exiting) would be a redundant search, in that one would either end up back at the starting point or, at best, would find the goal through a path that is topologically equivalent to the optimal one, but with a far greater cost of travel. In the example illustrated in Figure 4.7 and assuming knowledge of the bounding obstacle, simply intersecting the outer bound at E_{10} is insufficient to identify the direction of travel conducive to the solution. However, identifying the interior corners stretching the convex hull of the boundary (i.e. vertices C_3, C_{12}, K_3, K_{12}, and potentially $C_7, C_8, E_{12}, F_{12}, H_{12}, I_{12}$) would allow for a rapid termination of the east-bound path (when reaching either C_{12} or E_{12}). Such a preprocessing procedure can prove useful even in robotics applications operating in dynamic environments as changes to the topology of the outer boundary are far less likely to happen in realistic environments (e.g. exterior walls don't change often).

4.6 Algorithm properties

4.6.1 Multiple path solutions

If the algorithm is allowed to run longer, beyond having found a shortest path it can generate multiple alternative paths, if such paths exist and given that in its expansion RPF has encountered sufficient obstacles. This behaviour is trivial to implement and the only modification needed for Algorithm 9. The conditional statement at Line 17 presents an extra test, becoming:

if $(path.length > shortestLength) \vee (desired > size(clearedPathQueue))$

where *desired* corresponds to the number of desired paths to search. This unfortunately, is not enough to guarantee that the alternative paths founds are unique (don't overlap), nor that the target number of solutions can be reached.

Examining the paths found by RPF in Figures 4.8 & 4.9, one should remark that the multiple solutions found by the algorithm. In both scenarios, a number of 4 paths were requested. However, for the configuration in Figure 4.8, only 3 paths are identified. While a 4th path can be visually recognised

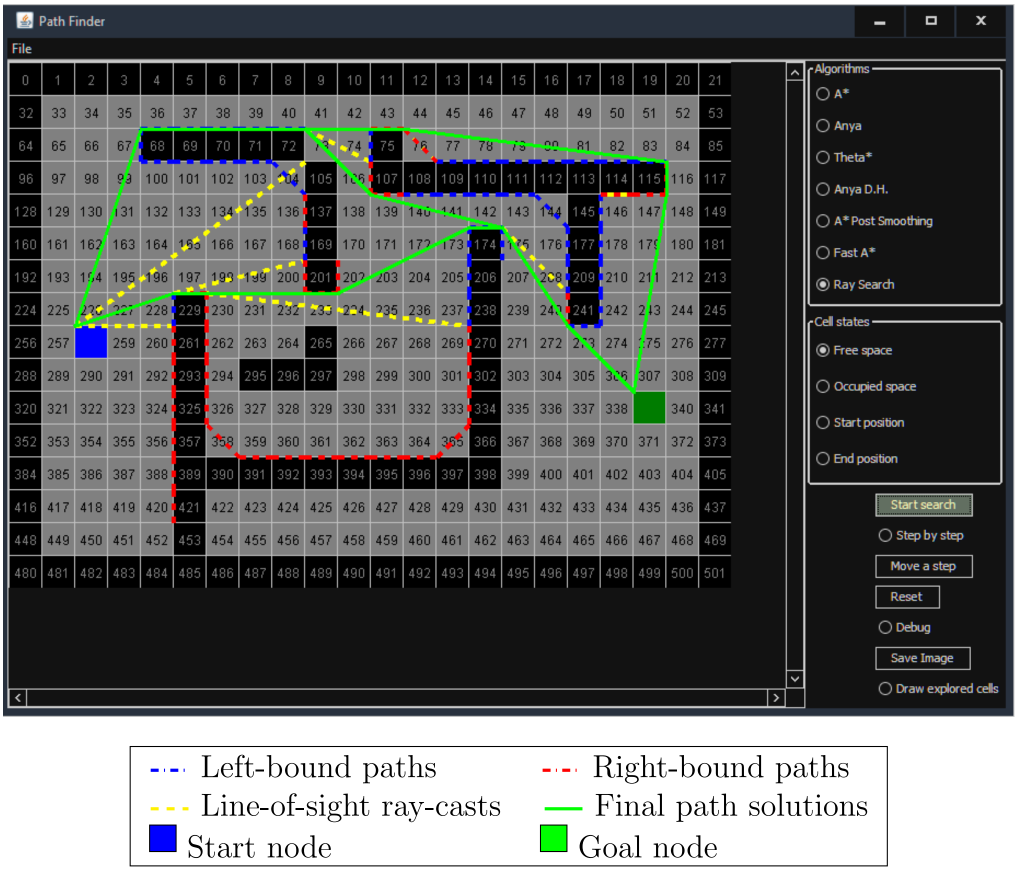


Figure 4.8: Screen-shot of multi-path solutions found by RPF

as passing south-bound of the obstacle $\{295, 296, 297, 265\}$, none of the paths of the algorithm ever encounter the obstacle and thus the path is never explored. This also implies that with the exception of the shortest path solution, the alternative paths are not discovered in ascending order of length, but rather as simple by-products of the root path branching off when it encounters obstacles.

Taking the example in Figure 4.9, a slight modification to the topology in the for of an obstacle at $\{235, 267\}$ allows the algorithm do discover the requested number of paths, while the south-bound path also takes priority as solution when the obstacle is discovered by the intersection at cell $\{265\}$.

Multi-path solutions found by RPF can be used if one wishes to consider alternative routes. This can be desirable to avoid congestion if multiple agents or AI characters in a game navigate together towards a target. A

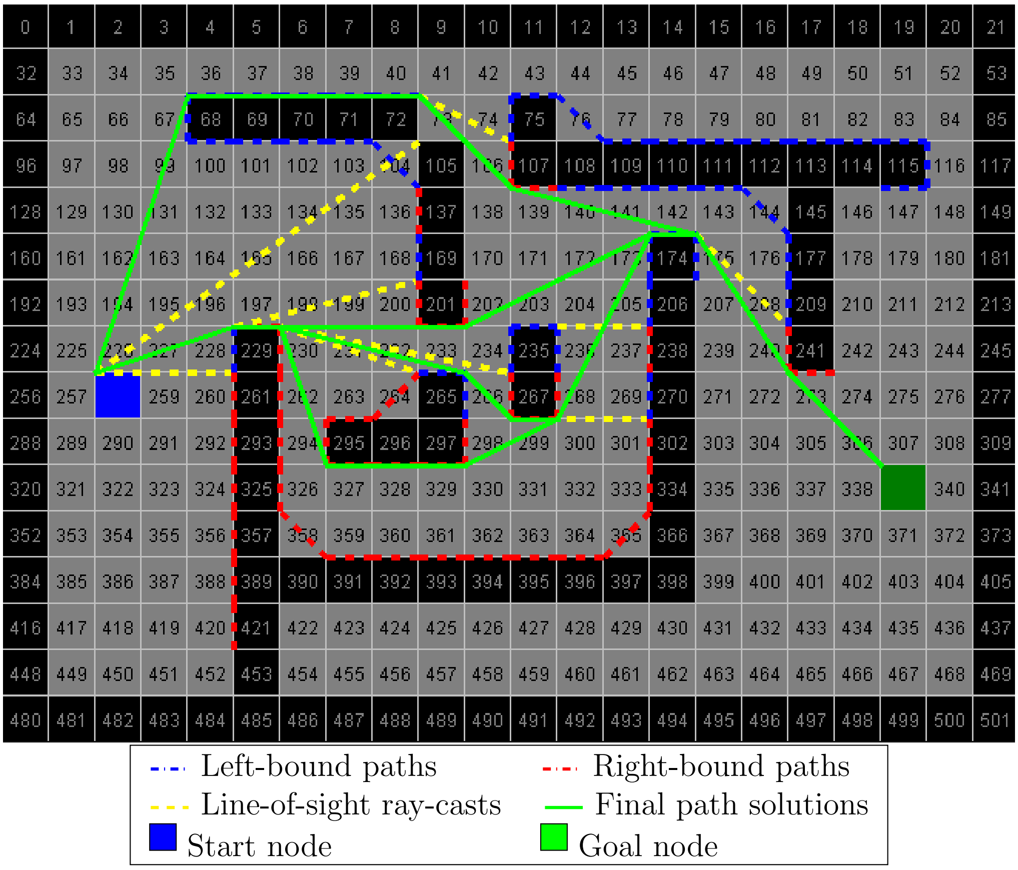


Figure 4.9: Screen-shot of alternate multi-path solutions found by RPF

post-processing stage can analyse paths based on criteria other that shortest length. For example, it can provide longer paths that prioritize fewer heading changes or low steering angles, such as S-Theta* [80]. Inflating the space around path trails can allow for clearing the minimum width and directing agents through different paths based on width or momentum, for example. Multiple paths may also be useful if one desires an algorithm capable of replanning, such as D* [81]. Given A mobile robot that discovers its planned path blocked by dynamic changes in the real-world environment, would be able to choose an alternate with very little replanning. If alternative paths are stored in memory while the robot moves towards a goal, it would require replanning only in switching from the blocked path to the new path, or by updating its map and performing a new search between its current location and the next expected heading change, similarly to clearing a path section – Line 42 in Algorithm 9.

4.6.2 Any-time nature

There are algorithms that provide any-time suboptimal solutions by inflating the heuristic cost. For example, Any-time A*, also referred to as ARA* [82], is a variant of A* that can provide any-time solutions to a path-finding problem even when it is interrupted before completion. It, however, achieves this by executing A* multiple times with decreasing cost functions and using the information from previous searches to minimise the length of the path.

An intriguing property of RPF is that, as a best-first search algorithm, it is an any-time algorithm, as it can provide a suboptimal solution before the algorithm completes. If a solution exists, the algorithm will have arrived at the goal node prior to arriving at a shortest path solution. The initial suboptimal path would consist of the segments described by the line-of-sight checks and the obstacle-adjacent nodes that trace the edge of the obstacle from the intersection node to the tangent node where the path abandons the object boundary. The reader must note that RPF does not explicitly assign parents to nodes (i.e. it only maintains lists of indices and their sidedness). Having nodes pointing back to their parents is characteristic that would be required if one were to retrace a suboptimal solution back to the origin, as is the case for the algorithms in the A* family. Enabling such a behaviour in RPF can represent an alternate avenue of research.

Accounting for and providing suboptimal solutions would incur an overhead that may impact performance. Availability of any-time solutions presents with a trade-off in respect to time and memory, as each path would require a separate structure to keep track of additional nodes. However, committing any-time suboptimal solutions to memory can be desirable in certain circumstances. For example, let us consider a hypothetical scenario in which RPF is implemented on a ground robot as an iterative life-long optimising bug algorithm. The robot would, thus, navigate between a start and goal point a multitude of times, attempting to shorten its path with each trip. If, after having located its target, the robotic agent navigates back to the starting position while at also attempting to optimise its current path,finds itself with a depleted power source and has deviated considerably from the initial path, either by following a very long or unsuccessful route, its higher level deliberative layer may choose to not allow it to attempt any more exploratory behaviour. Instead, it would direct the robot to return via the already known suboptimal path, which has a known length, and retrace it to the starting point within a window of safety.

4.6.3 Unknown 2D Environments

The focus of the research for the RPF algorithm is directed towards the problem of path planning in known 2D environments. However, there is a related class of challenges when considering path-planning in unknown 2D environments [83], [84], [85], [86]. Navigation for a robotic agent in known environments implies planning a path, and afterwards executing it. Inaccurate world models can compromise the validity of a plan (e.g. locked door, barrier) [15]. In unknown environments, the problem domain differs in that the robot must transverse its environment without prior knowledge of a map. Because of this, the navigation strategy is one of exploration, in which the robot only has information about its immediate environment through its sensors, and memory of the search-space it had previously visited.

The Ray Path Finder algorithm is presented as an online, any-angle path-planning method. Drawing a parallel between bug algorithms and RPF, the similarities lie in the behaviours of the latter's paths. However, Ray Path Finder operates in known environments, meaning that it possesses information on the entirety of the map at the start of the problem, and can formulate a solution based on this knowledge, before an agent engages in moving towards its target. A key difference between paths in the RPF algorithm and classical bug algorithms is that paths are allowed to terminate early, if they are deemed infeasible. A bug algorithm on the other hand may not terminate until it reaches its target, which implies that the paths it follows can intersect or loop freely.

While not addressed in this work, some of the strategies presented through the Ray Path Finder algorithm could, however, be adapted into a bug-like or multi-bug variant on unknown environments. The problem posed is of an agent navigating in an unknown terrain with a goal-seeking behaviour that aims to guide them to the target in the shortest amount of time. One can envision how such a behaviour would unfold in a real-world scenario. Let us consider a robotic agent placed in an environment for which it has no prior information. While navigating in a straight line towards its target, when the line of sight is broken, the robots reverts back to tracing object boundaries, pruning and inserting new nodes into its path that it keeps in memory. While wall following, it could infer that, by tracing the wall in a specific direction, it is moving too far away from the target, and it could decide to return to the point of intersection and trace the obstacle boundary by moving in the opposite direction, and repeat this strategy based on a heuristic estimation. The implemented algorithm has not been optimised for best performance. Due in part to the complexity of the algorithm relative to other algorithms, such a task can prove challenging. Potential future improvements (of the principles as well as of the implementation could further reduce the search-space and improve on run-time. For example, if the algorithm posses prior-knowledge about the environment, certain subroutines can be bypassed, for instance, in the case of an environment that only contains convex objects, checking for path redundancies becomes unnecessary (i.e. any two points on the object boundary would not go outside the object, thus a tangent line through any point on that boundary would not intersect the same object after leaving the object's edge). An equivalent bug variant would not employ the methodologies presented for wall following and boundary departure, but rather rely on sensor information. The behaviours of RPF could potentially be imitated by means of a compass and odometry.

In the case of non-convex objects that the robot may intersect multiple times while leaving the boundary, a strategy can be envisioned which takes note of redundant paths presented in Section 3.7. If, after having left an obstacle's boundary, it intersects the same obstacle again and begins tracing the wall in the opposite direction of the previous search, the robot reaches its previous point of departure from the wall boundary, it can simply revert back to moving in a straight line towards the target. It, thus, intersects the obstacle as it did previously, but traces its wall in the other direction and avoids retracing the redundant path.

Based on RPF's shortening of a path only after it has reached the goal, a similar behaviour would allow a robot to optimise its path on successive instance of moving between its start and goal. With a free-space assumption strategy, the robot can attempt to find a straight line-of-sight between its successive points of departure from obstacles' boundaries. If no line-of-sight exists, the robot may attempt to explore on the side which keeps it close to the wall boundary, as a free space is guaranteed given that the robot had previously discovered its lower bound while originally moving towards the goal. Other strategies on the RPF algorithm for navigating unknown 2D environments could be thought of, and may make useful additions to the class of Bug algorithms.

4.7 Chapter Summary

This chapter introduced the novel path-planning algorithm developed in this work - Ray Path Finder.

RPF, comes under the class of any–angle path–finding algorithms, applied within a 2D environment, in which nodes can have as a parent any other node with a direct line of sight. RPF minimizes the amount of collision check computations. It looks to connect the least amount of points belonging to a path without breaking the line of sight between them.

For certain topologies however, an additional step is necessary to prevent the path from mistakenly leaving the edge. RPF is able to provide sub-optimal solutions before the algorithm completes, also RPF is able to terminate early, not having to backtrack steps. If the algorithm is allowed to run longer, beyond having found the shortest path RPF is able to generate multiple alternative paths.

An important issue not addressed by RPF currently, is the underestimation of the path heuristic lengths for certain topologies, and thus impacts performance of RPF.

Chapter 5

Experimental results

This chapter describes the experimental setup of this work, and the results which were subsequently obtained. The developed interface, and the databases which were used are discussed, and analysis of the results acquired are explored in depth.

5.1 Interface

5.1.1 Graphical user interface

The path planning algorithms has been implemented using the Java Programming Language (Java 8). Often, path-planning algorithms can be difficult to debug, or even implement. The behaviour of an algorithm can also be challenging to describe or visualise. For these reasons, a graphical user interface (GUI) was constructed to allow for easy development, integration and testing of path-planning algorithms. The implementation provides a graphical user interface (GUI) developed using Swing and the JavaFx platform. The GUI is used for testing and development and for the visual inspection of solutions and for manual manipulation of grids through

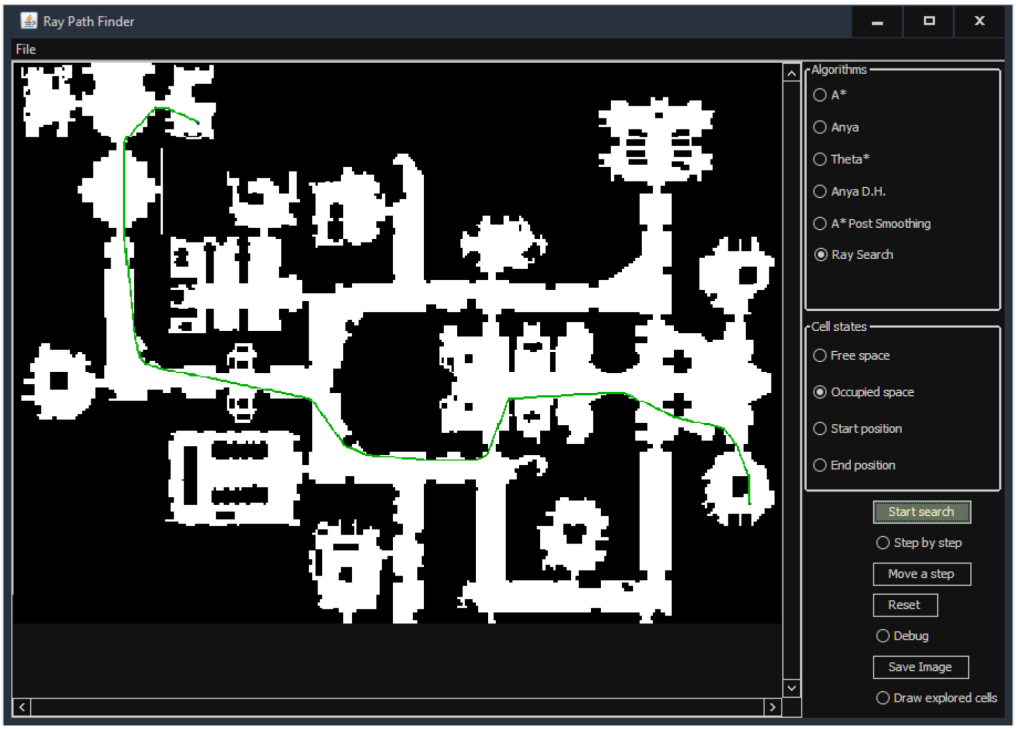


Figure 5.1: Screen-capture of path-planning visualisation tool

user input. A screen-capture of the user interface is illustrated in Figure 5.1.

