The Collaborative Iterative Search approach to Multi Agent Path Finding

Callum Gordon Rhodes

A thesis submitted for the degree of
Doctor of Philosophy

School of Computer Science, Newcastle University

October 2017

Abstract

This thesis presents a new approach to obtaining optimal and complete solutions to Multi Agent Path Finding (MAPF) problems called Collaborative Iterative Search (CIS). CIS employs a conflict based scheme inspired by the Conflict Based Search (CBS) algorithm and extends this to include a linear order lower level search. The structure of Planar Graphs is leveraged, permitting further optimization of the algorithm. This takes the form of reasoning-based culling of the search space, while maintaining optimality and completeness. Benchmarks provided demonstrate significant performance gains over the existing state of the art, particularly in the case of sparsely populated maps. The thesis draws to a conclusion with a summary of proposed future work.

Acknowledgements

I would like to take this opportunity to thank Dr Graham Morgan for believing in me and for his encouragement and support throughout my research. Thanks are also due to Dr Gary Ushaw for all his help and advice.

I am also indebted to Dr William Blewitt for his unwavering support through the many highs and lows; his encouragement, understanding and wealth of experience which he was willing to share.

Within the Computer Science Department I would like to thank my colleagues who were always there to share a joke, coffee and their knowledge.

To Amanda for her patience and kindness during times of stress.

And a special thanks to my Mum, Dad and Sister who have supported me and allowed me to reach this point in my studies.

Contents

List of Figures

List of Algorithms

List of Tables

Chapter 1. Introduction

1.1 Introduction

This thesis presents Collaborative Iterative Search (CIS) a solution to the Multi-Agent Path Finding (MAPF) problem. Inspired by the unique Conflict-Based approach of Conflict Based Search (CBS) we employ a two layered scheme implementing a new algorithm (CIS) with a linear lower level search. We prove the completeness and optimality of CIS. Using the Planar property we extend the CIS algorithm to allow culling of paths which converge on the same collision. We call this extension CISR.

1.2 Concept of MAPF

Path finding is a widely applicable area of computer science. The area of path finding is concerned with the construction of paths from a start location to a goal location. In path finding we abstract the nature of the problem by constructing a mathematical structure called a graph. A graph G is a collection of points called vertices V and a set of edges E between the vertices in V. The purpose of the path finding algorithm is to construct a path $\mathbf{s}, e_1, \mathbf{v}_1, e_2, \ldots, e_{n-1}, \mathbf{v}_{n-1}, e_n, \mathbf{g}$ from the start node $\mathbf{s}$ to the goal node $\mathbf{g}$.

Associated with each edge of the graph is a cost. The sum of costs along a path determine the fitness or quality of a path. An optimal path finding algorithm will return a path of minimal cost.

The path planning problem can be extended by the inclusion of multiple agents. The Multi Agent Path Finding (MAPF) problem is concerned with finding paths for all the agents in a scenario without any two agents colliding. Agents can pause on a vertex of the graph and also travel backwards in order to avoid collision. A collision may occur on a vertex or along an edge if two agents travel along the same edge in opposite directions at the same time. When optimality is a concern the cost of the solution is generally computed as the Sum of Costs (SOC) where the cost of each agent's path is summed to give a total solution cost.

1.3 Approaches to solving MAPF

There are many different approaches to MAPF. The approaches to MAPF are generally divided into Optimal and Non-Optimal.

When the goal of a problem does not specifically exclude collisions between entities then the problem need not be fully abstracted to graphs. Algorithmic techniques such as flocking[SW02] and crowd simulation[MBCT98] allow for the realistic movement of entities in open 3D environments.

The focus of our research is the graph based MAPF problem where collisions need to be avoided. It has been shown that completeness, the property that a particular problem has if a solution exists, is NP-Hard. This is proven using the 15-puzzle[JS79,

RW90, RW86, Gol11] as the 15-puzzle can be represented as a MAPF problem.

When a problem is known to have a solution and optimality is sacrificed solutions can be found in a tractable time. There are several techniques to solving non-optimal MAPF, some techniques specialize in solving specific sub-problems or are only applicable in special scenarios.

Decentralized MAPF[GMF06] uses distributed computation to share the computational load of the algorithm, however messages need to be passed between the computational units and solutions are often non-optimal. Rule based[LB11a, KHS11, RH12] solvers use sets of rules to swap and arrange the agents until a path to the solution is available at the cost of optimality. The HCA[Sil05] algorithm implemented a reservation based system. The algorithm was designed to achieve results near real-time. The HCA algorithm can however in certain circumstances introduce collisions between agents and does not always lead to a solution.

Optimal MAPF problems are a subset of the complete solutions with minimal cost. MAPF can be solved optimally by generalizing the A* algorithm to encompass the states of all agents[Sta10, SK11]. Each node in the search space represents a configuration of all agents and a time step associated with it. Early versions of this algorithm had large branching factors for each node as the branching factor depended on the combination of moves available to the agents. Later work[YMI00, GFS+14] reduced the branching factor by segregating search space or deferring the computation of branches until later.

Our inspiration comes from an algorithm called Conflict Based Search (CBS) which approaches the MAPF problem by focusing on collisions as a means of reducing the search space [SSFS12b]. The CBS algorithm is a two layer algorithm where the lower layer computes the path of individual agent. The paths of each agent are combined to construct a possible solution. If the possibility is collision free it is considered a solution. If however there exists a collision restrictions are placed on the relevant agents and the process begins again.

Collaborative Iterative Search (CIS), the solution presented in this paper, extends the concept of A*-focused collision-based path planning but takes a greedy approach more directly focused on sparse environments. Our new approach is more akin to the Dijkstra Algorithm than A*, as new paths are constructed from old using a predefined ordering. This makes our approach methodical, but informed.

Taking advantage of map geometry and a pre-calculated distances via Reverse Resumable A*[Sil05], CIS permits an additional level of reasoning which further culls the search space of the problem without sacrificing optimality or correctness. When the underlying search graph is planar, i.e. can be drawn onto a 2D surface without overlap, we can apply the extension to the algorithm which we call Backtracking. We call this extended algorithm CISR.

Furthermore, CIS and CISR can demonstrate significant performance gains over

existing solutions when employed to solve multi-agent navigation of grids of sizes 8x8, 16x16, 32x32 and of maps drawn from existing game titles [Stu12]. In addition to the gaming maps we also test CIS/CISR against a set of bespoke maps of our own creation which are shown in appendix A.

1.4 Research Contributions

In this thesis we present a new approach to the MAPF problem, which is both optimal and complete. We present two versions of our algorithm:

- First we construct CIS as a Conflict Based approach to MAPF. CIS has a linear lower layer and forms a basis for extension.
- Second we construct a culling algorithm for CIS called backtracking. The backtracking algorithm can be applied to planar graphs to remove equivalent solutions in a region. The CIS algorithm with the backtracking algorithm is called CISR.

The contributions these algorithms bring can be summarized as:

- A graph theory re-visitation of the underlying theories of optimal MAPF.
- New algorithms which provide answers faster than the existing state of the art.
- A more statistically predictable compute time.

Benchmarks demonstrate the performance improvements over our primary comparator CBS. We show that there are significant improvements in performance when CIS and CISR are applied in sparse environments.

1.5 Publications

There has been one publication linked too this work:

- Callum Rhodes, William Blewitt, Craig Sharp, Gary Ushaw and Graham Morgan “Smart Routing: A Novel Application of Collaborative Path-finding to Smart Parking Systems." in the Proceedings of the 2014 IEEE 16th Conference on Business Informatics.

This paper modified CIS and applies it to the problem of smart parking. The problem is generalized to accommodate capacities on nodes and edges and only detects a collision if an edge contains more agents than the indicated capacity. The implementation selects only a portion of the available paths to reduce the search space. This makes the algorithm sub-optimal however the solution times become tractable.

1.6 Thesis Structure

- *Chapter 2:* We begin this thesis by exploring the subject area in general. The subject area is broken into categories with common properties or techniques. We discuss the difference in representation of the environment and highlight that

the abstract graph representation allows us to reason about optimality and completeness. Non-optimal algorithms are discussed ranging from rule-based solvers such as Push and Swap[LB11a] to the tractable HCA[Sil05]. Moving onto optimal solvers we highlight the progression from generalized A* methods of solving MAPF to the Conflict-Based approach of CBS. We then discuss our main competitor Conflict Based Search (CBS).

- *Chapter 3:* The underlying mathematical notions are then rigorously defined and then explored. We begin this exploration with the basis for CIS. We define an ordering on the set of connections from each node. This allows us to extend the ordering to the entire path. We use this structure to build a incremental version of the conflict based scheme of CBS. We then reason about the algorithm and prove its completeness and optimality.

- *Chapter 4:* In this chapter we move onto the concepts needed to understand the extension CISR. We reason about the structure of a planar graph and its simplest paths in terms of path finding which we call the Non-Complex paths. We build a theory based on these paths and construct the basis for the Backtracking Algorithm. We define the Backtracking Algorithm in the terms of these mathematical concepts and prove its optimality and completeness as an extension to CIS.

- *Chapter 5:* After the mathematical construction of the Algorithm we discuss the structure of the implementation supplying pseudo code for the most prominent portions of code. We discuss the data structures involved in the construction of the algorithm which is used later to reason about the complexity of the algorithm.

- *Chapter 6:* We then test our implementation with a range of grids; 8x8, 16x16, 32x32, varying the number of agents navigating the environment and randomizing their start and goal nodes. We also test against a set of gaming maps used in benchmarking [Stu12] and a subset of bespoke maps that we used to test the implementations as illustrated in appendix A. Comparing the results of CIS, CISR and our primary comparator CBS we show in what circumstances each algorithm will experience a higher rate of success, and highlight our algorithm's strengths in sparse environments.

- *Chapter 7:* We revisit work we did previously in a case study chapter where we describe a modified version of the CIS algorithm. Removing the restriction of optimality from CIS and generalising the approach we extend our algorithm to the problem of smart parking. Nodes and edges of the graphs are allowed up to a fixed capacity until a conflict is registered and resolved.

- *Chapter 8:* Reviewing the work up to this point we discuss some of the limitations and advantages of our approach.

- *Chapter 9:* Finally we bring the thesis to a conclusion. We discuss possible extensions to the applicability of the algorithm, improvements to optimality that could be made and extensions of the algorithm to wider contexts.

Chapter 2. Background And Related Work

2.1 Path Finding

Path finding is a fundamental field of AI. Many areas of AI can be abstracted to a Graph $G(V, E)$ of nodes V and edges E. Some problems in these areas of AI require a minimal path from a start node s to a goal node g. There are many variants of algorithms which explore this problem[CGR96].

The Dijkstra Algorithm[Dij59] was a seminal work in this area. The Dijkstra Algorithm works by assigning nodes to three separate sets; nodes yet to be analysed, nodes currently under contention and nodes which have a defined minimal distance. Labels would be assigned to each node indicating the minimal distance to that node. One of the advantages of Dijkstra is its simplicity, it requires no information other than the distance between nodes.

After Dijkstra one of the most prominent path finding Algorithms is A* as explored by Hart et al[HNR68a]. The A* algorithm uses a function $h(\mathbf{x})$ called the heuristic function. The heuristic function is an approximating guess as to the distance to the Goal. The heuristic function is never allowed to be an overestimation of the distance to the Goal. If the heuristic function satisfies this property then it can be used to influence which nodes to expand and explore next without compromising the optimality of the path.

2.2 Collaborative Path Finding

Collaborative path finding is an extension of path finding concerned with multiple entities travelling to their own destinations. This subject has many applications from traffic simulation, crowd simulation, virtual environments, robotics and abstract problem solving such as the 15-puzzle and Goal Orientated Action Planning.

Collaborative path finding is a problem which can be framed in a number of ways. The problem can be framed in 2D or 3D space with entities which avoid or attract each other. The problem can also be framed in a more abstract manner by defining a node graph where the travelling entities avoid occupying the same node while traversing the edges.

2.2.1 *Motion Planning*

When solving navigation problems for several agents we need a representation of the environment which they occupy. As discussed earlier the environment can be represented as an abstract node graph; $G(V, E)$. As we will discuss later navigation through a graph is NP-Hard .

Some of the original research into the area of multi agent path finding where collision was disallowed originated from robotics and the need for multiple agents (robots) to navigate a physical environment without colliding. The original complexity of motion

planning [HSS84] by Hopcroft et al represented the problem as 3 dimensional shapes sliding in a 3 dimensional space and was able to prove that the general problem was P-SPACE Hard.

Work by Hopcroft et al on the motion of pivoting arms [HJW84] shows that the similar problem of the 2 dimensional linkages of robotic arms. The problem is similar in nature and it is also proven that moving a joint to a given position from initial configuration is P-SPACE Hard.

Erdmann et al on sliding objects within the plane[ELP87] builds an algorithm to solve the sliding shapes and the two dimensional linkage problem. The algorithm works by prioritising each individual agent/linkage, analysing the space available for the entity to move and then planning a path for the said entity.

2.2.2 *Boids and Flocking*

An alternative approach which avoids the NP-Hard nature of MAPF is to represent the environment as a 3D space and solve the agent direction by alternative methods. One such method is flocking which originates from Reynolds Boids[Rey87]. With the use of Boids groups of agents can navigate from one area to another, however the exclusivity of a node based graph navigation scheme does not apply in the same form. A balance of forces dictate the motion of the agents forcing them to maintain an even distance. The balance of forces keep the flock together, spread apart in an even distribution and travelling towards their joint goal.

Algorithms of this nature however do not restrict their agents from colliding, collisions can occur. If these collisions are to be resolved an algorithm will be needed to detect the interface of two of the agents (intersection of the the two shapes representing the agents). An interface resolution algorithm would then be needed to correct the intersection.

Flocking is a technique which finds application in virtual environments due to its real-time nature allowing for fast answers to complex behavioural problems; Sweetser et al [SW02] review a few of the current AI techniques in computer games. A related topic to flocking is crowd simulation [MBCT98, PGT08] which simulates a realistic group behaviour, where the collective behaviour of agents is more important than the goal of individual agents. Algorithms such as flocking do not provide an optimal path planning solution, however, as there is no consideration of intelligently navigating the environment.

When a 3D environment needs the rigorous nature of graph based navigation the environment can be split into nodes representing regions or geographical locations. For a 2D surface within a 3D environment a closer approximation of the open nature can be attained by the use of Navigation Meshes[vTCG11, vTCG12]. Navigation meshes allow for path planning algorithms to be applied to 2D polygonal surfaces, however the notion of collision does not apply in the same manner as in an abstract graph based path planning algorithm. This is due to a lack of normalisation in node distribution.

Navigation meshes allow agents to pass around each other within the space of a single node of the path. This leads to similar needs for interface detection and interface resolution as in the flocking algorithms.

2.2.3 *Ant Colony Optimisation*

Taking inspiration from ant behaviour Ant Colony Optimisation(ACO)[DMC96, SH00, DG97] is a form of swarm intelligence which isan abstract approach which can be used to solve certain combinatorial problems. Ants use a set of pheromones to mark territory that can lead to useful resources. The path traced by these pheromones leads to the relevant resources however the diffusion of the pheromone naturally balances which paths are more suitable.

By breaking a general combinatorial problem into a set of finite decision variables $X_i, i \in [1, n]$ then ACO can be applied to the problem by constructing a graph which encodes these decisions. Suppose that each X_i can take a value from the set $\mathbf{D}_i = \{v_i^1, \ldots, v_i^{|\mathbf{D}_i|}\}$ then we define a set $\mathbf{C}$ called the components of the problem where each element $c_{ij} \in \mathbf{C}$ represents the assignment $X_i = v_i^j$. Each of the c_{ij} can either be used as an edge or node of the graph we use to construct the solution. A path along this graph represents a set of decisions and a possible solution.

Particular edges or vertices can be removed to represent constraints on the decision variables. Solutions are explored by simulated traversal of this graph with agents representing the ants of the ant colony. As these agents traverse the graph in a stochastic manner they lay pheromones along the edges of the graph. As the algorithm steps forwards in time the pheromone decays leaving the most efficient paths with the highest concentration of pheromone. The probabilities with which the ants travel along each edge is dependent on the levels of pheromone present however a non marked edge will still have a finite probability of being visited.

ACO has a number of advantages such as a generalized approach applicable to a number of problems. Several version of the approach have been described in literature each having their own advantages and disadvantages. The ACO approach has an intuitive implementation inspired by a system found in nature and like most optimisation techniques the solution is found iteratively therefore can be stopped at any time giving a partial optimisation.

However ACO has disadvantages towards the problem we wish to solve (MAPF). The most optimal solution may never be found as the algorithm may find itself stalled in a blind alley of optimisation. The true optimal may also not be known meaning that when it is found the algorithm shows no indication that that point has been reached. In the case of MAPF the generalized solution means that a bespoke approach is not used and many facets of the original problem are not taken advantage of. MAPF has a large state expansion and can potentially explore an infinite number of decisions (which direction does an agent choose at each time step).

2.2.4 *Multi-Agent Path Finding*

The focus of this thesis is on the more abstract approach based on graphs. A graph is a mathematical construct consisting of a set of points V called vertices and a set of edges $E \subset V \times V$ between vertices. This version of collaborative path finding is called Mulit-Agent Path Finding (MAPF).

There is no universal definition of a MAPF scenario, though prior literature has favoured a formal definition which this work employs [GFS+12]. Following the example of Goldenberg et al, this work defines the inputs to a MAPF problem as:

1. A graph $G(V, E)$ with V vertices (nodes) and E edges
2. A set K of agents $i \in K$, each with a unique start node and goal node.

Graph G is normalised such that a single incrementation of time maps to the cost of traversing a single edge. Thus at every increment of time, each agent may *move* to a neighbouring node, or *wait* at its current node. Constraints placed upon the system, in terms of mutual exclusivity, define the properties of an incomplete solution. A solution shall be considered incomplete if:

1. A node is concurrently occupied by more than one agent in any given time-step.
2. An edge has been traversed by more than one agent in-between any two consecutive time-steps.

For the purposes of this work, the detection of either condition shall be considered a *collision*. A solution shall be considered *complete* if no collisions are detected; it shall be considered both *complete* and *optimal* if no collision is detected *and* the sum total of time agents spend away from their goal node is a minimum possible value. Figure 2.1 shows an example of swap collisions and collisions on a specific node on a square 3x3 grid.

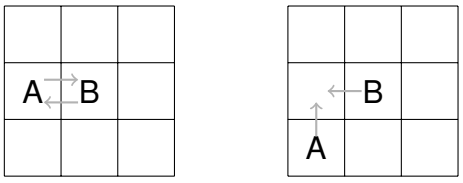


Figure 2.1: Example of the two types of collisions.

2.2.5 *Complexity*

The problem of MAPF is in general NP-Hard. The first work on the complexity of such problems comes from robotics. As discussed earlier Hopcroft et al prove that the motion planning of sliding objects[HSS84] and the motion planning of two dimensional linked arms[HJW84] is PSPACE-hard.

Later work into mathematical puzzles of the nature of the 15-puzzle are NP-Hard[JS79, RW90, RW86, Gol11]. The 15-puzzle is an example of a problem relating closer to our current work as can be seen if we consider the sliding pieces to be our agents. The Goal of the 15-puzzle is to slide the tiles in a square 4x4 grid until they reach the configuration shown in Figure 2.2. There is only ever one square free for agents to move to however the puzzle only allows certain configuration to return to the initial configuration shown in the Figure. The computing whether a solution is possible is where the NP-Hard nature of MAPF occurs.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Figure 2.2: Example of a 15-puzzle.

2.3 Non-Optimal MAPF

The approaches taken to solve MAPF be categorized into a number of groups. Depending on the focus of the solver one of several approaches can be taken to solve the problem. We will now focus on Graph based solutions to MAPF and give a brief summary of each of the main approaches to MAPF.

2.3.1 *Decentralized*

The MAPF problem naturally lends itself to a distributed processing approach. In a decentralized approach each agent of the MAPF problem is self interested and processes its own route. Collisions are avoided by communication between these agents. The decentralized approach mirrors the original focus of the research area of robotics. Each agent would have its own dedicated processing and would be inherently self interested.

In the work by Gilboa et al [GMF06] an algorithm was described which would work to uncover a hidden environment. As agents traversed the graph new nodes would be uncovered and the graph updated. Agents would communicate new nodes, their current location and future plans and then use a distributed algorithm to calculate their next move.

Work by Bhattacharya et al [BKL10] use optimization methods to calculate the result of MAPF problems. While agents are optimized individually information is shared so that the global problem can be optimized. By applying pairwise distance constraints between agents collisions are avoided.

DEC-A* by Falou et al [FBM12] uses a decentralized method based on spacial locality. The map is split into separate graphs, and the paths are solved in a decentralized approach by separating the computation among the sub-graphs.

The focus of this work is centralized techniques which consider the problem in its

entirety, meaning that agents are considered together such that an overall solution can be calculated.

2.3.2 *Rule Based Solvers*

One technique which is prevalent in congested environments is the restriction of agent movement defined by a set of rules. This allows for faster compute times at the cost of optimality.

In the work of Luna et al an algorithm called Push and Swap [LB11a, LB11b] is described. Push and Swap has two primitives, *push* will move an agent towards their goal until the agent can go no further, *swap* will interchange two agents if one of them was blocked from reaching their goal.

Push and Swap was extended by Wilde et al in an algorithm called Push and Rotate [dWtMW14]. Push and Rotate adds a new operation rotate that allows more freedom to the algorithm. The *rotate* sequence of movements allows two agents to swap in a congested cycle of agents.

The algorithm Tree based Agent Swapping Strategy (TASS) by Korshid et al [KHS11] specializes in the solving of tree based MAPF problems. Using a previous work by Masehian and Nejad [MN09] which shows a linear algorithm capable of checking the solubility of a tree based MAPF problem, TASS builds on the work by constructing an algorithm for solving such problems.

These algorithms described run in polynomial time however as they reduce the number of available options for movement they are not always applicable too all configurations, for instance the original Push and Swap was restricted to graphs which had two spaces for agents to move into.

As highlighted in work by Röger and Helmert[RH12] earlier work by Wilson [Wil74] which is further extended by Kornhauser[KMS84] fully solves non-optimal MAPF problems in polynomial time. This work comes under the name of Pebble motion on graphs and originates from a pure mathematics Graph theory perspective. Our own work is inspired by this philosophy.

2.3.3 *Reduction Approaches*

Other areas of computer science and the well known problems studied in those areas can be leveraged to compute sub-optimal solutions to MAPF problems. Work by Surynek[Sur12c] uses the boolean satisfiability problem (SAT) to optimise the sub-problem generated by existing sub-optimal solutions. The solutions used as a template for the optimization are generated by solvers such as their earlier work [Sur09] and Push and Swap [LB11b]. This work was later revisited [Sur12b, Sur12a] with a revised version of the SAT encoding employing a constraint called ALL-DIFFERENT to model the requirement that all agents must not collide.

Similarly work by Yu et al[YL13] utilizes Integer Linear Programming to compute optimal and complete solutions to congested environments. The graph which repre-

sents the MAPF problem is transformed into a network flow problem where the edges are directed and have capacities and costs associated with them. The network flow problem is then treated as a series of linear equations that need solving.

Work by Erdem et al [EKÖS13] explores the application of Answer Set Programming to the MAPF problem. A formalism is devised to solve the MAPF problem, rules are defined which must be satisfied. Heuristics are introduced to improve the efficiency of the algorithm and the quality of the solutions, but provides no explicit benefit in the computation of optimal solutions.

2.3.4 *MAPP*

By combining multiple techniques Wang et al [Wan11, WB11] design a sub-optimal algorithm MAPP which first pre-computes individual paths for each agent. When a collision occurs sliding techniques similar to that of rule based solvers such as Push and Swap[LB11a] are used.

2.3.5 *HCA*

In 2005 Silver[Sil05] contributed one of the first works (HCA) on MAPF restricted to the abstract set-up of agents occupying nodes on a graph. Silver describes a non-optimal solver with the main focus on real time answers to a given number of steps into the future. The algorithm works by maintaining a reservation table of points in space and time. As agents plan their path nodes will be reserved in the table restricting other agents from landing on those squares at the specified time. Silver also describes the use of reverse resumable A* a method that uses the g-value of A* to given an accurate gauge of distance from the goal.

HCA is not a complete algorithm. The algorithm is designed to compute solutions as close to real-time as possible. However the reservation policy can stall agents stopping themselves from ever reaching a goal and force agents to collide if there is no viable escape for them.

2.4 Optimal MAPF

2.4.1 *Costing Function*

All solvers need a method to evaluate the efficiency of the resulting solution. The costing function for MAPF problems is generally split into two concerns. First is the cost of individual paths. This can take a number of forms, for example the cost can be calculated to be the last time step that an agent moves, or the Fuel heuristic[FSBY+04] which uses the number of steps travelled and ignores any pauses during the journey to the goal.

The second part of the cost calculation is the method with which the individual costs are combined. This is generally handled by the Summation of Costs[Sta10] (SOC). With the summation of costs all the costs of the paths which combined to make the solution are summed to give a total cost.

2.4.2 *Optimal vs Non-Optimal*

The algorithms we have discussed so far have been non-optimal. An optimal solver minimizes the cost of a solution. Optimal MAPF solvers bring a number of guarantees and advantages over non-optimal solvers. While being generally more computationally expensive they guarantee completeness of the solution and minimum cost. Certain non-optimal solvers can have behavioural anomalies due to the method in which the problem is solved. This is not the case when optimal algorithms are applied as the minimality of the solutions constrains the behaviour of the agents.

Optimal solutions also give a better understanding of the nature of the problem. Optimal solutions serve as a bound for all solutions of a specific problem.

2.4.3 *Fundamental Approach to Optimal MAPF*

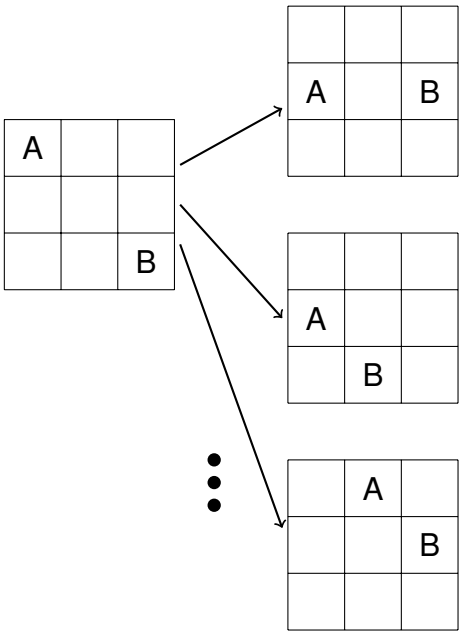


Figure 2.3: Example of the states of a generalized A* MAPF solver.

The original approach to optimal solutions to MAPF use a generalized A*[Sta10, SK11]. Each node in the generalized version of A* is a state of the map, i.e. the position of every agent on the map and the time associated with this configuration.

Each neighbouring state is a new configuration of agents on the map and each branch from that node represents one time step. The overall number of branches in this simple generalization of A* is exponential in the number of agents, i.e. roughly $O(k^n)$ where k is the number of connections to each node and n is the number of agents. This however does not take into account invalid moves; a branch can only be traversed if no two agents collide.

2.4.4 *Refinements to the A* approach*

This general approach to MAPF can be refined by a number of techniques. Standley defines an 'operator' as a possible connection between consecutive joint states of

agents. For a single agent he gives nine possibilities (eight compass directions, plus *wait*). He reduces this through *Operator Decomposition*: by splitting each operator into *n* intermediary states, each agent can be moved individually, reducing the order of the algorithm to $O(9nt)$.

Standley also provided the concept of the *Collision Avoidance Table* (CAT). This is a mechanism whereby unnecessary collisions are avoided between independent groups. All moves of other agents are added to the CAT table, which is then used to resolve tie-breaks between equal-cost choices.

The overall complexity of the problem can be reduced by splitting groups of agents into independent groups. This is called Independence Detection (ID) and can be achieved by running the solver on test groups and merging groups when conflicts occur between them. This strategy generally only incurs a constant multiple to the overall computation time because of the NP-Hard nature of MAPF.

Work by Yoshizumi et al [YMI00] on Partial Expansion A* of nodes which is later extended into Enhanced Partial Expansion A* [GFS+14] by Goldenburg et al focuses on the problem of nodes which branch a large number of times. When processing nodes with a large branching factor the branches expanded can be restricted by cost reducing the amount of processing and deferring the computation of further branches to later. Our own work utilizes a similar technique under the name of *Stems* which reduce the search through possible alternative routes when redirecting after a collision.

2.4.5 *Conflict Based Approaches*

The joint approach of these previous methods allows us to apply A* directly to the MAPF problem, however each node of the process incurs a large branching factor even when most agents have a least cost move towards the solution. Rather than applying A* to the whole problem we can split the problem into multiple processes. By solving each agent individually and then combining the agents afterwards we shift the perspective of the problem.

Sharon et al. extended Standley's work with an approach called ICTS (the *Increasing Cost Tree Search*) [SSGF11]. Taking a two-layer approach, ICTS implements a top layer which searches over potential cost distributions. The root node assigns the minimal cost to each agent such that it can reach its destination (ignoring all collisions). Each branch from a node in the top layer increments the intended cost of a single agent. Each node in the top layer is searched in increasing total cost until an overall solution is found.

The lower layer of ICTS creates a data structure for each agent called a *multi-value decision diagram* (MDD), representing all possible paths of the allotted cost for that agent. The MDD is an acyclic, directed graph from the start to the goal, with each node representing a coordinate in space-time within the graph. MDDs of increased cost can be constructed relatively easily from the MDDs of lower costs.