The application allows for constructing new maps by direct user input (i.e. clicking, dragging), choices of path-planning algorithms can be selected for evaluation, as well as visualising solutions. The GUI integrates with the MovingAI database and allows for saving new maps that follow the same map data format described in [70].

5.1.2 Synchronisation

The application is implemented on multiple threads of execution, with various features such as a graphical user interface (GUI), file input/output operations for loading or saving maps. Multi-threaded applications can suffer from thread interferences as one thread can randomly pause the execution of another, modify resources that are shared between them, etc. This can result in erroneous timing results and/or memory consistency errors. As such care must be taken so that other threads do not interfere with the thread responsible for executing the search, so that algorithms only get executed in isolation without the possibility of the search being interrupted from other threads.

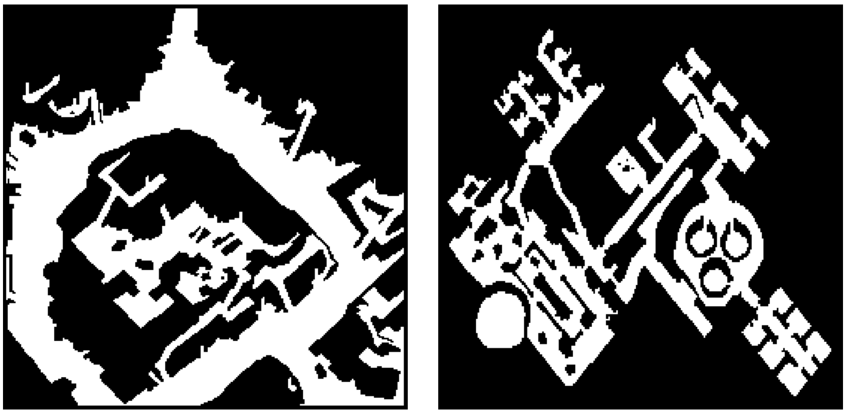
To ensure that the path-planning algorithms run reliably without externalities affecting their performance metrics, a separate worker thread is allocated the sole responsibility for running the search within a synchronised block of execution which guarantees that when the thread executes the synchronized function, all other threads which could in any way interfere with the search block, suspend execution until the worker thread had performed the search.

5.2 Database

Grid-based maps have been used as test-beds for path-planning by a wide variety of researchers. Furthermore, the paradigm is widespread, having been adopted in countless video-game developments, or in the form of occupancy grid maps in the field of robotics, including the ROS platform [87].

The Moving AI lab [88] is run by Prof. Nathan Sturtevant at the University of Denver, as is publicly available for download. The database is one of the most popular databases of 2D grid-maps in the path-planning literature. The database is often used by state-of-the-art algorithms for evaluation [55],[50], [51], [89], [83], [25], [57]. It provides a good selection of game maps and maze maps. Each map is provided with a large set of scenarios that provide as input map dimensions, the coordinates of the start and goal nodes along with the optimal length that assumes $\sqrt(2)$ diagonal cost and does not allow agents to cut corners through walls. For these reasons, the MovingAI map database [88] has been used in this work for evaluating the performance of the proposed methodology and for comparison against other algorithms in the literature.

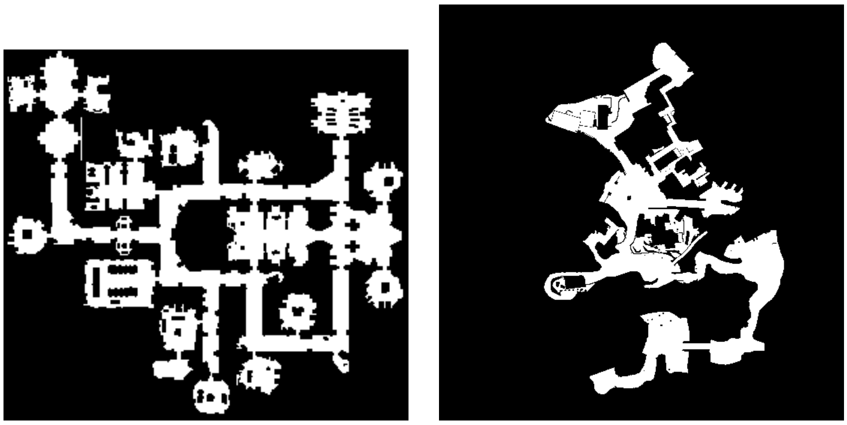
Map examples from the database can be seen in Figures 5.2, 5.3, 5.5, 5.4, 5.6. Baldur's Gate is a set of 75 maps taken from BioWare's video-game Baldur's Gate II: Shadows of Amn, with a total of 93160 scenarios. The scaled version of (512 x 512) is used for evaluation. Maps from this game generally present rooms with large open-space areas.



(a) AR0400SR (b) AR0406SR

Figure 5.2: Sample maps from Baldur's Gate II

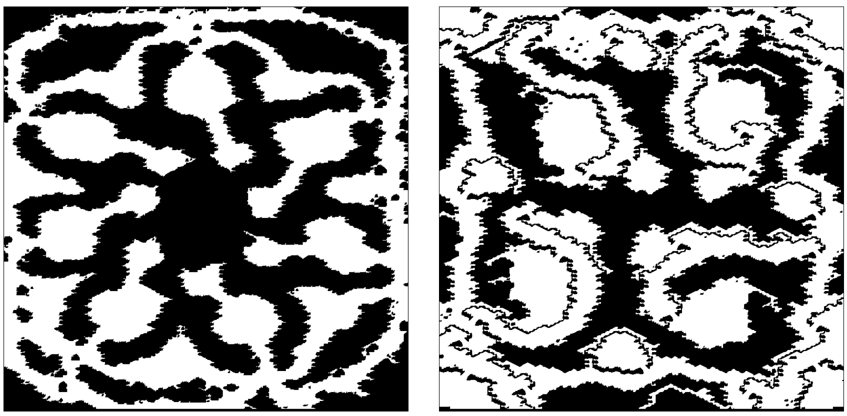
The largest of the four games in the database, BioWare's role-playing game Dragon Age: Origins consists of 156 maps ranging in size from 30 x 21 to 1104 x 1260, with a total of 159465 scenarios. Maps from the game are generally large is size, with long connected "corridor-like" regions and intricate topologies.



(a) hrt201d (b) lak100d

Figure 5.3: Sample maps from Dragon Age: Origins

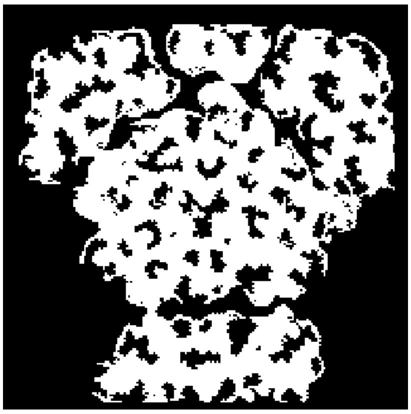
The popular military sci-fi game from Blizzard Entertainment, Starcraft, has 37 map with a total of 97650 scenarios. Starcraft maps are generally large with dimensions of 512x512 and above, and present with large regions of connected free-space.



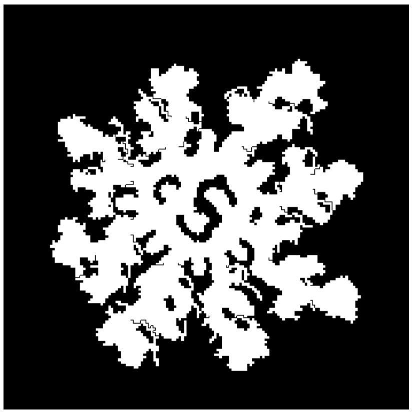
(a) BlackLotus (b) Inferno

Figure 5.4: Sample maps from Startcraft

Maps from the popular video-game franchise Warcraft III total 36 with a scenario number of 45101. Most similar to Starcraft in terms of topology.



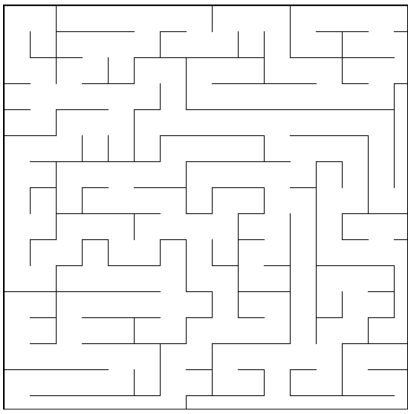
(a) riverrun



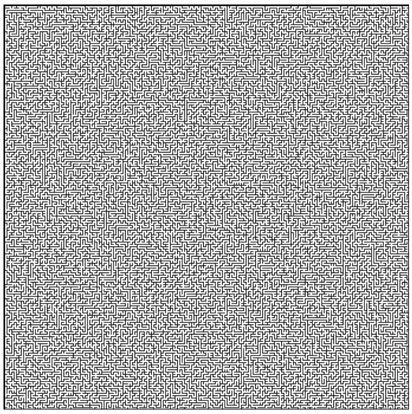
(b) thecrucible

Figure 5.5: Sample maps from Warcraft III

The Mazes database offers a set of closed mazes with fixed corridor widths ranging from 32 cells wide to 2 cells wide.



(a) maze512-32-1



(b) maze512-2-0

Figure 5.6: Sample maps from Mazes

The experimental evaluations use 391 maps, with a cumulative number of scenarios of 562170. Additionally, the maps are also used at double the original scale presented in the database. The numerical breakdown of the database is presented in Table 5.1.

Table 5.1: Map Database: Games & Scenarios

	Mazes 32W	Baldur's Gate II	Dragon Age: Origins	Starcraft	Warcraft	Total
Maps	10	75	156	37	36	314
Scenarios	60670	93160	159465	97650	45101	456046
Inputs Blocked	0	0	0	0	2446	2446
Double Corners	0	0	80	3	17	100
Invalid Scenarios	0	0	80	3	2463	2546

For simplicity, the algorithms imposes two restrictions on the global start and global goal vertices, namely that their corresponding grid-cell (vertex at upper left corner of the observed cell) must be unoccupied (consistent with the map database used for evaluation) and, that neither vertex can be a double-corner (scenarios from the database that don't respect the restriction are ignored for all evaluated algorithms). Each scenario from the Moving AI database provides metrics and restrict corner-cutting (i.e. crossing between two diagonally adjacent blocked grid-cells). For this reason, cutting corners is disallowed in all evaluated algorithms.

The rejected scenarios total 2546 in number, for a total number of **453500** valid scenarios.

5.3 Experimental results

5.3.1 Introduction

The current section presents results for metrics collected from a number of implemented algorithms. For comparison with other methodologies in the literature, five algorithms have been implemented or adapted for data collection. All the algorithms presented perform single source path-planning on 2D uniform-cost octile grids. The search agent is treated as a point object with no holonomic constrains, meaning that it can travel in any direction, unrestricted. Each algorithm performs a total number of 453500 searches, with the scenarios spread over 314 maps from the MovingAI database (presented in Table 5.1 of Section 5.2).

The first two algorithms are the standard benchmarking algorithm A* and its post-processing variant, A* with Post-Smoothing. Both algorithms use an octile heuristic for distance estimation. D_{Octile} (Equation 1.9) is a consistent heuristic for 2D grids with 8-neighbourhood connectivity, in which orthogonal movements have a step cost of 1 and diagonal steps have a cost of $\sqrt{2}$. A second any-angle algorithm after A* with Post-Smoothing, Theta*, was selected for implementation due to its near-optimal path solutions. Results from literature ([56], [55], [4]) suggest that Theta* provides a good trade-off between runtime and path length, by finding near-optimal paths that are shorter than the ones found by A*, and which look more realistic (i.e. have fewer heading changes outside of those around obstacle corners) while incurring a slight increased runtime relative to A*. The Anya algorithm is a recent addition to the any-angle path-planning family of algorithms. It is also the first to be optimal, finding true shor. Two implementations of the *Anya* algorithm have been integrated into our experimental setup. The first *Anya* algorithm version is implemented by Oh et al. [27] and available at [90]. The second version of Anya is implemented by its original authors, Harabor et al. and described in [24], [25]. The source-code is made public at [60]. We label this version as $Anya(Harabor\ et\ al.)$ to distinguish it from the version introduced by Oh et al. [27]. As grid-based any-angle algorithms, Theta* and Anya search over grid nodes but are not bound to move on grid edges. As such, both Theta* and Anya use the Euclidean distance ($D_{Euclidean}$ – Equation 1.10) as their heuristic estimation function. Finally, the novel path-planning methodology proposed in this work is implemented in two variations: the original implementation of RPF (labelled RPF On Cells) which operates on cell centres (using the grid-map tiles as nodes as opposed), and the latest implementation of the algorithm which operates on vertices. Both versions of the algorithm are any-angle path-planning algorithms and use Euclidean distance ($D_{Euclidean}$ – Equation 1.10) as the heuristic for distance estimation.

The experiments presented are performed on a 2.9GHz Intel Core i7 machine with 8GB of RAM running Windows 10.

A number of different metrics are considered for the evaluation of our algorithms:

- path length - the sum of the Euclidean distance for each pair of nodes belonging to a path that connects the start and goal nodes;
- run-time - the elapsed time between initiating the search for a path and reaching a solution;
- nodes expanded - the number of nodes within the search-space that the path-planning algorithm visits during a search;
- heading changes - number of nodes from start to goal in which the path changes its direction;
- memory usage - the amount of RAM memory that a path planning requires to perform the search;

The following sections describe the metrics and present the experimental results of the selected path planning algorithms. The experiments have been conducted on the popular maps from the MovingAI database (see Table 5.1 from Section 5.2). The experimental results informs us about the performance of our novel path-planning algorithm, Ray Path Finder, against the current state-of-the-art algorithms. A discussion on the implications of the results vis-a-vis the competitiveness and applicability potential of the algorithm follows in Section 5.4.

5.3.2 Path length

Solving the shortest-path problem has been a main focus of much of the research in the path-planning field. Path length is a key metric in establishing an algorithm's dominance. Minimising this metric implies converging towards the optimal solution. For a game character, shorter paths improves the perceived intelligence of the character. For a robotic agent, finding a shorter path has many benefits. Among them is the reduction of energy expenditure and increased battery life, as the robot would travel shorter distances. Additionally, it would limit the wear and tear of the platform.

Table 5.2 summarises the average path length data collected over five maps from the MovingAI lab database by the aforementioned algorithms.

Table 5.2: Average Path-length

	Baldurs Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes (32W)**
A*	252.97	391.44	552.68	256.58	1107.30
A*PostSmoothing	243.61	380.49	532.97	246.97	1076.01
Anya (Oh et al.)	241.18	373.82	525.11	242.15	1071.86
Anya (Harabor et al.)	241.18	375.14	525.12	244.11	1071.86
Ray Path Finder	241.20	375.50	525.74	244.13	1071.86
Theta*	241.27	375.43	525.47	244.24	1072.00

As expected, A* consistently has the longest paths of the six, because of its constraint to move along grid-edges. A* with Post-Smoothing, by its simple smoothing technique, improves on the base path length average of A* across all the games tested. Its biggest improvement is on the maze maps (last column in Table 5.2), reducing the length by 2.83%.

This is expected given the simple topologies of the maze maps, which only have right corner walls and no other obstacles.

The four remaining algorithms, namely the two Anya implementations, Theta* and the proposed algorithm, Ray Path Finder, performed similarly well to each other, and further improve on the path lengths identified by A* with Post-Smoothing.

If reducing path length is the main focus, simple smoothing solutions for paths are a useful tool for game characters, because of the simplicity of implementing such a solution. It offers a reduction in path length and improves on the perceived intelligence of a game character with minimal effort.

5.3.3 Run-time

Along with the path length, the run-time performance of a path-planning algorithm is one of the most important aspects of the shortest-path problem. A reason for this is that, general, they are antagonist metrics, as improving one degrades the other. Online algorithms such as Theta* and A* with Post-Smoothing have this trade-off because they require additional computation, and thus more time, to improve on the solution of their inherited A* base algorithm. The run-time overhead that the extra computation carries will not allow these algorithms to outperform A* vis-a-vis this metric. Ideally, we desire online algorithms to be as fast as possible while limiting trade-offs.

A summary of the run-time performance of the algorithms can be observed in Table 5.3.

Table 5.3: Average Run-time (ms)

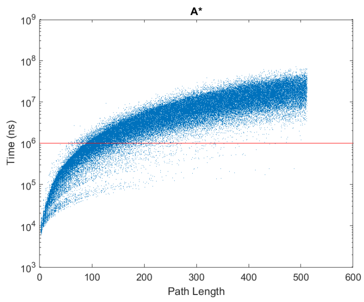
	Baldurs Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes (32W)**
A*	7.11	8.43	45.58	8.98	72.80
A*PostSmoothing	38.66	15.45	203.08	57.61	165.63
Anya (Oh et al.)	1.99	12.29	13.34	2.44	2.91
Anya (Harabor et al.)	0.16	1.01	1.54	0.22	0.38
Ray Path Finder	0.24	15.52	4.98	0.31	2.21
Theta*	26.06	35.13	225.17	31.63	505.39

Looking at A* Post-Smoothing and Theta* in terms of average run-time, the two algorithms alternate in outperforming each other, with Theta* being faster on maps from Baldur's Gate II and Warcraft, and A* Post-Smoothing taking the lead for the remaining games. However, both algorithms are an order of magnitude slower than A*. This is to be expected given the overhead of performing additional line-of-sight checks.

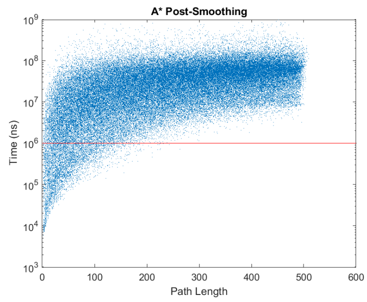
Both Ray Path Finder and Anya (Oh et al.) outperform A*, by an order of magnitude or more (e.g. RPF on Baldur's Gate II maps) with the exception of Dragon's Age: Origins, where A* is faster. The Anya algorithm by Harabor et al. is the fastest of the six algorithms across all maps. It is interesting to note that their implementation is an order of magnitude faster than that of Oh et al., which seems to slow down on the larger maps from the Starcraft and Dragon's Age: Origins games. Given that the principle behind the Anya algorithm is identical, the performance difference comes down to the level of implementation. This serves as a good example on how a good implementation can drastically improve on the run-time of an algorithm.

Overall, Ray Path Finder is the second fastest algorithm after Anya (Harabor et al.) with the exception of Dragon Age: Origins, where RPF's performance degrades, making it the second slowest after Theta*. Ray Path Finder is similar in performance to Anya (Oh et al.), but is surpassed in all instances by Anya (Harabor et al.). However, RPF has the same order of magnitude as Anya (Harabor et al.), with the exception of Dragon Age: Origins and Mazes (32W), where RPF is an order of magnitude slower. Given this discrepancy in run-time, it is very likely that the degradation in performance is due to the topology of the maps in Dragon Age: Origins. Maps from the aforementioned game are large in size, with long and narrow connected "corridor-like" regions, in which "overhead" paths that challenge RPF are much more likely to occur.

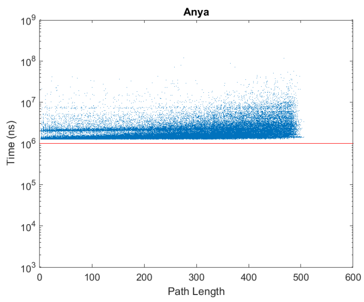
The following figures present the correlation between path length and run-time of the six tested algorithms across the five games. distributions of the data-points for each algorithm across The x axis represents the path length, expressed in cell units of a 2D grid-map. The logarithmic y axis represents time, expressed in nanoseconds. A red horizontal line is drawn at the 1 millisecond mark (10^6 nanoseconds) as a reference point.



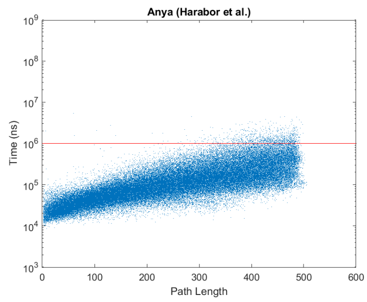
(a)



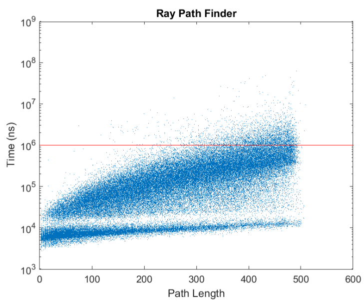
(b)



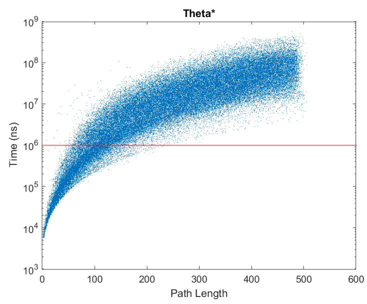
(c)



(d)



(e)



(f)

Figure 5.7: Baldur's Gate: 1x Scale. (Path Length vs. Time (ns))

Figure 5.7 illustrates the performance of the six algorithms on the game maps from Baldur's Gate II. Both A* and A* with Post-Smoothing (Figures 5.7a & 5.7b) exhibit a strong correlation between a path's length and the algorithm's run-time. This is due to the nature of the search performed by the A* family, in which three costs need to be calculated in relation to each expanded node. A* with Post-Smoothing presents an additional overhead of performing line of sight checks between the nodes of the solution. As such, it exhibits earlier signs of run-time degradation, with more of its run-time profile being distributed towards the 10^8 ns mark even for paths shorter than 100 units (Figure 5.7b). Theta* presents a similar profile to A*, but delayed in time. Similar to A* with Post-Smoothing, Theta* also has a wider distribution across the time axis but it is only evident for longer paths. Theta* is 177% faster than A* with Post-Smoothing on Baldur's Gate II maps, but 266% slower than A*.

A notable distinction among the algorithms is the variant of Anya by Oh et al. (Figure 5.7c), with a 72% improvement in run-time compared to A*. While the data-points are exclusively situated above the 10^6 ns mark, making it generally slower than the other Anya implementation (i.e. Harabor et al.), the profile is very compact and mostly concentrated below the 10^7 ns mark, which allows for a consistent, predictable performance on maps of the type found in Baldur's Gate II.

Ray Path Finder (Figure 5.7e) presents a similar profile to Anya (Harabor et al.) (Figure 5.7d), with the exception of the lower part of the distribution, which is more strongly associated with paths that have line-of-sight between start and stop nodes. Because of this, the algorithm only needs to perform a simple line-of-sight check that results in a straight-line solution. Figure 5.7e suggests that for such scenarios, Ray Path Finder outperforms Anya (Harabor et al.). and indeed all the other algorithms. Anya (Harabor et al.) concentrates more of its solutions in a narrow band between 10^4 and 10^6 ns while RPF has more outliers above the 10^6 ns mark, with a more sparse distribution. This indicates that for some scenarios, the algorithm does perform better than Anya (Harabor et al.) but for others it performs worse.

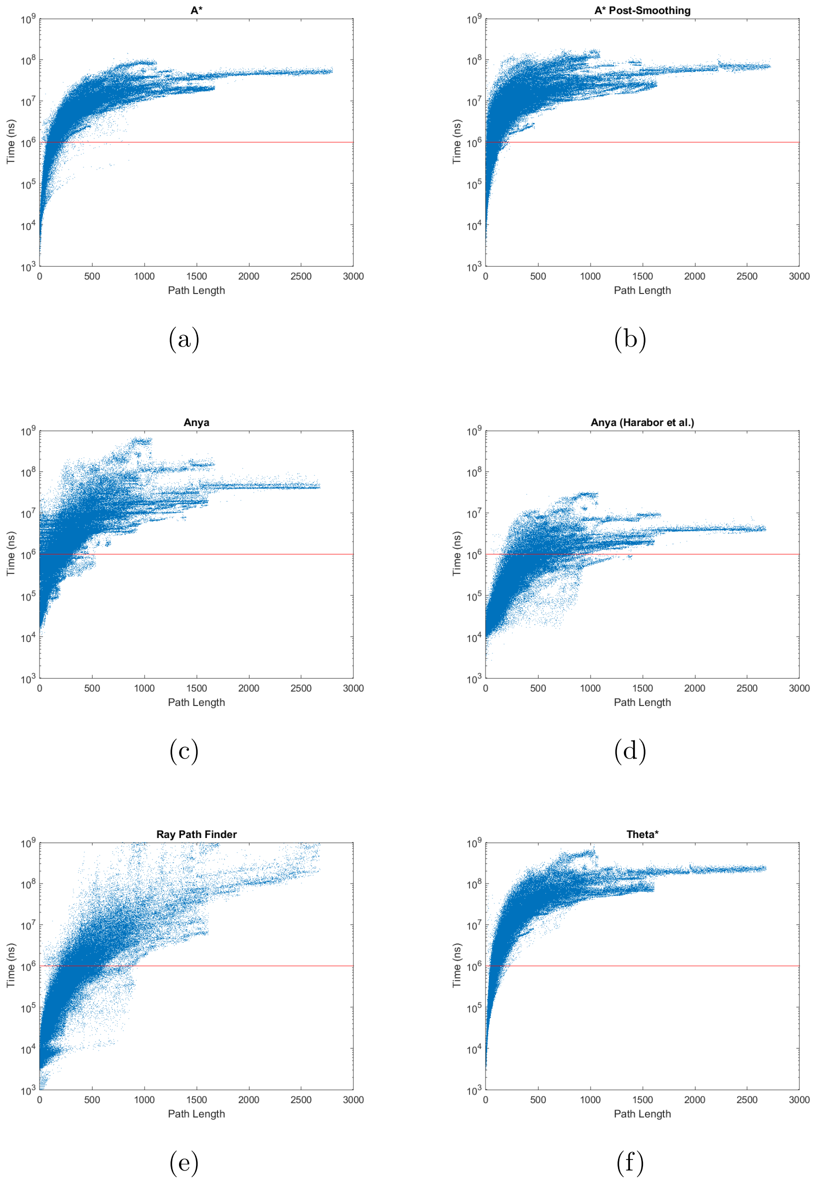


Figure 5.8: Dragon Age Origins: 1x Scale. (Path Length vs. Time (ns))

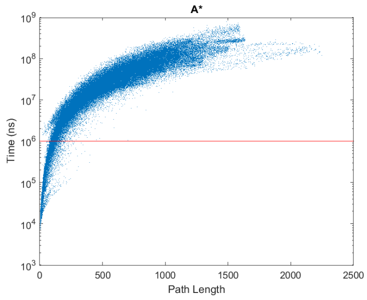
Examining the algorithms' behaviour on maps from Dragon's Age: Origins in Figure 5.8, it is evident that this map dataset is more challenging for all six algorithms, resulting in longer run-times. Of the three profiles among

A*, A* Post-Smoothing and Theta*, Theta* presents with the highest run-time overall, while A* Post-Smoothing presents a very sharp increase in run-time even for short paths. Anya (Oh et al.)'s profile (Figure 5.8c presents with a similar sharp increase in run-time for short paths. This is likely due to Dragon's Age: Origins having maps with more complex and cluttered environments.

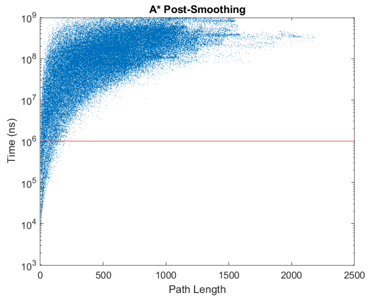
Anya (Harabor et al.) (Figure 5.8d) proves the most adept at solving the maps while maintaining a good run-time average around the 10^6 ns mark.

Ray Path Finder (Figure 5.8e) experiences large variability resulting in a sparse profile and some considerable degradation in run-time – one order of magnitude slower than Anya (Harabor et al.), with numerous outliers above 10^7 ns. It does, however, share a similar profile with Anya (Harabor et al.) for data-points below the 10^6 ns mark. Some maps from Dragon's Age: Origins present with longer than average paths (i.e. over 1000 units, with the average path length of approximately 375 units – see Table 5.2). Most algorithms show little increase in run-time in response to the longer path lengths (over 1000) provided by the scenarios. However, RPF struggles with these scenarios. One reason for RPF's poor performance on Dragon's Age: Origins may have to do with the specific topologies of the game's maps, some of which are large in size and have long and narrow corridors. In such instances, "overhead" paths are more likely to occur. These present a challenge for RPF as it underestimates the heuristic lengths of these paths, which leads to the algorithm performing extensive expansions.

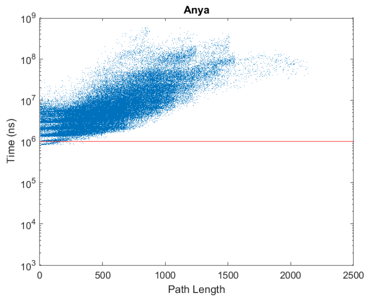
Figure 5.9 examines the algorithms on the Starcraft game maps. A* and Theta* present similar profiles, with Theta* suffering a steeper degradation in run-time for longer paths. A* with Post-Smoothing exhibits a sharp increase in run-time for all path-lengths.



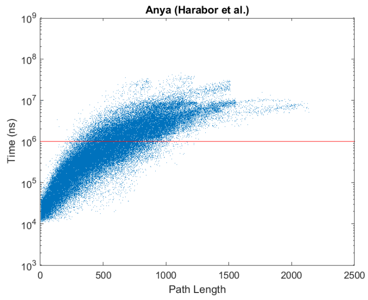
(a)



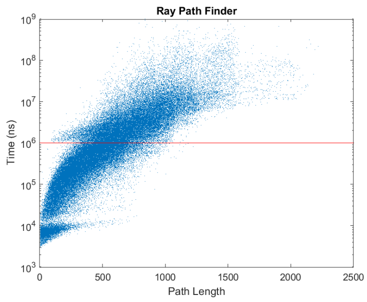
(b)



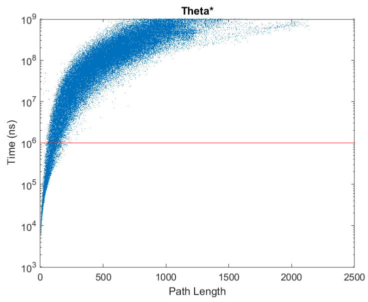
(c)



(d)



(e)



(f)

Figure 5.9: Starcraft: 1x Scale. (Path Length vs. Time (ns))

Anya (Oh et al.) has poor performance on Starcraft when compared to both Anya (Harabor et al.) and Ray Path Finder. Anya (Harabor et al.) and Ray Path Finder present with similar profiles, and RPF suffers a sharper increase in run-time for scenarios above the 10^6 ns mark, mostly for path-lengths above 250 units.

Figure 5.10 illustrates the performance of the algorithms on the popular game Warcraft. The profiles of the algorithms on Warcraft are reminiscent of the ones for Baldur's Gate II. Theta* presents a similar profile to A* with a longer run-time. A* with Post-Smoothing (Figure 5.10b) has a wider distribution across the time axis than A* with short paths having a longer run-time, even when compared to Theta* (Figure 5.10f).

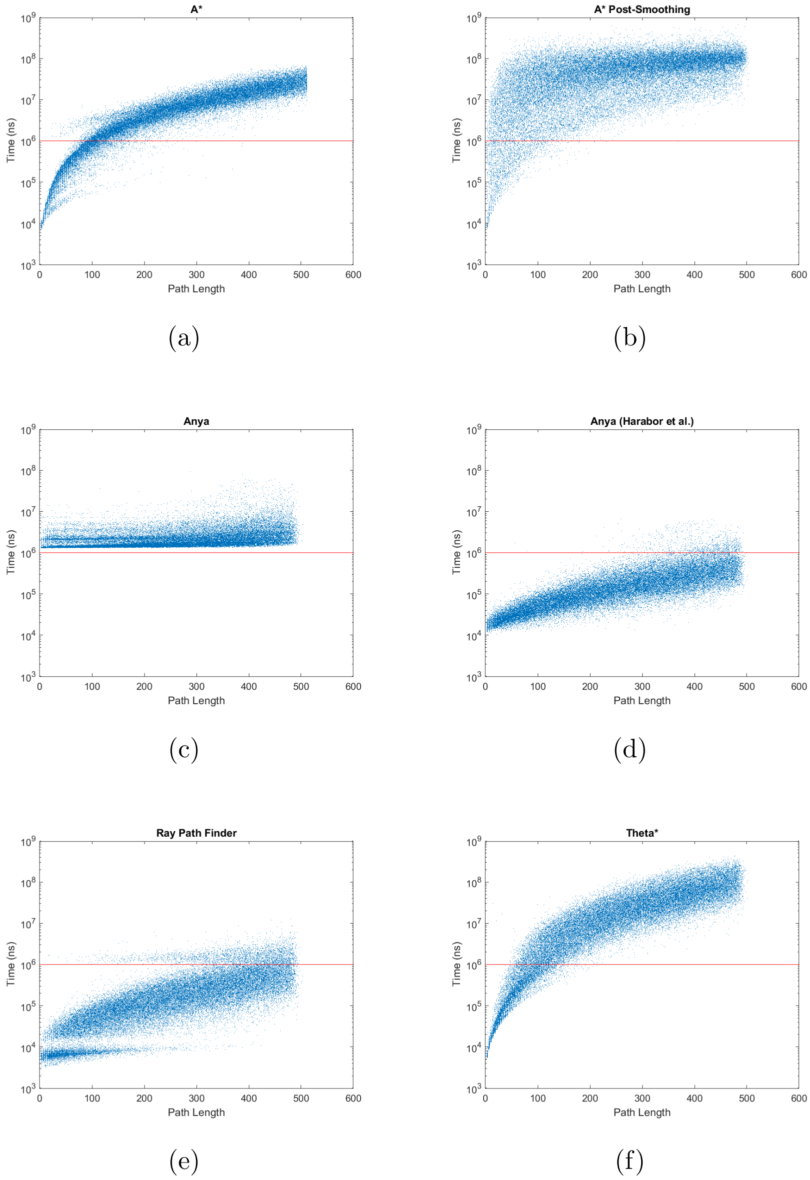


Figure 5.10: Warcraft: 1x Scale. (Path Length vs. Time (ns))

Anya (Oh et al.) (Figure 5.10c), while having a longer running time than both Anya (Harabor et al.) and Ray Path Finder, has a consistent run-time window between 10^6 and 10^7 ns, similar to the behaviour on maps from Baldur's Gate II. Ray Path Finder (Figure 5.10e) presents a similar profile to Anya (Harabor et al.) (Figure 5.10d). Similar to its profile on Baldur's Gate II, Anya (Harabor et al.) concentrates more of its solutions in a narrow cluster. Ray Path Finder, on the other hand, has a more sparse distribution along the time axis, implying that it has a higher variability in performance, with some searches running faster than Anya (Harabor et al.) and others running slower.

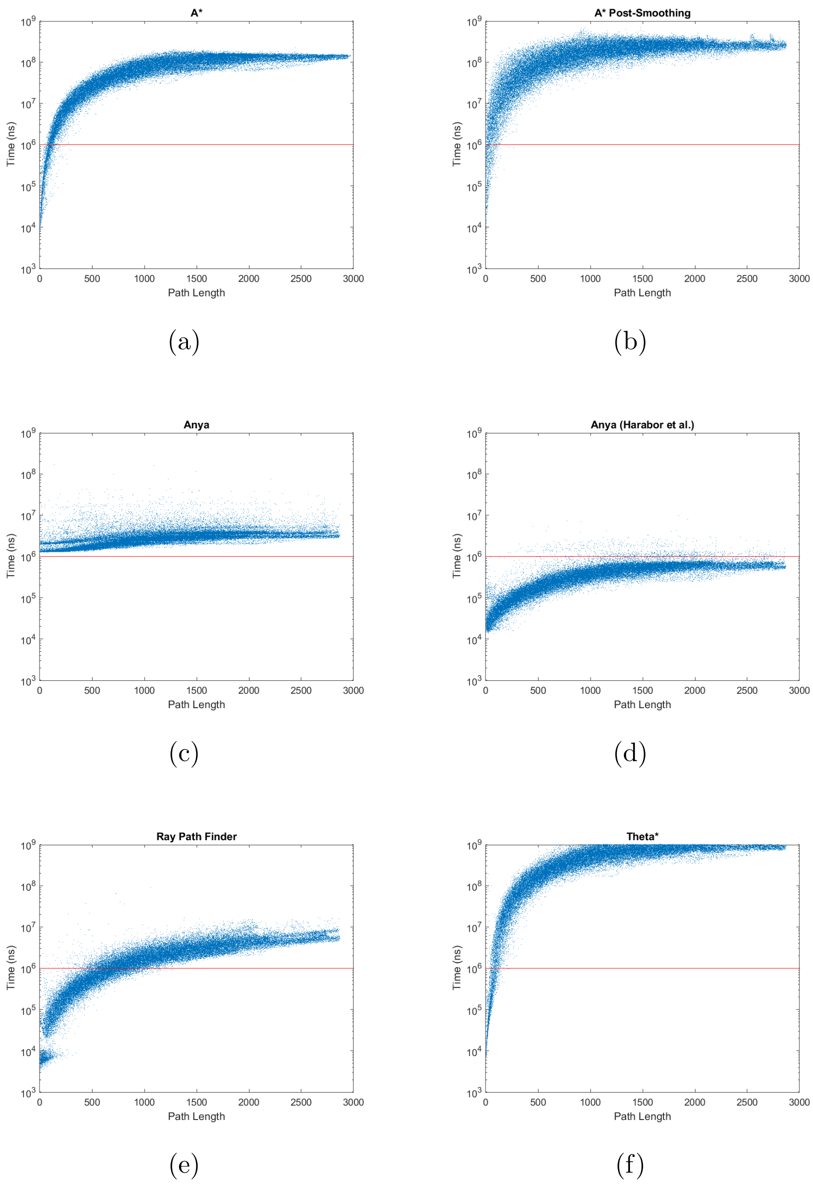


Figure 5.11: Mazes (32W): 1x Scale. (Path Length vs. Time (ns))

Figure 5.11 illustrates the performance of the algorithms on maze maps with 32-unit wide corridors. A*, A* with Post-Smoothing and Theta* exhibit a similar profile, with Theta* being the slowest of the three. All three algorithms are outperformed by Ray Path Finder and Anya (Harabor et al.), and by Anya (Oh et al.) for longer paths (Figure 5.11c). While slower than Anya (Harabor et al.) and Ray Path Finder, Anya (Oh et al.) presents an interesting profile, in that path length has very little influence on its run-time. As this is not the case for maps from Dragon's Age: Origins and Starcraft, Anya (Oh et al.)'s profile suggests that the general topology of the maps plays a role in the algorithm's response. With the exception of scenarios with line-of-sight solutions, Ray Path Finder (Figure 5.11e) is outperformed by Anya (Harabor et al.) (Figure 5.11d). RPF exhibits a shallow slowdown in run-time when compared to Anya (Harabor et al.), which mostly maintains its run-time below the 10^6 ns mark. Given that the maze maps have a fixed size of 512 by 512 cells, with wide corridors at right angles, the Anya algorithms would readily scan the majority of the map (as exemplified in Figure 5.12), while searching for long paths within the topology.