The lower layer search is completed by merging the MDD representations for each

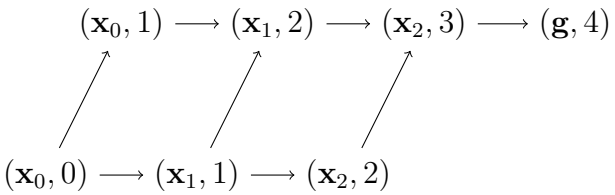


Figure 2.4: An Example MDD with a pause. The agent arrives at $\mathbf{g}$ at time step 4.

agent, removing conflicting pairs as it traverses. There exists a solution to the problem if the joint goal is reached by the merged MDDs; if the lower layer cannot find a solution the top layer creates new nodes where each agent has its allotted cost incremented. Figure 2.4 shows a simple example of an MDD where a pause is introduced to a line of nodes $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{g}$.

2.4.6 *CBS*

Sharon et al. extend ICTS into a new approach called *conflict based search* (CBS) [SSFS12a]. CBS can be seen as an evolution of ICTS, where a constraint is applied to the system rather than a distribution of costs. The nodes form a tree structure (*Constraint Tree Nodes*, or CT nodes). The constraints disallow specific actions based on an agent's situation relative to the rest of the system. For example, when a collision between two agents is detected two branches will be formed; each disallowing one of the agents from performing the action which triggered the collision. CBS is our main comparator in this thesis.

Like ICTS the CBS algorithm is a two-layer approach; the top layer assessing the cost and correctness of the *global* solution, while the bottom layer constructs individual paths following a *decoupled* schema, informed by a CAT. This provides performance benefits relative to ICTS [SSFS12a], but requires a full low-level re-computation of individual paths informed by the constraints, whenever a collision is detected.

Sharon et al further extends CBS into Meta Agent CBS[SSFS12b] or MA-CBS. In MA-CBS agents can be combined together to yield a meta agent. The meta agent then uses another MAPF solver such as the techniques from Standley and Korf. The advantage to this technique is that after a large number of collisions between two or more agents consistently the collision based scheme is less efficient as their paths are constantly being recomputed. However the more tradition A* approaches are not affected as much as CBS in these scenarios. We focus our comparison on the main algorithm CBS itself as MA-CBS needed an optimizing variable B for each case, making the comparison indirect. Future work may consider a comparison with MA-CBS and any analogies to our own work.

Boyarski et al[BFS$^+$15] introduce incremental improvements to MA-CBS with an algorithm called Improved Conflict Based Search (ICBS). The ICBS algorithm is an incremental improvement over the MA-CBS algorithm. Three modifications are made to the algorithm. First is a clean restart of the algorithm on the merging of agents

in MA-CBS, which improves the efficient integration of the new meta agent into the computation. The second improvement prioritizes which collisions to expand upon depending on the rise in cost due to the conflict. Conflicts which raise the cost are given priority as the conflict will eventually raise the cost of the solution later in the computation. The third improvement stops the low level solver from splitting paths unnecessarily by redirecting the agent when possible. This means that if possible the CT node does not need to split if the collision can be solved by recalculating the original node's path.

Barer et al[BSSF14a] take the CBS algorithm and relax the optimality condition producing 3 new algorithms with suboptimal solutions but much faster compute times. The first algorithm Greedy Collision Based Search (GCBS) relaxes the optimality condition of the upper layer search and lower layer search expanding nodes which are more likely to produce valid solutions. By applying an additional heuristic to the solution of the whole problem they GCBS takes a greedy approach which prioritizes nodes nearer the solution at the cost of overall solution value. The second algorithm Bounded Conflict Based Search (BCBS) splits the open list of the high level and low level search into different sets. The Focal set is a subset of the open list and allows the sub-optimality to be bounded by the selection of the Focal set. The last algorithm Enhanced Conflict Based Search (ECBS) improves upon BCBS by calculating appropriate bounding conditions during the course of computation.

2.5 Properties of the Graph

In this thesis we have taken the approach of analysing the underlying graph. We show that an understanding of the nature of the graph which structures the MAPF problem allows us to leverage its properties to cull equivalent solutions while improving our understanding of the area. The use of A* in MAPF is an application of the algorithm, however this application of A* does not improve our understanding of the structure of the solutions inherent in the MAPF problem.

2.5.1 *Path Ordering*

We have used two properties in the construction of our algorithm. First of all we have utilized an arbitrary preference for the connections of a node to construct an ordering[DP02a] of said connections. We can use this to construct an ordering on the paths themselves through a lexicographic ordering scheme[DP02b]. This ordering allows us to construct a conflict based algorithm which we call CIS with two layers where the lower layer has a linear order of complexity.

2.5.2 *Planar Condition*

We employ CIS as a basis to construct an algorithm CISR capable of culling regions of possibilities on a planar graph. A planar graph [Bol98] is a graph which can be projected onto a flat plane without its edges crossing. Figure 2.5 shows examples of

planar and non-planar graphs. As can be seen with the transformation μ certain graphs need to be transformed before they meet the planar condition if the planar condition can be met. The planar condition is present in all sub-graphs of square grids in which we test our results.

The planar condition allows us to contain a set of equivalent paths between two boundary paths. This allows us to search the boundary for potential routes which avoid collisions. This naturally leads to the exploration of a type of path we call Non-Complex paths. These Non-Complex paths form a structures which we reason about in order to construct the Backtracking Algorithm at the centre of our technique.

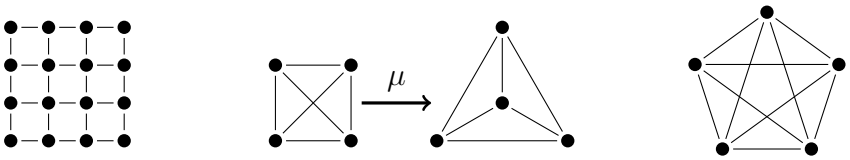


Figure 2.5: Examples of Planar Graphs.

2.6 Collaborative Iterative Search (CIS)

Collaborative Iterative Search (CIS), the solution presented in this thesis, extends the concept of A*-focused, collision-based path planning, but takes a greedy approach more directly focused on sparse environments. CIS is both optimal and complete.

An earlier work describes a modified version of the CIS algorithm for non-optimal solutions of a parking simulation[RBS+14]. The definition of a collision is generalized to include a capacity along a connection and on grid cells. Not all possibilities are explored leaving the algorithm to take a more holistic approach.

Taking advantage of map geometry and a perfect heuristic, CIS permits an additional level of reasoning which further culls the search space of the problem without sacrificing optimality or correctness. We call the extension presented in this thesis CISR.

Chapter 3. Theory of CIS

3.1 CIS Theory

In this chapter we aim to build the mathematical foundation of the CIS algorithm. We leave the pruning additions of the CISR algorithm to the next chapter. We will construct a definition of the search spaces through which we search for both the solutions of the overall MAPF problem and the search spaces used by individual agents as they navigate towards their individual goals.

We provide a purely mathematical approach as the problem is fundamentally based in graph theory as was the approach of the original Dijkstra paper[Dij59] and Coordinating pebble motion papers[Wil74], [KMS84]. In this chapter and the next we construct the notions which we need to describe the problem and build our solution. We show that a fundamental understanding of the underlying structures of a graph allow us to tailor the solution to the circumstances and improve performance.

On this basis this chapter will construct a definition of order which we apply to the paths on which agents navigate toward their goals. This is an important concept of CIS as it allows us to select an appropriate alternative for a path which collides with another agent while maintaining cost and being computationally efficient.

3.1.1 *The Problem Definition*

The MAPF problem is an abstract approach to the general problem of collaborative path planning. MAPF is defined on a graph structure $G = (V, E)$ where V is a set of vertices and E is a set of edges between the vertices in V. For our algorithm we restrict ourselves to the undirected simple graphs. This set is large and general but does not contain one-way edges, i.e. edges can always be traversed in both directions, no two nodes have more than one edge connecting them and no node has an edge looping to connect to itself. The MAPF problem is defined as a set of agents K navigating on the graph G simultaneously. With each agent $i \in K$ we associate a start node $\mathbf{s}_i \in V$ and a goal node $\mathbf{g}_i \in V$.

Definition 3.1.1 (MAPF, $G = (V, E), K$)**.** *We define the graph $G = (V, E)$ where $V = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{|V|}\}$ is a set of vertices and $E \subset V \times V$ is a set of edges. We also restrict the set E such that if $(\mathbf{x}_i, \mathbf{x}_j) \in E$ for $i, j \in [1, |V|]$ then $(\mathbf{x}_j, \mathbf{x}_i) \in E$, also we restrict E such that $(\mathbf{x}_i, \mathbf{x}_i) \notin E$ for $i \in [1, |V|]$.*

We define a MAPF problem on G using a set of agents K. For each agent $i \in K$ there exists a start node and a goal node $\mathbf{s}_i, \mathbf{g}_i \in V$.

We can also define a distance metric based on path length. This metric is our primary concept of distance and can be calculated using reverse A*.

Definition 3.1.2 (Distance, $D_{\mathbf{y}}(\mathbf{x})$)**.** *We define a distance functions given a location* $\mathbf{y}$ *on the graph* G *the value* $D_{\mathbf{y}}(\mathbf{x})$ *represents the minimum path distance from the node* $\mathbf{x}$ *to* $\mathbf{y}$*.*

Our CIS approach is a centralized scheme which solves the MAPF problem as one process however the technique shares commonality to the decentralized techniques as each agent is considered separately and then combined into one potential solution. This process is analogous to the two layer approach of algorithms such as CBS The upper layer of our algorithm works with collections of paths of the form $\mathbf{v} = (v_1, \ldots, v_{|K|})$ which may or may not contain collisions. These collections of paths are tuples; each indices of the tuple representing a path associated with a particular agent. We call a collection of paths of this form a Multiverse. We will fully define solutions and Multiverse's after we have discussed the components which constitute the Multiverses, the individual paths associated with each agent.

A path on G is a trace through the graph in time and space. The distinction between a location on the graph and a location in space-time is important for MAPF unlike other path finding algorithms. Since collisions between agents and unusual movement patterns such as pausing on a node or reversing direction are dependent on time we define our paths in terms of space-time points.

Definition 3.1.3 (Space Time Point, $\widetilde{\mathbf{p}} = (\widetilde{\mathbf{p}}_{\mathbf{x}}, \widetilde{\mathbf{p}}_t) \in \widetilde{\mathbf{V}}$)**.** *Given a graph* $G = (V, E)$ *we define a tuple of the form* $(\mathbf{x}, t)$ *to be a space time point where* $\mathbf{x} \in V$ *and* $t \in [0, \infty)$*. We use* $\widetilde{\mathbf{V}}$ *to represent the set of all space time points. We also use a special index* $\widetilde{\mathbf{p}}_{\mathbf{x}}$ *to represent the positional component of the space time point and* $\widetilde{\mathbf{p}}_t$ *to represent the time component, i.e.* $\widetilde{\mathbf{p}} = (\widetilde{\mathbf{p}}_{\mathbf{x}}, \widetilde{\mathbf{p}}_t)$*.*

Using these space time points we can define a path as a set of space time points. We restrict the set representing a path so that it can only describe a path along a graph G.

The first restriction we place on the paths is that they can only pass through a single point in space at any one time and must exist for every time step. This stops the agent from being in multiple locations at once and from disappearing from the graph when a time step is missing from the path. The resulting paths will also exist after the goal is reached which is important in MAPF as an agent can still collide with a stationary agent which has reached its goal.

We restrict paths so that each transition in a path must either be a pause or traverse an edge in G. This restriction is equivalent to the fact that the graphs that concern us are normalized as each edge takes a single time step to traverse. This together with the first restriction defines what we would consider the normal behaviour of an agent on the graph G.

However this agent could still traverse the graph G only resting on a single node for a finite number of time steps until the agent moves again. No solution can contain

a path which does not rest on its goal therefore our final restriction is that these paths have a finite number of transitions along the edges of the graph G and stop on a single node after a finite number of time steps.

Definition 3.1.4 (Path, $v \in A$)**.** *We define $v = \{\widetilde{\mathbf{p}}_0, \widetilde{\mathbf{p}}_1, \widetilde{\mathbf{p}}_2, \ldots\} \subset \widetilde{\mathbf{V}}$ a set of space time points to be a path on G if it satisfies the following properties:*

1. *The path v has a position at every time step from 0 to ∞. This position is also unique, i.e. $\forall t \in [0, \infty), \exists! \mathbf{x} \in V,$ s.t. $(\mathbf{x}, t) \in v$.*
2. *For every step in time from t to $t+1$ the path v either traverses an edge in G or pauses on a node, i.e. $\forall t \in [0, \infty), (\mathbf{x}, t), (\mathbf{y}, t+1) \in v \implies (\mathbf{x}, \mathbf{y}) \in E$ or $\mathbf{x} = \mathbf{y}$.*
3. *The path v eventually comes to rest, i.e. $\exists t \in [0, \infty), \exists \mathbf{x} \in V,$ s.t. $\forall t' \geq t, (\mathbf{x}, t') \in v$.*

We define the set of all such paths as A.

Each agent in $i \in K$ has a set of paths associated with it which starts at $\mathbf{s}_i$ and rest on $\mathbf{g}_i$. We define a subset $A_i \subset A$ to be the paths associated with agent $i \in K$ and call each of the paths in A_i the Agentverses of i. Each Agentverse is a potential solution for the agent i which excludes all other agents from consideration. This means that a Multiverse is a collection of Agentverses for each agent.

Definition 3.1.5 (Agentverse, $v \in A_i$)**.** *The path $v \in A$ is an Agentverse of $i \in K$ iff:*

1. *The path v begins at $\mathbf{s}_i$, i.e. $(\mathbf{s}_i, 0) \in v$.*
2. *The path v eventually rests at $\mathbf{g}_i$, i.e. $\exists t \in [0, \infty), \forall t' \geq t, (\mathbf{g}_i, t') \in v$.*

We define the set A_i to be the collection of all such paths associated with agent i.

For convenience we define a function $P_t(v)$ and $\widetilde{\mathbf{P}}_t(v)$ which allows us to access the position and space time co-ordinate of v at time t. We will then extend this definition to encompass all points traversed by the path. This will in essence be a projection of v onto the graph G as the positions returned will not have a time associated with them.

We also define a function $P(v)$ called the projection of v. The projection will later allow us to define subsets of similar paths based on the sets of points on the graph they form. Their projection will form partitions of paths which can be reasoned about together reducing the amount of work needed to dismiss unsuitable paths. This forms the basis of our culling algorithm in CISR.

Definition 3.1.6 (Position, $P_t(v), \widetilde{\mathbf{P}}_t(v)$)**.** *We define the function $P_t(v)$ to be the position of the agent on path v at time t, i.e. $P_t(v) = \mathbf{x} \implies (\mathbf{x}, t) \in v$. For convenience we also define the function $\widetilde{\mathbf{P}}_t(v)$ as the space time point at time t, i.e. $\widetilde{\mathbf{P}}_t(v) = (P_t(v), t) \in v$.*

Definition 3.1.7 (Projection, $P(v)$)**.** *We define a function called the projection $P(v)$. The function $P(v)$ is defined as the set of locations traversed by the path v, i.e. $P(v) = \{\mathbf{x} \colon \mathbf{x} \in V, \exists t \in [0, \infty), (\mathbf{x}, t) \in v\}$.*

We can now fully define the concept of a potential solution, i.e. a Multiverse, with respect to our definition of a path. Each Multiverse is a node on the wider search space of potential solutions to the MAPF problem but is made from the cross-product of the reduced problem search space for individual agents A_i.

Definition 3.1.8 (Multiverse, $\mathbf{v} \in S$)**.** *We define a tuple $\mathbf{v} = (v_1, \ldots, v_{|K|})$ where $v_i \in A_i$ to be a Multiverse. We represent the set of all such Multiverses by the set S, i.e. $S = A_1 \times \ldots \times A_{|K|}$*

In order to assess the correctness of a solution we need to detect the collisions which may be present in the tuple. We define the collision function $\chi(v, u)$ to be the earliest time step that a collision occurs or a half time step if the collision occurs between time steps.

Definition 3.1.9 (Collision function, $\chi(v, u)$)**.** *Given two paths $v, u \in A$ we define the collision function $\chi(v, u)$ as the smallest time step or half time step at which a collision occurs:*

$$\chi(u, v) = \min\{t \colon P_t(v) = P_t(u) \text{ or } (P_{t-1/2}(v) = P_{t+1/2}(u) \text{ and } P_{t+1/2}(v) = P_{t-1/2}(u))\}$$

When no collision occurs we define $\chi(u, v) = \infty$. We also define $\chi(\mathbf{v})$ over tuples of paths for convenience as the minimum collision over all pairs, i.e. $\chi(\mathbf{v}) = \min\{\chi(v_i, v_j) \colon i, j \in K, i \neq j\}$.

Using this definition of collision we can define the solution space as the set of all Multiverses without a collision.

Definition 3.1.10 (Solution space, S')**.** *We define the subset $S' \subset S$ to be the solution space of the MAPF problem, i.e. $S' = \{\mathbf{v} \colon \mathbf{v} \in S, \chi(\mathbf{v}) = \infty\}$.*

One last concept we need to fully describe our solution to the MAPF problem is our idea of cost. We use the Sum Of all Costs (SOC) metric to define the cost of a solution. This metric takes the individual paths and sums the costs together. However this leaves room for an arbitrary choice of how individual paths are assessed. We use a particular metric in this paper which simplifies the implementation which we call the time spent off goal, which counts the number of time steps the agent is away from its associated goal node. This has the property that any change in a path has a finite set of possible cost changes, i.e. each branch can add up to 2 units to the overall cost by moving away from the goal.

Definition 3.1.11 (Agentverse Cost Functions, $F(v)$)**.** *We define the function $F(v)$ to give the cost of an agentverse v. We use a metric where each time step spent away from the goal adds a single unit of cost to the path, i.e. $F(v) = \sum_{t=0}^{\infty}[P_t(v) = \mathbf{g}_i]$ where the $[\cdot]$ notation equals 1 when the condition inside is true and 0 when false. We define the cost of a Multiverse as the Sum Of all Costs, i.e. $F(\mathbf{v}) = \sum_{i=1}^{|K|} F(v_i)$.*

Other costing functions are possible however they require slight modification when dealing with larger cost changes.

3.1.2 *The Structure of* A_i

The set A_i of agentverses associated with an agent $i \in K$ has a complex structure. In order to highlight some of the structure inherit in these sets we can introduce a relation between the agentverses in A_i. Our approach uses the concept of an ordering on the paths in A_i. This ordering also allows us to compute the paths involved using simple greedy algorithms and search for alternatives along the ordering using linear time complexity searches.

The method by which we order these paths has several arbitrary options however we would like to include several desirable properties:

- Similar paths are close together in the ordering.
- Paths which branch from each other later in time are close together in the ordering.
- Paths with the same cost form a contiguous block together in the ordering.

We can construct an ordering which include these properties by using a lexicographic ordering scheme. We use the transitions between nodes as an implicit alphabet. We can illustrate the ordering that we implement by considering decimal numbers.

Given a path v we will construct a decimal number associated with it. We set the integer part of this representative to the cost of the path. We then consider each choice the agent makes numbering choices that lead towards the goal lower in value than choices which increase the distance to the goal. We assign an index called d_t to each choice. The index d_t ranges from 0 to the maximum number of choices for that time step, including the possibility of a pause. The form the representative then takes is $F.d_0d_1d_2\ldots$ which is unique for each path. However by assigning a particular order to the choices linked to given indices this number satisfies each of the conditions we set for the path ordering. The number representation however is not needed in the application of our algorithm as it is only illustrative of the properties of the incrementation algorithm.

We implement this strategy by applying a fixed priority mapping to each edge on the graph. For instance for a square grid we can apply a number from 0 - 4 on each of the cardinal directions.

Definition 3.1.12 (Preference, $\mathrm{Pref}(\mathbf{x},\mathbf{y})$)**.** *For any valid edge $(\mathbf{x},\mathbf{y})$ we define a preference to the edge using the function $\mathrm{Pref}(\mathbf{x},\mathbf{y})$. The preference can be dependent on the direction of the tuple $(\mathbf{x},\mathbf{y})$, i.e. the direction along which the agent travels along the edge. The preference function assigns values in a range from 0 to the maximum number of connections from the first argument. The values of the preference function do not repeat given a particular first argument.*

When we then consider an agent on a node we then convert this priority system into an index. Each node is assigned an index from 0 to the maximum number of choices. The first indices indicate the edges which lead towards the goal in order of priority, then comes the index of the pause and any edges which lead to nodes which are of equal distance to the goal, then the nodes which lead away from the goal are indexed.

Definition 3.1.13 (Indexing Function, $I_t(v)$)**.** *Suppose $v \in A_i$, we define a function $I_t(v)$ called the indexing function which indicates the index of the direction between time step t and $t+1$ with respect to preferential ordering at location $P_t(v)$, i.e. $\mathrm{Pref}(P_t(v), P_{t+1}(v))$.*

We define a number of helper functions before we define the index function. The three functions $\mathcal{D},\mathcal{R},\mathcal{P}$ will be used to help define the index function. The $\mathcal{D}_i(\mathbf{x},\mathbf{y})$ function calculates the relative distance to the goal from each end of the edge $\mathbf{x},\mathbf{y}$. Edges which are closer to the goal are prioritised over edges leading further away. The $\mathcal{R}_{\mathbf{x}}(r)$ function calculates the set of edges which have a relative distance of r from the goal and the $\mathcal{P}(\mathbf{x},\mathbf{y})$ function calculates the set of edges with a lower global preference via the Pref function.

Using these functions we define the index ordering the edges first with those leading to the goal, then the pause movement, followed by edges where each end is equidistant from the goal and finally edges leading away from the goal.

$$
\begin{aligned}
\mathcal{D}_i(\mathbf{x},\mathbf{y}) &= D_{\mathbf{g}_i}(\mathbf{x}) - D_{\mathbf{g}_i}(\mathbf{y}) \\
\mathcal{R}_{\mathbf{x}}(r) &= \{e \colon \forall e = (\mathbf{x},\mathbf{y}) \in E_{\mathbf{x}}, \mathcal{D}_i(\mathbf{x},\mathbf{y}) = r\} \\
\mathcal{P}(\mathbf{x},\mathbf{y}) &= \{e \colon \forall e \in E_{\mathbf{x}}, \mathrm{Pref}(e) < \mathrm{Pref}(\mathbf{x},\mathbf{y})\} \\
I(\mathbf{x},\mathbf{y}) &= \begin{cases}
|\mathcal{R}_{\mathbf{x}}(1) \cap \mathcal{P}(\mathbf{x},\mathbf{y})| & \text{If } \mathcal{D}_i(\mathbf{x},\mathbf{y}) = 1 \\
|\mathcal{R}_{\mathbf{x}}(1)| & \text{If } \mathbf{x} = \mathbf{y} \\
|\mathcal{R}_{\mathbf{x}}(1)| + |\mathcal{R}_{\mathbf{x}}(0) \cap \mathcal{P}(\mathbf{x},\mathbf{y})| + 1 & \text{If } \mathcal{D}_i(\mathbf{x},\mathbf{y}) = 0 \\
|\mathcal{R}_{\mathbf{x}}(1)| + |\mathcal{R}_{\mathbf{x}}(0)| + |\mathcal{R}_{\mathbf{x}}(-1) \cap \mathcal{P}(\mathbf{x},\mathbf{y})| + 1 & \text{If } \mathcal{D}_i(\mathbf{x},\mathbf{y}) = -1
\end{cases}
\end{aligned}
$$

For convenience we define the special notation of $I_t(v) = I(P_t(v), P_{t+1}(v))$ for indices along a path v. We also define a partial inverse $I(\mathbf{x},i)$ which takes an index i and returns the node $\mathbf{y}$ which holds that index with respect to $\mathbf{x}$, i.e. $I(\mathbf{x},i) = \mathbf{y} \Leftrightarrow I(\mathbf{x},\mathbf{y}) = i$.

The use of this index function allows for the greedy nature of the CIS algorithm. For each time step there are a range of values such that any choice of index in that range

constructs an optimal path to the goal from that point in space time. Each time step can be seen as a potential branch to an alternative path with a predictable cost. Therefore when given a path v which has a collision, if the path is represented in an index form we can search backwards in time from the collision for alternative routes.

We define a number of equivalence relations on the set of agentverses A_i. The equivalence relation $\sim_t$ relates all paths which agree up to time step t. When we consider a collision on the path v at time t the collision criteria would hold for any path from the equivalence class $[v]_t$.

Definition 3.1.14 (Equivalence, $\sim_t$)**.** *We define an equivalence relation $u \sim_t v$ up to time step t, i.e. $\forall t' < t, P_{t'}(u) = P_{t'}(v)$. We represent the equivalence class generated by this relation by $[v]_t$, for some representative v. We then define equivalence classes restricted to a particular f-value; $[v]_t^f = \{u \colon F(u) = f, u \sim_t v, u \in A_i\}$. The set of all equivalence classes, associated with that f-value and time step, is defined as $E_t^f = \{[v]_t^f \colon v \in A_i\}$.*

The preferential ordering of paths can now be defined with respect to the index function and the equivalence relation. The structure of the ordering reflects the nature of our search algorithm. When we search backwards in time for an alternative branch an index would never need to be decremented in order to search for the solution of the same cost as each step along the ordering has ruled out portions of the order due to earlier collisions.

Definition 3.1.15 (Preferential Ordering, $<_i$)**.** *We define $<_i$ to be a total ordering on the set of agentverses A_i. We define $u <_i v$ if and only if either:*

1. $F(u) < F(v)$*, or*
2. $F(u) = F(v)$*, and* $\exists t,$ *s.t.* $u \sim_t v, I_t(u) < I_t(v)$*.*

The structure of the preferential ordering can be extended to the equivalence classes. We will use this similarity between paths and equivalence classes to show that steps along the ordering which we call incrementations are mapped to the ordering in the same manner and in most cases the concept of a path and equivalence class can be swapped without consequence.

Lemma 3.1.16 (Preferential Relation Extension)**.** *The preferential relation can be extended to the equivalence classes. i.e. we define $[u]_t^f <_i [v]_t^f$ iff $[u]_t^f \neq [v]_t^f$ and $u <_i v$. This relationship is well formed.*

Proof. For this relationship to be well formed we need to show that this relationship holds for any representative of the class. Suppose $u' \in [u]_t^f$ and $v' \in [v]_t^f$. Since $[u]_t^f \neq [v]_t^f$ there exists $t' < t$ such that $P_{t'}(u) \neq P_{t'}(v)$, let t' be the minimum such value, i.e. $\forall t'' < t', P_{t''}(u) = P_{t''}(v)$. This implies $u \sim_{t'} v$ and by the definition of the relation since $P_{t'+1}(u) \neq P_{t'+1}(v)$ we must have $I_{t'}(u) < I_{t'}(v)$ for the relationship $u <_i v$ to hold.

However since $t' < t' + 1 \leq t$ and $u' \in [u]_t^f$ we must have $P_{t'+1}(u') = P_{t'+1}(u)$, similarly for v', which implies $I_{t'}(u') = I_{t'}(u) < I_{t'}(v) = I_{t'}(v')$. Which proves that this relation holds for any representative. □

3.1.3 *Path Incrementation*

The CIS algorithm is based upon the concept of incrementation. Whenever a collision occurs we find the next path in the ordering which avoids the collision. To achieve this we use an incrementation algorithm. Each incrementation is capped at the time of the collision with the other agent. We find the last index which can be incremented to achieve a path of the same cost. We define the top branch to be the last time step before the collision which can branch by incrementing an index creating an alternative path. We define a helper function $Q_t(v, i)$ which defines the additional cost the path v gains from taking index i. This means indices which describe the most efficient route return 0 while movement equivalent to a pause result in the value 1 while movement which backtracks away from the goal return 2.

Definition 3.1.17 (Relative Priority, $Q(\mathbf{x}, i)$)**.** *Suppose $v \in A_i$, we define a function $Q(\mathbf{x}, i)$ which evaluates to the relative priority change from taking the direction indexed by i, at position $\mathbf{x}$, i.e. $Q(\mathbf{x}, i) = D_{\mathbf{g}_i}(I(\mathbf{x}, i)) - D_{\mathbf{g}_i}(\mathbf{x}) + 1$. We also define a convenience function $Q_t(v) = Q(P_t(v), I_t(v))$. Using the helper function $\mathcal{D}$ from the definition of the index function we can define $Q_t(v) = 1 - \mathcal{D}(P_t(v), P_{t+1}(v))$.*

We also define idea of a truncated cost $F_t(v)$ as the cost of the path v up until time step t but the remainder of the cost being the time taken to reach the goal. This function is used to define the cost of a branch in the path v as the cost of the branching path will agree with v up until the time of the divergence between the two paths.

Definition 3.1.18 (Truncated cost, $F_t(v)$)**.** *We define the truncated cost function $F_t(v)$ as the cost of a path v up to the time t such that any deviations from moving towards the goal after time step t are ignored, i.e. $F_t(v) = D_{\mathbf{g}_i}(\mathbf{s}_i) + \sum_{t'=0}^{t} Q_{t'}(v)$*

Definition 3.1.19 (Top Branch, $\beta_t^f(v)$)**.** *The function $\beta_t^f(v)$ represents the last time step from which agentverse v can branch to an agentverse of cost f, before time step t. i.e: $\beta_t^f(v) = \max\{t' : t' < t, Q(P_{t'}(v), I_{t'}(v) + 1) + F_{t'}(v) = f\}$.*

Using the above ideas we now define the incrementation algorithm which we call *Next*. We have restricted ourselves to incrementing an index by one since subsequent incrementations will find all indices with the correct cost. We do however skip indices which would raise the cost. This will be corrected later by the inclusion of the concept which we call the stem.

Definition 3.1.20 (Next, $N_t^f(v)$)**.** *$N_t^f(v) = u$ is a partial function and defines a new agentverse u with the following properties:*

1. $v \sim_{\beta_t^f(v)} u$.

2. $I_{\beta_t^f(v)}(v) + 1 = I_{\beta_t^f(v)}(u)$.

3. $\forall t' > t, I_{t'}(u) = 0$.

The *Next* algorithm has the important property that it serves as a strict incrementation on the equivalence classes themselves. Using *Next* N_t^f on an element v of an equivalence class $[v]_t^f$ of the correspondence time cap t will increment the result to a new equivalence class $[u]_t^f$. However the result will be the least element of the new equivalence class and no equivalence class will have been skipped over to reach this value. These properties are important as this shows that no potential solutions are skipped over when we increment.