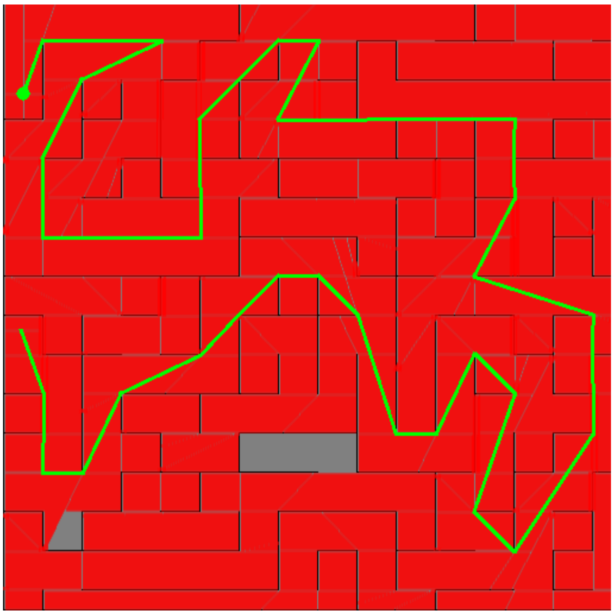


Figure 5.12: Anya Search-space on Mazes (32W): red – search-space scanned by Anya; green – path found

5.3.4 Node expansions

During the search process for a path between the start node and the target node on a 2D grid graph, the algorithm directs the incremental traversal of the graph from a node to its immediate free neighbours in the search-space. The number of nodes that are expanded by the algorithm can impact the efficiency of the algorithm. Performing occupancy checks on the nodes implies accessing the data structure that stores the map information. The nodes generally need to be stored in memory, in a data structure (e.g. open list) that requires insertion and extraction of elements. These operations can cumulatively reduce performance. As such, the number of node expansions during a search is desired to be minimal.

Table 5.4: Average Node-expansion (10^3 units)

	Baldurs Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes (32W)**
A	268.95	196.07	1094.19	395.75	1159.61
A*PostSmoothing	268.95	196.07	1094.19	395.75	1159.61
Anya (Oh et al.)	527.21	646.67	1023.21	528.20	535.70
Anya (Harabor et al.)	N/A	N/A	N/A	N/A	N/A
Ray Path Finder	2.43	73.50	21.57	2.93	30.60
Theta*	3784.46	3995.38	30708.76	4229.78	59869.25

Table 5.4 presents the average number of node expansions that the algorithms perform across the five games. As a note, node expansion information was not collected for the Anya (Harabor et al.) algorithm. Their version of Anya employs bit-level shifting and masking in a number of different procedures that scans along grid rows which are represented by integers in an 1D integer-array that encodes at the bit-level the occupancy of the grid. The optimisations adopted by Harabor et al. for their implementation of Anya results in a fast and efficient algorithm, but the resulting tightly coupled codebase makes collection of the node-expansion metric difficult. The version of Anya by Oh et al. does not employ this approach and as such, the node expansions were more easily obtainable.

Examining the node expansions in Table 5.4, Theta* stands out as the most costly of the five remaining algorithms. This is expected, given that it eagerly performs line-of-sight operations for all pairs of nodes in its open list. The two versions of A*, having the same underlining expansion policy expand the same number of nodes.

The proposed methodology, Ray Path Finder, expands the fewest number of nodes on average, by two orders of magnitude on maps from Baldur's Gate II, Starcraft and Warcraft and by one order of magnitude on Dragon's Age: Origins and maze maps. Because of its principle of operation, RPF works on free-space assumptions and only expands nodes when moving in free-space towards a target, or tracing the bounds of an obstacle.

Figures 5.13, 5.14 and 5.15 exemplify the nodes expanded by three of the tested algorithms: Theta*, Anya (Oh et al.) and Ray Path Finder.

Figure 5.13: Search-space of Theta*: red – expanded nodes; green – path solution

Theta* (Figure 5.13) presents a similar search-space to A*. However, as observed in Table 5.4, Theta* can perform an order of magnitute more node expansions than A*. While the number of distinct nodes expanded is similar to A* (i.e. the area covered by the search), expanded nodes are frequently revisited during the line-of-sight checks that Theta* performs, which negatively impacts the performance of the algorithm. Performing line-of-sight checks between each node and its parent results in the algorithm having a long run-time, which limits its applicability.

Figure 5.14: Search-space of Anya (Oh et al.): red – expanded nodes; green – path solution

We observed in Section 5.3.3 that the run-time performance of Harabor et al.'s implementation of Anya is unchallenged. One reason for this is its principle of operation. From a visual inspection of Anya's search-space, as depicted in Figure 5.14, we see that the surface of the explored free-space is typically comparable to algorithms such as A* or Theta* (Figure 5.13). Unlike A* and Theta*, however, which expand nodes sequentially, updating a node's parent and calculating a score for each node, Anya, instead, searches over intervals of free-space across grid rows. This proves to be very effective, as both implementations of Anya (Oh et al. and Harabor et al.) perform well in practice. The second reason for Anya (Harabor et al.)'s performance revolves around its efficient implementation that makes use of bit-level manipulation to expand over grid-rows very fast.

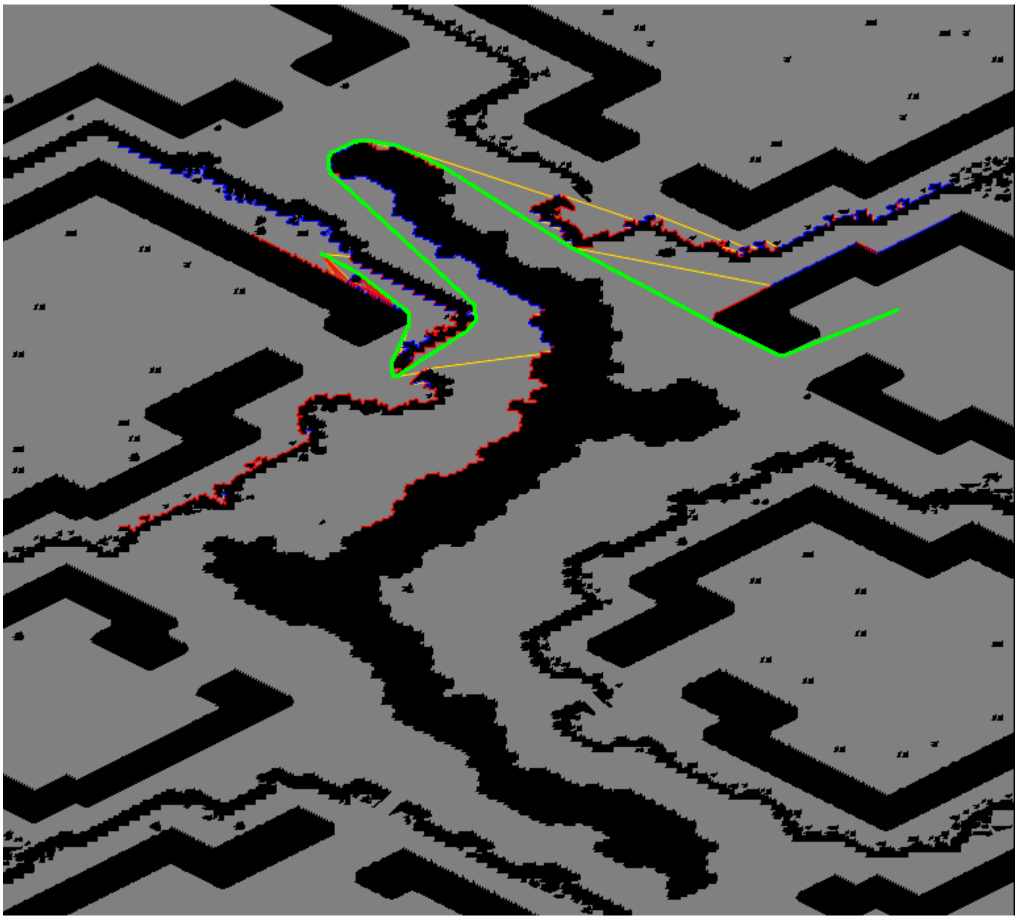


Figure 5.15: Search-space of Ray Path Finder: blue – expanded nodes by left-bound paths; red – expanded nodes by right-bound paths; yellow – expanded nodes by line-of-sight; green – path solution

A key attribute of the Ray Path Finder algorithm is its reduced search-space. An example of RPF's expansion policy is illustrated in Figure 5.15. The algorithm only expands nodes along object boundaries (marked with blue for nodes expanded by left-bound paths and with red for nodes explored by right-bound paths) and when traveling in a straight line towards a goal after leaving an obstacle's boundary (marked with yellow). Because in operates on the free-space assumption, it also delays performing line-of-sight between the nodes of a path, until it has reached the goal. This avoids the caveat of performing extensive line-of-sight checks that encumber algorithms such as Theta*.

As observed in this section, while RPF exhibits an order of magnitude reduction in the number of nodes expanded relative to Anya, this is not directly reflected in the run-time performance of the algorithm (as observed in Table 5.3). The probable causes and possible solutions for improving the run-time performance of RPF are discussed in Section 5.4.

5.3.5 Heading changes

A path on a 2D grid graph is composed of sequential segments that connect pairs of nodes from the start node to the end node. A holonomic agent following this path would require to change direction in any node of the path where the segments are not collinear. These nodes are referred to as heading changes and reflect on the perceived intelligence of the agent. Shortest paths only have heading changes around obstacle corners. Arbitrary heading changes of the path in free-space look unrealistic and are not conducive to optimal path solutions. In general, minimising the number of heading changes creates a more realistic path and contributes to the perceived intelligence of the robotic platform by making the subsequent path-following more efficient, reducing fuel consumption, minimising loss of momentum, etc..

Table 5.5: Average Heading-changes

	Baldurs Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes (32W)**
A*	51.23	52.46	95.65	44.83	60.51
A*PostSmoothing	5.97	19.71	23.32	5.84	24.66
Anya (Oh et al.)	4.71	17.36	15.01	5.04	23.97
Anya (Harabor et al.)	4.60	15.97	13.66	4.99	23.78
Ray Path Finder	4.85	17.35	16.27	5.30	24.75
Theta*	4.91	17.13	16.00	5.32	24.31

Table 5.5 summarises the average number of heading changes the algorithms perform. As the only grid-constrained algorithm of the six examined, A* exhibits the largest number of heading changes. A robot that

would use A* as its path-planning algorithm would behave very inefficiently, being restricted to movements of 90 deg and 45 deg and unnecessarily having to change its direction instead of navigating straight through free space. Even employing a simple smoothing technique, such as A* with Post-Smoothing would improve on the robot's behaviour, sometimes by an order of magnitude, as is the case for Baldur's Gate II and Warcraft. Among the any-angle algorithms, A* with Post-Smoothing is the least effective but nonetheless manages to significantly reduce the number of heading changes.

The remaining algorithms, Anya (Oh et al. and Harabor et al.), Theta* and Ray Path Finder, all improve on this metric to a similar extent. To justify why Anya (Harabor et al.) presents with a smaller number of heading changes, as compared to the implementation by Oh et al., we must note that the metric for the four algorithms was collected by retrieving the number of nodes contained in the path of a resulting solution. Collinear nodes (which are inconsequential to path length) that were identified by the algorithms but are in fact redundant do not get removed from the path, and, as such, are not discounted by the heading changes metric. Additionally, the algorithms do not account for the existence of different equal-cost paths that have the same length but are different in the number of heading changes, as minimising a path's length is what drives the search.

5.3.6 Memory footprint

The memory footprint during the algorithm's runtime is considered as an indicator of the resource usage of the methodology, which can affect performance and which can dictate the employability of an algorithm. Reducing the memory footprint of an algorithm reduces the load on resources, and frees them up to be used for other purposes and minimises power consumption, ownership cost of the hardware and response time. A space-time trade-off can occur when an algorithm produces fast results at the cost of higher memory requirements. For an algorithm to be efficient, we wish to have a fast algorithm that is not demanding of higher resources.

Table 5.6 presents the average memory expenditure of the algorithms.

Table 5.6: Average Memory (KB)

	Baldurs Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes (32W)**
A*	63.10	51.83	415.02	69.54	1553.47
A*PostSmoothing	78.13	58.17	444.52	81.25	1668.20
Anya (Oh et al.)	122.83	3415.76	3397.54	163.06	505.84
Anya (Harabor et al.)	88.82	807.27	1178.49	96.59	428.78
Ray Path Finder	18.99	999.22	209.68	296.37	469.93
Theta*	4973.34	2182.60	36545.15	5498.77	79919.66

Examining Table 5.6, no one algorithm stands out as the most memory efficient for all game maps. A* and A* with Post-Smooting present with a relatively consistent memory footprint for Baldur's Gate II, Dragon's Age: Origins, Warcraft and Starcraft, outperforming all algorithms apart from Ray Path Finder. However, their performance degrades considerably on Mazes, in which they are the most memory demanding after Theta*. Theta* is the poorest performing algorithms of the six evaluated, by as much as two orders of magnitude against A* on the maps from Baldur's Gate II, Dragon's Age: Origins, Warcraft and Starcraft. It has the highest memory footprint of all algorithms on Mazes, followed by A* with Post-Smooting and A*. The reason for this is that maze maps have a high concentration of connected, unoccupied nodes which have to be maintained in memory, with each node requiring to update their heuristic cost. This results in a flood-fill behaviour of the algorithms, in which they require to explore the entirety of the search-space up to the goal node. In addition to this requirement, Theta* also needs to perform line-of-sight checks for the unoccupied nodes.

Anya (Oh et al.) is the second poorest performing algorithm after Theta*. Anya (Harabor et al.) outperforms Anya (Oh et al.) for all maps, up to an order of magnitude on game maps from Baldur's Gate II, Dragon's Age: Origins and Warcraft. It also outperforms Ray Path Finder on Dragon's Age: Origins, Warcraft and Mazes. However, it is outperformed by Ray Path Finder, A* and A* with Post-Smooting on Baldur's Gate II and Starcraft.

Ray Path Finder has the smallest memory footprint for maps in Baldur's Gate II and Starcraft, but performs poorly on Warcraft, where it is the second most memory demanding after Theta*. The higher memory demand for these games is likely due to overhead paths, for which Ray Path Finder grossly underestimates the heuristic cost, and which forces the algorithm to maintain them in memory for longer. On Dragon's Age: Origins, it outperforms Anya (Oh et al.) but not Anya (Harabor et al.), and is an order of magnitude more memory intensive than A*.

5.3.7 Impact of Gridmap resolution

This section illustrates the impact on performance relative to the scale of the maps. Maps were scaled at two times (2X) their original dimensions.

The experiments presented are performed on a 2.9GHz Intel Core i7 machine with 8GB of RAM running Windows 10. For this particular experimental setup, the number of scenarios is reduced to a 1–in–3 sampling rate. The graphs presented are semi-logarithmic scatter plots on the Y axis. Tables 5.7, 5.8, 5.9, and 5.10 present the averages for the metrics collected on maps scaled at double their original size.

Table 5.7: Average Path-length

	Baldurs Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes (32W)**
A*	503.66	821.10	1094.77	512.67	2222.61
A*PostSmoothing	484.94	797.76	1055.57	493.49	2159.63
Anya (Oh et al.)	480.16	783.04	1040.13	484.19	2151.33
Anya (Harabor et al.)	480.16	786.68	1040.13	487.82	2151.33
Ray Path Finder	480.19	786.86	1041.29	487.87	2151.33
Theta*	480.26	787.04	1040.48	487.95	2151.35

As expected, the path length is double that of the original scale (see Table 5.2) for all algorithms tested.

Table 5.8: Average Run-time

	Baldurs Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes (32W)**
A*	35.73	39.78	213.27	43.07	373.95
A*PostSmoothing	317.22	113.97	2356.46	616.80	1147.10
Anya (Oh et al.)	5.96	31.85	32.52	6.07	8.18
Anya (Harabor et al.)	0.25	1.63	2.35	0.34	0.68
Ray Path Finder	0.37	24.58	7.23	0.42	4.54
Theta*	184.08	295.31	1852.82	226.00	2991.25

In Table 5.8, at the $2X$ scale, run–time of Theta* is substandard. It is seen to have an average of a 87% increase in comparison to the original scale, Table 5.3. A* Post–Smoothing is seen to suffer from magnitude degradation.

While Anya (Harabor et al.) and Ray Path Finder are approximately two times slower than the original scale, they are still the fastest algorithms of the six examined. Anya (Oh et al.) appears to slow down more due to the algorithm implementation.

Table 5.9: Average Node-expansion (10^3 units)

	Baldurs Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes (32W)**
A*	1059.44	780.84	4241.25	1533.82	4542.11
A*PostSmoothing	1059.44	780.84	4241.25	1533.82	4542.11
Anya (Oh et al.)	2102.57	2601.42	3979.59	2104.75	2119.72
Anya (Harabor et al.)	N/A	N/A	N/A	N/A	N/A
Ray Path Finder	4.85	283.70	52.77	5.90	62.89
Theta*	29697.33	35355.73	233104.12	33872.69	482717.58

Node expansions in 2X Scale is seen in Table 5.9. A* and A* Post-Smoothing have the same expansion principal, and would expand the same number of nodes. From examination of this table, Theta* remains standing out as the costliest of the algorithms (refer to Table at original scale 5.4) - this was again expected, similar to the 1X scale, has to perform line-of-sight checks for all pairs of nodes in its open list.