Lemma 3.1.21 (Equivalence Incrementation)**.** *When the value of $N_t^f(v)$ exists, there does not exist an equivalence class $[u]_t^f$ with the following properties, $[v]_t^f <_i [u]_t^f <_i [N_t^f(v)]_t^f$.*

Proof. For such an equivalence class to exist we need to find an agentverse u which satisfies $v <_i u <_i N_t^f(v)$. Let $\beta = \beta_t^f(v)$, by the definition of N_t^f we have $v \sim_\beta N_t^f(v)$. Let t' be the maximum time for which $v \sim_{t'} u$ holds. Suppose $t' < \beta$, then $I_{t'}(u) > I_{t'}(v) = I_{t'}(N_t^f(v))$ since $v \sim_\beta N_t^f(v)$. However this contradicts $u <_i N_t^f(v)$, therefore $t' \geq \beta$.

The fact that $t' \geq \beta$ implies that $u \sim_\beta N_t^f(v)$ since $u \sim_{t'} v \sim_\beta N_t^f(v)$. However by the definition of β_t^f, β is the last time step at which the correct cost f can be achieved. Otherwise if $t' > \beta$ this implies $F(u) > F(N_t^f(v))$ which in turn implies $u >_i N_t^f(v)$, therefore $t' = \beta$.

From $v <_i u <_i N_t^f(v)$ we have $I_\beta(v) < I_\beta(u) < I_\beta(N_t^f(v)) = I_\beta(v) + 1$. But since I_t^f is always an integer, we have a contradiction, therefore there is no equivalence class between $[v]_t^f$ and $[N_t^f(v)]_t^f$. □

Lemma 3.1.22 (Equivalence Minimum)**.** *When $N_t^f(v)$ exists, $N_t^f(v)$ is the minimum, w.r.t the $<_i$ relation, of the set $[N_t^f(v)]_t^f$. i.e. $\forall u \in [N_t^f(v)]_t^f, N_t^f(v) <_i u$ or $N_t^f(v) = u$.*

Proof. Given the agentverse v the top branch $\beta_t^f(v)$ is defined to be less than t. For all time steps t' more than $\beta_t^f(v)$ we have $I_{t'}(N_t^f(v)) = 0$. All elements of $[N_t^f(v)]_t^f$ share the same initial t time steps; i.e. $\forall u \in [N_t^f(v)]_t^f, u \sim_t N_t^f(v)$. So if u differs from $[N_t^f(v)]_t^f$ it must do so after time step t. Supposing $t' \geq t, I_{t'}(u) \neq 0 = I_{t'}(N_t^f(v))$ then since I_t^f is always positive we must have $I_{t'}(u) > I_{t'}(N_t^f(v))$ which implies $v <_i u$, as needed. □

We can also show a property mirroring the previous two, that every incrementation is the minimum of some equivalence class. This also shows that all paths that result from *Next* are finite and still satisfy the conditions required to be a path in definition 3.1.4.

Lemma 3.1.23 (The Resting Equivalence Class)**.** *If agentverse v is reaches its goal and stops, then there exists a time t such that v is the minimal element of $[v]_t^{F(v)}$.*

Proof. Since the agentverse v eventually rests there must be a time t beyond which $I_s(v) = 0$ for $s > t$. Let t' be the maximal time at which $I_{t'}(v) \neq 0$. Now construct an agentverse u such that $u \sim_{t'} v$, and $I_{t'}(u) = I_{t'}(v) - 1$. But by the definition of the top branch $\beta_{t'+1}^{F(v)}(v)$, t' is a branch which gives an agentverse of the correct cost ($F(v)$), but t' is also the maximum time step we consider. Therefore we have $N_{t'+1}^{F(v)}(u)$ and by Lemma 3.1.22, this means v is a minimum of $[v]_{t'+1}^{F(v)}$. □

We highlight a special path called the α path. This path is defined to be the path of minimal preference in a given Verse set A_i. We will show that this agentverse is the root of all agentverses and that all agentverses can be generated by a sequence of *Next* invocations. We also introduce the idea that a path v covers a path u when there exists a sequence of *Next* invocations which transform v into u. The result showing that the α path is the ancestor of all paths shows that it also covers all of these paths.

Definition 3.1.24 (α path, α_i)**.** *The α_i path is the unique minimum of the preferential ordering $<_i$, i.e. $\forall v \in A_i, \alpha_i <_i v$, or equivalently $\forall t, I_t(\alpha_i) \equiv 0$.*

Corollary 3.1.25 (Ancestor)**.** *Every agentverse v other than α_i has an ancestor u such that for some t, $N_t^{F(v)}(u) = v$.*

Proof. If an agentverse v contains a time t such that $I_t(v) \neq 0$ then we can construct an agentverse u as we did in the last lemma. Otherwise the index function is identically 0, which is the definition of the α_i agentverse. □

Definition 3.1.26 (Cover, $C(v)$)**.** *The agentverse u covers agentverse v iff there exists a sequence of agentverses, $u_0, u_1, \ldots, u_n$ with the following properties:*

1. *$u_0 = u$ and $u_n = v$.*
2. *$u_{i+1} = N_{t_i}^{f_i}(u_i)$ where $f_{i+1} \geq f_i$.*

Note that v covers v when $n = 0$. We define the set of all agentverses covered by v by the function $C(v) = \{u \colon v \text{ covers } u\}$. We say that a set of agentverses covers an agentverse v if any of its elements covers v. We also extend the definition to tuples piece wise. i.e. (u_i, u_j) covers (v_i, v_j) iff u_i covers v_i and u_j covers v_j. Similarly a subset $W \subset A_i \times A_j$ covers $\mathbf{v} = (v_i, v_j)$ if one of its elements covers $\mathbf{v}$.

Lemma 3.1.27. *The α_i agentverse covers every element of A_i.*

Proof. Given an agentverse $v \in A_i$. Consider the sum of index values; $\sum_{t=0}^{n} I_t(v)$ (call this sum $S(v)$), where n is the max non-zero time step for v (via Lemma 3.1.23). By Lemma 3.1.25 there exists an agentverse u such that $N_t^{F(v)}(u) = v$. By construction the only difference between these two agentverses is at time step t', $I_{t'}(u) = I_{t'}(v) - 1$, which implies $S(u) = S(v) - 1$.

If we continue in this manner creating ancestors $u_k = N_{t_k}^{F(u_k)}(u_{k+1})$, and so forth, we will eventually reach a point where the sum is zero; $S(u_n) = 0$. Since all indices are positive this implies that the index function of u_n must be identically 0, which is the definition of α_i. Therefore, via the sequence u_k, α_i covers v. □

3.1.4 *Completeness and Optimality*

We can now work towards a proof that our algorithm is complete and optimal. Each iteration of the top layer of our algorithm brings the process closer to the solution. This initial multiverse covers all multiverses and as such covers all solutions. As the algorithm progresses we create a chain of sets of multiverses we call working sets. Each working set in the chain removes multiverses which have collisions with a set of alternatives which cover all solutions the original multiverse did. Each set in the sequence is related by a single multiverse being replaced by a set called a branching set.

Definition 3.1.28 (Working set, W)**.** *The working set W is a finite subset of the search space $S = A_1 \times \cdots \times A_{|K|}$. The working set represents the currently considered possibilities for solutions. Each element $\mathbf{v} \in W$ is a Multiverse. During the main loop of the algorithm we remove the lowest element (sum of individual costs) and replace it with a branching set; defined next.*

Definition 3.1.29 (Branching set, B)**.** *Given a tuple of agentverses $\mathbf{v}$, we call B a branching set of $\mathbf{v}$ if it satisfies the following properties:*

1. *$\mathbf{v}$ is excluded from B, i.e. $\mathbf{v} \notin B$.*
2. *$\mathbf{v}$ covers B, i.e. $B \subset C(\mathbf{v})$.*
3. *B covers all solutions that $\mathbf{v}$ covers, i.e. $C(\mathbf{v}) \cap S' \subset C(B) \cap S'$.*

We now use the idea of a Branching set to prove completeness and optimality. Using a cap on the length of paths we consider we can prove that the working set sequence is finite since the set of possiblities is reduced by each branching set replacement. Because of the properties of branching sets no solution will be skipped over therefore an optimal solution will eventually be found.

Theorem 3.1.30 (Incremental Limit)**.** *Suppose we have a sequence of working sets W_k defined as follows:*

1. *$W_0 = \{(\alpha_0, \ldots, \alpha_n)\}$.*
2. *Given W_k we select the minimal element $\mathbf{v}$. If $\mathbf{v}$ is a solution we stop. If this element was not a solution we calculate a branching set B and produce $W_{k+1} = (W_k - \{\mathbf{v}\}) \cup B$.*

If a solution exists this process will end in a finite number of steps and produce a minimal solution.

Proof. Suppose $\mathbf{u}$ is an arbitrary minimal solution with cost C. Take the maximum time step t where $I_t(u_i) \neq 0$ of all agents $i \in K$. Since G is finite there exists a node with

the maximum number of edges, call this number of edges c. This means $I_t(u_i) \in [0, c]$ which implies there are only a finite number of combinations of unique agentverses which have non-zero indices below t ($(c+1)^t$ combinations, or $(c+1)^{tn}$ for all agents). We will call this set the restricted cover.

The restricted cover contains at least one minimal solution, i.e. $\mathbf{u}$. Given a multiverse $\mathbf{v}$ if B is the branching set of $\mathbf{v}$ then B covers less elements of the restricted cover, since at least $\mathbf{v} \notin B$ and $B \subset C(\mathbf{v})$. This implies that the sequence of working sets must end, since at some point at most one element would be left; the unselected solution $\mathbf{u}$ which would end the process returning $\mathbf{u}$ as the solution.

We now need to prove this solution's optimality. Suppose a non optimal solution $\mathbf{v}$ were selected. This suggests that a non optimal solution was part of the working set and was considered the minimal element. However by the definition and use of the branching set, the working set must always cover all solutions. This implies an element of the working set covers the true minimal solution $\mathbf{u}$, call this element $\mathbf{v}'$. Since $\mathbf{v}'$ covers $\mathbf{u}$ we must have $\forall i, v'_i \leq_i u_i$ which implies the corresponding cost satisfies $F(v'_i) \leq F(u_i)$. Summing these costs to get the cost of the element $\mathbf{v}'$ we get; $F(\mathbf{v}') = \sum_i F(v'_i) \leq \sum_i F(u_i) = F(\mathbf{u}) < F(\mathbf{v})$, contradicting the fact that v was minimal in the working set.

This proves the theorem that an optimal solution will be selected in a finite number of steps. □

We have proven the process can work if a suitable Branching set exists. We construct our branching set using our incrementation algorithm *Next*. However the cost targeted nature of *Next* requires us to take account of higher cost solutions which may have been skipped over if we do not include them in the search. We define a Stem to be a set of incrementations of a given path at higher and higher costs. When implemented the Stem does not need to be fully computed, although it is hypothetically infinite in size, only the lowest cost possibility needs consideration at any given time. Using the stem we define an appropriate branching set and complete the proof that CIS is optimal and complete.

Definition 3.1.31 (Stem, $\mathcal{S}_t^f(v)$)**.** *We define the stem* $\mathcal{S}_t^f(v) = \{N_t^f(v), N_t^{f+1}(v), \ldots\}$.

Using a Stem we can define an appropriate branching set which removes the collision from consideration. We first show what form the cover of the stem takes and then define a branching using the stem as a basis using a substitution function $\mathrm{Sub}_i(\mathbf{v}, B)$ which replaces the agentverse representing i by the elements of B.

Lemma 3.1.32. *The stem set of agentverses* $\mathcal{S}_t^{F(v)}(v)$ *covers all agentverses that* v *covers except for a subset of* $[v]_t$*. i.e.* $C(v) - [v]_t \subset C(S_t^{F(v)}(v))$.

Proof. Given $u \in C(v)$, suppose no element of the cover sequence has $\beta_{t_k}^{f_k}(v_k) < t$, then the index set of u is the same as that of v until at least time step t. This implies $u \in [v]_t$ which contradicts our assertion. Therefore there must exist k such that $\beta_{t_k}^{f_k}(v_k) < t$.

Select the first k which satisfies this property. All elements before k have a top branch greater than t and therefore only differ from v after time step t. However $N_{t_k}^{f_k}(v_k)$ only depends on the first $\beta_{t_k}^{f_k}(v_k)$ time steps, which excludes all the changes from the first $k-1$ iterations. Therefore we can redefine the sequence starting from step k, erasing the first $k-1$ steps. Conversely we can insert $N_t^{f_1}(v)$ as step one if $\beta_{t_1}^{f_1}(v) < \beta_t^{f_1}(v)$ since by the same logic this will only change indices which are above $\beta_{t_1}^{f_1}(v)$. This proves that $\mathcal{S}_t^f(v)$ covers u since $N_t^{f_1}(v) \in S_t^f(v)$. □

Theorem 3.1.33 (General Universal Branching)**.** *If we define the general branching set* $B = \mathrm{Sub}_i(\mathbf{v}, \mathcal{S}_t^{F(v_i)}(v_i)) \cup \mathrm{Sub}_j(\mathbf{v}, \mathcal{S}_t^{F(v_j)}(v_j))$*, where:*

$$\mathrm{Sub}_i(\mathbf{v}, B) = \{(v_0, \ldots, v_{i-1}, b, v_{i+1}, \ldots, v_n) \colon b \in B\}$$

Then B *forms a proper branching set when* $\chi(v_i, v_j) = t$*.*

Proof. If v_i, v_j clash at time step t, then so do all combinations of agentverses from $[v_i]_t$ and $[v_j]_t$, since they all share the position at time t. By Lemma 3.1.32 our branching set either uses elements which exclude a subset of $\mathrm{Sub}_i(\mathbf{v}, [v_i]_t)$ or elements which exclude a subset of $\mathrm{Sub}_j(\mathbf{v}, [v_j]_t)$. These elements all contain clashes at time step t and therefore do not contain solutions. Therefore B covers all solutions. B also does not contain $\mathbf{v}$, and is covered by $\mathbf{v}$. Therefore B is a branching set and Theorem 3.1.30 applies. □

Chapter 4. Theory of CISR

4.1 Introduction

In the previous chapter we built a theory containing all the paths available to an agent called CIS. We will use this algorithm as a basis for further work. The CIS algorithm considers each path equally by placing all paths in an ordering. The minimal increment through the ordering is used to attempt to avoid collisions. However this strategy does not always succeed in removing the collision.

CIS and several of the competing algorithms may have a number of recurrent behaviours. In these recurrent behaviours a collision or deadlock is considered more times than necessary. These behaviours may occur in a number of ways from choosing/computing equivalent paths with a pause shifted through time to enclosed regions of equivalent paths being explored when all paths in the region fail.

The CIS algorithm is a good basis for extension as the base computation of the paths is linear in nature. This is the minimal cost that a multi agent path finding algorithm can search for a new candidate path once the amortized calculation of distances through reverse A* is taken into account. The CIS algorithm also serves as a uniform basis for constructing culling algorithms based on these recurrent behaviours for improving performance because of its uniform treatment of paths.

4.1.1 *Recurrent Behaviour*

We define a recurrent behaviour as an aspect of a MAPF algorithm which repeats the work of solving a sub-problem of the current overall MAPF scenario. This repeated work may come in the form of a single search node which is equivalent to another attempted search or a number of complete path searches which repeatedly encounter conflicts from the same set of sources.

We use the idea of recurrent behaviour as an informal measure of the strengths of various MAPF algorithms. Different algorithms may remove or mitigate the recurrent behaviour by identifying the behaviour or computing results from previous ones. For instance the CBS removes collision space time points by adding a constraint, however this requires the re-computation of paths via the A* algorithm.

In this section we identify a number of recurrent behaviours that CIS encounters and compare this to how ICTS and CBS are effected by these behaviours. Then we describe the backtracking algorithm and explain why it is an effective strategy against certain types of recurrent behaviour and mitigates other kinds.

Collision

The simplest form of recurrent behaviour is a collision. An algorithm may attempt to remove a particular collision in space and time from a pair of paths by recalculating

one of the paths. This does not however guarantee that the recomputed path does not collide in the same location in space time. Depending on the strategy used to avoid collision several configurations of time complexity verses recurrent collision mitigation is possible.

Consider the CIS algorithm, when a collision occurs the algorithm performs a linear search for an alternative path. This alternative path is the first path of appropriate cost nearest the time of collision, however this does not guarantee that the new path does not collide at the same point in space and time. In a sparse system where a variety of alternatives occur the CIS algorithm is effective as only slight deviations from the collision are likely to solve the collision.

If we consider the ICTS algorithm each collision cannot be taken in isolation as all possibilities of a particular cost are considered at the same time. The ICTS algorithm builds a data structure called the multi-value decision diagram (MDD). The MDD will not reach the goal node of the problem when collisions occur on every path to the goal. Therefore ICTS avoids collisions at the cost of exploring all paths of a given cost. Even if the agent does not encounter a collision all paths are considered, this is a recurrent behaviour although it is spread over one computation.

Pause Migration

There are many patterns that can form in the sets of paths used to explore a graph for a solution. Many of these patterns can lead to the same collisions and conflicts between agents repeatedly causing the conflict to propagate through the search. One such example of a continual state of collision is what we call *Pause Migration*.

Pause Migration occurs when a search algorithm attempts to find an alternative to a route which contains a pause. If the only alteration that happens to the path is such that the pause is shifted backwards in time this process is called *pause migration*. When the CIS Algorithm attempts to find an alternative to such a path the algorithm will first search for paths which have the same cost but avoid the collision. This will force the algorithm to consider a set of paths which pause at earlier and earlier time steps until the start of the path is met.

Figure 4.1 shows an example of this behaviour using a space time graph. The x-axis represents each of the nodes visited by the path v and the y-axis represents the time step at which the path visits that node.The highlighted path is the original path v:

$$v = \{(\mathbf{x}_0, 0), (\mathbf{x}_1, 1), (\mathbf{x}_2, 2), (\mathbf{x}_3, 3), (\mathbf{x}_4, 4), (\mathbf{x}_5, 5), (\mathbf{x}_5, 6), (\mathbf{x}_6, 7)\}$$

The pause can be seen to be the vertical section of the graph as the path maintains its position for a single time step. However as the algorithm explores alternatives to the path v because of a collision after time step 6 the pause moves backwards in time. This behaviour is indicated by the grey arrow: $\longleftarrow$.

Consider a collision which occurs before the pause in v. To resolve the collision

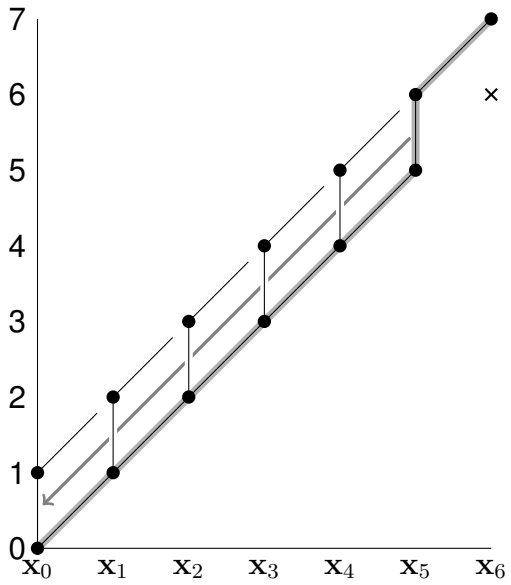


Figure 4.1: A graph illustrating Pause Migration. The x-axis represents a location on the Graph G and the y-axis represents the time an individual path visits that location. The x at $(\mathbf{x}_6, 6)$ marks the original point of collision.

the original pause can be moved backwards in time to the point before the collision as this will have an effect on the state of the path at the time of collision. However if a collision occurs after the pause in time then moving the time at which the pause occurs backwards in time whilst keeping the original positions of the path v will not have an effect on the collision which we are trying to resolve.

Bypasses and Backtracking

In a planar graph where the connections between nodes cannot cross another form of recurrent behaviour can occur. If we consider a region bounded by two paths which meet at a point of collision then all paths that lie between these two paths also lead to that collision. Figure 4.2 shows an example of recurrent selection of paths of this nature. If the graph is planar all paths which escape this fate must cross one of the bounding paths and leave the region in question. We call a region of this form a bypass.

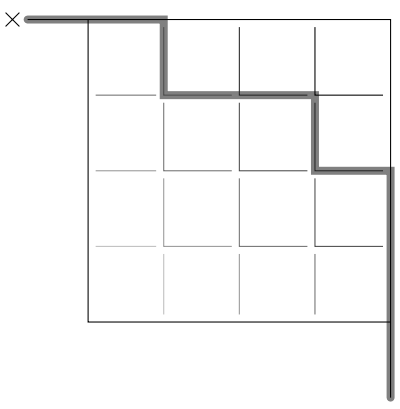


Figure 4.2: Bypass

The Backtracking Algorithm we introduce in this chapter takes advantage of the form that a bypass takes. Using the bounded form of the bypass we can search this boundary for possible alternatives to the paths which lie in the bypass. The outer boundaries can be traced by selecting a cyclic direction, either clockwise or anti-clockwise, and searching for the first connection towards the start node. Every step towards the start node the backtracking algorithm searches for a connection which leads out of the bypass region and heads towards the goal. If such a connection is found it is considered a potential solution.

From this point onwards the backtracking algorithm tries to connect this potential solution to the original path. This is done by reversing the cyclic direction of the backtracking selection of nodes, i.e. from clockwise to anti-clockwise or visa versa. The algorithm will eventually meet the original path creating a new path which leads away from the collision.

This process works because the planar condition ensures that all other paths leading out of this region must cross the boundary at a node meaning that the backtracking search will detect it. The algorithm also removes pause migration as it can be seen as a sub-case of the same process although with the boundary paths being the same as the original path itself.

Comparisons with Backtracking

Backtracking targets the most frequent recurrent behaviour of CIS and is the focus of our approach to improving the performance of the CIS algorithm. In this section we compare the backtracking algorithm to two approaches to MAPF. The generalized A* approach and the collision based approach of CBS.

Generalized A*

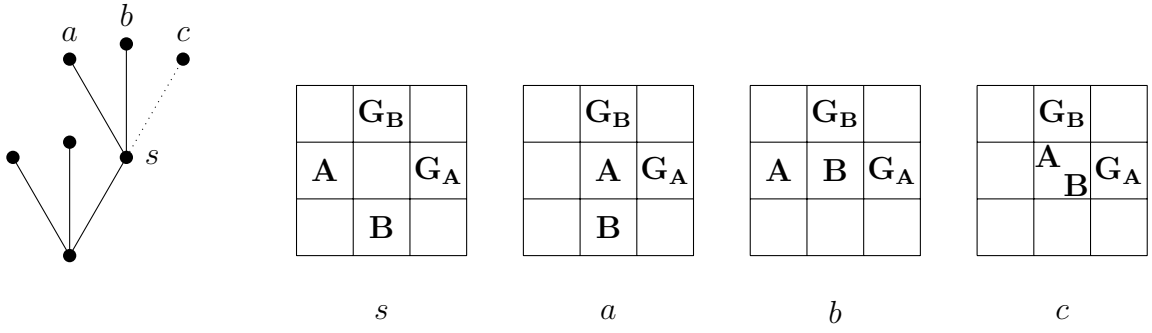


Figure 4.3: Comparison of methods: Generalized A*.

The most basic approach to optimal MAPF comes from generalizing A* to multiple agents. Each node of the graph which A* explores contains a configuration of agents. Each branch is a step forwards in time for the whole system moving each agent to a new position. As the Generalized A* algorithm explores the graph of configurations it

will ignore branches which lead to invalid configurations or transitions such as agents colliding on a node or between nodes during transition.

Figure 4.3 shows a simplified example of this approach. Each node on the left representing the positions of several agents. The node s tries to transition to a number of possibilities a, b, c but finds that c has a collision and therefore is ignored (indicated by the dotted line).

This generalized approach can be extended and modified however at its core it explores a multitude of branches at each time step. The branching cost for each node can grow exponentially with the number of interacting agents.

CBS

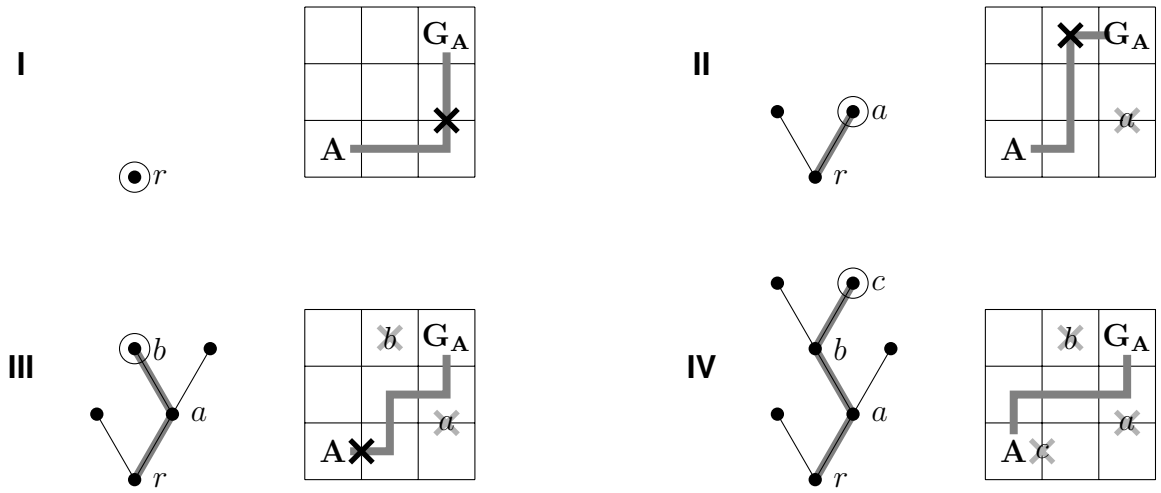


Figure 4.4: Comparison of methods: CBS.

The CBS algorithm takes a retroactive approach of reconciling collisions after they occur rather than the proactive approach of planning every time step. Figure 4.4 shows an example of CBS in action. Each of the diagrams (**I**-**IV**) represent one pass of the main algorithm and shall be used to illustrate the CBS algorithm. The CBS algorithm first computes trial paths for each agent using the standard A* algorithm ignoring interaction between the agents. The CBS algorithm then searches for a collision between the paths it has generated.

Figure 4.4(**I**) shows the initial state of a single agent A. A collision occurs between time steps 2 and 3 due to a swapping action with another agent. The CBS algorithm builds a tree representing constraining factors as shown on the left of the figure. The initial tree only contains the root which adds no constraints to the graph.

Once the collision has been detected constraints are added to the tree. One branch for each agent involved in the collision. The branches represent mutually exclusive possible solutions to the collision. Figure 4.4(**II**) shows the constraint labelled as a and how it applies to the agent A. The constraint stops the agent from making the same choice which caused the collision forcing the agent to take a different route. The A*

algorithm is rerun for agent A resulting in another path.

The process repeats and another collision is detected at timestep 3. Additional constraints are added as leaves to the node a and represent additional constraints on top of the constraint given by a. Figure 4.4(**III**) shows the addition of constraint b and how it affects the computation of the path for A. When considering a constraint in the constraint tree all constraints below must be taken into consideration.

These constraints in essence modify the underlying graph, meaning that the A* path finding algorithm has to be run again from the beginning taking into account the new graph which has been pruned of edges or nodes. The constraint tree is searched for a constraint set which will erase all collisions from the graph. Each node gains two children when a collision occurs to add constraints to either agent restricting them from colliding on that location at that time from that node onwards. Figure 4.4(**IV**) shows the finished configuration leading to the solution given.

The CBS algorithm has the advantage over Generalized A* that it only needs to recompute the path of an agent when a collision occurs. The less collisions a system of agents has the more efficient the algorithm is. Conversely in a congested environment with many collisions the algorithm will need to recompute paths numerous times, also the longer the distance the further the A* algorithm has to compute the agent meaning large environments can also have a detrimental effect.

CISR

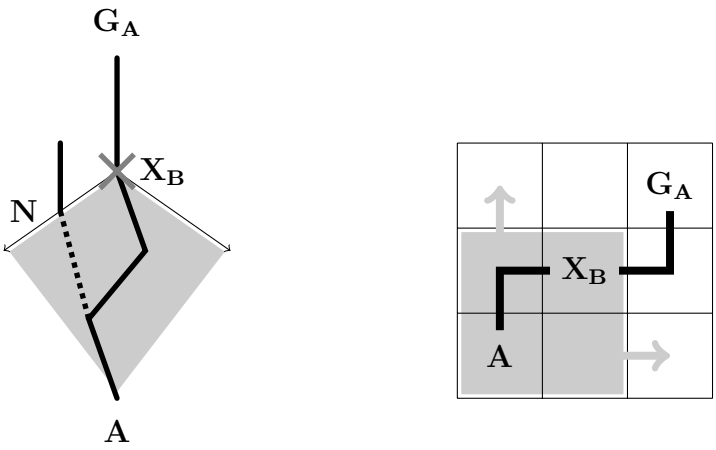


Figure 4.5: Comparison of methods: CISR.

The CISR algorithm works in a similar manner to CBS, modifying paths which collide with other agents in order to avoid the collisions. Whereas the CBS algorithm essentially modifies the graph in order to remove collisions, the CISR algorithm modifies the given path. By searching for an optimal side step around the collision the work done to avoid the collision is minimal negating the need to use A* altogether. The CISR algorithm steps back from the point of collision looking for connections to nodes which guarantee a path around the point of collision.

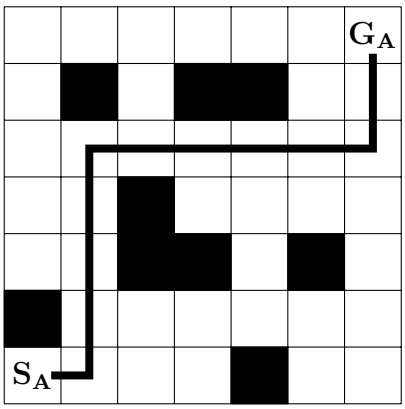


Figure 4.6: Planned path for agent A.

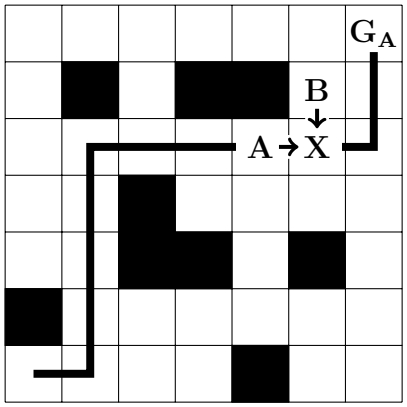


Figure 4.7: Collision between agents A and B along the path of A. Collision happens in the next time step at X.

Figure 4.5 illustrates the CISR algorithm in action. The diagram on the left shows an abstraction of the process without nodes. The agent $\mathbf{A}$ travels towards goal $\mathbf{G_A}$ and collides at point $\mathbf{X_B}$ with agent $\mathbf{B}$. From this collision point $\mathbf{X_B}$ the CISR algorithm steps backwards in time searching for a connection to a node which bypasses the collision. There is a region of nodes indicated by the grey area which may still reach the point of collision. By tracing the edge of this region the CISR algorithm can guarantee a path which avoids the collision by selecting a node which escapes this grey region. The new node is then connected to the old path indicated by the dotted line. The right side of the figure shows a simplified version on a square grid.

Using the next few diagrams we shall illustrate the process of backtracking for a solution in detail. Figure 4.6 shows an arbitrary path for agent A. Agent A starts at the start node indicated by $\mathbf{S_A}$ and travels until the goal node marked $\mathbf{G_A}$. Agent A travels along the path in Figure 4.7 colliding with agent B on the node marked as $\mathbf{X}$.

Once the collision has occurred if the path is suitable then backtracking can occur. The algorithm attempts to find alternative routes around the point of collision. An alternative path of the same cost is needed by the algorithm to proceed or proof that no such path exists. Both the left side of the collision and the right side need to be searched for a viable sidestep around the collision. The backtracking algorithm does not search backwards along the original path. The area searched encloses all nodes

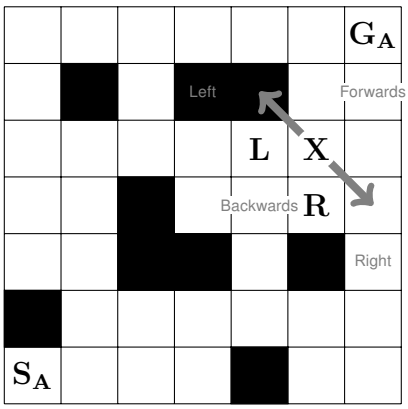


Figure 4.8: Indication of the left side and right side of a agent A.

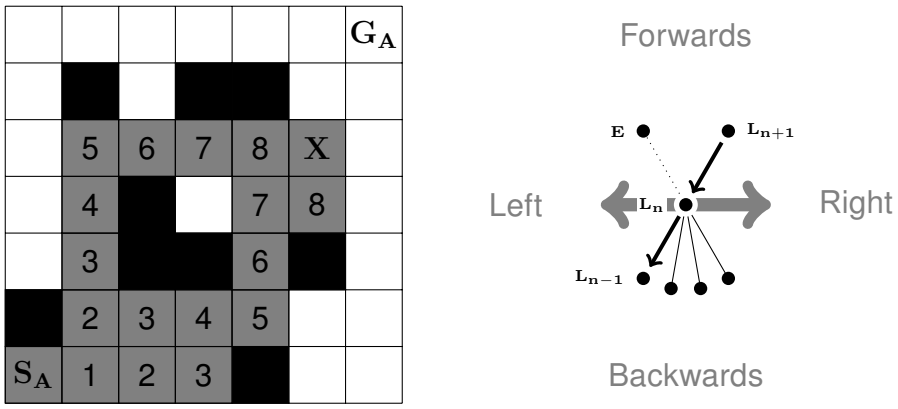


Figure 4.9: Left: Region of nodes which have a collidable path with the collision at X. Right: Abstract illustration of backtracking along the left side of the grey region by selecting the left most option $\mathbf{L_{n-1}}$ and checking for escape nodes such as $\mathbf{E}$.

which could reach the collision point $\mathbf{X}$.

Figure 4.8 labels these directions with respect to the agent A. The nodes indicated by $\mathbf{L}$ and $\mathbf{R}$ indicated the first steps backward from the collision node $\mathbf{X}$. The node labelled by $\mathbf{R}$ is not one from the original path traced by A, however can still reach the point of collision $\mathbf{X}$ and is therefore included in the search for an escape route. These directions can be calculated using the distance to the start node and goal node as indicators. Nodes which travel towards the start are backwards nodes and nodes which travel towards the goal are forwards nodes.

Figure 4.9 shows a shaded grey region. These are all the nodes which are contained within paths which can reach the collision point with minimal cost. If a path can escape this region without increasing the cost over the original path then the collision can be avoided. The white square in the middle of this region is not included as it cannot be reached from the forwards direction from any other node in the grey region. The left side of this region can be traced by taking the most extreme left option backwards in time one step at a time. Similarly for the right side by tracing consecutive right most backwards options. The numbers in Figure 4.9 indicate the time step at which agent A can visit that square. By taking the most extreme option left or right respectively we guarantee that we have selected the most extreme node which can enter our current

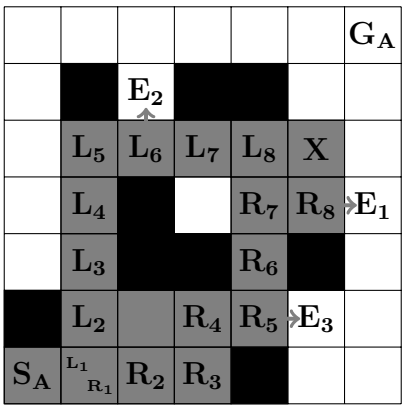


Figure 4.10: The nodes traced on the left side indicated by $\mathbf{L_n}$ and the right side $\mathbf{R_n}$. Escape nodes are indicated by $\mathbf{E_k}$.

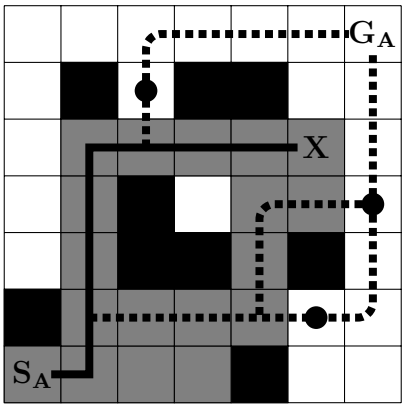


Figure 4.11: Connecting the escape nodes to the original path for agent A.

node from the earlier time step.

The right side of Figure 4.9 is an abstract illustration of this process. If the node $\mathbf{L_n}$ indicates the left most option at time $\mathbf{n}$ which can reach $\mathbf{X}$ at the time of collision then in order to explore the grey area we must select the left most option $\mathbf{L_{n-1}}$ as indicated. Any other option would enter the inside of the grey region and potentially miss options which escape the grey region. Given this configuration of nodes if an option such as $\mathbf{E}$ occurs to the left of $\mathbf{L_{n+1}}$ which travels towards the goal then this option escapes the collision and cannot possibly revisit the collision in minimal cost (otherwise it would be contained within the grey region and would either be visited by the algorithm or bypassed altogether).

Figure 4.10 marks the left nodes $\mathbf{L_n}$ and the right nodes $\mathbf{R_n}$ which follow this backtracking process. Possible escape nodes are indicated by $\mathbf{E_1}, \mathbf{E_2}, \mathbf{E_3}$ with grey arrows indicating the direction of escape from the grey region. Only $\mathbf{E_1}$ and $\mathbf{E_2}$ are needed for the current collision as they indicate escape paths on the left and right however the algorithm is continued to illustrate the complete process. Figure 4.11 shows how the escape paths are minimally connected to the original path to reduce the disturbance to earlier computation.

In this manner the CISR has the advantage of minimal cost for computing an alternative route. The algorithm does not compute the new path using A* as only a slight

change to the existing path using a linear search for the minimal sidestep around the point of collision.

Conclusion

In conclusion Generalized A* searches the largest search space and is the most verbose in its search for a solution. However this method does not suffer in a congested search space where many collisions may happen.

The CBS algorithm is collision based and its complexity rises with the number of collisions. When there are few collisions the CBS algorithm shows an improvement over the Generalized A* method because it does not branch at every time step and has no exponential rise in complexity with the number of agents. The CBS algorithm recalculates A* each time an agent needs avoid a collision.

The CISR algorithm works in a similar manner to a CBS algorithm by recomputing paths which collide. However the CISR algorithm reroutes paths rather than recomputes with A*. The rerouting algorithm is minimal in cost since it only requires a linear search backwards in time for an alternative node which leads on a route which avoids the collision. The CISR algorithm has the advantage over the CBS algorithm in sparse environments where either the number of collisions are low or the size of the environment is relatively large. The sparse environment means that the recomputation of A* that CBS needs is relatively expensive compared to the small deviation that CISR computes by backtracking through time.

4.2 Properties of Backtracking

In this section we will prove the correctness of the complete algorithm. This includes the properties of backtracking and its combination with the *Next* algorithm. To facilitate this proof we first describe the foundation behind the algorithm using mathematical definitions and terminology.

The main properties which serve as the basis for the backtracking technique is the planar condition and the equivalence of paths which converge to a common point in space and time. The planar condition is the property that allows for a region of points to be contained in a boundary formed by two paths.

In this section we will describe the properties which we use for a basis of the proofs which use this property. We restrict our analysis to a particular useful subtype of path and describe the structures which are formed from them. Using the structure of the faces of the planar graph we can describe the algorithm and prove correctness and completeness.

4.2.1 *General Properties*

First we need to build a basis for the constructions needed to make our backtracking algorithm. The Backtracking algorithm applies to planar graphs taking advantage of enclosed regions to bypass large regions of nodes which lead to the same points of

collision. A planar graph is defined as a graph which can be drawn without edges intersecting on a plane. Planar graphs generally have more than one configuration they can be drawn in to satisfy this condition. As such we will keep all planar graphs fixed in one planar configuration and call the planar projection function μ which fixes the position of a node to the 2D plane.

Planar Properties

The Backtracking algorithm relies on the fact that the graph can be drawn without any edges which intersect. We also require a number of similar ideas to help describe the backtracking algorithm later so we describe the graph with the perspective that the goal node is considered 'forwards' and the start node considered 'behind'. As such we describe nodes to the left or right in relation to this assumption.

Definition 4.2.1 (Planar projection, μ)**.** *A planar graph is a graph which has a projection onto a plane such that no edges cross. Call an arbitrary such projection μ. μ will be held constant for each map we are given.*

Definition 4.2.2 (Connection Side, $\{Left, Right\}$)**.** *Given a path v through a slice we divide the connections leading to the node into two subsets (excluding connections involved in the path). We split these paths based on cyclic order; by traversing the connections starting from the incoming connection and travelling clockwise.*

We represent the incoming connection by i, the outgoing connection by o and an arbitrary connection by c. A connection which gives the cyclic order $(i\ c\ o)$ we define the connection c as a $Left$ connection. A connection which gives the cyclic order $(i\ o\ c)$ we define the connection c as a $Right$ connection.

Layers

In order to describe the pruning method of bypassing collisions we need a new definition of equivalence. We can define two paths as equivalent if they reach the same point in time and space. This is called space time equivalence and is the basis of describing paths which collide at the same point in time and space. It is these same solutions we wish to avoid when we are looking for alternative solutions. The extreme paths of these cases which travel furthest left and right form a boundary on which we will be searching for alternative solutions. This is the essence of the backtracking algorithm.

To describe our new search space we define a number of concepts building to the definition the new equivalence relation on space time points. We first define a Layer as all the paths of one agent belonging to a single F-value.

Definition 4.2.3 (Layer, L_i^f)**.** *Define a layer L_i^f as all paths of cost f for agent i. i.e. $v \in A_i, F(v) = f \Leftrightarrow v \in L_i^f$.*

At each stage of the algorithm an exhaustive search is performed in order to find a solution or prove that no solution exists at that level of cost. We introduce a notation to specify the context in which a Layer is considered, we call this notation the Configuration:

Definition 4.2.4 (Configuration, $\mathbf{v}_{-i}$)**.** *Given a tuple* $\mathbf{v}$ *of agentverses let* $\mathbf{v}_{-i}$ *be the tuple of agentverses excluding agent* i*, i.e. if* $\mathbf{v} = (v_1, v_2, \ldots, v_{|K|})$ *then:*

$$\mathbf{v}_{-i} = (v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{|K|})$$

When performing an exhaustive search of the solution space each point in space and time will coincide with a number of paths. Considering all paths which reach a given space time point before the goal, all subsequent choices will only depend upon the reachability of that point in space and time.

When specifying a Configuration in the context of a Layer we restrict the Layer to all paths which make their non-zero choices before the first collision point. We call the combination of a Layer and a Configuration a Layer Configuration:

Definition 4.2.5 (Layer configuration, $L_i^f(\mathbf{v}_{-i})$)**.** *We define the layer configuration* $L_i^f(\mathbf{v}_{-i})$ *as a subset of* L_i^f*, where each agentverse* $v \in L_i^f(\mathbf{v}_{-i})$ *has the property that* $\forall s \geq \min_{j \neq i} \chi(v, v_j), I_s(v) = 0$.

We can restrict our perspective to paths which pass through a single point. This becomes important as it allows us to reason about paths which share a common point but ignore how the path got to that point if that is unimportant. We define an equivalence class over these points and prove its correctness:

Definition 4.2.6 (Layer Equivalence)**.** *Given a layer configuration* $L_i^f(\mathbf{v}_{-i})$ *we define the equivalence class of agentverses with respect to one space time point* $L_i^f(\mathbf{v}_{-i})[\widetilde{\mathbf{p}}]$*. i.e.* $v \in L_i^f(\mathbf{v}_{-i})[\widetilde{\mathbf{p}}]$ *implies* $v \in L_i^f$ *and* $P_{\widetilde{\mathbf{p}}_t}(v) = \widetilde{\mathbf{p}}_{\mathbf{x}}$*. Equivalence classes with the same associated time form a disjoint union of all agentverses of the given Layer, i.e.* $\cup_{\mathbf{x}} L_i^f(\mathbf{v}_{-i})[(\mathbf{x}, t)] = L_i^f(\mathbf{v}_{-i})$.

Lemma 4.2.7. *Definition 4.2.6 is well defined. i.e. the sets* $\cup_{\mathbf{x}} L_i^f(\mathbf{v}_{-i})[(\mathbf{x}, t)] = L_i^f(\mathbf{v}_{-i})$*, form a disjoint union.*

Proof. Given the fact that an agent cannot occupy two locations at any one time each of the sets $L_i^f(\mathbf{v}_{-i})[(\mathbf{x}, t)]$ must be distinct and therefore disjoint. We can also prove that every agentverse belongs to at least one of these sets by observing $\forall v \in L_i^f(\mathbf{v}_{-i}), v \in L_i^f(\mathbf{v}_{-i})[(P_t(v), t)]$. □

Space-Time Point Equivalence

When reasoning about the events which occur after a point which is shared by a number of paths in our search we would like to apply the same logic to all of these paths. For this reason we define the idea of space time point equivalence:

Definition 4.2.8 (Space Time Point Equivalence)**.** *We call two agentverses* $u, v \in L_i^f$ *in configuration* $\mathbf{v}_{-i}$ *equivalent at* $\widetilde{\mathbf{p}}$ *and write* $u \sim_{\widetilde{\mathbf{p}}} v$ *iff* $P_{\widetilde{\mathbf{p}}_t}(u) = P_{\widetilde{\mathbf{p}}_t}(v) = \widetilde{\mathbf{p}}_{\mathbf{x}}$ *and* $\widetilde{\mathbf{p}}_t < C_i^f(v), \widetilde{\mathbf{p}}_t < C_i^f(u)$*.*

However we need to prove that this equivalence is well defined with respect to the wider picture. For this purpose we need to check what properties a space time point equivalence has upon a branching set.

Lemma 4.2.9 (Space Time Point Equivalence)**.** *Point equivalence is a well defined property. Given a point* $\widetilde{\mathbf{p}}$ *all paths which reach point* $\widetilde{\mathbf{p}}$ *can be considered equivalent, if no collision is introduced on or before* $\widetilde{\mathbf{p}}_t$*.*

Proof. We will prove this lemma by constructing a bound for a new branching set. If the new branching set still satisfies the conditions of a branching set we will have proven that the equivalence of paths given a shared space time point is well defined.

Suppose we are constructing a branching set for $\mathbf{v}$ given that all paths which reach $\widetilde{\mathbf{p}}$ are equivalent for agent i. We can take a branching set B of $\mathbf{v}$ and construct a bound for our equivalence condition.

Construct a path m which is the minimum representative of the equivalence for space time point $\widetilde{\mathbf{p}}$ which is covered by B:

$$m = \min\{u_i \colon \mathbf{u} \in C(B), P_{\widetilde{\mathbf{p}}_t}(u_i) = \widetilde{\mathbf{p}}_{\mathbf{x}}\}$$

We next define three sets which bound the results of the target Branching set. The first set represents paths which agree with the minimal representative up to the space time point $\widetilde{\mathbf{p}}$. This version only uses the m path to represent the given space time point, removing the other extraneous possibilities:

$$D_e = \{\mathbf{u} \colon \mathbf{u} \in C(B), C(u_i, \mathbf{u}_{-i}) \geq \widetilde{\mathbf{p}}_t, u_i \sim_{\widetilde{\mathbf{p}}_t} m\}$$

Next we preserve possibilities which conflict with the given minimal path m. These paths will need to avoid the collision before the equivalence can be maintained:

$$D_x = \{\mathbf{u} \colon \mathbf{u} \in C(B), C(m, \mathbf{u}_{-i}) < \widetilde{\mathbf{p}}_t\}$$

Lastly we account for all paths which are a part of the cover but break the equivalence by going through another point at time $\widetilde{\mathbf{p}}_t$.

$$D_c = \{\mathbf{u} \colon \mathbf{u} \in C(B), P_{\widetilde{\mathbf{p}}_t}(u_i) \neq \widetilde{\mathbf{p}}_{\mathbf{x}}\}$$

Together these sets form a bound on a new branching set B' which takes account of the equivalence, call the joint set $D = D_e \cup D_c \cup D_x$. i.e. if the cover of B' contains the elements of the set $D \subset C(B')$, and maintains the properties of a branching set then space time point equivalence is a well defined property.

It can be noted that $\mathbf{v} \notin C(B)$ which implies that $\mathbf{v} \notin D \subset C(B)$. Which implies that we will not be forced to make $\mathbf{v}$ an element of B'.

We also have to prove the second condition that $\mathbf{v}$ covers B' is possible. We need to prove that the set $D \subset C(\mathbf{v})$. This can be seen from the fact that $B \subset C(\mathbf{v})$ and $D_e, D_x, D_c \subset C(B)$, which implies that $C(\mathbf{v}) \supset C(B) \supset D_e \cup D_x \cup D_c = D$.

For the third condition we need B' to cover all solutions that $\mathbf{v}$ does. However the bound does not effect this property therefore meaning if a set with the above conditions can be found the equivalence property is well formed. □

We would also like to define a representative of these points freely. Considering the fact that all paths which lead to these points have an equal cost it does not matter which order we use each representative.

Lemma 4.2.10 (Point Representative)**.** *We can select any representative of a space time point, as long as that path is removed from future computation.*

Proof. This lemma follows from lemma 4.2.9 and the fact that all representatives have equal cost. By rearranging the order of the equivalent paths the new representative can be utilized and then removed from later computation. □

4.2.2 *Complex Paths*

To reduce the complexity of the problem to be solved we split the paths that we analyse into two categories; Complex and Non-Complex. These two categories form an almost arbitrary boundary between the Complex cases of paths which can need to be redirected around collisions from the Simpler cases to be redirected. This categorisation allows us to build a theory of backtracking which covers most cases but allows for a general solution when the path structure becomes more complex. The choice we make for the division is chosen because it is simple to describe but serves the purpose of dividing the two categories of paths simply and still covers a broad number of paths.

The simplest case of path is a path which travels the most efficient path towards the goal, i.e. at every step the path moves towards the goal. We call these paths a Rudimentary path and form the basis for complexity from them. We consider a path v Non-Complex if there exists a rudimentary path r which shares its positions with v, i.e. $P(v) = P(r)$ where $P(v) = \{P_t(v) \colon \forall t \in [0, \infty)\}$. This can be intuitively thought of as the path v sharing the same projection onto the underlying graph as r.

This definition of Non-Complex paths allow the rudimentary path to form the backbone of the backtracking algorithm. Steps along the graph can be thought of in several directions either towards the goal or away from it and clockwise or anticlockwise around the goal. The definition of Complex/Non-Complex depends on what the agents start and goal nodes are however as they dictate the set of Rudimentary paths which exist. All nodes and edges which can be traced by these paths are called Non-Complex. Paths which do not project onto a Rudimentary path are called Complex and points

and edges which can't be reached from one of the rudimentary paths. Several Non-Complex paths can project onto a single Rudimentary path as this encompasses all paths which pause and reverse direction along the Rudimentary path at any time.

Definition 4.2.11 (Rudimentary Path)**.** *We define a path r to be a rudimentary path if each step has minimal relative cost, i.e. r is rudimentary iff $\forall t \geq 0, Q_t(r) = 0$. An alternative definition is that the set of Rudimentary paths R is defined as $R_i = \{r \colon F(r) = D_{\mathbf{s}_i}(\mathbf{g}_i)\}$.*

Definition 4.2.12 (Non Complex Agentverse)**.** *We define a path v to be Non-Complex iff there exists a rudimentary path $r \in R_i$ which shares the same position set $P(r) = P(v)$.*

We define points that can be reached by a rudimentary path Non-Complex points (NC points), and branches that can be made by rudimentary paths Non-Complex branches (NC branches). We call points and branches which cannot be reached by a rudimentary path Complex points and branches, similarly paths u which do not share their point set $P(u)$ with a rudimentary path are known as Complex paths.

We use the notation $\mathbf{x} \in \mathrm{Cx}$ to indicate a complex point and $\mathrm{Cx}_t(v)$ to indicate that the path v makes a complex branch at time t.

The backtracking algorithm finds alternative routes to avoid collisions. We use the concept of a bypass to find the earliest route around the collision. A bypass is represented by two rudimentary paths which encompass the maximum area which colliding paths can pass through in order to meet the collision point. By searching the boundary of the bypass we can find the alternative routes which 'bypass' the collision.

To reduce the search space we aim to remove portions of the search space. We construct paths which bound areas which will cause redundant calculations. We call a pair of paths which contain a region a bypass and define it as such:

Definition 4.2.13 (Bypass)**.** *Given a planar graph and two Non-Complex paths $u, v \in L_i^f$, If u and v coincide at two individual space time points $\widetilde{\mathbf{p}}, \widetilde{\mathbf{q}}$ after the last change in cost but remain separate in between these two paths, we call the region contained a bypass. i.e. suppose $\widetilde{\mathbf{p}}_t > \widetilde{\mathbf{q}}_t$ then $u, v \in L_i^f(\mathbf{v}_{-i})[\widetilde{\mathbf{q}}], u, v \in L_i^f(\mathbf{v}_{-i})[\widetilde{\mathbf{p}}], \forall s,$ **s.t.** $\widetilde{\mathbf{q}}_t < s < \widetilde{\mathbf{p}}_t, P_s(u) \neq P_s(v), \forall s \in (\widetilde{\mathbf{q}}_t, \widetilde{\mathbf{p}}_t), Q_s(u) = Q_s(v) = 0$.*

Figure 4.12 shows an example of a bypass and labels relevant details. We categorize points with respect to the bypass. Points inside the bypass are called interior points. Points outside the bypass are called exterior points, this includes the Goal node. Points which lie along the bypass on u or v are called boundary points.

We use the property of a planar graph to show that any path through the centre of a bypass will have to pass through the boundary of the region:

Lemma 4.2.14. *All paths which coincide with the first space time point $\widetilde{\mathbf{p}}$ of a bypass and remain on or inside the boundary will coincide with the second space time point $\widetilde{\mathbf{q}}$ or they will pass through a point on the boundary.*

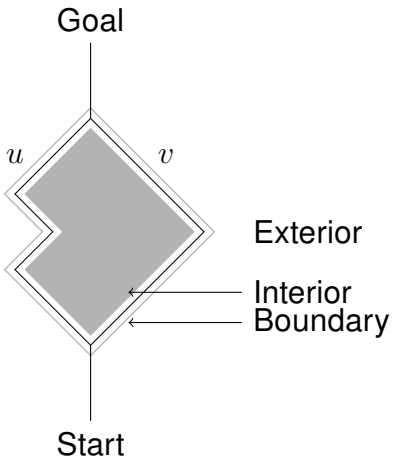


Figure 4.12: Example of a bypass u, v.

Proof. The segments of the paths u, v form a single loop between the points $\widetilde{\mathbf{p}}, \widetilde{\mathbf{q}}$. As the loop segregates the space into two portions, the interior and exterior, any path which begins in the interior must cross the loop to reach the goal. However the graph is planar by assumption and therefore the path must pass through a point on this loop. □

4.2.3 *Properties of the Non-Complex Subgraph*

Restricting the graph to the Non-complex nodes and edges of a specific agent we get the Non-Complex Subgraph of that agent. The Non-Complex Subgraph is all that is required in order to calculate the Backtracking algorithm and because of its construction it has a regular structure with certain features. These features dictate the shape and properties of the faces included in this subgraph and the subsets of nodes at a fixed distance which we call Slices.

Definition 4.2.15 (Non-Complex Sub-graph)**.** *A Non Complex sub-graph is a sub-graph which contains only Non-Complex points and branches with respect to a particular agent. The graph will have a single Start node and a single Goal node.*

Figure 4.13 shows an example of a Non Complex sub-graph. The grey highlighted path from the Start to the Goal is an example of a rudimentary path on this Non Complex sub-graph. Each dotted line connects nodes of equal distance to the Goal. Non of the nodes within a slice are interconnected within the subgraph as the connection would be Complex. Lines which connect equidistant nodes of this form are called Face Loops and will be discussed later.

The Face Loops in the figure highlight the fact that there is no particular prominent node to a Slice, as paths may wind around the Start or Goal node a number of times depending on the structure of the graph. We will analyse the structure of these slices by assigning an index to each node allowing the construction of intervals of indices to reason about contiguous arcs of nodes around the Face Loops.

Definition 4.2.16 (Slice, S_i^d)**.** *Given a particular Non Complex sub-graph $G \subset M$ we define a subset of nodes from G called a Slice; a Slice is the subset of NC nodes at a*

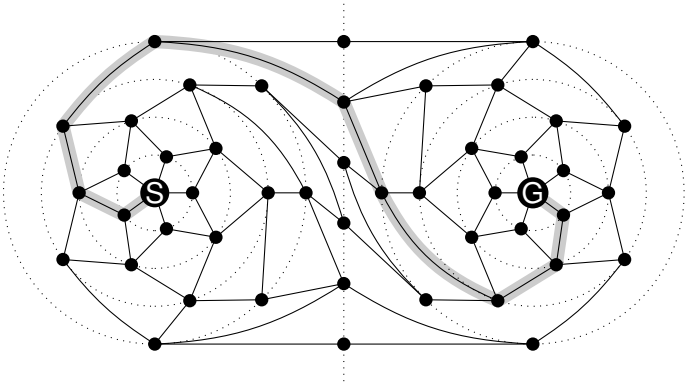


Figure 4.13: An example of a Non Complex Subgraph. (S=Start, G=Goal)

specified distance. i.e. $S_i^d = \{\mathbf{x} \colon \mathbf{x} \in M, D_{\mathbf{g}_i}(\mathbf{x}) = d\}$*. Given a node* $\mathbf{x} \in S_i^d$ *we define* $S_i^{d'}(\mathbf{x})$ *as the set of nodes in* $S_i^{d'}$ *which connect to* $\mathbf{x}$*.*

We now study the structure of the sub-graph. Since we deal with a planar graph, as the original was planar, we can define a face as an undivided region contained by a set of edges. Each of the faces of the Non-Complex Subgraph follow the same pattern. They have a single node nearest the start node and a single node nearest the goal node. Each side of the face between these two start and end points have an equal number of nodes. This regular pattern is important in establishing the structure of each Slice as a single loop through neighbouring faces in the Face Loop. Together with the idea of a Face Loop we have a complete picture of the structure of the Non-Complex Subgraph. The following lemmas work towards proving the structure of each face is as described.

Lemma 4.2.17 (Neighbour Property)**.** *Given any bidirectional map the distances between the Goal and two neighbouring points can only differ by 1.*

Proof. To see this consider two neighbouring nodes $\mathbf{x}, \mathbf{x}'$. If $\mathbf{g}, \mathbf{x}_1, \ldots, \mathbf{x}_{d-1}, \mathbf{x}$ is a minimal path from the Goal to $\mathbf{x}$ of length d then $\mathbf{g}, \mathbf{x}_1, \ldots, \mathbf{x}_{d-1}, \mathbf{x}, \mathbf{x}'$ is a path from the Goal to $\mathbf{x}'$ of length $d+1$. This means that the distance from the Goal to $\mathbf{x}'$ is at most $d+1$. The same logic can be applied in with a minimal path from the Goal to $\mathbf{x}$ proving that the respective distances can differ by at most 1. □

Definition 4.2.18. *Three consecutive nodes* $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2$ *on the boundary of a face are defined to be a local minimum if the distances to the Goal from the nodes are of the form* $D_{\mathbf{g}_i}(\mathbf{x}_0) = D_{\mathbf{g}_i}(\mathbf{x}_2) = D_{\mathbf{g}_i}(\mathbf{x}_1) + 1$*, and are defined to be a local maximum if they are of the form* $D_{\mathbf{g}_i}(\mathbf{x}_0) = D_{\mathbf{g}_i}(\mathbf{x}_2) = D_{\mathbf{g}_i}(\mathbf{x}_1) - 1$*. Nodes not of either of these forms are known as 'side' nodes.*

Lemma 4.2.19 (Maximum Number of Local Maxima/Minima)**.** *On a Non-Complex subgraph there exists one and only one local minimum and local maximum per face.*

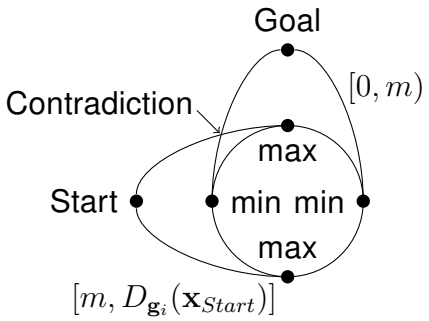


Figure 4.14: An illustration of the contradiction formed by having multiple maxima/minima.