Table 5.10: Average Heading-changes: 2X Scale

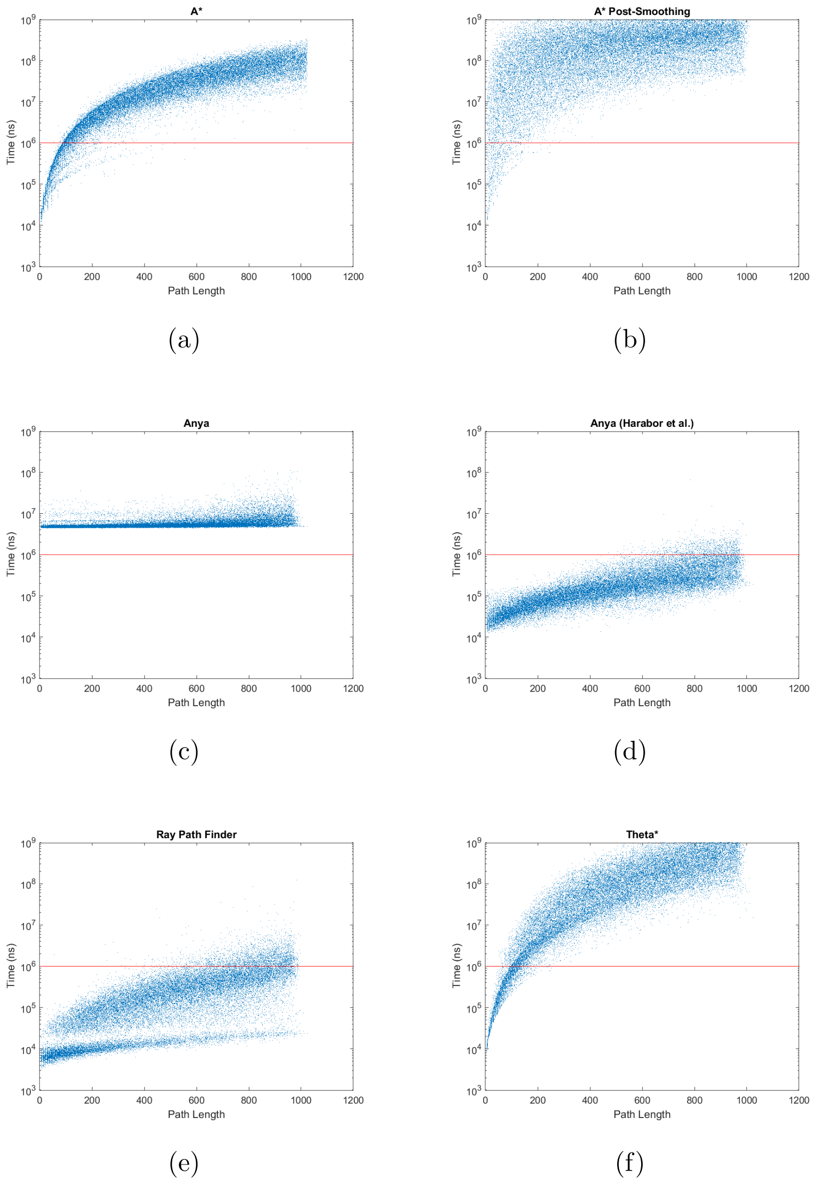
	Baldurs Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes (32W)**
A*	92.08	74.06	142.68	79.56	91.34
A*PostSmoothing	5.91	20.87	23.08	5.80	24.77
Anya (Oh et al.)	4.68	18.31	14.82	5.02	24.08
Anya (Harabor et al.)	4.56	16.80	13.48	4.97	23.89
Ray Path Finder	4.83	19.97	16.61	5.28	24.86
Theta*	4.97	18.81	16.30	5.41	24.47

With the exception of A*, heading changes are not affected by scaling, as the topologies of the paths don't change, but merely their length. This can be observed in 5.10. A*, as discussed previously, is constrained to moving along grid edges and hence would have to perform a higher number of heading changes whilst attempting to move diagonally at angles other than 45°.

In comparison to A*, A* with Post-Smooting and Theta*, scaling has less of an impact on Ray Path Finder overall. RPF's metrics preserve the same order of magnitude compared with those seen at the original scale (i.e. in Table 5.5). which demonstrates RPF's resilience to map scaling.

It has also been observed that Anya (Harabor et al.) appears to share RPF's resilience to map scaling. However, Anya (Oh et al.) is not as fortunate, as its run-time degrades faster.

Looking at Figures 5.16, 5.17,5.18, 5.19, and 5.20, the profiles of these algorithms appear to remain almost identical in comparison to their $1X$ profiles, Figures 5.7, 5.8, 5.9, 5.10, 5.11, respectively.



(a) (b)

(c) (d)

(e) (f)

Figure 5.16: Baldur's Gate: 2x Scale. (Path Length vs. Time (ns))

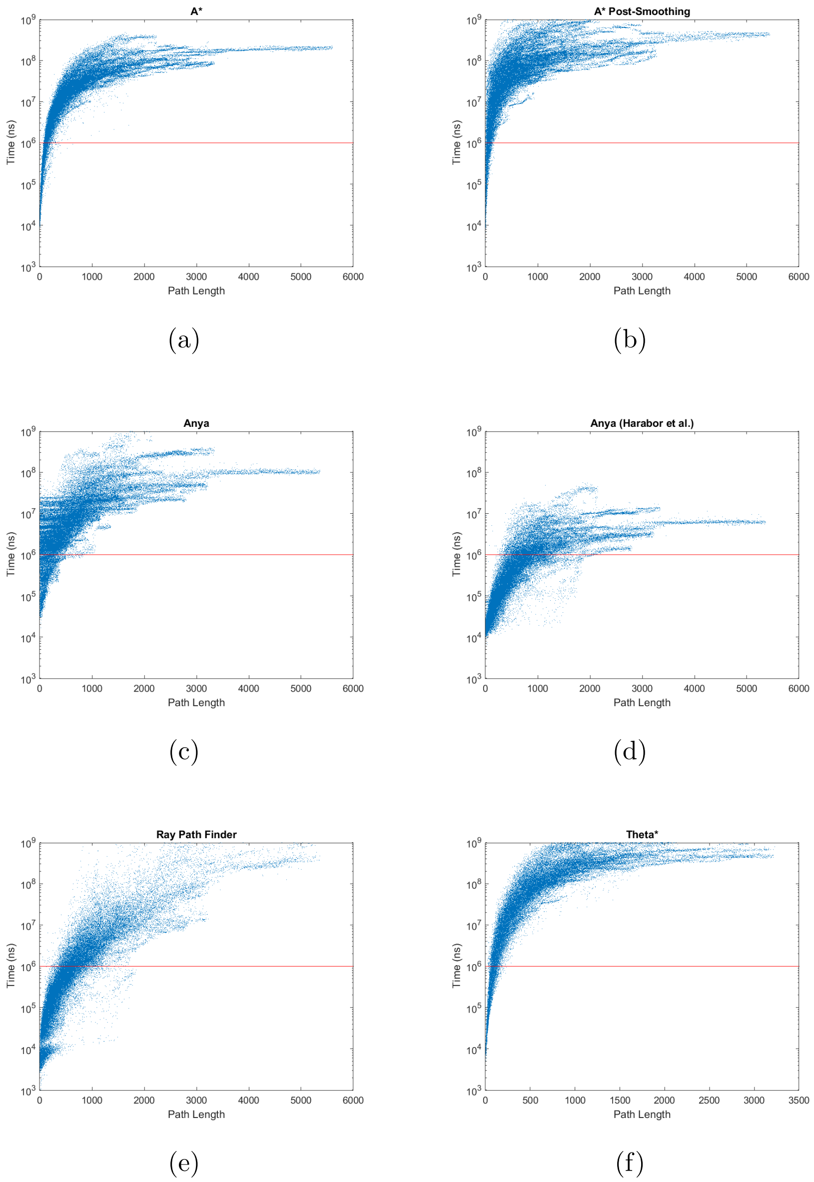


Figure 5.17: Dragon Age Origins: 2x Scale. (Path Length vs. Time (ns))

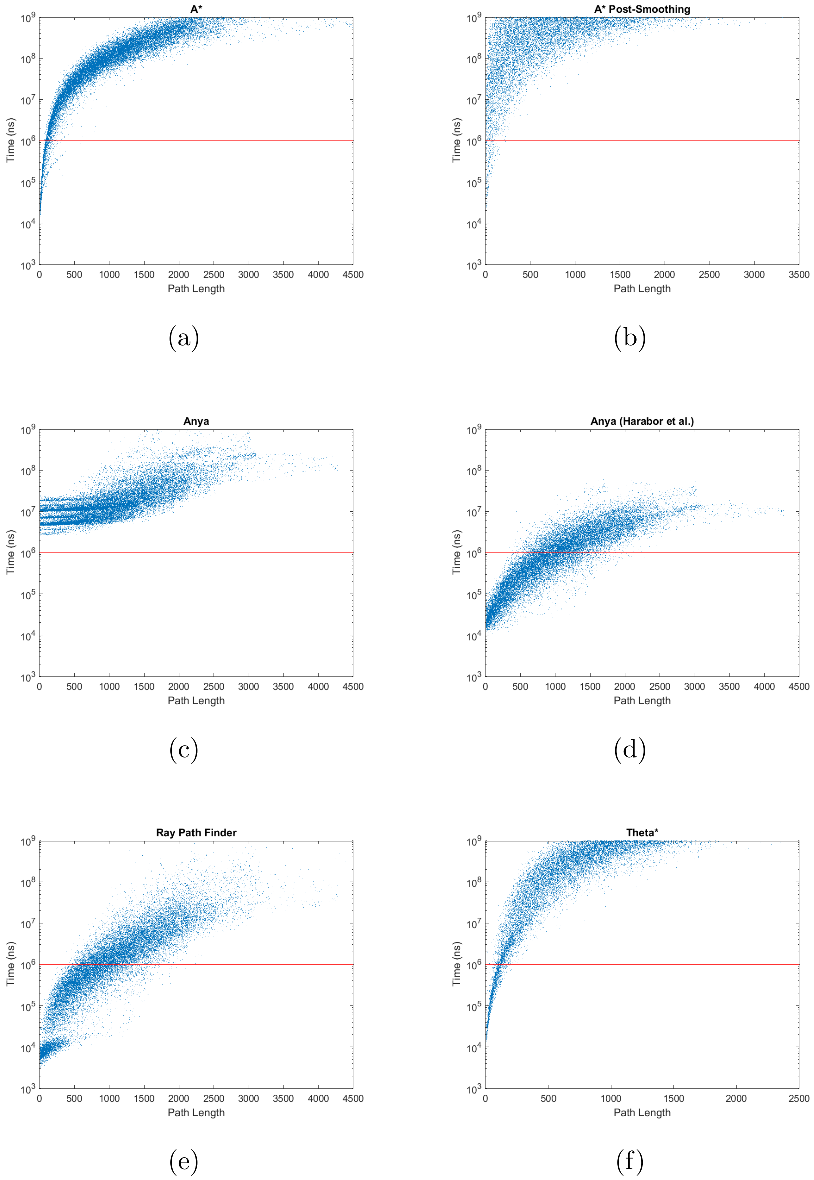


Figure 5.18: Starcraft: 2x Scale. (Path Length vs. Time (ns))

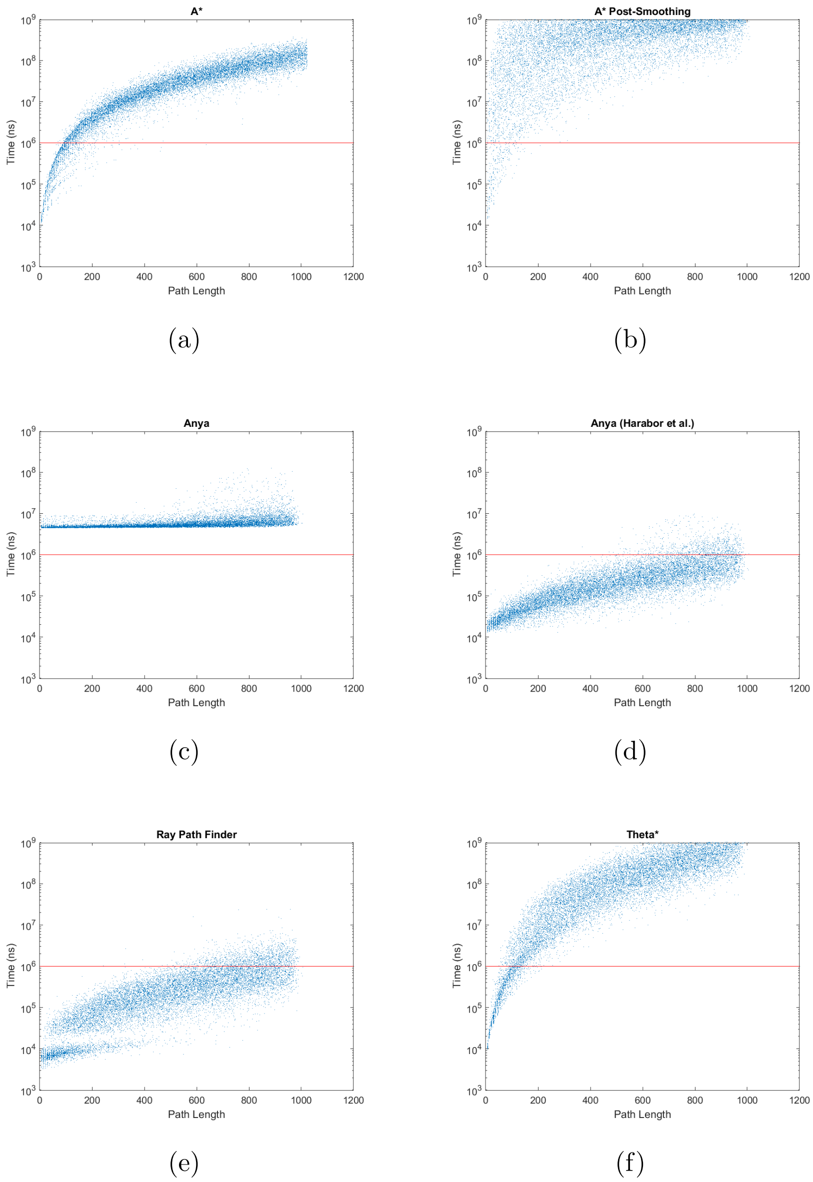


Figure 5.19: Warcraft: 2x Scale. (Path Length vs. Time (ns))

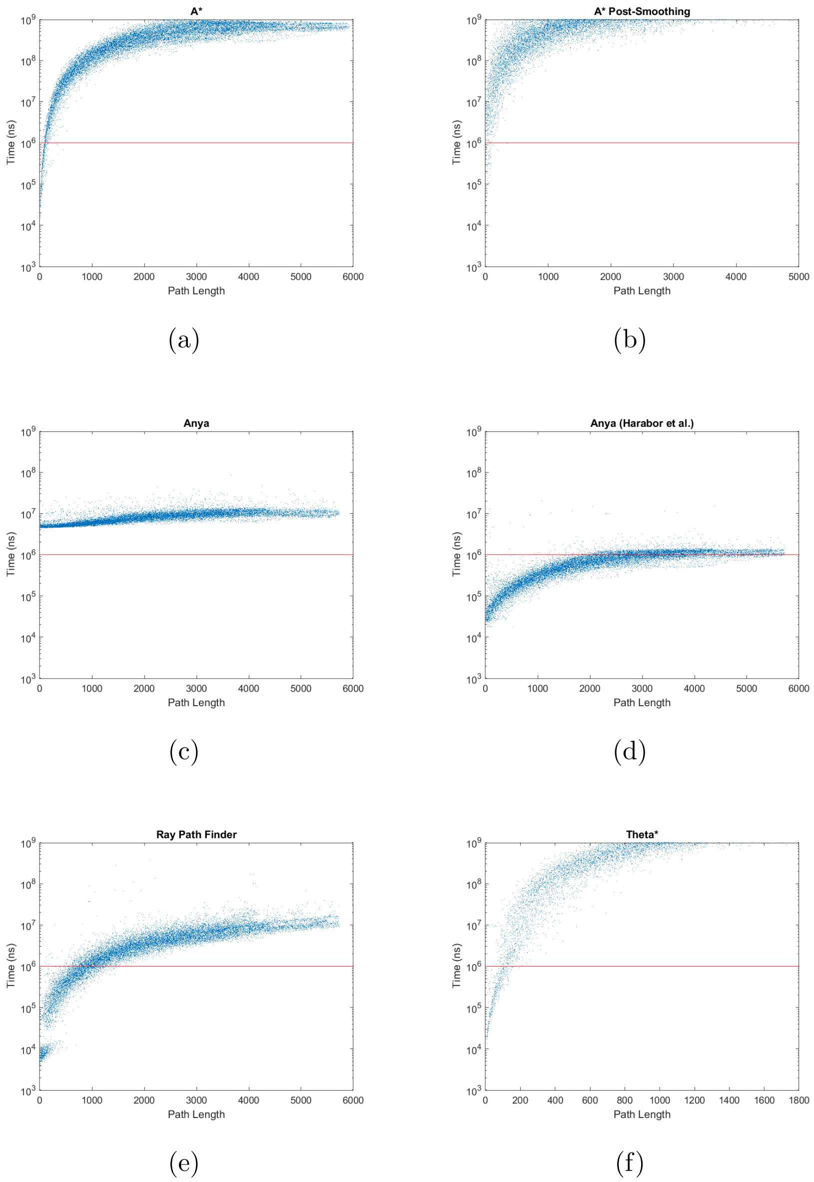


Figure 5.20: Mazes (32W): 2x Scale. (Path Length vs. Time (ns))

5.3.8 RPF versions

This section compares two different implementations of the Ray Path Finder algorithm, the variant described in the current work and refered to as Ray Path Finder or RPF, and a legacy variant labeled *RPF on Cells*. The development of *RPF on Cells* was subsequently abandonded in favour of RPF, which favours a cleaner and less complex implementation.

Ray Path Finder, the newer version of the algorithm operates over vertices on a 2D uniform-cost octile grid-graph indexed as a 1D array. The legacy variant, *RPF on Cells*, operates over cell centers on a 2D uniform-cost octile grid-graph. It indexes the grid-map as a 2D array, maintaing the x and y coordinates for each node, as opposed RPF which treats nodes as single integer values in the 1D array representation of the grid.

The principle of operation of both algorithms is the same, but given their different implementations, it serves as a useful showcase to justify the adoption of the vertex based solution, embodied by RPF. The implementation of *RPF on Cells* avoided using vertices, but rather operated directly on the grid-cells. However, this version presented with functional issues when exploring narrow spaces, such as corridors with minimum widths of 1 cell. For such scenarios, a cell could be visited repeatedly from different directions, and additional checks were required to identify these situations. Furthermore, because obstacles could occasionally share the same corner cell and the search paths require keeping track of such corner nodes, paths could pass through the same cell but attempt to indentify it as a distinct corner, which would result in an unwanted self-intersection. For example, two diagonally opposing squares that are spaced one cell apart have the same cell as a corner, one as its lower-right corner, and the other as its upper-left corner. Because of this, a path could end up passing through the same cell while following the edge of the two separate obstacles, but falsely conclude that it had intersected itself. To account for such cases, a workaround required keeping track of pairs of cells for each corner, both the free corner node that belonged to the path and the occupied corner that belonged to the object boundary. In addition to list which maintains the path, the implementation required additional lists to keep track of the corner pairs. Unfortunately, these solutions considerably increased the complexity of the algorithm further, and resulted in a degradation of performance from the additional overhead. The vertex based solution was adopted instead, as it proved more elegant, and allowed for better performance. The vertex-based Ray Path Finder avoided the pit-falls that plagued the previous cell-centric variant, as each vertex uniquely indentifies a corner of an obstacle.

A set of experiments pits the two implementations of RPF against each other. Scenarios were selected from the entire database, with a 1–in–3 sampling step. The experiments were performed on an 1.8GHz Intel Core i3 with 4GB RAM running Windows 10. The following tables represent the averages of the collected metrics. The new algorithm improves on the metrics of the original, with an average 55.75% decrease in run-time (Table 5.11), and an average 80.01% decrease in memory expenditure (Table 5.12).

Table 5.11: Average Time (ms) on maps

	Baldur's Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes 32W**
RPF	0.45	4.95	9.18	0.51	2.96
RPF On Cells	1.02	9.34	14.80	1.25	13.90

Table 5.12: Average Memory usage (KB)

	Baldur's Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes 32W**
RPF	0.55	37.81	127.36	0.65	5.27
RPF On Cells	9.59	1862.47	1730.36	0.77	1425.85

Table 5.13 shows an improvement in path length due to paths wrapping tighter around an obstacle boudary, i.e. the path passes through the corner

vertex which neighbours an occupied cell.

Table 5.13: Average Path Length

	Baldur's Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes 32W**
RPF	235.65	186.66	496.89	239.80	1024.70
RPF On Cells	236.74	190.93	505.74	240.93	1174.91

Additionally, the search procedure also presents a speed-up in converging to the any-time solution. Table 5.14 summarises the average time in milliseconds for the algorithm to first encounter the goal, before it proceeds to optimise a solution.

Table 5.14: Average Time (ms) to reach goal on first encounter

	Baldur's Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes 32W**
RPF	0.12	3.48	3.75	0.14	1.14
RPF On Cells	0.19	6.50	5.88	0.19	4.56

While an improvement on its experimental predecesor, the implementation of the Ray Path Finder algorithm has not been optimised for performance. Refinement of RPF is a subject for future research and has the potential to further improve on the algorithm's metrics.

5.3.9 Anya vs. RPF

This section compares Ray Path Finder, the proposed algorithm, with the two implementations of Anya, identified in the previous sections as the best performing algorithms. The metrics used for comparison with RPF for taken from Anya (Harabor et al.), with the exception of node-expansions which are not available. The node expansion metrics are instead extracted from Anya (Oh et al.). The following experiments were performed on an 1.8GHz Intel Core i3 with 4GB RAM running Windows 10. Scenarios were selected from the MovingAI database, with a 1–in–3 sampling step.

Tables 5.15, 5.16 & 5.17 summarise the averages of the collected metrics. Anya has a 48.88% higher memory footprint on average, and covers 89.21% more of the search-space, but outperforms RPF in run-time by 65.68%. Table 5.18 presents the percentage of scenarios for each game in which Ray Path Finder outperforms Anya (Harabor et al.) with respect to run-time. The first row in the table indicates the total percentages in which RPF is faster. This includes the scenarios in which the start and stop goal have a direct line-of-sight to eachother. The second row in the table excludes the aformentioned scenarios, looking only at scenarios in which there is at least one obstacle breaking the line of sight between the start and destination. Even for such cases, RPF outperfroms Anya (Harabor et al.) between 40% and 49% of the time, with the one exception being Mazes, in which it only outperforms it on 15% of occassions. Coupling this with the information from Table 5.17, in which it was observered that Anya (Harabor et al.) is faster on average than RPF, it becomes apparent that there are scenerios for which the performance of RPF is severely degraded. Empirical observations made from comparisons of the two algorithms on individual maps indicate that the ocassions for which RPF suffers considerable degradation in run-time occur when the scenarios under observation lead the algorithm to form “over-head” paths for which the heuristic length function greatly underestimates the cost of travel, and prioritises them over viable paths with a consistent heuristic cost. This is likely to happen for certain map topologies, as best can be observered for the cases with the Maze maps, in which overhead paths are very common due to long, winding corridors.

Table 5.15: RPF vs. Anya (Harabor et al.): Average Memory (KB)

	Baldur’s Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes 32W**
Anya (Harabor et al.)	470.69	630.07	1105.92	490.04	621.89
RPF	173.43	374.19	1177.74	470.46	583.61

Table 5.16: RPF vs. Anya (Oh et al.): Average search-space expansion

	Baldur’s Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes 32W**
Anya (Harabor et al.)	49382.35	69968.98	476788.60	77352.50	186510.63
RPF	2269.59	11501.90	19237.24	2893.57	46861.51

Table 5.17: RPF vs. Anya (Harabor et al.): Average Run-time(ms)

	Baldur’s Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes 32W**
Anya (Harabor et al.)	0.26	0.80	2.40	0.37	0.73
RPF	0.50	6.62	11.38	0.51	5.27

Table 5.18: RPF vs. Anya (Harabor et al.): Scenarios where RPF has better Run-time than Anya (Harabor et al.) (%)

	Baldurs Gate II	**Dragon Age: Origins**	**Starcraft**	**Warcraft**	**Mazes (32W)**
RPF - obstacles ≥ 0	48%	45%	52%	52%	16%
RPF - obstacles ≥ 1	40%	41%	49%	44%	15%

The graphs illustrated in Figures 5.21, 5.22, 5.23, 5.24 and 5.25 are logarithmic scatter plots which present the behaviours of RPF and Anya over the five database games. They indicate that with the exception of scenarios where start and goal nodes have direct line-of-sight, Anya outperforms RPF in run-time. Both algorithms present a similar behaviour profile between path length and run-time for all types of maps, with Anya

(Harabor et al.) having a faster run-time than RPF. These results can be explained by Anya (Harabor et al.)'s performance optimisations curtsey of the authors, and by RPF's inefficient implementation coupled with the performance hit due to the liberal expansion of "over-head" paths which advertise grossly underestimated heuristic lengths to the driving Best-first search algorithm of RPF.

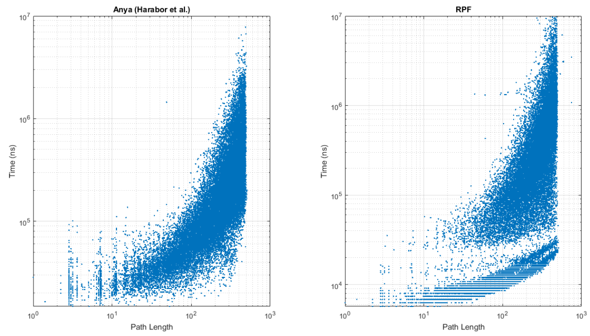


Figure 5.21: Anya (Harabor et al.) vs. RPF: Path Length - Time(ns) on "Baldur's Gate II" maps

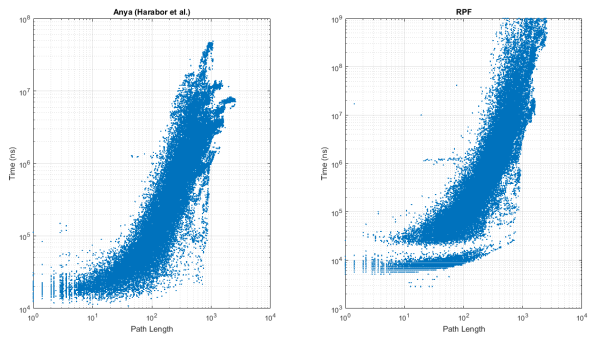


Figure 5.22: Anya (Harabor et al.) vs. RPF: Path Length - Time(ns) on "Dragon Age: Origins" maps

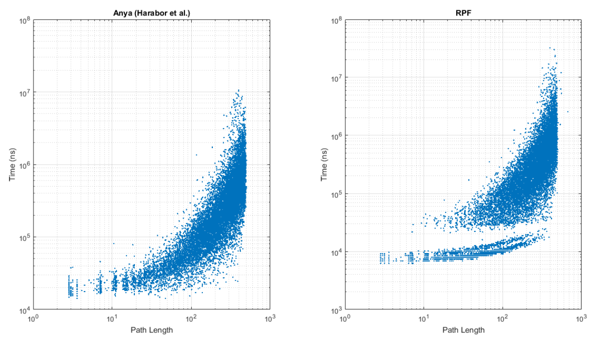


Figure 5.23: Anya (Harabor et al.) vs. RPF on Warcraft maps: Path Length - Time(ns)

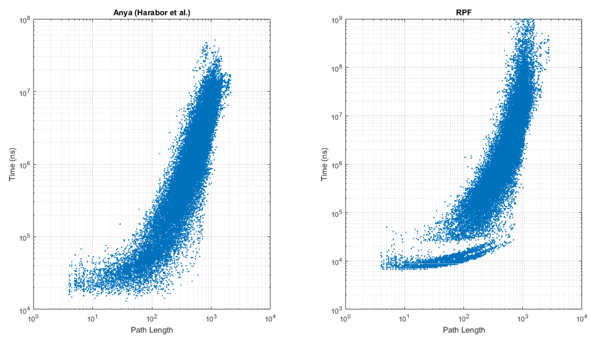


Figure 5.24: Anya (Harabor et al.) vs. RPF: Path Length - Time(ns) on “Starcraft” maps

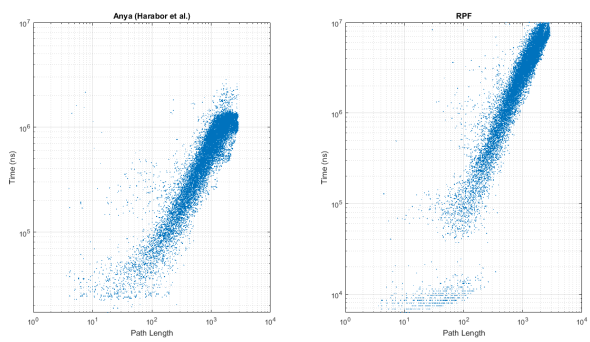


Figure 5.25: Anya (Harabor et al.) vs. RPF: Path Length - Time(ns) on “Mazes 32W” maps

Figure 5.26 presents a sub-sample of Figure 5.24 which isolates the data points for scenarios in which start and target have a direct line-of-sight. As can be observed, RPF performs slightly better in these circumstances because the search is equivalent to a single line-of-sight check.

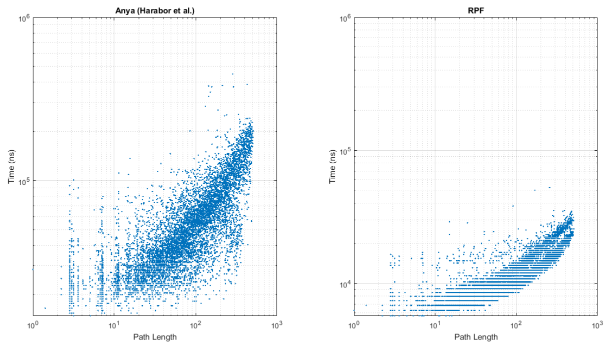


Figure 5.26: Anya (Harabor et al.) vs. RPF: Path Length - Time(ns): sub-sampling of direct line-of-sight data-points

5.4 Discussion on RPF

A novel strategy for finding any-angle paths on octile grids was introduced in the form of Ray Path Finder. To the author's knowledge, RPF is the only algorithm to combine a best-first-search strategy and bug-like path searches to perform online any-angle path-planning.

In this chapter we presented the experimental evaluation of the RPF algorithm against five popular state-of-the-art online path-planning algorithms from the literature, namely A*, A* with Post-Smooting, Theta*, and two distinct implementations of the Anya algorithm (Anya (Oh et al.) and Anya (Harabor et al.)). Each algorithm was evaluated over 453500 scenarios on 314 maps from five popular game maps from the MovingAI database (presented in Table 5.1 of Section 5.2).

The algorithm has been shown experimentally to be competitive on a number of different metrics against the other five algorithms. The metrics used for this evaluation were the length of the identified path, run-time of the algorithm, nodes expanded during search-space exploration, heading changes of the path, and memory requirements of the algorithm.

Regarding path length, Ray Path Finder finds short paths on the same level as Anya (Oh et al.), Anya (Harabor et al.) and Theta* (as observed in Section 5.3.2). Looking at run-time (Section 5.3.3), Ray Path Finder is the second fastest algorithm, overall, after Anya (Harabor et al.) on four of the five evaluated games, and is similar in performance but generally faster than the variant of Anya by Oh et. al. which present a non-optimised implementation. One game represents the exception. RPF suffers a reduction of performance on Dragon's Age: Origins, making it the second slowest after Theta*. The limitation of RPF described in Section 4.5 regarding the underestimation of the heuristic cost of "over-head" paths (i.e. paths that the interior contour of an obstacle that contains the start node) is another aspect that is subject for future improvement. As there exists no strategy to inform on which scenarios or maps present with topologies that can lead RPF to create and expand "over-head" paths, no straightforward conclusion can be drawn as to the negative impact that these scenarios have on the performance of the algorithm. Our conjecture is that, if an elegant solution for accurately estimating the heuristic length of "over-head" paths were to be identified and implemented, the performance of Ray Path Finder would improve, and possibly make RPF more competitive against the dominant algorithm in our experimental results, namely Anya (Harabor et al.). Such a solution would come in the form of an efficient way of implementing a methodology similar to the one described in Section 4.5, to allow for an accurate estimate of the heuristic of "over-head" paths. Because such a solution does not exist in practice as of this writing, the optimality of the algorithm is compromised. The Best-first search algorithm prioritises paths that promise the smallest heuristic. As such, over-head paths can find themselves being evaluated first, as they underestimate their true heuristic cost, which can result in additional run-time.

As observed in Subsection 5.3.4, the proposed methodology explores fewer nodes within the search-space compared to other on-line path planning algorithms. However, while Ray Path Finder exhibits a reduction in the number of nodes expanded relative to Anya, this is not always reflected in the run-time performance of the algorithm. The version of Anya by Harabor et al. is, on average, faster than our proposed methodology across all game maps, as observed in Subsection 5.3.3.

Subsection 5.3.6 shows Ray Path Finder to have a smaller memory footprint than other algorithms for game maps from Baldur's Gate II and Starcraft, and a memory footprint comparable to Anya (Oh et al.) and Anya (Harabor et al.) for the other evaluated games. Both Ray Path Finder and Anya (Harabor et al.) present with decent resilience to scaling up of maps, as observed in Subsection 5.3.7.

Because RPF, in its current form, is generally dominated by Anya (Harabor et al.), it may not be the preferred search method for any-angle path-planning on octile grids. However, RPF possesses properties that can make it appealing for certain applications. If a faster run-time is desirable over path-length, the algorithm can be allowed to terminate early, after at least one path has converged to a sub-optimal solution. Given its ability to produce multiple solutions, for instance, gives the option for a higher-level planner to chose alternative paths that may have other favourable characteristics, such as shallow turning angles for preserving momentum.

The variability in performance of the algorithm requires further investigation, as it could be attributed to a number of different factors. One key reason for this comes down to the implementation of the Ray Path Finder algorithm. Inefficiencies in Ray Path Finder's implementation may contribute to the algorithm's poor performance relative to Anya (Harabor et al.). One aspect of Anya (Harabor et al.)'s implementation is the use of a Fibonacci heap for operating its priority queue, which improves an algorithm's asymptotic run-time. In contrast to Anya (Harabor et al.), RPF has not been optimised to use a Fibonacci heap for its priority queue data structure, but uses only a generic queue provided by the Java API library. The second reason for Anya (Harabor et al.)'s performance revolves around its efficient implementation that makes use of bit-level manipulation to expand over grid-rows very fast. This allows the algorithm to scan the free search-space efficiently, but tightly-couples the algorithm to the data structure representing the grid-map. For their implementation, Anya (Harabor et al.) use a bit-packed integer matrix where each bit of an integer element represents the occupancy state of the search-space. While Ray Path Finder along with the other algorithms use the same underlining data-structure for consistency of the experimental results, they are not able to take advantage of the bit-packed matrix, having to access the data-structure one bit at a time. Because of these reasons, the algorithm provides room for improvements of its implementation. As well, other additions to its principles of operation could help it make further headway in improving its performance. One caveat of RPF's implementation is how path objects are represented and handled internally by the algorithm. Individual paths are treated as linked lists which undergo look-ups, insertions, extractions and cloning. These processes can cumulatively take a long time and consume resources. Cloning the paths, for instance, can be a relatively expensive procedure, as memory needs to be allocated on the heap for each new object (i.e. list). The main Best-first search algorithm is implemented as a priority queue that extracts and reinserts paths with each iteration. With each insertion and reinsertion, the heuristic length of each path is calculated in order to compare them with each-other, and to order each path in the queue from shortest to longest. Recalculating the path of each path with each expansion is expensive and unnecessary. A better solution that stores the path's length and only updates it during a change in its topology would be preferable. As an additional example of inefficiency, given a large number of intersections with obstacles, the priority queue may hold many paths in the queue, out of which only a handful may prove useful in leading to a solution.

The positive aspects in Ray Path Finder's performance present a promising avenue for further research into the algorithm. The evidence to date presents RPF as a competitive algorithm as compared to other state-of-the-art algorithms. It has been observed in Subsection 5.3.9 that RPF can outpeform Anya (Harabor et al.) in run-time, up to 52% of the time on certain game maps. The algorithm's main defficiency lies in RPF's inability to accurately estimate the cost of "over-head" paths, as no appropriate solution has been developed as of this writing. Future research into Ray Path Finder that extends beyond the scope of the current work, may provide a solution to the problem of "over-head" paths. If such a solution is indeed identified, it could address a large array of scenarios for which RPF currently exibits a substantial degradation in performance. It is likely that additional optimisations brought to the algorithm would improve the run-time and potentially further reduce the number of node expansions. Such optimisations would include, but not be limited to, using a Fibonacci heap to operate RPF's priority queue, caching the results of line-sight checks that have succeeded, information sharing between paths to indicate search-spaces that have already been explored, and possibly adopting/implementing a better data structure representing the expanded paths, e.g. using a tree structure. A solution that accurately estimates the heuristic of "over-head" paths would also allow RPF to converge to solutions faster. Applying a smarter branch-and-bound strategy and heuristics to the Best-first search algorithm may be able to discard paths early would also prove useful in reducing the number of paths in the queue.