Proof. To prove this lemma we aim to show a contradiction in the assumption that there can be multiple maxima/minima. We will show that if there could be multiple maxima there would be an equal number of minima and vice versa. However each minima must have a direct path to the Goal and each maxima must have a direct path to the Start. When these paths are considered in the plane they must cross in at least one case leading to a point which must be closer to the Goal than the local minima and closer to the Start than the local maxima. This is a contradiction disproving the fact that there can be multiple local maxima/minima. Figure 4.14 shows a illustration of the process described in the proof.

First suppose that there are two local minima on a single face. There must be a local maximum since there are only a finite number of nodes between two minima there must be a node which achieves the maximum distance from the Goal. To show that a node $\mathbf{x}$ which achieves the maximum distance is a local maximum consider the nodes either side $\mathbf{y}, \mathbf{y}'$. Since an edge between two nodes of the same slice would be a Complex branch we know that $\mathbf{x}$ can not be the same distance from the Goal as $\mathbf{y}, \mathbf{y}'$. From lemma 4.2.17 we see that the difference between $D_{\mathbf{g}_i}(\mathbf{x})$ and $D_{\mathbf{g}_i}(\mathbf{y}), D_{\mathbf{g}_i}(\mathbf{y}')$ can be at most 1, also by assumption of $\mathbf{x}$ being a maximum we have $D_{\mathbf{g}_i}(\mathbf{x}) > D_{\mathbf{g}_i}(\mathbf{y}), D_{\mathbf{g}_i}(\mathbf{x}) > D_{\mathbf{g}_i}(\mathbf{y}')$. This implies that $D_{\mathbf{g}_i}(\mathbf{x}) - 1 = D_{\mathbf{g}_i}(\mathbf{y}) = D_{\mathbf{g}_i}(\mathbf{y}')$ and is therefore a local maximum. Similarly between two local maxima there must be a local minimum.

Let $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}$ represent the nodes of the face; traversing them in an order such that $\mathbf{x}_j, \mathbf{x}_{j+1}$ are neighbours and $\mathbf{x}_{n-1}, \mathbf{x}_0$ are neighbours, where n is the number of nodes around the face and $j \in [0, n-1)$. Now suppose that $\mathbf{x}_k$ and $\mathbf{x}_{k'}$ are two local minima, without loss of generality we assume $k < k'$. Using these two nodes the loop can be split into two sections: the segment $\mathbf{x}_{k+1}, \ldots, \mathbf{x}_{k'-1}$ and the segment $\mathbf{x}_{k'+1}, \ldots, \mathbf{x}_{n-1}, \mathbf{x}_0, \ldots, \mathbf{x}_{k-1}$. Using the properties of local minimum there must exist a local maximum in each segment of value at least $m = \max(D_{\mathbf{g}_i}(\mathbf{x}_k), D_{\mathbf{g}_i}(\mathbf{x}_{k'})) + 1$ since each of these has neighbours at a distance at least one higher than the local minimum. Suppose $\mathbf{x}_l, \mathbf{x}_{l'}$ are two such maxima, one in each segment.

The two minima $\mathbf{x}_k, \mathbf{x}_{k'}$ must have an unbroken path to the Goal, where each node on the path takes a step towards the Goal. Connecting these two paths together there is

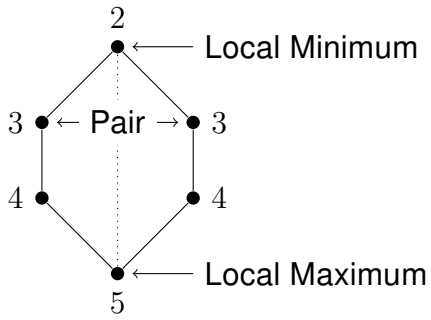


Figure 4.15: Example face from a Non Complex sub-graph.

an unbroken path from $\mathbf{x}_k$ to $\mathbf{x}_{k'}$ where each of the nodes takes a distance from the set $[0, m)$ (where $m = \max(D_{\mathbf{g}_i}(\mathbf{x}_k), D_{\mathbf{g}_i}(\mathbf{x}_{k'}))+1$ as before). Since the graph is constructed from Non-Complex paths there must be a path from the Start to each node in the graph where each step along the path decreases the distance to the node. Connecting a path from each of $\mathbf{x}_l, \mathbf{x}_{l'}$ to the Start we can then construct a path from $\mathbf{x}_l$ to $\mathbf{x}_{l'}$ via the start where each of the nodes takes a distance from the set $[m, D_{\mathbf{g}_i}(\mathbf{x}_{\text{Start}})]$.

The path from $\mathbf{x}_k$ to $\mathbf{x}_{k'}$ must intersect the path from $\mathbf{x}_l$ to $\mathbf{x}_{l'}$, since the paths cannot travel through the face (as this would split the face into multiple faces) and the local maxima are interleaved between the local minima around the face. However since the graph is planar this must occur on a node, and since each node from the first path takes a distance from the set $[0, m)$ and the second path takes a distance from the set $[m, D_{\mathbf{g}_i}(\mathbf{x}_{\text{Start}})]$ we show the contradiction as these two sets are disjoint. This means the original assumption that there were two local minima is false meaning there is only one local minimum, similarly there can only be one local maximum. □

Lemma 4.2.20 (Face Structure)**.** *The faces of a Non Complex sub-graph have the following properties:*

1. *Each face has a minimum and maximum node. These nodes have the nearest and farthest distances to the Goal respectively.*

2. *Each face has a symmetrical property. Given a node other than the local maximum and local minimum a corresponding node the same distance from the Goal will be present on the other side of the face.*

3. *Following from 2. There exists the same number of nodes on both sides. Starting from the local maximum the nodes incrementally get closer to the Goal until they reach the local minimum on the face.*

Figure 4.15 gives an example of a face from a Non Complex sub-graph.

Proof. We have proven via lemma 4.2.19 that there exists only one local minimum and one local maximum per face. We also know via lemma 4.2.17 that the distance from

the Goal can only differ by 1 between neighbouring nodes on the face. Now suppose $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}, \mathbf{x}_n$ is a path along the face such that $\mathbf{x}_0$ is the local maximum and $\mathbf{x}_n$ is the local minimum. From lemma 4.2.17 we have $|D_{\mathbf{g}_i}(\mathbf{x}_j) - D_{\mathbf{g}_i}(\mathbf{x}_{j+1})| \leq 1$, however this difference cannot be 0 as this would make the branch from $\mathbf{x}_j$ to $\mathbf{x}_{j+1}$ a Complex branch. We also cannot have $D_{\mathbf{g}_i}(\mathbf{x}_j) < D_{\mathbf{g}_i}(\mathbf{x}_{j+1})$ before $j = n$ since this would make $\mathbf{x}_j$ a second local minimum. Therefore we have $n = D_{\mathbf{g}_i}(\mathbf{x}_0) - D_{\mathbf{g}_i}(\mathbf{x}_n)$ since the distance is decremented each step along the path. This logic also applies to the path which leads from the local maximum to the local minimum from the other side making the face symmetrical. □

Since we now know the structure of a face, we wish to study the relationship between nodes within a single slice. In order to describe the structure of the face loop and the relationship of the nodes between slices we define an index for each node on the loop. We call this the Slice Index. A unique Slice Indexing can be found by applying an indexing scheme. We index the node on the slice which is a part of the α path from the start to the goal as 0. We then follow the faces that are a embedded in the Slice around the goal indexing them from left to right incrementally. This scheme is always well defined and leads to a unique indexing.

Definition 4.2.21 (Slice Index, $J(\mathbf{x})$)**.** *We identify an index $J_d(\mathbf{x})$ with each of the nodes $\mathbf{x} \in S_i^d$. When the Slice is implicit we may remove the index and refer to the index by writing $J(\mathbf{x})$. Each index is unique and ranges from $[0, |S_i^d|)$. The indices are assigned using the following rules:*

1. *If the location $\mathbf{x} \in S_i^d$ lies on the α path from the Start to the Goal then $J(\mathbf{x}) = 0$.*
2. *If the two locations $\mathbf{x}, \mathbf{x}' \in S_i^d$ lie within the same face then their indices are consecutive (mod $|S_i^d|$): $J(\mathbf{x}') \equiv J(\mathbf{x}) \pm 1 \pmod{|S_i^d|}$.*
3. *By extending a temporary edge from the point which lies on the α path, the node $\mathbf{x}$ which lies on the $Right$ side of the α path is assigned index 1. i.e. $J(\mathbf{x}) = 1$.*

We define for convenience a function $\mathcal{J}_d(v)$ as shorthand for the index of the node at distance d from the Goal on path v:

$$\mathcal{J}_d(v) = J(P_{D_{\mathbf{g}_i}(\mathbf{s}_i)-d}(r_v))$$

Figure 4.16 shows an example of the process starting from $\mathbf{x}_0$ on the alpha path we assign the index 0. To the right of $\mathbf{x}_0$ at the same distance from the Goal and contained within the same face we assign the value 1. This process continues until we reach the node to the left of $\mathbf{x}_0$ where we assign the index $n - 1$ where n is the number of nodes in the Slice $S_i^d \ni \mathbf{x}_0$.

Lemma 4.2.22. *Apart from the Start node and the Goal node, all nodes are contained on the 'side' of at least one face.*

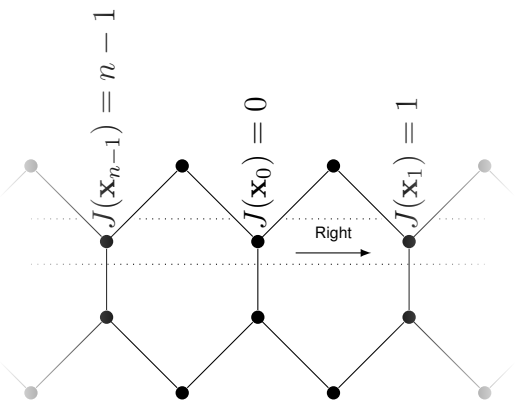


Figure 4.16: An example of index assignment.

Proof. Consider a single node $\mathbf{x}$ in the Non-Complex sub-graph. Suppose that the node $\mathbf{x}$ is a local minimum of all the faces in which it is contained, this implies that it is the nearest node to the Goal in all the faces of which it is part. The only node which can satisfy this condition is the Goal node, otherwise there would not be a path available to the Goal. Similarly if the node $\mathbf{x}$ were a local maximum in all the faces in which it is contained this would imply that it was the Start node, otherwise there would be no path to the node from the Start.

Suppose there exists a face where $\mathbf{x}$ is a local minimum and a face where $\mathbf{x}$ is a local maximum. Now consider the order in which these faces occur when each face is visited in a cycle around the node $\mathbf{x}$. When traversing this cycle there must be a point at which $\mathbf{x}$ is a local maximum and in one face but $\mathbf{x}$ is a local minimum in the next face. Let $\mathbf{y}$ denote the node which adjoins $\mathbf{x}$ along the edge which separates the faces in which $\mathbf{x}$ is a local maximum and local minimum. Given the face $\mathbf{x}$ was a local maximum in one face that implies $D_{\mathbf{g}_i}(\mathbf{x}) - 1 = D_{\mathbf{g}_i}(\mathbf{y})$, however in the other face it was a local minimum implying $D_{\mathbf{g}_i}(\mathbf{x}) + 1 = D_{\mathbf{g}_i}(\mathbf{y})$ which contradicts the previous statement. This means that there cannot exist a face in which it is only a local maximum and a local minimum, meaning that other than the Goal and the Start it must be a part of a side. □

We have also proven the following corollary:

Corollary 4.2.23. *Given a node $\mathbf{x}$, a face in which the node $\mathbf{x}$ is a local maximum cannot share an edge with a face in which $\mathbf{x}$ is a local minimum.*

Lemma 4.2.24 (The Slice Index Function is Well-Defined)**.** *The definition of the Slice Index function $J(\mathbf{x})$ is well defined and unique.*

Proof. In lemma 4.2.22 we have proven that other than the Start node and the Goal node all nodes exist as a side node to at least one face. We will show that if we exclude the last condition for the index function $J(\mathbf{x})$ then there are two possible ways to assign the indices to a slice. We first proceed by constructing a loop of faces. Starting from the initial node $\mathbf{x}_0$ indexed as 0, we select a face in which $\mathbf{x}_0$ is a side node. On the other

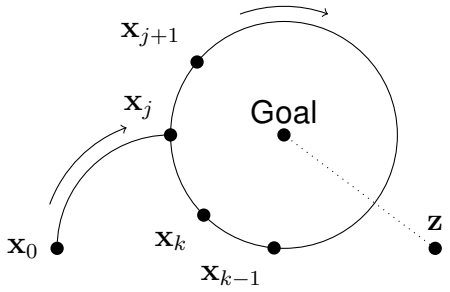


Figure 4.17: Illustration of the contradiction generated when $\mathbf{x}_j$ and $\mathbf{x}_k$ share a face.

side of the given face another node $\mathbf{x}_1$ exists which is the same distance from the Goal as $\mathbf{x}_0$. By the definition of the index function there are two options $J(\mathbf{x}_1) = 1, |S_i^d| - 1$.

From corollary 4.2.23 even if $\mathbf{x}_1$ exists as a local maximum and a local minimum traversing the faces in cyclic order there would be two transitions from minimum to maximum and from maximum to minimum. By corollary 4.2.23 in-between these transitions the node $\mathbf{x}_1$ must be a side node.

Next we need to prove that there exists two faces where $\mathbf{x}_1$ is considered a side node of the face. Consider that there must be edges which lead to the Goal and from the Start which passes through $\mathbf{x}_1$. Suppose the node is not a local maximum; let $\mathbf{w}$ represent the node connected to $\mathbf{x}_1$ on the path to the Goal. The edge from $\mathbf{x}_1$ to $\mathbf{w}$ is contained in two faces (if we exclude the possibility that the slices with $\mathbf{x}_1$ and $\mathbf{w}$ only contain one element), if we continue around the two faces with $\mathbf{y}$ and $\mathbf{y}'$. The arcs $\mathbf{w}, \mathbf{x}_1, \mathbf{y}$ and $\mathbf{w}, \mathbf{x}_1, \mathbf{y}'$ cannot form local maximum by assumption, therefore making $\mathbf{x}_1$ a side node of two faces.

Using this property we can find $\mathbf{x}_2$ which now only has one index it can take by construction: $J(\mathbf{x}_2) = 2$ if $J(\mathbf{x}_1) = 1$ or $J(\mathbf{x}_2) = |S_i^d| - 2$ if $J(\mathbf{x}_1) = |S_i^d| - 1$. We can continue this construction, however this process will eventually run out of nodes in the slice S_i^d and must connect to a node earlier in the sequence. Let $\mathbf{x}_k$ be the last node assigned an index by this sequence of indices before we connect to an earlier node which we represent by $\mathbf{x}_j$. Figure 4.17 shows the sequence of nodes from $\mathbf{x}_0$ to $\mathbf{x}_k$ and how it loops back to $\mathbf{x}_j$.

We now need to show that this loop contains all nodes in the slice S_i^d. The nodes $\mathbf{x}_j$ to $\mathbf{x}_k$ now form a loop as shown in Figure 4.17. By constructing a curve through the faces which connect the nodes in this loop we have split the Goal node from the Start node. Now consider any node $\mathbf{z} \in S_i^d$ not on this curve. There must exist a rudimentary path from the Start node to the Goal node which passes through $\mathbf{z}$, however by construction it must also pass through one of the nodes from the sequence $\mathbf{x}_j, \ldots, \mathbf{x}_k$, since the graph is non-planar and we constructed the curve through faces or nodes. This leads to a contradiction as the node that is passed through on the curve is at distance d, however so is the node $\mathbf{z}$ which conflicts with the original assumption that this path was rudimentary. Therefore the curve must contain all nodes at distance d meaning one of two indexing functions can be constructed.

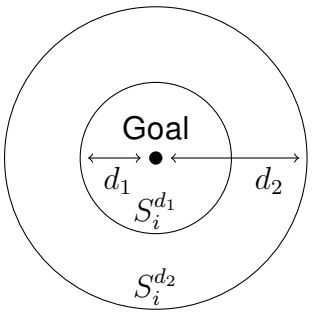


Figure 4.18: An illustration of the disjoint Face Loops of lemma 4.2.26.

Using the the last property we can decide between the two versions as it uniquely defines the direction of the sequence of indices. This proves the indexing function is well defined and unique. □

Now that we have proven that Indexing the nodes of a Slice is possible and well defined we can now define a Face Loop.

Definition 4.2.25 (Face Loop)**.** *Given a slice S_i^d a face loop is a set of additional edges added between each of the nodes in S_i^d embedded in the faces between the nodes. i.e. an edge is defined between nodes $\mathbf{x}, \mathbf{y} \in S_i^d$ where $J(\mathbf{x}) \equiv J(\mathbf{y}) \pm 1 \pmod{|S_i^d|}$.*

This definition is well defined as proven in Lemma 4.2.24. We can also prove that no two Face Loops will cross:

Lemma 4.2.26 (Disjoint Face Loops)**.** *Two Face Loops of different slices do not cross.*

Proof. Each Face Loop contains the Goal node on one side and the Start node on the other. This can be seen as the nodes present on the Face Loop are of a fixed distance. All rudimentary paths pass through these Face Loops, at said fixed distance. Given two face loops through slices $S_i^{d_1}, S_i^{d_2}$ and supposing $d_1 < d_2$ then all nodes of $S_i^{d_1}$ lie closer to the Goal node and therefore lie on the same side of the Face Loop of the $S_i^{d_2}$ slice. This implies the two face loops cannot intersect. Figure 4.18 illustrates these two Face Loops. □

4.2.4 *Cyclic Intervals*

Before we can discuss the Backtracking algorithm we need a new definition to facilitate the description of neighbouring nodes. We have shown that the nodes on a slice all lie on one loop around the goal node. Depending on the structure of the Non-Complex Subgraph Rudimentary paths may loop around the goal in one or more turns before they reach the goal. This leads to parts of our theorem where the indexing function loops round to zero as we describe the path through the graph. This cannot however be fixed by renumbering the nodes such that this never occurs as the geometry of the graph may still allow the path to loop all the way around the graph.

We define a generalization of the concept of an interval of numbers i.e. $[j,k],(j,k),[j,k)$ by allowing a new type of interval called the Cyclic Interval; $\langle j,k\rangle_n$. The cyclic interval is defined within a range of values and allows the interval to loop around at the maximum value n given. This leads us to a natural definition of the set of nodes $T \subset S_i^{d-1}$ neighbouring a given node $\mathbf{x} \in S_i^d$ as a cyclic interval starting from 0 and going up to and including $|S_i^{d-1}| - 1$. In the following section we will define a number of important relationships between neighbouring nodes using this these intervals.

Definition 4.2.27 (Cyclic Interval, $\langle j,k\rangle_n$)**.** *We define the Cyclic Interval to be a set of the form:*

$$\langle j,k\rangle_n = \begin{cases} [j,k] & \text{if } j,k \in [0,n), j \leq k \\ [0,k] \cup [j,n) & \text{if } j,k \in [0,n), j > k \\ \langle j \bmod n, k \bmod n\rangle_n & \text{otherwise} \end{cases}$$

All modulo of the form $(a \bmod n)$ *are assumed to result in a value in the range* $[0,n)$*. We also define two accessors functions:*

$$\mathrm{Left}(\langle j,k\rangle_n) = j$$
$$\mathrm{Right}(\langle j,k\rangle_n) = k$$

The last cases allows for the notational convenience of writing expressions such as $\langle j,k+a\rangle_n$ without much concern for looping around. We may now use this definition to describe a region of a slice contained within a bypass.

Lemma 4.2.28. *The region of a slice contained within a bypass between* u,v *is an cyclic interval of the indexing function. i.e. the region contained by* u,v *in* S_i^d *is of the form* $\langle j,k\rangle_{|S_i^d|}$*.*

Proof. The points in which the paths u,v intersect with S_i^d can only happen at two locations $P_t(u), P_s(v)$ for some t such that $D_{\mathbf{g}_i}(P_t(u)) = D_{\mathbf{g}_i}(P_s(v)) = d$ from the definition of a Non Complex path. Using a subset of the edges of the Face Loop for the S_i^d slice we construct a curve through the faces which connect the $P_t(u)$ to $P_s(v)$ the curve will not meet the boundary while traversing the faces or nodes. However by construction the indices between neighbouring nodes are consecutive, unless we loop to the beginning, meaning the indices included will be of the form $\langle j,k\rangle_{|S_i^d|}$. □

We now show that the neighbours of a given node form a contiguous block of indices.

Lemma 4.2.29. *Given a node* $\mathbf{x} \in S_i^d$ *and an adjacent slice* $S' = S_i^{d\pm1}$ *the neighbouring nodes* $S'(\mathbf{x})$ *have indices of the form* $J(S'(\mathbf{x})) = \langle j,k\rangle_{|S'|}$*. (we take* $\pm$ *to mean one or the other in this case).*

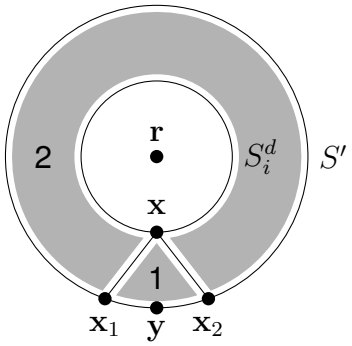


Figure 4.19: Regions between S_i^d, S' split by $\mathbf{x}_1, \mathbf{x}, \mathbf{x}_2$.

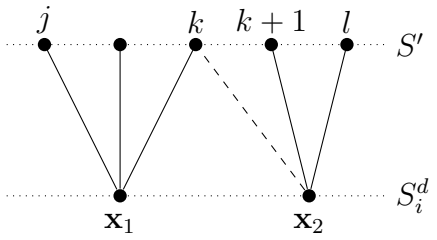
Proof. Consider a node $\mathbf{x} \in S_i^d$ with a neighbouring slice $S' = S_i^{d\pm1}$. Now suppose that the neighbouring $S'(\mathbf{x})$ are not a Cyclic interval. Let the set $T = J(S'(\mathbf{x}))$ represent the set of indices representing the neighbours of $\mathbf{x}$. Let $\langle j, k\rangle_{|S'|}$ be the smallest set such that $T \subset \langle j, k\rangle_{|S'|}$. By assumption there exists $\mathbf{y} \in \langle j, k\rangle_{|S'|}, J(\mathbf{y}) \notin T$. Also note that there exists $\mathbf{y}' \in S', J(\mathbf{y}') \notin \langle j, k\rangle_{|S'|}$ otherwise the cyclic interval $\langle J(\mathbf{y}) + 1, J(\mathbf{y}) - 1\rangle_{|S'|}$ could have been used to cover T.

Now consider the two face loops through S_i^d and S'. Lemma 4.2.26 shows that two distinct face loops will not intersect. This implies that a region will be formed between the two Face Loops. Consider the Face Loop through S'. Depending on the side which $\mathbf{x}$ lies on we select a representative of the centre of the loop. If $S' = S_i^{d+1}$ we select the Goal node as the representative, otherwise if $S' = S_i^{d-1}$ we select the Start node as the representative. We have selected this representative so that the node $\mathbf{x}$ lies on the same side of S' as the representative which we will label $\mathbf{r}$. Note that all nodes on the slice S_i^d have an unbroken path to the representative.

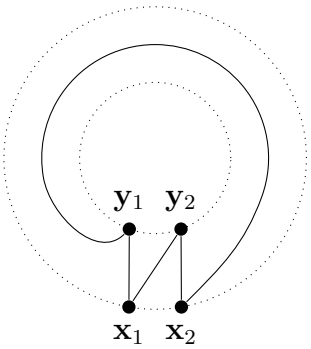
Now consider the fact that the Cyclic Interval $\langle j, k\rangle_{|S'|}$ can be broken into two disjoint Cyclic intervals at $\mathbf{y}$, while still covering the neighbours of $\mathbf{x}$: as such $T \subset \langle j, J(\mathbf{y}) - 1\rangle_{|S'|} \cup \langle J(\mathbf{y}) + 1, k\rangle_{|S'|}$. Select a node which lies in each half: $\mathbf{x}_1, \mathbf{x}_2 \in S', J(\mathbf{x}_1) \in \langle j, J(\mathbf{y}) - 1\rangle_{|S'|}$, $J(\mathbf{x}_2) \in \langle J(\mathbf{y}) + 1, k\rangle_{|S'|}$.

Now consider the path $\mathbf{x}_1, \mathbf{x}, \mathbf{x}_2$ which splits the region contained between S_i^d and S' into two regions. Figure 4.19 illustrates one of the two possible arrangements of nodes. The representative must be contained within one of these two regions, however either $\mathbf{y}$ or $\mathbf{y}'$ will be within the region which does not contain the representative by the method of construction. By assumption the nodes $\mathbf{y}, \mathbf{y}'$ are not connected to the node $\mathbf{x}$, meaning there is not a rudimentary path from one region to the other, disconnecting at least one of $\mathbf{y}, \mathbf{y}'$ from the representative. This is a contradiction meaning our initial assumption must be incorrect and the neighbours $S'(\mathbf{x})$ must satisfy $J(S'(\mathbf{x})) = \langle j, k\rangle_{|S'|}$ for some $j, k \in [0, |S'|)$. □

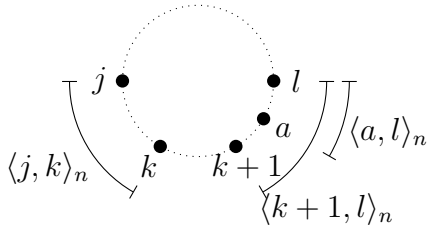
The neighbours of neighbours also follow a strict pattern. The cyclic intervals for neighbours are adjacent, however they may share a single node in between or be separate. If the neighbours contain the entire slice this property may apply to both



(a) The two cases for neighbouring intervals of neighbouring nodes $\mathbf{x}_1$ and $\mathbf{x}_2$.



(b) The counter example in case 1 when $|J(S'(\mathbf{x}_1))| = 2$.



(c) The relationship between the two cyclic intervals $J(S'(\mathbf{x}_1))$ and $J(S'(\mathbf{x}_2))$.

Figure 4.20: An illustration of the properties used in lemma 4.2.30.

ends.

Lemma 4.2.30 (Neighbour Interval Property)**.** *Suppose* $\mathbf{x}_1, \mathbf{x}_2 \in S_i^d$ *and* $S' = S_i^{d\pm1}$ *then: If* $J(\mathbf{x}_1) \equiv J(\mathbf{x}_2)+1 \pmod{|S_i^d|}$ *then the neighbours are of the form* $J(S'(\mathbf{x}_1)) = \langle j, k\rangle_{|S'|}$ *and* $J(S'(\mathbf{x}_2)) = \langle k', l\rangle_{|S'|}$ *for some* $j, k, l \in [0, |S'|), k' = \{k, k+1\}$.

Proof. Figure 4.20a illustrates the two cases in which two neighbours can have joint or separate neighbouring intervals. We aim to show that these intervals cannot overlap and cannot have a gap between them. We will consider two cases separately. In Case 1 we consider the possibility that the intervals overlap. In Case 2 we consider that the intervals have neighbours between them.

Case 1. Suppose $\mathbf{x}_1, \mathbf{x}_2 \in S_i^d$ and $S' = S_i^{d\pm1}$. However $J(S'(\mathbf{x}_1)) = \langle j, k\rangle_{|S'|}$ and $k-1 \in J(S'(\mathbf{x}_2))$, where $|J(S'(\mathbf{x}_1))| \geq 3$. Figure 4.20b illustrates why the interval needs to have more than 2 indices as the condition can still be met by wrapping around the Slice.

Now consider the region formed between the two face loops through S_i^d and S'. The arcs to the neighbours of $\mathbf{x}_1$ which have indices $k-2, k$ split the region between the two face loops into two segments. However the segment which contains the neighbour with index $k-1$ can only connect to $\mathbf{x_1}$. This contradicts the assumption that $k-1$ was an element of $J(S'(\mathbf{x}_2))$ and therefore the two neighbours cannot overlap.

Case 2. Suppose $\mathbf{x}_1, \mathbf{x}_2 \in S_i^d$ and $S' = S_i^{d\pm1}$. However $J(S'(\mathbf{x}_1)) = \langle j, k\rangle_{|S'|}$ and $J(S'(\mathbf{x}_2)) = \langle a, l\rangle_{|S'|}$ where $j, k \in [0, |S'|), l \in \langle k+1, j\rangle_{|S'|}, a \in \langle k+1, l\rangle_{|S'|}$. Figure 4.20c

illustrates how these Cyclic Intervals are related to one another. Now consider the region between the two face loops S_i^d and S'.

Also consider the arcs from $\mathbf{x}_1 \in S_i^d$ to $\mathbf{x}_1' \in S'$ and from $\mathbf{x}_2 \in S_i^d$ to $\mathbf{x}_2' \in S'$ where $J(\mathbf{x}_1') = k$ and $J(\mathbf{x}_2') = a$. These two arcs split the region between the face loops similarly to the last case. However the nodes which have indices in the interval $\langle k+1, a-1\rangle_{|S'|}$ cannot connect to either of $\mathbf{x}_1, \mathbf{x}_2 \in S_i^d$ by construction. This leads to a contradiction as there are no nodes between with which to connect to, from the condition $J(\mathbf{x}_1) \equiv J(\mathbf{x}_2) + 1 \pmod{|S_i^d|}$. Therefore $J(S'(\mathbf{x}_2)) = \langle k', l\rangle_{|S'|}$ where $k' = \{k, k+1\}$. □

For convenience we augment an existing definition when applied to indices:

Definition 4.2.31. *We augment the definition of the neighbour function $S_i^d(\mathbf{x})$ to take indices as parameters, i.e. $S_i^d(j)$, and output the intervals of the next slice. We use a positive index to indicate the neighbouring slice towards the Goal and a negative index, i.e. $S_i^d(-j)$, to indicate the neighbouring slice towards the Start.*

$$S_i^d(j) = J_d(S_i^d(J_{d+1}^{-1}(j)))$$
$$S_i^d(-j) = J_d(S_i^d(J_{d-1}^{-1}(j)))$$

We can now prove a generalized neighbourhood property for a contiguous block of nodes. We can show that the neighbours of an interval are an interval, and that the resulting interval is independent of the interior of the original interval.

Lemma 4.2.32. *For a segment of nodes in S_i^d, with cyclic interval $\langle j, k\rangle_{|S_i^d|}$, the cyclic interval of the neighbours in $S' = S_i^{d\pm1}$ is of the form $\langle j', k'\rangle_{|S'|}$. Specifically:*

$$S'(\pm\langle j,k\rangle_{|S_i^d|}) = \begin{cases} J(S') & \text{if } \mathrm{Left}(S'(\pm j)) = \mathrm{Right}(S'(\pm k)) \\ & \text{and } |S'(\pm\langle j,k\rangle_{|S_i^d|})| \neq 1 \\ \langle \mathrm{Left}(S'(\pm j)), \mathrm{Right}(S'(\pm k))\rangle_{|S'|} & \text{otherwise} \end{cases}$$

Proof. Using lemma 4.2.30 we can see that any two neighbouring nodes will have neighbouring intervals. Let $S' = S_i^{d-s}$ where $s = 1, -1$ and as such can be used to indicate the direction of the neighbouring Slice S'. If $\mathbf{x}_1, \mathbf{x}_2 \in S_i^d$ are neighbouring nodes then the two intervals are either of the form $J(S'(\mathbf{x}_1)) = \langle j, k\rangle_{|S'|}, J(S'(\mathbf{x}_2)) = \langle k, l\rangle_{|S'|}$ or of the form $J(S'(\mathbf{x}_1)) = \langle j, k\rangle_{|S'|}, J(S'(\mathbf{x}_2)) = \langle k+1, l\rangle_{|S'|}$. In either case $J(S'(\mathbf{x}_1)) \cup J(S'(\mathbf{x}_2)) = \langle j, l\rangle_{|S'|}$. Continuing this process inductively, assuming the whole face loop is not traversed in the process, we find that:

$$\bigcup_{l \in \langle j,k\rangle_{|S_i^d|}} S'(sl) = \langle \mathrm{Left}(S'(sj)), \mathrm{Right}(S'(sj))\rangle_{|S'|}$$

Eventually the entire face loop is traversed leaving the entire set as the cyclic interval $J(S')$. □

4.2.5 *Backtracking Algorithm*

Now that we have a description of the underlying graph and properties of neighbouring nodes we consider the Backtracking algorithm itself. To describe the algorithm we define a data structure called a winding. A winding represents a Cyclic Interval bounded by a path. We use the windings to define and calculate the path of the Backtracking algorithm. However only the outer edge is ever needed for the calculation.

Definition 4.2.33 (Winding, $\langle j, v]_n^d$, ${}_n^d[v, k\rangle$)**.** *We define a left winding with the notation $\langle j, v]_n^d$ and a right winding with the notation ${}_n^d[v, k\rangle$. The right and left winding represent the same set as an equivalent cyclic interval:*

$$\langle j, v]_n^d = \langle j, \mathcal{J}_d(v)\rangle_n$$
$${}_n^d[v, k\rangle = \langle \mathcal{J}_d(v), k\rangle_n$$

We also define the function $\mathrm{Edge}(w)$ *to access the outer index of the winding:*

$$\mathrm{Edge}(\langle j, v]_n^d) = j$$
$$\mathrm{Edge}({}_n^d[v, k\rangle) = k$$

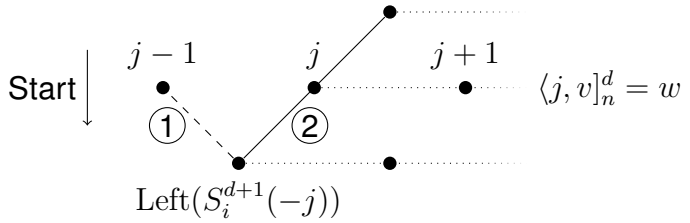
And the function $\mathrm{Side}(w)$ *to determine the type of winding:*

$$\mathrm{Side}(\langle j, v]_n^d) = Left$$
$$\mathrm{Side}({}_n^d[v, k\rangle) = Right$$

Note that we may use the $\mathrm{Side}(w)$ function to define the $\mathrm{Edge}(w)$ function as such $\mathrm{Edge}(w) = \mathrm{Side}(w)(w)$ if we treat the result as a function. For example, consider the expression $\mathrm{Side}(\langle j, v]_n^d)(\langle j, v]_n^d)$:

$$\mathrm{Side}(\langle j, v]_n^d)(\langle j, v]_n^d) = \mathrm{Left}(\langle j, v]_n^d) = \mathrm{Left}(\langle j, \mathcal{J}_d(v)\rangle_n) = j$$

For convenience we will write $\mathrm{Side}^c(w)$ to denote the complement, i.e. the reverse direction:



① $\mathrm{Inf}(w) - w = \{j-1\}$

② $\mathrm{Inf}(w) - w = \emptyset$

Figure 4.21: An illustration of the Influence function. Case ① includes the dashed line to the node indexed $j-1$ in the result of $\mathrm{Inf}(w)$. Case ② includes only up to the node indexed j in the result of $\mathrm{Inf}(w)$.

$$\mathrm{Side}^c(\langle j, v]_n^d) = Right$$
$$\mathrm{Side}^c({}_n^d[v, k\rangle) = Left$$

The Backtracking algorithm is split into two portions, the first portion is a search for a suitable diversion around the collision point. As such we need to detect when a new option occurs. We achieve this by the use of the Influence Function $\mathrm{Inf}(w)$. The Influence Function calculates when a winding can be extended by its neighbouring nodes. If the winding cannot be extended in this way then the winding is self contained and therefore all options lead to the collision.

Definition 4.2.34 (Influence function, $\mathrm{Inf}(w)$)**.** *We define a function* $\mathrm{Inf}(w)$ *called the influence function.*

$$\mathrm{Inf}(\langle j, v]_n^d) = \langle \mathrm{Left}(S_i^d(\mathrm{Left}(S_i^{d+1}(-j)))), v]_n^d$$
$$\mathrm{Inf}({}_n^d[v, k\rangle) = {}_n^d[v, \mathrm{Right}(S_i^d(\mathrm{Right}(S_i^{d+1}(-k))))\rangle$$

Figure 4.21 illustrates how the influence function detects new options. Two possibilities are shown, case ① shows the case where the node indexed by $\mathrm{Left}(S_i^{d+1}(-j))$ on slice S_i^{d+1} has an additional option of $j-1$. Case ② the figure shows the case where no additional options are available other than those given by w.

In the subsequent definitions label common variables to associate them with particular portions of the Backtracking Algorithm. Variables labelled with the letter o will be associated with the First Option Algorithm, for example windings such as w_a^o, and nodes such as $\mathbf{x}_a^o$. Variables labelled with a c are associated with the Minimal Connection Algorithm, for example windings such as w_a^c, and nodes such as $\mathbf{x}_a^c$. We also

associate variables labelled with b with the boundary of the of the Backtracking Search, variables such as w_d^b and $\mathbf{x}_d^b$ are indexed by their distance from the Goal.

Using the Influence Function and the idea of a winding we define the first portion of the Backtracking Algorithm which we call the First Option Algorithm:

Definition 4.2.35 (First Option)**.** *We define the First Option algorithm as a recursive application of the following function:*

$$\mathrm{Out}(\langle j, v]_n^d) = \begin{cases} \langle \mathcal{J}_{d+1}(v) + 1, v]_{n'}^{d+1} & \text{if } \mathcal{J}_{d+1}(v) \in S_i^{d+1}(j) \\ \langle j', v]_{n'}^{d+1} & \text{otherwise } S_i^{d+1}(j) = \langle j', k' \rangle_{n'} \end{cases}$$

$$\mathrm{Out}({}_n^d[v, k\rangle) = \begin{cases} {}_{n'}^{d+1}[v, \mathcal{J}_{d+1}(v) - 1\rangle & \text{if } \mathcal{J}_{d+1}(v) \in S_i^{d+1}(j) \\ {}_{n'}^{d+1}[v, k'\rangle & \text{otherwise } S_i^{d+1}(j) = \langle j', k' \rangle_{n'} \end{cases}$$

The algorithm proceeds in the following steps:

1. *Given an initial winding* $w_1^o = \langle \mathcal{J}_d(v), v]_n^d, {}_n^d[v, \mathcal{J}_d(v)\rangle$*.*

2. *If* w_a^o *is of the form:*

   - *If* $w_a^o = [0, |S_i^d|)$ *then the algorithm exits. (The winding* w_a^o *has travelled around the entire graph, leaving no options available to avoid the collision).*
   - *If* $\mathrm{Inf}(w_a^o) - w_a^o = \emptyset$ *we continue. (No new options are available)*
   - *Else if* $\mathrm{Inf}(w_a^o) - w_a^o \neq \emptyset$ *we exit with the result* $w = \mathrm{Out}(w_a^o)$*, and the index* $I = \mathrm{Side}^c(w_a^o)(\mathrm{Inf}(w_a^o) - w_a^o)$*. (Additional options have been discovered)*

3. *We replace* w_a^o *with* $w_{a+1}^o = \mathrm{Out}(w_a^o)$ *and repeat step 2.*

When we refer to windings used in the First Option algorithm we denote them as $w_1^o, w_2^o, \ldots, w_{N^o}^o$*, where* N^o *is the number of windings traversed in the calculation. We refer to nodes on the outer edge of the windings as* $\mathbf{x}_a^o = J_d^{-1}(\mathrm{Edge}(w_a^o)), w_a^o = \langle j, v]_n^d, {}_n^d[v, k\rangle$*. We may also refer to each winding with respect to its distance from the goal; e.g.* $w_l^o = \langle j, v]_n^d = w_d^b$*.*

After a suitable point has been discovered to avoid the collision a path has to be constructed to connect the edge of the winding to the original path with the minimal diversion. We define the the Minimal Connection Algorithm for this purpose.

Definition 4.2.36 (Minimal Connection)**.** *We define the Minimal Connection algorithm*

as a recursive application of the following function:

$$\mathrm{In}(\langle j,v]_n^d) = \begin{cases} \langle \mathcal{J}_{d+1}(v), v]_{n'}^{d+1} & \textit{if } \mathcal{J}_{d+1}(v) \in S_i^{d+1}(j) \\ \langle k', v]_{n'}^{d+1} & \textit{otherwise } S_i^{d+1}(j) = \langle j', k' \rangle_{n'} \end{cases}$$

$$\mathrm{In}({}_n^d[v,k\rangle) = \begin{cases} {}_{n'}^{d+1}[v, \mathcal{J}_{d+1}(v)\rangle & \textit{if } \mathcal{J}_{d+1}(v) \in S_i^{d+1}(j) \\ {}_{n'}^{d+1}[v, j'\rangle & \textit{otherwise } S_i^{d+1}(j) = \langle j', k' \rangle_{n'} \end{cases}$$

The algorithm proceeds in the following steps:

1. *Given an initial winding* $w_1^c = \langle j,v]_n^d, {}_n^d[v,k\rangle$.
2. *We label the edge of the winding* $w_a^c = \langle j,v]_n^d, {}_n^d[v,k\rangle$ *as the node* $\mathbf{x}_a^c = J_d^{-1}(\mathrm{Edge}(w_a^c))$.
3. *If* w_a^c *is of the form* $w_a^c = \langle \mathcal{J}_d(v), v]_n^d, {}_n^d[v, \mathcal{J}_d(v)\rangle$ *then the algorithm exits with the sequence* $\mathbf{x}_1, \ldots, \mathbf{x}_a^c$ *as the result.*
4. *We replace* w_a^c *with* $w_{a+1}^c = \mathrm{In}(w_a^c)$ *and repeat step 2.*

When we refer to windings used in the Minimal Connection algorithm we denote them as $w_1^c, \ldots, w_{N^c}^c$*, where* N^c *is the number of windings traversed in the calculation. W refer to nodes on the outer edge of the windings as* $\mathbf{x}_a^c = J_d^{-1}(\mathrm{Edge}(w_a^c)), w_a^c = \langle j,v]_n^d, {}_n^d[v,k\rangle$*. We may also refer to each winding with respect to its distance from the goal; e.g.* $w_l^c = \langle j,v]_n^d = w_d^b$

We apply the First Option and Minimal Connection Algorithms on both sides of the path v. If the algorithm calculates at least one solution they can be used to continue the search. The boundary of the search can be used as a bypass to avoid further calculation within the searched area.

A collision can occur in two forms. One form of collision is where two agents occupy a node at the same time step. With this first form of collision we must remove the point of collision from consideration therefore we ignore branches from the node of the collision. The second form of collision is a swap between nodes. With this form of collision the agent can occupy the node before the collision if there exists another connection which bypasses the connection on which the collision occurs.

Definition 4.2.37 (Backtracking Algorithm)**.** *We define the Backtracking Algorithm by concatenating the First Option and Minimal Connection Algorithms.*

The algorithm proceeds in the following steps for a path v *at a distance* d *from the Goal:*

1. *Define 2 windings* $w_1 = \langle \mathcal{J}_d(v), v]_n^d, w_2 = {}_n^d[v, \mathcal{J}_d(v)\rangle$.
2. *If the collision was of the form of a swap:*

   (a) *If* $\mathcal{J}_{d-1}(v) - 1 \in S_i^{d-1}(\mathcal{J}_d(v))$ *then we assign* $w_1' = w_1$ *and* $I_1 = \mathcal{J}_{d-1}(v) - 1$. *We then skip Step 3 for the left side of the algorithm.*

   (b) *If* $\mathcal{J}_{d-1}(v) + 1 \in S_i^{d-1}(\mathcal{J}_d(v))$ *then we assign* $w_2' = w_2$ *and* $I_2 = \mathcal{J}_{d-1}(v) + 1$. *We then skip Step 3 for the right side of the algorithm.*

3. *Apply the First Option Algorithm to both* w_1, w_2 *giving the results* w_1', w_2' *and indices* I_1, I_2 *for the connecting nodes. If either application of the algorithm was successful continue. Otherwise the collision is unavoidable at this cost.*

4. *Apply the Minimal Connection Algorithm on* w_1', w_2' *resulting in the two sequences* $\mathbf{x}_1^1, \ldots, \mathbf{x}_{N_1}^1$ *and* $\mathbf{x}_1^2, \ldots, \mathbf{x}_{N_2}^2$. *If either application of the algorithm was successful continue.*

5. *Concatenate the node indexed by* I_1 *to sequence 1, and Concatenate the node indexed by* I_2 *to sequence 2. These two paths connect to the latest point in time for which* $\mathbf{x}_{N_1}^1$ *and* $\mathbf{x}_{N_2}^2$ *lie on* v *respectively.*

6. *We define two types of results, the boundary formed from the windings used in First Option and Minimal Connection;* $\mathbf{x}_d^b = J_d^{-1}(\mathrm{Edge}(w_d^b))$*, and the resulting path.*

There are two important properties that we need to prove about this algorithm. The first is that the algorithm calculates the minimal divergence from the original path which avoids the target space time point. The second is that we still reach all possible permutations of paths given the opportunity, this proves that we cover all possible solutions. It may be noted that all permutations need not be computed if a collision does not force the calculation. One property which we do not prove is that paths will not be duplicated during computation. This is remedied in the implementation by detecting combinations of paths which have been attempted before and pruning them, irrespective of this, this property is not necessary for correctness.

Lemma 4.2.38 (Minimal Avoidance)**.** *Using the backtracking algorithm a path which minimises* $\widetilde{\mathbf{p}}_t - \widetilde{\mathbf{q}}_t$ *where* $\widetilde{\mathbf{p}}$ *is the point of collision and* $\widetilde{\mathbf{q}}$ *is the point of connection selected by the Minimal Connection algorithm.*

Proof. Suppose there existed a path u which diverged at $\widetilde{\mathbf{q}}'$ from path v as opposed to $\widetilde{\mathbf{q}}$ (shown in figure 4.22a), where $\widetilde{\mathbf{q}}_t' > \widetilde{\mathbf{q}}_t$ avoiding the space time point $\widetilde{\mathbf{p}}$. Note that we can assume that this path will only intersect again with v after time $\widetilde{\mathbf{p}}_t$ since we could otherwise cut out the earlier part of the diversion while still avoiding $\widetilde{\mathbf{p}}$. As shown in figure 4.22b the upper portion of either example could be used ignoring the earlier divergence from v. Without loss of generality we can assume that the first point of divergences happens to the $Left$ of the path v.

From the definition of $\mathrm{In}(w)$ and $\mathrm{Out}(w)$ the $\mathbf{x}_d^b$ form a connected path and the path formed by the $\mathbf{x}_d^b$ and v form a bypass. Any solution u which starts at $\widetilde{\mathbf{q}}'$ will start inside

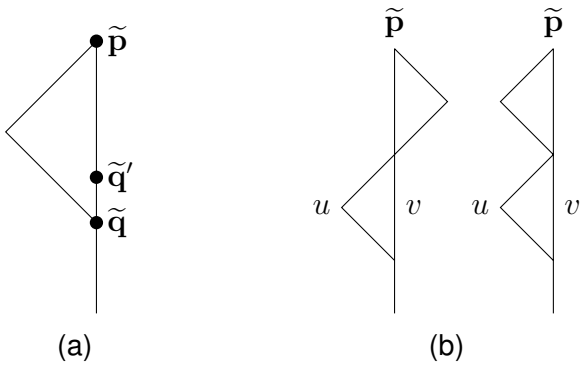


Figure 4.22: An illustration of the space time points related to the Minimal Avoidance proof and an illustration of the simplifications that can be made to a solution with multiple intersections with v.

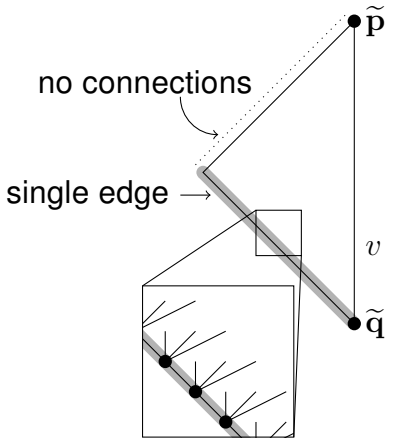


Figure 4.23: An illustration of the Backtracking Algorithm, showing the properties that the First Option and Minimal Connection Algorithms have to neighbouring nodes.

this bypass, and will at some point exit the region. Since the path u does not pass back through v before the end of the bypass it must intersect during the First Option or Minimal Connection phase of the algorithm.

The First Option phase happens during the sequence of windings $w_1^o, \ldots, w_{N^o}^o$. By Lemma 4.2.32 we know that the neighbours of a cyclic interval $\langle j, k \rangle_n$ form a cyclic interval $\langle j', k' \rangle_{n'}$ where the neighbours of the edges j, k match the edges of the cyclic interval j', k'. This shows that the Influence Function computes the additional options which can be reached from a winding, i.e. the Influence of w_l^o is defined to be the extent to which the earlier winding w_{l+1}^o can extend the scope of w_l^o. During the processing of the First Option phase of the algorithm we continue while $\mathrm{Inf}(w_l^o) - w_l^o = \emptyset$; which implies that w_{l+1}^o cannot reach any nodes which are not already contained within w_l^o without crossing v. Therefore there is one node in $w_{N^o}^o$ which can avoid the collision, the node indexed by $\mathrm{Edge}(w_{N^o}^o)$.

The Minimal Connection phase occurs during the windings $w_1^c, \ldots, w_{N^c}^c$. During the Minimal Connection phase each of the windings is constructed from the nearest neighbour of the edge $\mathrm{Edge}(w_l^c)$ to the path v by using $\mathrm{In}(w_l^c)$. This implies that $\mathrm{Edge}(w_l^c)$ has only one connection to w_{l+1}^c. Figure 4.23 illustrates the properties of the First Option

and Minimal Connection Algorithms as discussed above.

Now consider the hypothetical path from $\widetilde{\mathbf{q}}'$ which avoids $\widetilde{\mathbf{p}}$. We have proven that there can be no exit for u from within the first windings, $w_1^o, \ldots, w_{N^o-1}^o$, and that the edge of the windings $w_1^c, \ldots, w_{N^c-1}^c$ has only one connection to the next winding in the sequence, meaning that there is only one path into the edge of $w_{N^o}^o$ from within the bypass. This contradicts the fact that there existed a path u parting from v at a later time than $\widetilde{\mathbf{q}}_t$. □

Lemma 4.2.39. *The application of the backtracking algorithm is a permutation of the equivalent paths of a space time point.*

Proof. To prove this lemma we need to show that any path which can represent a point can be generated by application of the Backtracking Algorithm with an appropriate sequence of collision points. Given a path v, supposing that a collision occurs at $\widetilde{\mathbf{p}}^c$ and that a path u is a solution which avoids $\widetilde{\mathbf{p}}^c$, we will show that backtracking will find a path identical to u or one that diverges from v sooner than u. We will also assume that u does not rejoin v after it separates as this could be reduced into multiple applications of the Backtracking Algorithm.

Without loss of generality we will assume the solution u exits on the $Left$ side of the path v. From lemma 4.2.38 we know that the difference between the point at which the collision occurs $\widetilde{\mathbf{p}}^c$ and the point at which the result of backtracking diverges $\widetilde{\mathbf{q}}^c$ is minimized. From this we can see that the point of divergence $\widetilde{\mathbf{q}}'$ of u from v satisfies $\widetilde{\mathbf{q}}'_t \leq \widetilde{\mathbf{q}}^c_t$. The Minimal connection algorithm always picks the inner-most neighbour in $w_{N^c-1}^c$ that not on v.

Suppose $\widetilde{\mathbf{q}}' = \widetilde{\mathbf{q}}^c$ and the algorithm is iterated to find a solution. Consider the constructed path to the Goal. The path will be constantly travelling towards the Goal each branch travelling from one slice to the next. There are a finite number of slices towards the Goal and a finite number of nodes within each slice. This implies that only a finite number of paths to the Goal can be generated. As the algorithm proceeds collisions will be detected at every point not on the solution path u.

We must show that all paths will eventually be considered. First consider the slice S_i^d which contains the output node from the Minimal Connection Algorithm. From Lemma 4.2.38 we know that if there were a collision on this node and other neighbours were available at this step in the algorithm then they would have a time step difference of 1 between $\widetilde{\mathbf{p}}, \widetilde{\mathbf{q}}$. Since the inner most node will always be considered first, the algorithm will increment through each of these potential neighbours.

When an earlier branching point $\widetilde{\mathbf{q}}$ is selected we know from Lemma 4.2.30 and iterative application of Lemma 4.2.32 that the next set of nodes passed through in slice S_i^d will be disjoint and continue from the previous set. In this manner all nodes of a given slice will be considered eventually, starting from the latest slice. Once all the nodes up to a given time t pass along a solution they need not be considered again.

Nodes after time step t will be searched until at least the node at $t+1$ becomes part of a solution. This will continue until the Goal is reached or all points after $\widetilde{\mathbf{q}}'$ are exhausted.

If on the other hand $\widetilde{\mathbf{q}}'_t < \widetilde{\mathbf{q}}^c_t$ then the iteration process will first saturate all possible branches at $\widetilde{\mathbf{q}}^c$ forcing the algorithm to proceed to the next available point at which it can branch from v. The potential solutions of the new branch point will be explored until they are saturated and the process will continue until $\widetilde{\mathbf{q}}' = \widetilde{\mathbf{q}}^c$; in which case as before a solution will eventually be found. □

To apply the algorithms which we have described so far, the Backtracking Algorithm and the *Next* Algorithm, we need to split potential solutions into a number of cases. The Backtracking Algorithm is designed to be applied to Non Complex paths. This results in a redirection of a path around a collision, however this cannot solve collisions which occur on the Goal node as the Goal node cannot be avoided. To solve collisions after the Goal has been reached we apply the *Next* Algorithm.

Another split in the solution search space is Complex solutions. We analyse complex paths by splitting the path into a Complex Segment and a Non Complex Segment. The Non Complex Segment is defined by the steps before the first Complex Branch. We will show that we need only analyse the Complex paths which branch from the boundary the bypasses computed in the Backtracking Algorithm. We show that there will always be an equivalent path to a solution which branches within a bypass, or the centre of the bypass will eventually be considered.

Once a path has made a complex branch we apply the *Next* algorithm to the complex portion. The point at which the algorithm selects a complex branch can be considered a new start node allowing the application of the *Next* algorithm.

Definition 4.2.40 (Journey and Stationary Segments)**.** *We define a function* $\mathrm{Journey}(v)$ *to equal the time the path* v *takes to initially reach the Goal:*

$$\mathrm{Journey}(v) = \min\{t\colon D_{\mathbf{g}}(P_t(v)) = 0\}$$

We define the segment of the path v *before the time* $\mathrm{Journey}(v)$ *to be the Journey Segment of the path* v *and the segment after the time* $\mathrm{Journey}(v)$ *to be the Stationary Segment of the path* v*.*

Lemma 4.2.41. *Suppose a solution* u *of cost* c *exists, along with an algorithm* Sol *which can solve for* u' *where* $u' \sim_{\mathrm{Journey}(u)} u$*. A minimal multi-agent solution* $\mathbf{u}$ *can be solved in two Segments split by the time* $\mathrm{Journey}(u_i)$ *for each path* u_i *in* $\mathbf{u}$*.*

Proof. To prove this lemma we will split the process of solving for a complete solution $\mathbf{v}$ into two steps. The first step will remove from consideration any collision for agent i in configuration $\mathbf{v}_{-i}$ after $\mathrm{Journey}(v)$. These are the initial segments of the full solution $\mathbf{v}$. By assumption the *Sol* Algorithm can solve for these initial segments, therefore the first step is to apply *Sol*.

The second step is achieved by applying the *Next* Algorithm to an initial segment v after the time $\mathrm{Journey}(v)$ without ignoring any collisions from $\mathbf{v}_{-i}$. From theorem 3.1.33 we know that a minimal solution will be calculated from v to include the collisions after $\mathrm{Journey}(v)$. Complete multi agent solutions $\mathbf{v}$ produced by step two will be sorted by the overall cost including any diversions after the Journey Segments. Eventually by assumption the *Sol* algorithm will produce the required initial segment of a minimal solution which will then be solved by the *Next* algorithm giving an overall minimal solution. □

Definition 4.2.42 (Complex Branch, $\mathrm{CB}(v)$)**.** *We define the function* $\mathrm{CB}(v)$ *to indicate the first complex branch of a path* v*:*

$$\mathrm{CB}(v) = \min\{t \colon \mathrm{Cx}_t(v)\}$$

We will now define the Path Selection Algorithm which serves the purpose of the *Sol* Algorithm in the previous lemma. A path v is processed searching for a solution up to the time $\mathrm{Journey}(v)$. The Path Selection Algorithm works in passes. Each pass of the Path Selection Algorithm takes a tuple of data. Each tuple can either be a path from a potential solution $\mathbf{v}$ or a variation on a Stem. Three Stems are considered, the *Next* Stem, the Non Complex Stem and the Complex Stem. Each Stem explores the solution search space by extending the cost of certain portions of the given path v, which we will show in a later Theorem.

Definition 4.2.43 (Path Selection Algorithm)**.** *Here we define the algorithm used to select a set of options for the next path for the algorithm to consider, for both the same cost and higher cost solutions. Any path produced is combined with the given configuration* $\mathbf{v}_{-i}$ *which is the context of the collision. Per application of this algorithm a single number is selected based on the which condition is met (Complex/Non Complex/Next Stem/etc...), subsequently each letter is processed producing new possibilities to be considered. During the processing of this algorithm we ignore collisions after* $\mathrm{Journey}(v)$ *as they are handled by the technique given in Lemma 4.2.41.*

The input of this algorithm is a tuple of data, giving the path v *to be operated on, a point* $\widetilde{\mathbf{p}}$ *or collision time* t*, the label given to the tuples selects which part of the algorithm is used to process the tuple. There may also be a target cost* c *for the solution when the tuple is in the form of a Stem. The algorithm proceeds in the following steps:*

1. v *Non-Complex or* $\widetilde{\mathbf{p}}_t \leq \mathrm{CB}(v)$*:*
   (a) *Apply the Backtracking Algorithm producing two paths* u, u' *which avoid the collision at* $\widetilde{\mathbf{p}}$*.*
   (b) *We construct a Complex Stem from the two boundary paths of each Backtracking application before* $\widetilde{\mathbf{p}}_t$*.*
   (c) *We construct a Non-Complex stem before* $\widetilde{\mathbf{p}}_t$ *on* v *at cost* $c = F(v) + 1$*.*

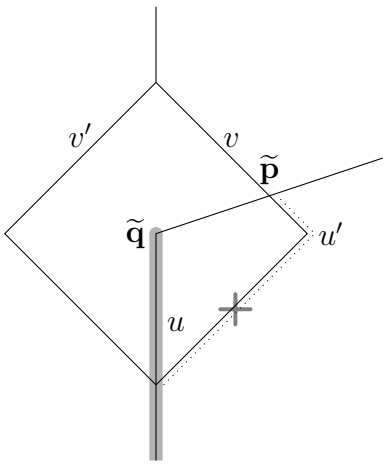


Figure 4.24: An illustration of a Complex solution which makes its Complex branch within the bypass v, v'.