Chapter 6

Conclusion

This chapter presents the summary of the findings of the developed any-angle path-planning algorithm. Conclusions and recommendations are offered for future research work, including some potential applications of the novel algorithm.

6.1 Summary

The thesis introduces a novel best-first search algorithm for finding any-angle paths on grid-constrained graphs. To the author's knowledge, RPF is the only algorithm to combine a best-first-search strategy and bug-like path searches to perform online any-angle path-planning.

We have developed and implemented an online any-angle path-planning algorithm based on "bug-like" paths with free-space assumptions and conducted by a best first-search algorithm. The paths travelling towards a goal (using a variant of Bresenham's line algorithm introduced in 3.2) bifurcate when encountering an obstacle and split off in opposite directions. The paths perform wall-following (using a novel contour tracing algorithm introduced in 3.3), while identifying corner points that can be added to the path. While wall-following, each path investigates its tautness to preserve the optimality of the best-first search that handles the priority of each path based on which path estimates it is the shortest. For this purpose it prunes nodes that compromise a path's tautness at each step to retain a consistent heuristic. It greedily searches for a solution among the most promising paths and only performs line-of-sight checks between path vertices after the path has arrived at a solution.

The proposed algorithm is shown experimentally to be competitive on a number of performance metrics with state-of-the-art any-angle algorithms. It also presents desirable properties that allow it to have a reduced search-space and make it suitable for providing any-time solutions.

Employing the algorithm can reduce the search space considerably (Figure 6.1) and finds solutions fast. The algorithm presents with competitive metrics, comparable to Anya, the fastest state-of-the-art online any-angle path-planning algorithm. It also allows for multi-path and any-time solutions, making it a good candidate for robotic platforms or applications that impose time constraints.

Figure 6.1: Search space comparison: A* (left), Theta*(middle), RPF (right)

Additionally, a graphical user interface was developed in order to:

- provide a general and simple tool for evaluating path-planning algorithms.
- perform simulations and visualise results regarding algorithm behaviours.
- provide compatibility with the MovingAI map database.
- enable large-scale data collection of algorithm metrics.

This would allow future researchers to use the tool and integrate their algorithm with the application, allowing them to focus on the development of their respective path-planning methodology rather than having to implement, manage and handle low-level interactions with incompatible map databases.

RPF operates on 2D grid-maps, represented by a 2D occupancy matrix that encodes a cell as block or unblocked. As such, its implementation is specific to operations performed on the matrix, i.e. traversing the encoded free-space of the occupancy grid by incrementally moving in the Moore neighbourhood of a cell, as well as tracing along the contours of blocked regions in the same manner. Through its principle of operation, however, RPF could also operate on a ground robot by employing behaviours similar to those used by bug algorithms (e.g. tactile and/or range sensors along with odometry). Essentially, the paths that the algorithm propagates towards the goal node act in a similar way to individual bugs.

As part of the Ray Path Finder algorithm, a novel contour tracing algorithm has been developed, which can provide a good alternative to other methodologies in the literature and which can have potential applications beyond RPF (e.g. image segmentation).

6.2 Future Work

The research presented in this work has been undertaken as part of the Cognitive Assisted Living Ambient System (COALAS) Project (Nr. 4194) [91]. The COALAS project was selected under the European cross-border cooperation programme INTERREG IV A France (Channel) – England, and co-funded by the European Regional Development Fund (ERDF), with the aim of developing an "autonomous cognitive platform, combining an intelligent wheelchair coupled with the assistive capabilities of a humanoid robot" [91]. The project aimed to develop a system consisting of a humanoid robot, powered wheelchair, and sensors in order or develop an assistive navigation system, which has been a key issue of development for the disabled. The COALAS wheelchair falls under the category of assistive technology, with semi-autonomous (collision avoidance) and autonomous functionality (mapping, planning). The wheelchair has been supplied with Udoo Quad on-board mini PC, a LiDAR sensor and wheel-encoders. We have adapted the system for collision avoidance, remote control, odometry estimation and integrated the ROS navigation stack for the purposes of mapping and autonomous navigation. The ROS framework for robotics development and visualisation along with the Gazebo Robotic simulation environment (Figures 6.2, 6.3, 6.4) allow for a safe testing and experimentation environment. Complementary to the development of the Ray Path Finder algorithm, the ROS-enabled smart wheelchair serves as a mobile robotic agent capable of mapping the environment and autonomous navigation, and represents the prime candidate for a future implementation of a ROS-based RPF path-planning solution.

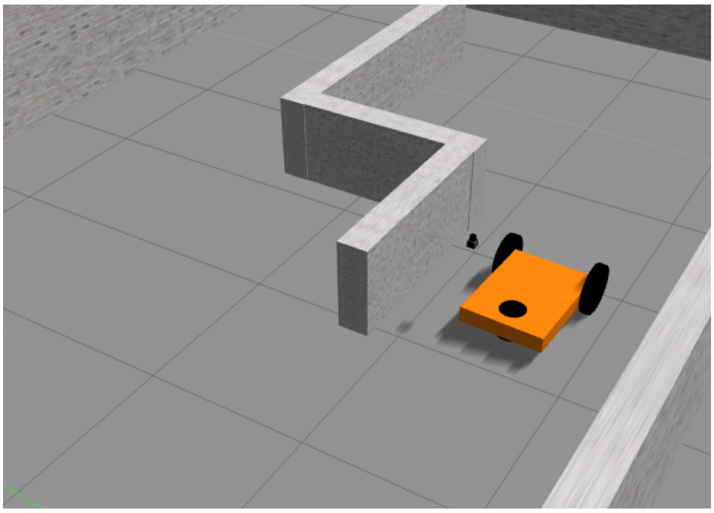


Figure 6.2: Screen-capture of platform model with LiDAR sensor inside maze constructed using Gazebo simulation environment [92]

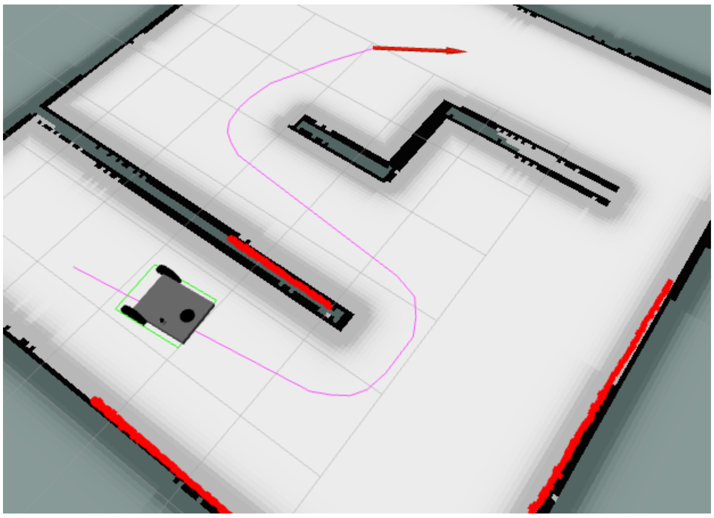


Figure 6.3: Screen-capture of platform model with LiDAR sensor in RViz following path found by Theta* implementation [92]

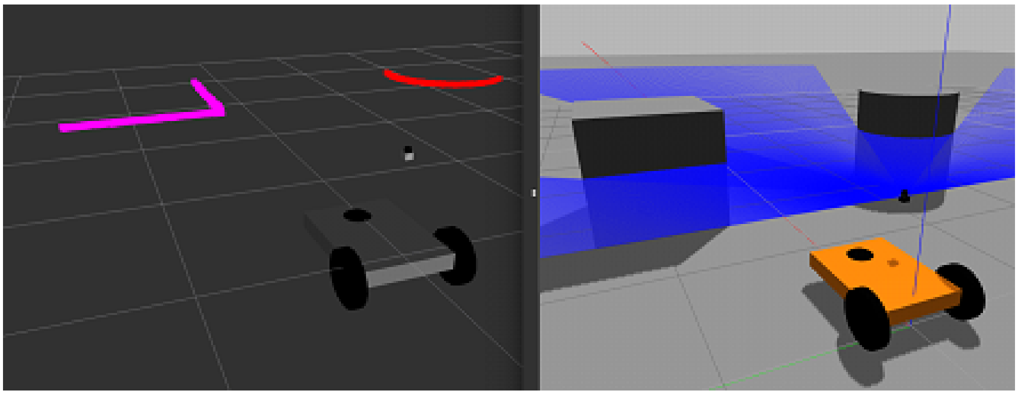


Figure 6.4: Left: screen-capture of platform model with LiDAR sensor feedback as viewed in RViz (ROS compatible robotics visualisation tool); Right: screen-capture of platform with LiDAR in simulated world with obstacles in Gazebo [92]

6.2.1 Optimisations

This section presents a brief discussion on potential optimisations of the RPF algorithm.

The current implementation of Ray Path Finder performs ray-casting between subsequent node pairs until all node pairs have a clear line of sight between them. This presents some redundancy. After a path has reached the goal, line-of-sight checks are performed between subsequent nodes in the path. If both the path and its bifurcated descendent share common vertices other than the root (start node) and continue to be expanded after having reached the goal (i.e. they have similar lengths and/or the algorithm requests multiple solutions), the line-of-sight checks between the shared vertices are performed independently for each path, leading to redundancy. This can be prevented by sharing information between paths (i.e. maintaining a history of the performed line-of-sight checks). A possible implementation would be to cache the results of the Bresenham algorithm explorations already performed in order to avoid redundancy in the line-of-sight expansions.

The direction monitoring functionality of each path is implemented by maintaining references to subsequent vertices on the obstacle's edge and computing the discrete number of turns away from the desired direction of travel towards a goal node. This solution is part of a legacy implementation, but the same functionality could be achieved in other ways. One such way would be to perform angle calculations, which could prove more appropriate for implementing RPF on a robotic platform lacking a grid-map discretisation of the environment. Another possible alternative, which would operate on grid maps would be to add the number of inner corners and subtract the number of outer corners.

As of this writing, a drawback of the current implementation of the RPF algorithm relates to how a path bifurcates at the point of intersection with an obstacle. Each individual path is implemented as a self-contained object, and maintains its own references to path nodes it had previously visited, and the list of nodes under consideration to be part of the path. When an obstacle is encountered, a new copy of the path in question is constructed, i.e. copies of the data structures containing the node references are assigned to the newly created path. Given an environment or configuration conducive to a high number of intersections, the copying process could delay the runtime of the algorithm with memory management tasks. The redundancy of the information in each path could be reduced. This remains an elusive problem and among the most prominent open questions that have arisen from the current research.

Path creation

In an environment with many concave objects, paths could intersect repeatedly with edges of the same object, which would lead to the generation of a new redundant path for each new intersection. From an implementation stand-point, object creation and duplication of the data structures from one path object to another can strain resources and impact performance. Given that one drawback of RPF is that it requires the creation of a new path object with each wall intersection, it can prove desirable to avoid unnecessary path generation. This could be achieved in a number of ways and we will discuss three possible options:

- Off-line obstacle labelling - in a preprocessing stage, obstacles inside a map can be uniquely identified (by means of contour tracing, for example) and the cells belonging to the same obstacle can share an identifier unique to each individual object; by this means, in the online path-planning search, when an intersection occurs with a cell that has the same identifier as the ones of the previous wall following step, the algorithm can avoid creating a new path to follow the obstacle in the opposite direction, as the path would intersect itself and be discarded as redundant. One should note that if the goal node were located inside the convex hull bounding an obstacle, the procedure described above could fail to find a solution. Therefore, additional information may be required from the preprocessing stage, for example, a bounding box associated with each object.

- Online lookahead step - after a path determines that it can leave the edge of an obstacle and resume travel in a straight line, before engaging in a ray-cast towards a goal node (or internal goal node), the algorithm can perform a lookahead step, in which it allows the wall following to carry on for a number of steps (chosen stochastically); the lookahead procedure would terminate in one of three ways, either by intersecting the path, in which case the obstacle would not be intersected again, or by crossing on the opposite side of the M-line defined by the goal node and the internal start node (point of departure from obstacle edge), in which case the wall following can be allowed to continue normally (i.e. a new path would be redundant), or, finally, by exhausting the number of steps, in which case a conclusion would not be drawn, and the algorithm would resume normal ray-casting behaviour. The latter case would prove costly as a redundant path may still be generated; as such, choosing the number of lookahead steps could have an either a positive or negative impact on performance.

- Avoidance of path structures - one could circumvent the issue by avoiding the task of path creation altogether. The implementation choice for moving away from node objects (e.g. implementations of algorithms in the A* family), and using path objects instead was done on the basis of simplicity. The implementation uses structures such as linked lists for appending, inserting and deleting nodes which are represented by signed integer values (the absolute value represents the index corresponding to the vertex, and the sign reflects the side of the path at the moment of exploration). The main difficulties of using individual node objects is the problem of nodes having multiple potential parents and also maintaining the sides of the paths passing through the node. Algorithms such as R* [93] and LIAN [94] could provide alternatives methods for handling multiple parents. A solution that would address these issues is reserved as a potential avenue for future research.

Multi-threading

The algorithm could benefit from parallelisation (eg. parallel Dijkstra [95]) as the searches executed by each path are performed independently of each other. From an implementation point of view, multiple threads running simultaneously would have access to shared resources such as the grid graph. For static maps however, occupancy queries (read operations from memory and/or cache) would not require synchronisation (preventing multiple threads from accessing the object at the same time). Shared resources that would require synchronisation to prevent concurrency faults would be the priority stack maintaining references of the active paths. The decision of adopting parallelisation would take into consideration the boost in performance and weigh the potential reduction im runtime against the overhead of multi-threading (thread object creation, shared resource synchronisation, processor architecture, etc.).

6.2.2 Open Questions and Potential Applications

The following section presents a brief discussion on some potential applications and variations on the RPF framework. Some questions that could hopefully fuel future research directions are considered:

One avenue of investigation, that has not been explored in detail by the current work, would be to alter the algorithm to perform path clearing procedures (i.e. checking for line-of-sight between subsequent nodes of a path) in reverse order (from stop to start) after having reached the goal node at least once. Such a behaviour would be desirable and expected of a robotic agent that would enact a RPF-like navigation strategy. In an unknown environment the robot would learn about its surroundings by multiple attempts of travelling between the start and stop goals, assimilating more information about the environment and optimising the path it travels with each iteration. Such a scenario would imply travelling towards the goal and returning to the start node, but rather than retracing its steps, the robot can attempt to optimise the path by minimising the travel score between path nodes. Preliminary experiments indicate that reversing the order of the nodes in a path (Line 40 of Algorithm 9) and/or reversing the path's side are insufficient to accomplish this behaviour.

6.2.3 Hybridisation

One can imagine a hybrid algorithm that combines the initial stage of the Ray Path Finder algorithm (racing a path towards an end goal) with other solutions such as Theta* or Anya to clear a path that has reached a goal. During the development stage, brief experimentation with a RPF- A* variant allowed for the discovery of solutions with fewer node expansions than a purely A* algorithm, but had the same limitations in a path's any-angle and optimality attributes as A*. In an RPF hybrid algorithm, a path is allowed to search until it terminates or reaches its target, after which a second algorithm is allowed to perform internal searches between nodes of the path. To arrive at an optimal solution, any such hybrid should also maintain a path's tautness (node backward and forward pruning), which would introduce a higher complexity to the implementation.

Similar to other algorithms, such as Bidirectional A* [96], RPF on known environments may benefit from a bidirectional variant, in which two simultaneous searches from start and goal meet to form a solution. A bidirectional RPF could, ideally, improve performance, but with a trade-off of having a higher memory demand and complexity. Other hybrids that more closely fit the any-angle paradigm would see a RPF variant that can handle non-holonomic movement, such as in the case of Theta*-RRT [97].

Swarm robotics

In a hypothetical adaptation of RPF in a robot swarm application, robots can be sent out and allowed to perform a search for the target, after which more robots can attempt to optimise the solutions found. If multiple paths are discovered, the robots can be distributed among them so as to avoid congestion.

Cellular automata

A discrete computational model with applications in diverse fields of study, from biology to mathematics, cellular automata are a useful tool for generating complex using relatively simple rules. In general, a cellular automaton consists of a cell grid where each cell represents an entity that exists in one of a finite number a states [98]. Given a set of rules applied to each cell based on the states of the cells in its neighbourhood, the state of the cell can change with each generation. Some classes of resulting patterns that evolve with each generation can exhibit complex behaviour with few simple governing rules. Conway's Game of Life, one of the most popular examples of a cellular automaton, has been determined to be Turing complete [99], making it a powerful method of computation, and is designed around four simple rules:

- a live cell with fewer than two neighbours dies, due to under-population
- a live cell with more than three neighbours dies, due to over-population
- a live cell with two or three neighbours survives to the next generation
- a dead cell with exactly three neighbours becomes a live cell.

Hypothetical algorithms combining RPF inspired behaviours with a cellular automaton are considered as a future research direction. One can envision such a cellular automaton in a path planning strategy having a few simple rules:

- automaton starts off with only one cell (start node)
- a dead cell which finds itself in an open space, and has a live cell as neighbour (either an open space or a leave point) or next t
- a live cell with more than three neighbours dies, due to over-population
- a live cell with two or three neighbours survives to the next generation
- a dead cell which finds itself on an object boundary with at least one living neighbour becomes a live cell (i.e. contour tracing), unless the neighbour is a leave point

Rather than expanding one cell at a time, a cellular-automata solution would be able to cover a larger surface depending on the size of the desired object or based on the momentum of a navigating agent. This would also be applicable to solutions found by RPF, where cellular automata would re-trace the path or paths found by the algorithm, expanding larger surface areas to account for desired properties or behaviours from the robotic agent, such as corner clearance to avoid collisions with moving targets, or to approach the corner from a better angle of attack to preserve momentum and have a smoother, more natural trajectory.

An RPF inspired cellular automaton variant may provide a solution to path-planning on grids with non-uniform traversal costs, as the automaton may allow the search-space to dilate locally, in order to account for paths with different weights. A cellular automaton approach to RPF may also be able to tackle the problem of path-planning in 3D environments.

Bibliography

[1] J. Hawkins and S. Blakeslee, *On Intelligence: How a New Understanding of the Brain will Lead to the Creation of Truly Intelligent Machines.* Macmillan, 2007.

[2] K. Magzhan and H. M. Jani, "A Review And Evaluations Of Shortest Path Algorithms," *International journal of scientific & technology research*, vol. 2, no. 6, pp. 99–104, 2013.

[3] J. Jones, "A Morphological Adaptation Approach to Path Planning Inspired by Slime Mould," *International Journal of General Systems*, vol. 44, no. 3, pp. 279–291, 2015.

[4] A. Nash, "Any-Angle Path Planning," Ph.D. dissertation, University of Southern California, 2012.