2. *v Complex and $\widetilde{\mathbf{p}}_t > \mathrm{CB}(v)$*

   (a) *We apply the* Next *algorithm along the path defined after $\mathrm{CB}(v)$ on the path v.*

   (b) *We construct a regular* Next *Stem after $\widetilde{\mathbf{p}}_t$ on v at cost $c = F(v) + 1$.*

   (c) *We construct a Non-Complex Stem before $\mathrm{CB}(v)$ on v' at cost $c = F_{\mathrm{CB}(v)}(v) + 1$. Where v' is a non-complex path which satisfies $v \sim_{\mathrm{CB}(v)} v'$ and $\forall t > \mathrm{CB}(v), I_t(v') = 0$.*

3. *Next Stem v, t, c:*

   (a) *We apply the* Next *algorithm along the path defined after $\mathrm{CB}(v)$ before the collision at time t targeting the cost c.*

   (b) *We construct a regular* Next *Stem before time t on v at cost $c' = c + 1$.*

4. *Non-Complex Stem v, t, c:*

   (a) *We apply the* Next *algorithm along the path v restricted to the subgraph defined by the set $P(v)$ before the collision at time t targeting cost c.*

5. *Complex Stem v, t:*

   (a) *We construct a complex path u for each time such that $u \sim_{\mathrm{CB}(u)} v, \mathrm{CB}(u) < t$ and $F(u) = F_t(v), F_t(v) + 1, F_t(v) + 2$.*

Lemma 4.2.44. *Given a path u which makes a Complex Branch within a bypass v, v', the path u must coincide with a Non-Complex point on the boundary of the bypass v, v'.*

Proof. We are given two paths v, v' which form a bypass. Suppose there exists a solution u which makes its complex branch within the region enclosed by v, v'. The paths v, v' both lie on the non-complex subgraph however complex paths are also restricted

by the planar condition of the entire graph, therefore there exists a spacetime point $\widetilde{\mathbf{p}}$ on u which crosses over either v or v' at a node which lies on either path. □

Lemma 4.2.45. *Given a solution u which makes a Complex Branch within the bypass v, v', the Path Selection Algorithm will eventually compute a path u' which makes a Complex Branch that coincides with u. The path u' will not have any collisions before* $\mathrm{CB}(u')$.

Proof. To prove that the Path Selection Algorithm will select an appropriate path we will analyse a series of events. Either a path will be found along the boundary of the bypass or a collision will occur along said boundary. If a collision has occurred this will allow the application of the Backtracking Algorithm, which will give a new set of boundary paths v_1, v_1'. Continuing this process we either construct a new path along one of the boundary paths or we eventually converge along u allowing the construction of a Complex Stem which branches along the same initial complex branch. Figure 4.24 illustrates the configuration of paths u, u', v, v' and several important space time points. A collision is also illustrated by a cross along the dotted line representing u'.

By lemma 4.2.44 there exists a space time point $\widetilde{\mathbf{p}}$ on the path u which crosses the boundary of the bypass v, v'. Consider the boundary path v or v' which contains this point, i.e. the path $b_v = v, v'$ such that $\widetilde{\mathbf{p}}_{\mathbf{x}} \in P(b_v)$. Suppose the current path under consideration is called u' then there exists a time at which u' reaches $\widetilde{\mathbf{p}}_{\mathbf{x}}$, i.e. $\exists t \leq \widetilde{\mathbf{p}}_t, P_t(u') = \widetilde{\mathbf{p}}_{\mathbf{x}}$.

If there is no collision along u' before t then the Path Selection Algorithm can select the appropriate branch which coincides with u. If the branch is Non Complex the Backtracking Algorithm will select the appropriate branch otherwise if the branch was Complex then a Complex Stem will select the appropriate branch. This path u' will satisfy the conditions of the lemma and therefore prove the result.

However if there is a collision before t along u' then either another application of a Non Complex stem will occur raising the cost of the path until $t = \widetilde{\mathbf{p}}_t$ or there will be an application of the Backtracking Algorithm which leads to an appropriate solution.

Consider the point $\widetilde{\mathbf{q}}$ where the path u makes a complex branch. The space time point $\widetilde{\mathbf{q}}$ is a non-complex point which belongs to the bypass v, v'. Points on u before $\widetilde{\mathbf{q}}_t$ have no collisions as u is a solution. By application of lemma 4.2.39 if a solution isn't found a path to $\widetilde{\mathbf{q}}$ will be found in a finite number of steps. Therefore either an appropriate solution will be found along a boundary before $\widetilde{\mathbf{q}}$ is reached or the path up to $\widetilde{\mathbf{q}}$ will be reached and a Complex Stem will discover the complex branch along u. □

Theorem 4.2.46. *The application of the Path Selection Algorithm as the* Sol *algorithm of Lemma 4.2.41 will find an optimal solution if one exists from the set of equivalent solutions.*

Proof. To prove this theorem we assume a minimal solution u exists. We will split the solution u into possible cases, proving u or an equivalent solution will be found.

We make the assumption that no collision occurs after $\mathrm{Journey}(v)$ by applying lemma 4.2.41. The following list gives the structure of the individual cases so that they can be discussed with reference to their case letter:

- u is Non-Complex: Case (a).
- u is Complex:
  - u not on a boundary: Case (b) continue using the new u', v.
  - u on a boundary v:
    - u satisfies the condition $F_{\mathrm{CB}(u)}(u) = F(u) - 2, F(u) - 1, F(u)$:
      - u and v satisfy $F_{\mathrm{CB}(u)}(v) = F_{\mathrm{CB}(u)}(u)$: Case (c).
      - Otherwise $F_{\mathrm{CB}(u)}(v) < F_{\mathrm{CB}(u)}(u)$: Case (d).
    - u satisfies $F_{\mathrm{CB}(u)}(u) < F(u) - 2$: Case (e).

Case (a). u is non-complex. By lemma 4.2.39 eventually a representative of the Goal Node, which traces the same rudimentary path as a solution, will be selected by the Backtracking Algorithm. Supposing that the solution u is picked as a representative for the Goal node when it is reached, we can assume the solution has a rudimentary form r_u. Following the Path Selection Algorithm as it proceeds the *Next* algorithm will be applied repeatedly on the paths which can be reduced to r_u as the potential solutions increase in cost. However if we restrict the graph to the nodes which r_u traverses Theorem 3.1.33 shows that an optimum solution will be found.

Case (b). Lemma 4.2.45 shows that either a new solution u' will be found on a boundary v, or a boundary v covering u will be constructed. If a boundary is constructed containing the Non Complex Segment of u a Complex Stem will discover the complex branch along u. The algorithm will continue, cases (c), (d) and (e) cover the further possibilities after the correct complex branch is found.

Case (c). Each complex branch which does not extend from the Goal adds an additional cost of 0, 1 or 2 to the overall cost of a path. When the complex solution u satisfies one of the equalities $F_{\mathrm{CB}(u)}(u) = F(u) - 2, F(u) - 1, F(u)$ then u has a single Complex branch before the rest of the path travels directly towards the Goal.

For this case we assume that the boundary v is considered and satisfies $F_{\mathrm{CB}(u)}(v) = F_{\mathrm{CB}(u)}(u)$ in relation to the solution u. When the Complex Stem is considered which covers v then $F_{\mathrm{CB}(u)}(v) = F_{\mathrm{CB}(u)}(u) = F(u) - 2$ which implies that u will be selected up to time $\mathrm{CB}(u)+1$ from part 5 of the Path Selection Algorithm. After repeated application of part 2 of the Path Selection Algorithm sdf(specifically part 2a) the solution u or an equivalent solution will be constructed as proven by Theorem 3.1.33.

Case (d). For this case we retain the condition that $F_{\mathrm{CB}(u)}(u) = F(u) - 2$, however the boundary v does not meet the solution u at the correct time, i.e. $F_{\mathrm{CB}(u)}(v) < F_{\mathrm{CB}(u)}(u)$. Since the first condition is met, $F_{\mathrm{CB}(u)}(u) = F(u) - 2$, there will be a path

u' considered which follows the complex branch of u at time $\mathrm{CB}(u)$ however since $F_{\mathrm{CB}(u)}(v) < F_{\mathrm{CB}(u)}(u)$ this will occur at an earlier time, i.e. $\mathrm{CB}(u') < \mathrm{CB}(u)$.

As the Path Selection Algorithm proceeds the path u' will be replaced by higher cost paths. When part 2c of the Path Selection Algorithm is applied it increases the cost of the earlier Non Complex portion of the boundary v. Another application of the Complex Stem will produce the same Complex Branch and the process can repeat. Eventually we will attain $\mathrm{CB}(u') = \mathrm{CB}(u)$ and case (c) will apply.

Case (e). Through the earlier cases we have shown that the correct boundary path v will be computed and that the correct Complex Branch will be selected at the right time. For this case we need to show that the correct Complex Segment of the path u can be constructed. This can be seen as an application of part 2a of the Path Selection Algorithm and follows from Theorem 3.1.33 applied after the Complex Branch at time $\mathrm{CB}(u)$. □

Chapter 5. Implementation

5.1 Algorithm

In this section we discuss the implementation of the original and extended version of the Algorithm. First we discuss data-structures we need during the execution of the algorithm. Then we describe the structure of the *Next* and *Backtracking* Algorithms; outlining the details which were ambiguous during the Methodology (Chapters 3 & 4) sections. The pseudo code for the algorithms in this chapter can be found in appendix B.

5.1.1 *Distance & Order Preference*

As described in Chapter 3 the idea of distance is important to the correct and fast calculation of minimal paths. We use a modified version of the Dijkstra's Algorithm to calculate the minimal distance for any node $\mathbf{x}$ to any target node $\mathbf{y}$ (this includes any Start and Goal node for any agent). This algorithm is analogous to the Reverse Resumable A* [Sil05] used in other papers on the MAPF problem. We use the term Reverse Resumable Dijkstra to describe this Algorithm (RRD). The RRD Algorithm works by applying the Dijkstra Algorithm in reverse from the target node $\mathbf{y}$ until the referenced node $\mathbf{x}$ is encountered, at which time the computation is frozen for later use. In this manner any subsequent distance lookups $\mathbf{x}'$ which involve $\mathbf{y}$ can use already discovered distances or resume calculating the Dijkstra's Algorithm until $\mathbf{x}'$ is encountered.

5.1.2 *Agentverses, Path Representation*

In Chapter 3 we discussed the existence of the most preferred paths called α paths. To represent arbitrary paths in our algorithm we construct a data-structure called an Agentverse based on these α paths. Every path can be described in terms of α paths. We can construct an arbitrary path inductively on the number of non-preferred options on the path, i.e. induction on:

$$\sum_{t=0}^{\infty}[I_t(v) \neq 0] = n$$

when we have $n = 0$ we have an α path and the Agentverse stores the entire α path. Otherwise suppose s represents the last non-preferred time step we store the α path after time step s in our Agentverse. We then reference the Agentverse with $n - 1$ non-preferred steps as our parent by removing the non-preferred option at time step s. This makes the basis of the Agentverse data-structure which represents v the parent $v.parent$, the start time $v.startTime$ and the remaining α path representing the final steps of v.

To facilitate the *Next* Algorithm we store additional information in the header of

the Agentverse data-structure and along side each times step within the α path. The Agentverse data-structure without Backtracking is as follows:

```
struct TimeUnit:
    Agentverse * next;
    location pos;
    unsigned short relCost: 2;
    unsigned short priority[2];
end
struct Agentverse:
    Agentverse * next;
    Agentverse * parent;
    unsigned char ind: 7;
    unsigned char complex: 1;                    // Used with Backtracking
    unsigned int id;
    unsigned int startTime;
    unsigned short length;
    unsigned short cost;
    unsigned short ext;                          // Used with Backtracking
    unsigned short cb;                           // Used with Backtracking
    unsigned short pq[3];
    TimeUnit * steps;                           // Steps of the alpha path
end
```

Algorithm 1: Agentverse Data-structure.

Within the Agentverse v we store more than the basis of the path ($v.startTime$, $v.parent$ and $v.steps$). Certain values are stored to avoid re-computation of their value, i.e. $v.cost, v.length$. Given an arbitrary step s at time step t the variable $s.next$ is used to avoid the re-computation of paths which branch from time step t. The $v.next$ variable of an Agentverse allows for linked list of all Agentverses which branch from time step t. The $v.ind$ variable identifies what index the Agentverse v branches from time step t.

The $v.id$ variable allows for a unique identification of this Agentverse which is needed for construction of a closed set for collections of Agentverses. The variables $v.pq$ from the Agentverse v and $s.relCost, s.priority$ for the time step s are used in the next algorithm and will be discussed next. The variables $v.complex, v.ext, v.cb$ are used in backtracking and will be discussed later.

Figure 5.1 shows an illustration of an example Agentverse with one non-trivial branch. The close-up section highlights the TimeUnit data structure used to store the path and meta data for the Agentverse.

5.1.3 *The* Next *Algorithm*

From the proof in Chapter 3 we know that all paths can be placed in an ordering. The basis of our algorithm is to increment along this order trying permutations of possible path configurations. The *Next* Algorithm operates on paths incrementing them along to the next available path in the ordering of the required cost. The pseudo code in

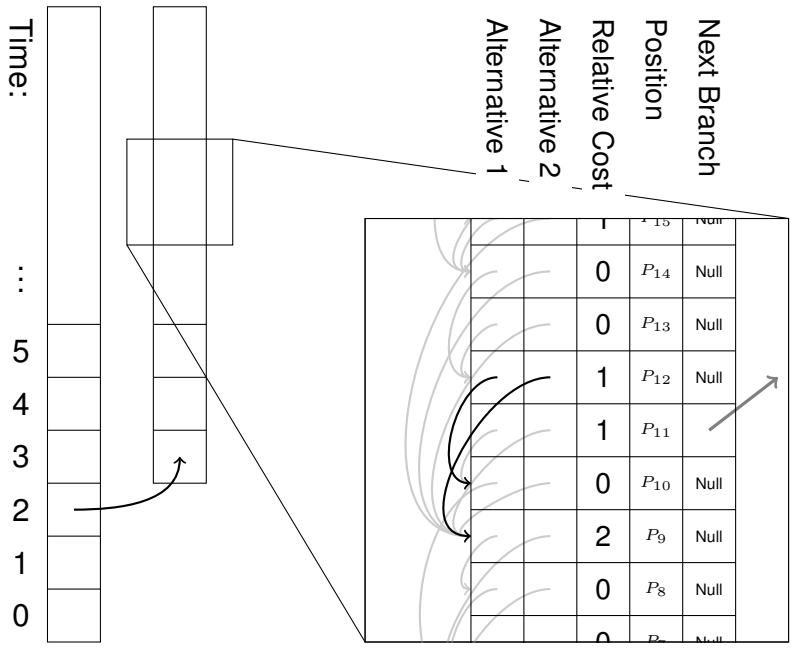


Figure 5.1: Agentverse structure and time step data.

Algorithm 3 shows the basis for the *Next* algorithm, and the pseudo code in Algorithm 4 shows how we search for an appropriate branch:

The *Next* Algorithm works by selecting a branch nearest the point of conflict which changes the cost of the current path by the desired amount. This new branch is selected by a search facilitated by a set of linked list embedded into the Agentverse.

Suppose s is the TimeUnit before the collision. The TimeUnit s contains a variable $s.relCost$ which contains the relative cost of the branch indexed by 1. When the *Next* Algorithm searches for a path of the same relative cost as $s.relCost$ then it can return the time step equivalent to s. Otherwise a time step is required at one of the other two possible relative costs. The 2 element array $s.priority$ contains the latest times of two alternative branches at the two alternative costs. An exception to this is the case where the correct branch occurs from the same time step at which a child Agentverse v branches from its parent $v.parent$. In this case the priority of the next branch after $v.ind$ is considered.

This process skips alternatives which may occur later in the indices of the a single TimeUnit, however these possibilities will eventually be uncovered if needed as proven in Section 3.

5.1.4 *Stem*

From each Agentverse there are a large number of possible branches. In general a search algorithm such as A-star or Dijkstra's search algorithm will process each node sequentially and then store each of its neighbours for later processing. When a node has a large number of neighbours the processing required to manage the neighbours could become prohibitive. Work on the Partial Expansion A* Algorithm [YMI00] and its enhanced version [GFS+14] works around this problem by splitting a node based on

the increase in cost.

To mitigate this problem we also consolidate a set of nodes with a common property into a single meta node called the Stem. We leverage these meta nodes to separate groupings of paths which can safely be deferred for calculation later in the Algorithm. Stem nodes can then be inserted into the priority queue with priority equal to the minimum of the group of nodes it represents. When the Stem is popped off of the queue it can be expanded into a subset of the nodes it represents together with another Stem of the remaining nodes. Figure 5.2 illustrates this idea.

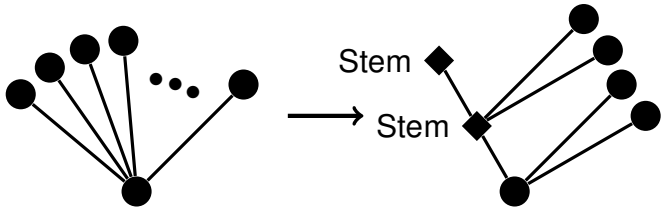


Figure 5.2: Reducing branching by using Stems.

The *Next* Algorithm can be iterated at a particular target cost and in doing so all paths at that cost and later in the ordering will be traversed. However by using this approach we skip branches of higher cost closer to the point of collision. We include these potential solutions by combining paths of a higher cost into stems. The property of target cost then becomes the property by which we define the stem and in doing so reduce the amount of Multiverses created in each loop of the Algorithm.

5.1.5 *Multiverse*

The Multiverse is the data structure we use to store collections of Agentverses. A Multiverse can either be a potential solution or a stem representing a set of branches not yet computed. We use the $v.id$ which uniquely identifies each Agentverse to compute a hash code for each Multiverse. This allows us to use a closed set (a hash set) to determine when a Multiverse has been computed before and dismiss the possibility before it has entered the priority queue. This can be extended to stems if we include the additional data associated with stems. Algorithm 2 shows the data structure used to store Multiverses and stems.

The main loop of our search is described in Algorithms 5 and 6. The main loop forms a search over Multiverses in a similar fashion to Dijkstra's Search Algorithm and the A-star Algorithm. Each Multiverse is pushed onto a priority queue with priority associated with its overall cost. Multiverses are then popped off of the queue and processed. If the Multiverse represents a stem it is expanded and the loop continues. However if the Multiverse is not a stem a search is performed to detect any collisions.

The collision search is performed in temporal order. All Agentverses are considered at each time step incrementally until all agents have reached their Goal. The search is conducted in this manner so that earlier collisions are not reintroduced.

struct VerseUnit:
    Agentverse * $verse$;
    unsigned int $stem$;
end
struct Multiverse:
    unsigned int $hash$;
    unsigned int $cost$;
    unsigned int $stemType$;
    unsigned int $stemAgent$;
    unsigned int $stemTime$;
    unsigned int $stemSide$;
    VerseUnit * $verses$;
end

Algorithm 2: Multiverse Data-structure.

5.1.6 *Backtracking*

The *Backtracking* Algorithm is an additional culling step which we use to remove permutations of paths which share a common conflicting point in spacetime. As was described in Section 4 the *Backtracking* Algorithm is applied on planar graphs and works by removing paths which remain in an enclosed region. The *Backtracking* Algorithm traces an outer boundary to the left and right of the original path searching for options which leave the region enclosed.

The implementation of the *Backtracking* Algorithm can be divided into two concurrent parts. We define the Solver which traces the outer boundary of the region we are removing from consideration. We define the Scanner which traces the original path such that every iteration of the Scanner produces a point $\mathbf{x}$ on the path which matches the distance from the Goal of the current position $\mathbf{y}$ of the Solver, i.e. $D^G(\mathbf{x}) = D^G(\mathbf{y})$.

The Scanner

We restrict the *Backtracking* Algorithm to Non-Complex paths. This implies that any point $\mathbf{x}$ returned by the Scanner will be the unique point on the path with distance $D^G(\mathbf{x})$. The Scanner always return the most recent occurrence of the location $\mathbf{x}$ on v prior to the collision. This reduces the amount of re-computation required, due to the fact that earlier costs have already been explored. Algorithm 8 illustrates how we determine the next time step the Scanner considers.

The Solver

The Solver has the task of analysing the outer boundary of possible paths reaching the point of conflict. The Solver begins by determining a *forwards* direction from the point of conflict. We are attempting to find potential options on the left or right side of the path. In order to achieve this we cycle clockwise or anticlockwise from the *forwards* direction until we cycle to a connection which leads towards the start. If at any point

while cycling the connection an edge leading to the Goal is discovered we have found a diversion around the conflict. After this option is discovered the cyclic direction is reversed and the nodes are stored along the path backwards towards the initial path. Once a path from the discovered option is calculated we use this data to construct Agentverses which lead to that option.

Non-Complex Paths & Preprocessing

When the *Backtracking* Algorithm has produced a potential path the result may not initially be a Non-Complex path. This can occur when a path v retraces its own steps. A Non-Complex path has a unique node on each slice between the Start and the Goal. When a path retraces its steps and then branches in a different direction one or more slices may have multiple intersections with the new path. This makes the path Complex breaking assumptions that were used in the proof in Section 4.

This Complexity can be removed by stripping additional nodes from the end result u creating a simpler path u' which may have a lower cost. The lower cost however can be recuperated later when the *Next* algorithm is applied but restricted to the nodes given by u'.

During the creation of an Agentverse two key values can aid future calculations. The complex branch $v.cb$ is the time step at which an Agentverse first takes a complex option. The complex branch is used to determine which algorithm to apply to a particular conflict. A conflict before a complex branch can use the *Backtracking* Algorithm to avoid the conflict. We also define a value called the extent $v.ext$ of an Agentverse. The extent determines the closest point an Agentverses parent comes to the Goal before the start of the given child Agentverse v. The extent can be used to aid in the reconstruction of Non-Complex Agentverses as discussed in the previous section by curtailing the computation when the extent matches the current Agentverse in the calculation.

Both calculations can be done at once. By stepping from child to parent we can keep track of the distance to the Goal. In this manner once the extent of the parent is larger than the extent of the starting position of the original Agentverse v we can exit the algorithm since all nodes nearer the start are already accounted for. In this same process we can also determine the initial complex branch by comparing the successive starting locations of the parents as we iterate through them.

Main Loop with Backtracking

The main search is extended to uses the Backtracking Algorithm by using a number of cases as discussed in the definition 4.2.43 the Path Selection Algorithm. Algorithm 12 shows the modification to the search algorithm to accommodate the cases used for the Backtracking Algorithm. Algorithms 13, 14, 15, 16 show the cases used to search for solutions using the Backtracking Algorithm, Complex branches from Non-Complex

paths, Non-Complex solutions and Complex solutions after a branch point. While some of these Algorithms 15, 16 are a slight modification to the makeBranch Algorithm 5, the Complex branch Algorithm 14 is a brute force search for Complex branches and Algorithm 13 is a direct application of the Backtracking Algorithm 4.2.37.

Chapter 6. Analysis/Results

6.1 Experimental Setup

To best assess the performance of CIS against CBS, our comparator of choice for reasons outlined earlier, an implementation of both was written in C++. We include implementations of CIS both with and without reasoning to show the improvement that the reasoning additions make to the Algorithm. CIS with reasoning will be labelled CISR and CIS without reasoning will remain labelled CIS from this point onwards. Some slight amendments were required to the CBS algorithm as published [SSFS12b] to avoid the cross-over of agents (a situation where two agents swap occupied squares, physically passing through one another). Neither implementation benefited from any machine-level or multi-threading optimisations.

(a) hrt201d (b) lak303d (c) oth000d (d) rmtst

Figure 6.1: A selection of maps used in testing.

In defining our experiments, we borrow from Sharon et al. [SSFS12b]. We defined three different sets of test batches. The first set is constructed as 3 sets of square grids of sizes 8x8, 16x16 and 32x32. For each grid size we range the number of agents; for the 8x8 grids we range between 4 to 16 agents, for the 16x16 grids we range between 4 to 25 agents and for the 32x32 grids we range between 4 to 35 agents. Each grid size and agent combination has 2000 different seeds tested and timed.

In the second case we consider maps drawn from Sturtevant's [Stu12] work on grid-based navigation benchmarking. Fifteen maps were selected from Sturtevant's *Dragon Age: Origins* map set, of a variety of sizes; a sample of these is shown in figure 6.1. Table 6.1 lists the maps employed in ascending order of open (passable) grid squares, along with their maximum dimensions along both axes.

It is obvious from the ordering of the table, relative to the dimensions of the maps, that the complexity of a given map was not directly related to its maximum dimensions; several larger maps have smaller numbers of navigable nodes, which is appropriate to the context of our work. A broad range of map sizes was selected in order to explore system performance over a variety of path lengths.

We varied the number of agents as follows: 4, 6, 8 and 16. Each map was tested using 2000 random seeds (per agent count) determining start and goal nodes for each agent, with both CIS and CBS.

Map	Dimensions (WxH)	Open Squares
ost102d	28x22	249
lak103d	49x49	861
rmtst	182x50	5589
lak202d	159x182	6240
ost003d	194x194	13214
lak303d	194x194	14784
oth000d	384x384	17601
den012d	310x350	22682
hrt201d	297x272	23572
den520d	256x257	28178
brc202d	530x481	43151
lak401d	401x593	43567
orz999d	632x698	43893
brc501d	225x288	57719
orz900d	1491x656	96603

Table 6.1: Details of testing maps

Finally in the third batch of tests we include selection from a set of bespoke maps which were used in initial testing of the algorithms. These maps are shown in appendix A. Each map is a square grid in nature certain squares made impassible. Agents can only travel to adjacent squares in one of the four cardinal directions. Figure A.1 shows the key for the maps. The maps in this set of tests have predefined start and end points for each of the agents. The start points are marked by circles and goals by rings so that a square used as both a start and end point can be distinguished. A total of 30 runs was done for each map and algorithm pair, however this is sufficient as the configuration of the map and agents does not change between runs.

A cap was applied to limit the amount of time permitted for each test; this reflected our interest in systems moving towards real-time computable solutions, and a similar approach was taken by Sharon et al. [SSFS12b]. The employed cap was 5 seconds for the first two tests and 100 seconds for the bespoke maps; if a solution had not been found within that time, the testing system was deemed to have failed (timed-out).

Experiments were performed using a desktop computer equipped with an Intel i7-2600K CPU at stock speeds, and 16GB of DDR3 RAM @ 1600MHz. Solution costs were compared for validity, and results were recorded in terms of compute time.

6.2 Results

We have used two methods to organise the data for our results. The first method is that which the our competitor CBS uses [SSFS12a]. We run the algorithm until a fixed time cap of 5 seconds and then compare the number of Wins vs Losses, i.e. the number of times CBS has a computation time than CIS and visa versa. Our second approach is to use The Inter Quartile Mean (IQM) to analyse the results, i.e. the mean of the mid 50% of results.

Our logic behind the use of the IQM is that the complexity of the task is not solely linked to the size of the Graph. The complexity can change based on the arrangement of agents and how they interact with each other. The IQM allows us to reduce the number of apparent outliers and compare a range of scenarios with a similar complexity.

It should be noted that in some cases shown in figures D.1-D.7 there were enough failed seeds to require the third quartile to include those failed results; those cases, indicated on the figures with a hat, should be seen as lowest possible values for the mean and third quartile as failed results are assumed to have taken 5 seconds for the purposes of calculation. Similarly in this figure, cases where there were no successful results for a given system are omitted from the graph, as they provide no useful information.

6.2.1 *Broad Analysis*

Taking the results of the three grids 8x8, 16x16 and 32x32 together we get the picture that CIS and CISR improve as the sparsity of the system increases. Each jump in grid size reduces the density of the map by a factor of $\frac{1}{4}$. Each step in density is accompanied by a reduction in the number of successes for CBS of approximately $\frac{1}{4}$ while the results for CISR increase by an equivalent portion. This supports the idea that CISR benefits from the sparser scenario. Although for the 8x8 grid we are overtaken in successes by 14 agents the results show that CISR is a competing algorithm in these cases.

Table C.1 shows the results for the 8x8 square grid. When the number of agents are low, i.e. 4-8 agents, the success rate for CISR is above the 90% range. Starting from a 99.1% success for CISR rate at 4 agents the number of wins slowly decreases with respect to CBS as the number of agents increase. Similar trends can be seen for the 16x16 grid and the 32x32 grid case, however for the larger grids the results for CBS tend to peak and then reduce again. This is due to the artificial cap of 5 seconds. As the complexity of the scenario increases the solution time approaches the 5 second cut off, when the majority of solutions are above 5 seconds the number of results start to curtail.

MAPF is a NP-Hard problem and therefore is restricted by the properties of NP-Hard problems. In particular the unpredictable nature of NP-Hard problems preclude the possibility of knowing the true complexity of a scenario beforehand. This is our reasoning behind using the IQM within our results. When we originally run a scenario we do not know the true complexity, making statistics like the average of a set of seeds an unreliable indicator of success. However we can view the resulting time as an indicator of the complexity post run. By restricting the average to the mid 50%, as is done for the IQM, we restrict the range of complexity to a subset of the overall complexity and remove some of the more extreme outliers.

Figures D.1-D.7 show Lower Quartile, IQM's and Upper Quartiles in a simplified set of box and whisker plots. The figures are displayed with a logarithmic scaling

to expose the relationship of the three algorithms. The 3 algorithms tend to trace a shallow ‘s’ shape. The shallowness of the curve show that the agent-time relationship of the algorithms is close to exponential however the results peter out at the 5 second cap.

The times of the gaming maps show similar results. The results have been ordered in order of open blocks as shown in table 6.1. The tables C.4 and C.5 show the relative success ratios for CISR and CBS. The graphs D.4-D.7 show the Inter Quartile data of the maps with 4 agents, 6 agents, 8 agents and 16 agents.