[5] M. Pala, N. Osati, F. López-Colino, A. Sanchez, A. de Castro, and J. Garrido, "HCTNav: A Path Planning Algorithm for Low-Cost Autonomous Robot Navigation in Indoor Environments," *ISPRS International Journal of Geo-Information*, vol. 2, no. 3, pp. 729–748, 2013.

[6] J. Antich, A. Ortiz, and J. Mínguez, "A bug-inspired algorithm for efficient anytime path planning," in *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on.* IEEE, 2009, pp. 5407–5413.

[7] "Real Time Collision," http://realtimecollisiondetection.net/blog/?p=56, (Accessed on 01/05/2017).

[8] R. G. Golledge, "Path Selection and Route Preference in Human Navigation: A Progress Report," in *International Conference on Spatial Information Theory.* Springer, 1995, pp. 207–222.

[9] "Harvard Medical School - Imagery Lab," http://www.nmr.mgh.harvard.edu/mkozhevnlab/, (Accessed on 01/04/2017).

[10] A. Madkour, W. G. Aref, F. U. Rehman, M. A. Rahman, and S. Basalamah, "A Survey of Shortest-Path Algorithms," *arXiv preprint arXiv:1705.02044*, pp. 1–26, 2017.

[11] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT press, 2005.

[12] J. Aulinas, Y. Petillot, J. Salvi, and X. Lladó, "The SLAM Problem: A Survey," in *Proceedings of the 2008 Conference on Artificial Intelligence Research and Development: Proceedings of the 11th International Conference of the Catalan Association for Artificial Intelligence*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2008, pp. 363–371. [Online]. Available: http://dl.acm.org/citation.cfm?id=1566899.1566949

[13] N. Karlsson, E. Di Bernardo, J. Ostrowski, L. Goncalves, P. Pirjanian, and M. E. Munich, "The vSLAM Algorithm for Robust Localization and Mapping," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*. IEEE, 2005, pp. 24–29.

[14] R. F. Salas-Moreno, R. A. Newcombe, H. Strasdat, P. H. Kelly, and A. J. Davison, "SLAM++: Simultaneous Localisation and Mapping at the Level of Objects," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1352–1359, 2013.

[15] G. Dudek and M. Jenkin, *Computational Principles of Mobile Robotics*. Cambridge University Press, 2010.

[16] P. Yap, "Grid-based Path-finding," in *Conference of the Canadian Society for Computational Studies of Intelligence*. Springer, 2002, pp. 44–55.

[17] A. Patel, "Red Blob Games," https://www.redblobgames.com/, 2000, (Accessed on 01/04/2017).

[18] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[19] A. Botea, M. Müller, and J. Schaeffer, "Near Optimal Hierarchical Path-finding," *Journal of Game Development*, vol. 1, no. 1, pp. 7–28, 2004.

[20] A. W. Kring, A. J. Champandard, and N. Samarin, "DHPA* and SHPA*: Efficient Hierarchical Pathfinding in Dynamic and Static Game Worlds," in *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2010.

[21] R. Dechter and J. Pearl, "Generalized Best-first Search Strategies and the Optimality of A," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 505–536, 1985.

[22] D. Ferguson and A. Stentz, "Field D*: An Interpolation-Based Path Planner and Replanner," *Robotics research*, pp. 239–253, 2007.

[23] R. E. Korf, "Depth-first Iterative-deepening: An Optimal Admissible Tree Search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[24] D. Harabor and A. Grastien, "An Optimal Any-Angle Pathfinding Algorithm," *ICAPS 2013 - Proceedings of the 23rd International Conference on Automated Planning and Scheduling*, pp. 308–311, 2013.

[25] D. Harabor, A. Grastien, D. Öz, and V. Aksakalli, "Optimal Any-angle Pathfinding in Practice," *Journal of Artificial Intelligence Research*, vol. 56, pp. 89–118, 2016.

[26] A. G. Michael Cui, Daniel D. Harabor, "Compromise-free Pathfinding on a Navigation Mesh," in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, 2017, pp. 496–502.

[27] S. Oh and H. W. Leong, "Edge N-Level Sparse Visibility Graphs: Fast Optimal Any-Angle Pathfinding Using Hierarchical Taut Paths," *arXiv preprint arXiv:1702.01524*, 2017.

[28] T. Uras and S. Koenig, "Identifying Hierarchies for Fast Optimal Search," *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence Identifying*, pp. 878–884, 2014.

[29] T. Uras and S. Koenig, "An Empirical Comparison of Any-angle Path-planning Algorithms," in *Eighth Annual Symposium on Combinatorial Search*, 2015.

[30] D. D. Harabor, A. Grastien *et al.*, "Improving Jump Point Search," in *ICAPS*, 2014.

[31] V. J. Lumelsky and A. A. Stepanov, "Path-Planning Strategies for a Point Mobile Automaton Moving Amidst Unknown Obstacles of Arbitrary Shape," *Algorithmica*, vol. 2, no. 1, pp. 403–430, 1987.

[32] V. J. Lumelsky, S. Mukhopadhyay, and K. Sun, "Dynamic Path Planning in Sensor-Based Terrain Acquisition," *IEEE Transactions on Robotics and Automation*, vol. 6, no. 4, pp. 462–472, 1990.

[33] A. Sankaranarayanan and M. Vidyasagar, "Path Planning for Moving a Point Object Amidst Unknown Obstacles in a Plane: The Universal Lower Bound on the

Worst Path Lengths and a Classification of Algorithms," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on.* IEEE, 1991, pp. 1734–1741.

[34] H. Noborio and T. Yoshioka, "An On-line and Deadlock-free Path-planning Algorithm Based on World Topology," in *Intelligent Robots and Systems' 93, IROS'93. Proceedings of the 1993 IEEE/RSJ International Conference on*, vol. 2. IEEE, 1993, pp. 1425–1430.

[35] I. Kamon and E. Rivlin, "Sensory-Based Motion Planning with Global Proofs," *IEEE transactions on Robotics and Automation*, vol. 13, no. 6, pp. 814–822, 1997.

[36] I. Kamon, E. Rimon, and E. Rivlin, "TangentBug: A Range-Sensor-Based Navigation Algorithm," *The International Journal of Robotics Research*, vol. 17, no. 9, pp. 934–953, 1998.

[37] M. Weir, A. Buck, and J. Lewis, "POTBUG: A Mind's Eye Approach to Providing BUG-Like Guarantees for Adaptive Obstacle Navigation Using Dynamic Potential Fields," in *International Conference on Simulation of Adaptive Behavior*, vol. 4095. Springer, 2006, pp. 239–250.

[38] R. A. Langer, L. S. Coelho, and G. H. Oliveira, "K-Bug, a New Bug Approach for Mobile Robot's Path Planning," in *Control Applications, 2007. CCA 2007. IEEE International Conference on.* IEEE, 2007, pp. 403–408.

[39] J. Ng and T. Bräunl, "Performance Comparison of Bug Navigation Algorithms," *Journal of Intelligent & Robotic Systems*, vol. 50, no. 1, pp. 73–84, 2007.

[40] Y. Gabriely and E. Rimon, "CBug: A Quadratically Competitive Mobile Robot Navigation Algorithm," *IEEE Transactions on Robotics*, vol. 24, no. 6, pp. 1451–1457, 2008.

[41] A. Yufka and O. Parlaktuna, "Performance Comparison of the BUG's Algorithms for Mobile Robots," in *International Symposium on Innovations in Intelligent Systems and Applications (INISTA'09)*, 2009, pp. 416–421.

[42] M. Pala, N. O. Eraghi, F. López-Colino, A. Sanchez, A. de Castro, and J. Garrido, "HCTNav: A Path Planning Algorithm for Low-cost Autonomous Robot Navigation in Indoor Environments," *ISPRS International Journal of Geo-Information*, vol. 2, no. 3, pp. 729–748, 2013.

[43] J. Ng, "An Analysis of Mobile Robot Navigation Algorithms in Unknown Environments," Ph.D. dissertation, University of Western Australia, 2010.

[44] P. Corke, *Robotics, Vision and Control: Fundamental algorithms in MATLAB*. Springer, 2011, vol. 73.

[45] H. Choset, "Prinicipals of Robot Motion: Theory, Algorithms and Implementations," *Robotica*, vol. 24, no. 2, pp. 271–271, Mar. 2006.

[46] S. Shrotriya and A. Pandey, *Imitating Humans: A Technical Approach*, ser. Imitating Humans. LULU Press, 2013.

[47] "Open Robotics," http://blog.daum.net/pg365/115, (Accessed on 01/05/2017).

[48] B. C. V, A. T, and R. B, "A New Multi-Bug Path Planning Algorithm for Robot Navigation in known Environments," in *2016 IEEE Region 10 Conference (TENCON)*, Nov 2016, pp. 3363–3367.

[49] S. Skiena, "Dijkstra's Algorithm," *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pp. 225–227, 1990.

[50] A. Nash, S. Koenig, and C. Tovey, "Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D," *Third Annual Symposium on Combinatorial Search (SOCS-10) Lazy*, pp. 153–154, 2010.

[51] D. Harabor and A. Grastien, "Online Graph Pruning for Pathfinding On Grid Maps." *AAAI Conference on Artificial Intelligence*, pp. 1114–1119, 2011.

[52] J. E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

[53] K. I. Joy, "Breshenham's Algorithm," *Visualization and Graphics Research Group, Department of Computer Science, University of California, Davis*, pp. 1–15, 1999.

[54] C. Thorpe and L. Matthies, "Path Relaxation: Path Planning for a Mobile Robot," in *OCEANS 1984*, Sept 1984, pp. 576–581.

[55] A. Nash, S. Koenig, and M. Likhachev, "Incremental Phi*: Incremental Any-Angle Path Planning on Grids*," *IJCAI International Joint Conference on Artificial Intelligence*, pp. 1824–1830, 2009.

[56] A. Nash, K. Daniel, S. Koenig, and A. Felner, "Theta*: Any-Angle Path Planning on Grids," in *AAAI*, 2007, pp. 1177–1183.

[57] S. Oh and H. W. Leong, "Strict Theta*: Shorter Motion Path Planning Using Taut Paths," in *ICAPS*, 2016, pp. 253–257.

[58] A. Nash, "Lazy Theta*: Faster Any-Angle Path Planning," http://aigamedev.com/open/tutorial/lazy-theta-star/, July 2013, (Accessed on 01/04/2017).

[59] Š. Kopřiva, D. Šišlák, D. Pavlíček, and M. Pěchouček, "Iterative Accelerated A* Path Planning," pp. 1201–1206, 2010.

[60] D. Harabor, "Bitbucket," https://bitbucket.org/dharabor/pathfinding, 2017, (Accessed on 01/3/2017).

[61] L. E. Dubins, "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents," *American Journal of Mathematics*, vol. 79, no. 3, p. 497, jul 1957. [Online]. Available: https://doi.org/10.2307/2372560

[62] W. Cai, M. Zhang, and Y. Zheng, "Task assignment and path planning for multiple autonomous underwater vehicles using 3d dubins curves †," *Sensors*, vol. 17, no. 7, p. 1607, jul 2017. [Online]. Available: https://doi.org/10.3390/s17071607

[63] L. Chuan-ling, T. Jing, and Y. Jing-yu, "Path planning of mobile robot using new potential field method in dynamic environments," in *2011 Seventh International Conference on Natural Computation*. IEEE, jul 2011. [Online]. Available: https://doi.org/10.1109/icnc.2011.6022190

[64] J. Guldner and V. Utkin, "Sliding mode control for gradient tracking and robot navigation using artificial potential fields," *IEEE Transactions on Robotics and Automation*, vol. 11, no. 2, pp. 247–254, apr 1995. [Online]. Available: https://doi.org/10.1109/70.370505

[65] H. H. Triharminto, O. Wahyunggoro, T. B. Adji, A. I. Cahyadi, and I. Ardiyanto, "A novel of repulsive function on artificial potential field for robot path planning," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 6, no. 6, p. 3262, dec 2016. [Online]. Available: https://doi.org/10.11591/ijece.v6i6.11980

[66] P. Shi and Y. Cui, "Dynamic path planning for mobile robot based on genetic algorithm in unknown environment," in *2010 Chinese Control and Decision Conference*. IEEE, may 2010. [Online]. Available: https://doi.org/10.1109/ccdc.2010.5498349

[67] J. Guo, Y. Gao, and G. Cui, "Path planning of mobile robot based on improved potential field," *Information Technology Journal*, vol. 12, no. 11, pp. 2188–2194, nov 2013. [Online]. Available: https://doi.org/10.3923/itj.2013.2188.2194

[68] P. Bhattacharya and M. Gavrilova, "Roadmap-based path planning - using the voronoi diagram for a clearance-based shortest path," *IEEE Robotics & Automation Magazine*, vol. 15, no. 2, pp. 58–66, jun 2008. [Online]. Available: https://doi.org/10.1109/mra.2008.921540

[69] A. Sgorbissa and R. Zaccaria, "Planning and obstacle avoidance in mobile robotics," *Robotics and Autonomous Systems*, vol. 60, no. 4, pp. 628–638, apr 2012. [Online]. Available: https://doi.org/10.1016/j.robot.2011.12.009

[70] N. Sturtevant, "Moving AI Lab," http://www.movingai.com/, (Accessed on 01/04/2017).

[71] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: Any-angle Path Planning on Grids," *Journal of Artificial Intelligence Research*, vol. 39, pp. 533–579, 2010.

[72] A. Gheuneim, "Contour Tracing - Square Tracing Algorithm," http://www.imageprocessingplace.com/downloads_V3/root_downloads/tutorials/contour_tracing_Abeer_George_Ghuneim/square.html, 2000, (Accessed on 01/04/2017).

[73] J. Seo, S. Chae, J. Shim, D. Kim, C. Cheong, and T. D. Han, "Fast Contour-tracing Algorithm Based on a Pixel-following Method for Image Sensors," *Sensors (Switzerland)*, vol. 16, no. 3, pp. 1–27, 2016.

[74] H. Abelson and A. A. DiSessa, *Turtle Geometry: The Computer as a Medium for Exploring Mathematics.* MIT press, 1986.

[75] V. Lumelsky, "A Comparative Study on the Path Length Performance of Maze Searching and Robot Motion Planning," *IEEE Trans. Robotics and Automation*, vol. 7, no. 1, pp. 57–67, 1991.

[76] R. E. Korf, "Linear-space Best-first Search," *Artificial Intelligence*, vol. 62, no. 1, pp. 41–78, 1993.

[77] N. R. Vempaty, V. Kumar, and R. E. Korf, "Depth-First Versus Best-First Search," in *AAAI*, 1991, pp. 434–440.

[78] B. Bonet and H. Geffner, "Planning as Heuristic search: New Results," in *European Conference on Planning.* Springer, 1999, pp. 360–372.

[79] H. Farreny and H. Prade, "Heuristics—intelligent search strategies for computer problem solving, by judea pearl. (reading, ma: Addison-wesley, 1984)," *International Journal of Intelligent Systems*, vol. 1, no. 1, pp. 69–70, 1986. [Online]. Available: https://doi.org/10.1002/int.4550010107

[80] P. Muñoz and M. D. R-Moreno, "S-Theta*: Low Steering Path-planning Algorithm," *Res. and Dev. in Intelligent Syst. XXIX: Incorporating Applications and Innovations in Intel. Sys. XX - AI 2012, 32nd SGAI Int. Conf. on Innovative Techniques and Applications of Artificial Intel.*, pp. 109–121, 2012.

[81] A. T. Stentz, "Optimal and efficient path planning for partially-known environments," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '94)*, vol. 4, May 1994, pp. 3310 – 3317.

[82] M. Likhachev, G. J. Gordon, G. Gordon, and S. Thrun, "ARA*: Formal Analysis," 2003, unpublished Manuscript.

[83] L. Shelton, "Optimizing Modern Path Finding Methods in Imperfect 2D Environments," *Journal of Game Behavior*, vol. 1, 2014.

[84] T.-K. Wang, Q. Dang, and P.-Y. Pan, "Path Planning Approach in Unknown Environment," *International Journal of Automation and Computing*, vol. 7, no. 3, pp. 310–316, 2010.

[85] P. K. Das, A. Konar, and R. Laishram, "Path Planning of Mobile Robot in Unknown Environment," *Special Issue of IJCCT*, vol. 1, no. 2, p. 3, 2010.

[86] R. Vatcha and J. Xiao, "Perceived CT-Space for Motion Planning in Unknown and Unpredictable Environments," *Springer Tracts in Advanced Robotics*, vol. 79, pp. 883–897, 2014.

[87] "ROS," http://www.ros.org/, (Accessed on 01/04/2017).

[88] N. R. Sturtevant, "Benchmarks for Grid-based Pathfinding," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 2, pp. 144–148, 2012.

[89] T. Uras and S. Koenig, "Speeding-Up Any-Angle Path-Planning on Grids," pp. 234–238, 2015.

[90] "Github - ohohcakester/any-angle-pathfinding: A collection of algorithms used for any-angle pathfinding with visualisations." https://github.com/Ohohcakester/Any-Angle-Pathfinding, 2017, (Accessed on 09/12/2018).

[91] N. Ragot, G. Caron, M. Sakel, and K. Sirlantzis, "COALAS: A EU Multidisciplinary Research Project for Assistive Robotics Neuro-rehabilitation," in *IEEE/RSJ IROS Workshop on Rehabilitation and Assistive Robotics: Bridging the Gap Between Clinicians and Roboticists, Chicago, USA*, 2014.

[92] W. Wycherley, "A Robot Navigation System based on Image Processing," 2017, unpublished Manuscript.

[93] M. Likhachev and A. Stentz, "R* Search," *Lab Papers (GRASP)*, p. 23, 2008.

[94] K. Yakovlev, E. Baskin, and I. Hramoin, "Grid-Based Angle-Constrained Path Planning," vol. 9324, pp. 208–221, 2015.

[95] N. Jasika, N. Alispahic, A. Elma, K. Ilvana, L. Elma, and N. Nosovic, "Dijkstra's Shortest Path Algorithm Serial and Parallel Execution Performance Analysis," in *2012 Proceedings of the 35th International Convention MIPRO*, May 2012, pp. 1811–1815.

[96] G. Nannicini, D. Delling, D. Schultes, and L. Liberti, "Bidirectional A* Search on Time-dependent Road Networks," *Networks*, vol. 59, no. 2, pp. 240–251, 2012.

[97] L. Palmieri, S. Koenig, and K. O. Arras, "RRT-based Nonholonomic Motion Planning Using Any-angle Path Biasing," in *Robotics and Automation (ICRA), 2016 IEEE International Conference on.* IEEE, 2016, pp. 2775–2781.

[98] P. Singhal and H. Kundra, "A Review Paper of Navigation and Pathfinding using Mobile Cellular Automata," *Int. J. Adv. Comput. Sci. Commun. Eng.(IJACSCE)*, vol. 2, no. I, pp. 43–50, 2014.

[99] "Game of Life," http://web.stanford.edu/~cdebs/GameOfLife/, (Accessed on 01/04/2017).