The vast majority of the maps tested have a great many more squares than the 3 test grids. The results show that CISR outstrips CBS in all cases shown up to 16 agents. The closest CBS comes to CISR is in the map names lak103d at 16 agents where CBS has 278 success and CISR 519 successes, there were however a large number of time outs for both algorithms (1252 time outs for CBS and 1422 time outs for CISR).

We also include a selection of maps which were used for testing. These are the bespoke maps and are displayed in appendix A. The time cap was extended to 100 seconds as the sample size did not need to be as large since an average time over 30 runs was taken, in the form of another IQM. However each of the 30 runs are identical in setup as the map is identical and the agent positions/goals are fixed. The averaged run times can be seen in table C.8 and a graphs of the data can be seen in graphs D.8-D.11 sorted by the run time for CISR for ease of reading.

The aim of the Permute maps, A.2-A.5, is to reverse the order of a set of agents. Each map has a line of n agents to the left side of a $n \times 3$ grid. The goals for each agent is placed in a mirror position in the x-axis. Agents have to navigate past each other to reach their respective goal however this setup does not give much opportunity for CISR to apply the backtracking algorithm as most agents need to move off of the only non-complex path available to them. However the results shows little proportional difference between CIS and CISR. Both CIS and CISR outperform CBS in these cases by a large margin. This margin grows larger as the number of agents in this scenario increases.

This shows the power of the underlying algorithm CIS and how it contributes to the gain in performance in both versions of the algorithm. CISR outperforms CIS on the Permute 4 map, perhaps due to the gain in performance given by the removal of pause migration. The other permute maps suffer a small loss of performance from the overhead of the backtracking algorithm although small.

The Outline set of maps, A.6-A.9, were made to test the behaviour of agents which have coinciding but generally non conflicting routes. A set of agents may travel as a group from one location to another with little interference. However one wrong movement may leave an agent with little room to manoeuvre.

The CISR algorithm does the best out of the three algorithms especially as the

number of agents grow. The Backtracking algorithm allows for quick resolution of congestion due to agents running one another into an obstacle. The CIS algorithm suffers on large grids due to recurrent behaviour such as pause migration and alternate routes however CISR negates these problems successfully and out performs CBS in all but the simplest outline map A.6, in which CIS outperforms CBS.

Our third group of maps contain geometry in the form of obstacles to manoeuvre around. The Crossroad maps, A.10-A.12, contain a crossroad at the centre which agents have to contend with. An agent which starts at the centre of this crossroad also has its goal in the centre meaning that it will always return to the centre. Other agents negotiate with this agent in order to reach their own goals. Another map belonging to this set is the Geometry 2 map A.13 which consists of a set of corridors with agents at the ends of them. Agents need to wait at intersections for congested intersections to clear before moving on.

Both of CIS and CISR outperform CBS on the crossroad maps. The most complex version of the crossroads maps A.10 the CISR algorithm outperforms CIS showing that the improvements to CIS in CISR apply to pause migration in this case. The other two crossroad maps however do not have enough complexity for CISR to outperform CIS. Similarly for the Geometry 2 map A.13 both CIS and CISR are outperformed by CBS. This could be interpreted as the success of the constraint based technique at quickly proving that the f-value needs to be raised to allow agents to pass one another by waiting for congestion to clear.

The last set of maps have no additional geometry to navigate around however these maps are similar in nature to the permute maps. Each agent starts at one end of the map and navigates to the other end in some permutation of he order it started as in its start locations. Multiple groups attempt this swapping action from either end simultaneously from either end. These maps come in several forms from the bypass map A.14, the swaptest maps A.15-A.20 and the pass maps A.21-A.23.

Most of the swaptest maps are confined to a small area and CBS tends to outperform CIS/CISR, the only success for CIS/CISR is when CIS outperforms CBS on swaptest4. For the bypass and pass maps the CIS algorithm outperforms CBS however CISR is slightly behind CBS and CIS. The pass maps are considerable in number of agents however the solutions need little in terms of intervention by simply nudging agents out of the paths of each other and hence are well suited for CIS. CISR seems to hamper itself with the additional overhead of computing the backtracking algorithm when a simpler approach of CIS gets to the answer quicker.

Overall the bespoke maps surprisingly show that the CIS algorithm has its own niches where it can outperform CISR and CBS depending on the complexity of the situation. CIS performs well when just about any choice will resolve a collision or progress the search, it also performs well when the solution does not increase the overall f-value by much. The other area where CIS performs well is the Permute maps

A.2-A.5 and congested geometry maps A.10-A.13 where the backtracking algorithm has little leeway to improve performance. However non of these cases are a large detriment to the CISR algorithm where only a constant improvement can be made to CISR by using the CIS algorithm. This shows that CISR has greater stability than the CIS algorithm.

This does not however undermine the effectiveness of CISR as the larger grids and gaming maps show the performance gain when graphs increase in size or sparseness. Certain bespoke maps also show this improved performance when the complexity increase such as the last Permute map A.5 and the larger Outline maps A.7-A.9.

6.2.2 *Analysis of Time Complexity*

The CBS algorithm uses A* as its low level solver. A* is known to be of order $O(n \log n)$ in computational complexity. CIS is linear in the lower level search. The getBranch and getNext Algorithms are of order $O(n)$ and $O(p)$ respectively where n is the time step of the collision and p is the number of parent Agentverses.

Many properties of an Agentverse can be calculated with a time complexity depending on p. One notable example is finding the last location at distance d from the goal along the path v (which we will henceforth refer too as $\mathcal{P}_d(v) = P_{D^G(Start)-d}(v)$). When the distance d occurs after the start of an Agentverse v we can use the expression $v.startTime + D^G(P_{v.startTime}(v)) - d$ to calculate the correct time step to query for the required position. When the distance d occurs before the start of the Agentverse we traverse up the parent link and try again until successful. This makes the function $\mathcal{P}_d(v)$ of order $O(p)$. An algorithm dependent on $\mathcal{P}_d(v)$, or other such properties, on every step of a linear process dependent on n would have at most order $O(np)$. The preprocessing Algorithm 10 has order $O(p^2)$ as the value $\mathcal{P}_d(v)$ is calculated for each parent with differing values of d.

The quantity p is bounded by the length quantity n, i.e. $p \leq n$. The quantity p is dependent on two processes; 1) every new branch from an existing Agentverse, from the execution of the *Next* algorithm, has the possibility to increment the value of p, 2) the execution of the Backtracking algorithm has the possibility of adding a value up to the number of steps taken by the Minimal Connection Sub-algorithm. According to the Minimal Avoidance Lemma 4.2.38 the Backtracking Algorithm is proven to take the smallest deviation possible, minimizing the addition to p. This means that p is related to the number of collisions encountered in the past iterations of the path v being processed or in other terms as a function of depth of the upper level of the solver.

The value of p may rise and fall depending on where branches are taken from, as all future branches are erased in favour of the new branch. Also the value of p can only be at most incremented on complex as the *Next* algorithm is the only algorithm applied to them. The value of p has a complex behaviour as the Algorithm proceeds making p an unpredictable other than its dependence on depth.

As the depth increases in a complex or congested environment a number of factors

that change the balance of the algorithm introducing a larger portion of complex paths requiring the use of the *Next* algorithm. The *Next* algorithm can be prone to a process which we call Pause Migration.

Consider a path which includes a pause in-order to avoid a location at a specific time t. When *Next* is applied to resolve a collision after t we run the risk of replacing the pause with an earlier pause which has no bearing on any events after time t. The old path and the new path may trace the same rudimentary path and may even share an identical segment after the two paths meet again before time t. This becomes solution redundant and has no value in solving the later collision while also increasing the branching factor. The Backtracking Algorithm eliminates this behaviour when it can be applied.

The behaviour of p together with the change in algorithm, from Backtracking to *Next*, as complexity increases dues to congestion means that the algorithm shifts in nature. We begin with a near linear lower level search with a low branching factor due to Backtracking. Then over the iterations we consider more complex solutions which require *Next* with a higher branching factor and the value of p also increases with the depth of the computation.

6.2.3 *Conclusion*

This shows that CISR has a duality. The results show that in sparse environments the linear low branching behaviour dominates and CISR has substantially lower times than CBS. The lower quartile also demonstrates that less complex scenarios in a congested environment share this trend, as it remains much lower than the lower bound of CBS for longer. The 16x16 and 32x32 grid results in figures D.2 and D.3 exemplify this fact. The interquartile range, i.e. the difference between the Upper Quartile and the Lower Quartile, demonstrate that when CISR excels the range of solution computation times also remain small. This shows that CISR is predictable and has a more statistically predictable range of results.

We can also see that when the results are sparse enough CIS outperforms CISR. This is due to the additional cost of the Backtracking Algorithm. However the results of CIS quickly grow above that of CISR and then CBS. Results in tables C.6 and C.7 show comparative results for CIS and CISR showing that CIS out performs CISR in the most sparse of cases. CIS excels in the game map examples frequently beating both CISR and CBS due to the large and sparse nature of the maps.

In conclusion both CIS and CISR show best performance in sparse environments. CIS can be more sensitive to the conditions of the map and fall short of CISR when congestion occurs. However CIS can outstrip CISR when the environment and conditions allow. CISR shows more stable results overall by removing the recurrent behaviour of CIS. The CBS algorithm shows improvement over CIS and CISR in the more congested environments as shown in the later square grids especially in the smaller grids such as the 8x8.

Chapter 7. Case Study

7.1 Case Study: Smart Parking

In this chapter we show the derivative work based on CIS called Smart Parking. Using a relaxed suboptimal version of CIS we produce an algorithm which attempts to plan paths for agents to find suitable parking spaces within a map.

Several aspects of the algorithm have been modified to cater for the change in context. By allowing for a capacity on individual squares and a modified definition of collision between agents a flow of traffic can be simulated at a larger scale than individual nodes representing the space of a single agent. The selection of branches is also modified to reduce the computation time of the algorithm allowing a smaller selection of alternative routes to be explored speeding up the computation of earlier cost configurations in favour of later less congested solutions.

7.2 Introduction

Drivers searching for a vacant car-parking space can account for more than 30% of traffic in a metropolitan area at any particular time [ARS+05]. Clearly a smart parking system which can efficiently guide motorists to available parking spaces could alleviate this problem. Traffic authorities in many cities have instigated parking guidance and information (PGI) systems, providing drivers with up-to-date information on the availability and location of parking spaces [TL06]. The information may be presented to drivers via dynamic street signage, or over the internet.

Many smart parking schemes exist based on resource allocation and reservation [YYRO11, WH11, GC11], whereby the PGI system knows how many spaces are currently available at each site and drivers are directed accordingly. The systems are typically based on locating the car-park or street with available spaces which is nearest to either the driver's entry point into the controlled area, or the driver's intended destination within that area. Many systems also identify the most suitable space by including a pricing factor, sometimes based on auction or electricity trading (in the case of electric or hybrid vehicles) [HKI13]. When the target parking space has been identified and reserved, a Global Positioning System (GPS) can be used to plot the driver's route to the parking destination. This can result in multiple vehicles being directed toward the same parking garage at the same time, or along routes which cross over one another, which can lead to further traffic congestion along those routes.

In this paper we introduce the concept of collaborative path-finding to the field of smart parking. We adapt a standard A-star path-finding algorithm to incorporate multiple agents plotting paths concurrently, while taking into account one another's progress along their assigned routes. In our simulation, the agents represent drivers being assigned a parking space, the destinations are the locations of the parking spaces

themselves, and the nodes of the path-finding grid are the streets and junctions of the metropolitan area. Our approach considers multiple scenarios wherein agents have taken different decisions in order to avoid over-occupying the same node on the path-finding grid. A selection technique is employed to identify the scenario which provides the most efficient solution for all agents at any particular time.

We show that employing this "smart routing"scheme within a PGI system can be beneficial in a number of ways. Congestion is reduced, as drivers are sent along routes which do not interfere with one another. A dynamic approach ensures that, as new drivers enter the controlled area, they are not only assigned an available space, but are assigned a route which causes minimal further congestion. Journey times for drivers are therefore reduced. The approach also leads to greater efficiency for the parking garages themselves, as spaces are vacant for smaller amounts of time, so revenue is earned over a greater proportion of the day.

7.3 Background and Related Work

In this section we present a brief background to the fields of smart parking in order to arrive at our contribution of smart routing.

7.3.1 *Smart Parking*

Parking guidance and information systems play an increasingly vital role in most major metropolitan areas worldwide [YYRO11]. Car parking is a revenue generator, rather than a cost centre, in most cities. Utilising a smart parking system can have a positive effect on that revenue due to improved occupancy rates, market-sensitive pricing and more efficient revenue collection. Further to this, the benefits to both commerce and the environment of reducing traffic congestion make smart parking an attractive proposition.

Earlier implementations of PGI schemes involved informing drivers on the availability of spaces and guiding them toward parking garages or streets identified as having free spaces [TL06]. This could often result in many drivers being directed toward the same place while car-parks with only a few spaces were being ignored, even if they presented a better solution. More recently, schemes have introduced the concept of reservations, whereby a a driver is allocated a specific space which is then marked as unavailable until the specific driver arrives [WH11]. The reserved space may be password protected until the assigned driver arrives (passwords are communicated through SMS) [HBD10]. The reservation approach has been augmented by the introduction of, for example, auctioning, price factoring, and trading of electricity, in the case of hybrid and electrical car schemes [HKI13].

Technology for detecting whether a parking space is occupied (including inductive loops, weight sensors, pneumatic road tubes, etc.) is beyond the scope of this paper; for our purposes it is assumed that information on the occupancy and location of spaces is available and correct.

Little or no work has been carried out on planning how drivers reach their allocated parking space. Existing systems typically rely on GPS navigation for individual drivers, or dynamic roadside signage for directing many vehicles along a shared route [GC11]. These approaches take no account of the congestion, and therefore time delays, introduced by sending multiple vehicles along shared routes toward reserved smart parking spaces. Further congestion can be introduced at streets or traffic junctions where directed routes intersect. In this paper we address for the first time the issue of planning route information for multiple vehicles approaching allocated parking spaces, by the application of a novel collaborative path-finding approach.

7.4 Protocol

In this section, we present our smart routing protocol for multi-agent path finding. Our algorithm is a variant on multi-agent path finding using A-star. Utilising a square grid we represent a portion of a city, each square containing data concerning the capacity of the corresponding area of the road network. Using the idea of reversing the direction of A-star we determine the exact distance to a goal. Decisions can be precomputed for every configuration of neighbors and their relative distances to the goal. We compile these decisions into a 'specification' which allows us to reference a lookup table during the processing stage of the algorithm.

We apply an ordering to each cardinal direction, enabling us to remove arbitrary decisions from the system. This implies an ordering on paths from their respective starts to their goals. Using specific paths relating to this ordering we can construct any possible route on the grid. We call routes constructed in this manner vehicle routes or *agentverses*. We select groups of vehicle routes together to represent all the vehicles in the system. We call these groups collective routes or *multiverses*.

The goal of our algorithm is to select a collective route which minimizes the total congestion within the system. We achieve this by applying A-star to the collective routes. We run a set number of iterations of A-star over the collective routes, storing results with low congestion while the algorithm maintains lower total path length. When congestion is detected within a collective route, the algorithm redirects random portions of the traffic to create a new collective route which is reintroduced back into the algorithm.

In our experiments the path finding algorithm is applied once every time step of the system. This simulates an evolving system, allowing us to introduce new vehicles into the system to test the adaptability of the protocol. Key elements of the algorithm are discussed below.

7.4.1 *Path finding*

Our approach to optimised car-park routing requires the definition of several algorithmic terms that are used throughout this work. In this section we outline the concepts underpinning the algorithm.

Grid Square: In this context, grid squares of the map (or graph) can be considered to represent likely points of intersection between flows of traffic rather than a truly representative route-map of a city. The grid square structure contains several important variables in our algorithm. Specifically, the structure permits us to determine the position of a grid square, its implicit neighbours, and a capacity value for each direction (including 'pausing', which is an increment through time rather than space). These capacities represent sustainable throughflow of traffic.

Node: One approach to solving multiagent path planning is to represent the state of every agent at every timestep as a node. In our simulation, the term node represents a collection of potential, complete routes - one for each vehicle within the system.

Reverse A-star: Reverse A-star is a useful technique to calculate a perfect heuristic [Sil06]. The principles of generic A-star [HNR68b] are applied to every square on the graph, from the agent's goal. The heuristic cost is therefore not a 'best guess', but an accurate cost from a given square *n* to the goal assuming no changes to the environment occur during the journey. Given that an A-star system is always aware of the absolute cost taken to reach the square under consideration, Reverse A-star provides an exact and absolute cost for any given route. For our purposes, this should be considered the exact cost (in terms of time to travel the route) between a given carpark and any point at which a given car might be located in the map, assuming the car does not need to avoid traffic choke points.

Node Possibilities and Costs: Assuming a normalised graph, the application of Reverse A-star facilitates a valuable algorithmic optimisation. Specifically, it permits us to reason that if we consider a given square *n*, that the neighbouring squares to *n* can only have one of three heuristic pathing costs associated with them: h, $h+1$ or $h-1$, where h is the cost of the considered square *n*.

In the special case of a square grid the possibilities are reduced to $h+1$ and $h-1$. In our applied protocol, when combined with the perfect heuristic outlined above, this permits us to employ our path-planning in a step-by-step fashion. In terms of engineering optimisation, this enables us to assign a specification to each square and reference a look-up table for swift decision-making when rerouting.

Path Ordering: A consequence of the normalised graph in conjunction with Reverse A-star is that the system will often be required to decide between squares of equal heuristic cost (if re-pathing, $h+1$). In order to assist that decision-making, we introduce a new property to the pathing algorithm which favours (prioritises) one direction over another of equal cost.

For our purposes when plotting routes for multiple agents through time, pausing is considered a direction in its own right, and indicates not only consequences of traffic flow but, additionally, opportunities which careful traffic flow management can encourage. The inclusion of a direction priority implicitly defines an ordering on the paths from a square to a goal; from each square on the graph we can select a preferred, unique

path to the goal, which we call the α path. A path with a higher f value would be higher on the list, but within the same f value set ordering is decided lexically with respect to the ordering of directions. We call this process 'preferential ordering'.

7.4.2 *Vehicle Route or Agentverse*

We employ the term vehicle route, or agentverse, as a means of differentiating between the path proposed by a single vehicle to reach its destination, and the overall collective route or multiverse. A vehicle route is the path proposed for a given vehicle to reach its destination carpark, with no consideration of the other cars moving through the map. A key assumption in our approach is that vehicle routes will be largely similar to their most optimal paths, meaning they rank lower (better) in terms of preferential ordering. The collective route, functionally, is a collection of suitable vehicle routes which reduce congestion.

We represent vehicle routes in our algorithm as segments of optimal paths. As such, we can use segments of α paths to represent all paths. Vehicle routes are represented using a start time, an α path, and a past. The past is a reference to another vehicle route from which the currently considered route branched. In this fashion, the solution's memory footprint and computational complexity is lowered. The beginning of a particular vehicle route segment connects to a point along another route segment for the same vehicle.

These connections are the only way a vehicle route can transition in a manner out of the preferred order. They are triggered generally in a case where an α path is found to be potentially non-viable due to square occupancy (too many vehicles being advised to pass through a given intersection, potentially leading to unmanageable congestion).

Let us consider the following example, at time step t. The directions South and West are toward the goal. South is favoured by preferential ordering, but we can construct a vehicle route with start time $t + 1$ starting directly to the West of the current square. This vehicle route, with the original as its past, would have the same f value as the current vehicle route but would represent a path which took West as its next direction at time t.

The same can be done with the North and East directions but this will yield a vehicle route with an increased cost of $f + 2$. A pause can also be represented. This is done by repeating the current square as the start of the α path, with the same start time of $t + 1$ and past, as the other examples.

The only vehicle route segments without a past are the initial vehicle routes computed at the beginning of the planning algorithm. This initial vehicle route stores the complete α path from the vehicle's start square to the goal, with a start time of 0. All vehicle route segments will ultimately form a chain that leads back to this initial vehicle route, forming a traceable tree from which the final vehicle route is assembled.

One potential issue with regards process efficiency and memory management is the recreation of vehicle route segments which already exist. We circumvent this issue

by storing forward pointers alongside the α path step data. Each step along the path can store a pointer to a single vehicle route. If the time of that step is t then the vehicle route would have start time $t + 1$, meaning that that step would be the corresponding vehicle route's immediate past. There can be up to at most 4 vehicle routes that would share that timestep as its immediate past, (other directions and pause at time step t + 1). The other vehicle routes are stored in a linked list; each vehicle route stores a *next* pointer to facilitate the linked list.

7.4.3 *Path Progression "Next"*

As highlighted in the previous section, multi-agent path-planning is a PSPACE-hard problem, with computation time exponentially connected to the number of agents being considered. In order to mitigate this issue, some form of pruning of the state space is required, based upon domain-specific assumptions. Our main assumption is that vehicles will pursue routes which mostly progress towards the goal. Hence we use combinations of fully formed vehicle routes as our nodes, which we term a collective route, which shall be outlined later in this section. The branches of these collective routes are made by replacing individual vehicle routes with like routes ranked higher (worse) in terms of preferential ordering. The algorithm we use to do this we call 'next'.

When a vehicle route is constructed the decision of which step to take is made immediately, comparing this to the traditional A-star approach (in which nodes represent a single timestep of the system), this results in branching decisions being made out of order. In situations where the traditional A-star approach would have selected from equivalent nodes on the priority queue to resolve a conflict, our algorithm has made the default decision of moving forwards, creating congestion which we need to resolve.

We can resolve these conflicts by adding in some of these decisions afterwards. In order to resolve these conflicts the full A-star algorithm would have picked a node representing an earlier timestep which could potentially avoid the conflict. Representing this in our algorithm we need to select a branch from a current conflicting vehicle route or one of its past vehicle routes. This branch must happen before the timestep of the conflict. We also prefer to branch later in time, as this avoids adding conflicts at timesteps we have already solved. The resulting vehicle route is of the same f value or higher, or equivalently higher on the preferential ordering list.

Given these facts our *next* algorithm needs to select the highest timestep before the conflict with a branch at a preferred f value. The algorithm can be split into two cases, 1) the current timestep we branch from is part of an α path of the vehicle route, 2) the timestep is a branch of the current vehicle route chain (i.e. the current timestep is t and the vehicle routes have start time $t + 1$).

In case 1) there is no need to consider directions past the second in our preferential ordering of directions (this includes Pause if no other direction moves forwards). If there is a need to consider these cases, they will eventually be reached as a subset of case 2), a branch of an vehicle route. This means the 'second priority' of a timestep becomes

important in our algorithm. The second priority can be looked up using the specification of the current square, calculated from our augmented reverse A-star algorithm.

In order to speed up the process of finding the needed timestep each vehicle route stores an array of three indices (pq), and each timestep stores a single index (pq_next). The pq array stores the maximum time that a timestep has a second priority branch with a corresponding f value offset. This means that $pq(0)$ would store the maximum time step at which a branch can be made that would not effect the f value. The index of $pq(1)$, and $pq(2)$ correspond to vehicle routes with f value + 1, and + 2 respectively. The pq_next index creates a linked list of timesteps with the same second priority.

Using this extra data we can skip ahead to a known priority offset, then we can follow the linked list until we are below the conflict time. This can be interleaved with another technique; by starting at the timestep before the conflict and incrementally testing for the correct second priority value. Together one or the other process will eventually terminate, either with a result, or by proving there is no branch at that priority.

7.4.4 *Branch Compression "Stem"*

To reduce the number of branches we compress a set of nodes (each being a collective route) with a common property into a single meta node called the stem. This node can be inserted into the priority queue with priority equal to the minimum of the group of nodes it represents. When the stem is popped off the queue it can be expanded into a subset of the nodes it represents together with another stem of the remaining nodes.

This technique has the advantage of postponing the processing of a large number of nodes until they are needed. Stems work most efficiently if the stem can be constructed to have a higher f value than the remaining nodes, which will be processed before the stem.

In the case of the *next* incrementing algorithm, the stem is used to correctly split the higher f value branches from the lower ones. Given a vehicle route, repeatedly calling *next* with its result will give vehicle routes with successively lower branches, as these branches are always available to the algorithm. In this manner all branches at a particular f value can be represented by one node. However if a higher f value is needed those solutions may be cut off before they can be considered as possibilities. If a clash happens at timestep t and the first branch available is at timestep $s < t - 1$, then if the solution lies at a higher f value, branches made between s and t will not be considered (since the new branch replaces this time period).

Using a *stem* we can use *next* with the same vehicle route but using successively higher f values. This solves the issue of missing higher f value solutions.

7.4.5 *Collective Route or Multiverse*

The collective route, or multiverse, is a collection of vehicle routes, and represents a potential solution. These data structures make up the nodes of the main A-star algorithm. Each node has a cost associated with the sum length of the individual

vehicle routes, and each vehicle route within the collective route has a corresponding integer value representing its current stem (i.e. the current f value *next* should look for). When a collective route is popped off the priority queue it is checked for congestion (starting from time 0 until all agents have reached their goal). Points of contention are placed onto a priority queue, with priority proportional to the amount of congestion.

After a congestion is detected and the point of highest congestion is popped off the priority queue, a number of random combinations of vehicle routes are incremented via *next*. These combinations form individual collective routes, and a corresponding collective route has its vehicle route's stems incremented. We maintain these stem nodes to preserve the structure of branches within our algorithm. As discussed earlier, if we did not maintain the stem, certain possibilities would be unreachable.

For each vehicle in a collective route we store the corresponding goal node. In this manner we can mix collective routes which use different goal nodes allowing a vehicle to choose between viable goals. However we restrict the number of possibilities, since if all combinations of vehicles and goals were considered the system would become unmanageable (this could be rectified with another stem system).

7.4.6 *Traffic Planner*

We define a data structure called the traffic planner. Its job is to simulate a use case of the algorithm. Every timestep a full path plan is completed for all vehicles currently in the system. Depending on the result we move each vehicle one step along the planned path. During each round of the full path planning we cap the maximum number of iterations, i.e. maximum number of collective routes considered. This ensures the algorithm takes a reasonable time to complete.

As vehicles get added to the system we randomly pick one goal in the simulation and free one space. This ensures the simulation can never reach deadlock, where a vehicle does not have a goal to reach. This does not change the semantics of the problem since if there wasn't a goal for a vehicle to use it would not have been entered into the system. As vehicles enter the system we do a full path plan with all goals available. The resulting goal from this solution becomes that vehicle's permanent goal.

A priority queue is maintained using a fitness criterion to decide priority. All the collective routes we iterate get pushed onto this results priority queue, such that if we reach the maximum number of iterations we do not select a collective route which introduces more congestion.

7.5 Simulation

We model a portion of a city with a 9x9 grid map, each grid square representing a collection of intersecting roads. Each square corresponds to the same journey time segment (i.e. each square takes the same time to traverse), as opposed to a geometric correspondence. In practice these may correspond to geometric locations, but for our purposes a normalized cost applies.

Each cardinal direction of each square was assigned a random capacity between 1 and 3. These capacities simulate the different road throughputs. A capacity of 1 might represent a slow road, a multi-lane road subject to roadworks, or a road with arbitrarily low traffic throughput; similarly, a capacity of 3 might correspond to a high speed limit or a multi-lane road operating at full capacity. All non-goal pauses were left at capacity 1. This defines the maximum capacity of our simulated segment of city to be roughly 730 (the number of capacity slots a vehicle can occupy across the entire map). We assigned 9 goals to the map, representing car parks, and randomly distributed 125 available parking spaces among them.

We designed two scenarios for simulation; each scenario was run several times, generating two sets of results. Each set varied the number of agents in increments of 20 vehicles, starting from 20 and continuing up to 500 vehicles. Our proportional occupancy of the road network across its length equated to between approximately 3% and over 68%, giving us a broad spectrum of occupancy proportions for analysis.

The first scenario deals with a constant flow of traffic. Half the vehicles are initialised at the beginning of the simulation; these vehicles are positioned on the map using a pseudo-random algorithm which avoids goal squares (parking spaces). The remainder are introduced over the next 9 time steps. We call this scenario the 'standard traffic scenario'. This represents a situation where a large initial spread of agents are navigating towards parking spaces, and a lower, but still proportionally significant, number of vehicles enters our portion of the city over subsequent time-steps, during which all are cooperatively and reactively navigating. This ensures an approximate constant flow since the maximum distance a vehicle will travel is 18 on a 9x9 grid, on average a vehicle will travel less than half this distance. Extending the start time any higher than 9 means that we are no longer testing this vehicle in conjunction with the majority of the original vehicles (equivalent to the same results with less vehicles but higher throughput).

The second scenario starts all vehicles at time step 0. This scenario simulates initially high throughput, which tapers off as vehicles find their destination; this will give a higher peak in activity but will not introduce unexpected factors such as vehicles arriving to our portion of the city later in the simulation. The scenario can be considered representative of a traffic stress point, where the road network begins with proportionally high occupancy, relative to the number of vehicles on it.

Whenever vehicles are positioned in the system, they are done so using this pseudo-random method. An additional constraint, aside from the limitation regarding goal squares, is that the maximum number of vehicles which can begin in a square is equal to the sum of all its capacities (four cardinal directions, and a single pause). Agents are always added to the simulation in the same ordering. As agents are added a pseudo random goal is picked and its capacity is incremented. The sequence of goals which are picked is always the same ensuring comparisons between methods are meaningful.

Each scenario places strain on different aspects of our algorithm. The first shows how our protocol handles unexpected input into the system. Vehicles can be added to the area leading to congestion where there would not have been before. Our system tends to spread vehicles out among the grid squares, which may introduce congestion when we add new vehicles, but the system should be better adapted to handle this change.

The traffic stress point scenario has the advantage that vehicles are not introduced after the first time step, meaning that the original simulation results can be built upon. Successive time steps will pass the point in time at which a particular congested square was a problem and redistribute iterations onto the later time steps. However the beginning of the simulation will start with a high number of vehicles in arbitrary clumps, which will be spread out through the network in later time steps; in this fashion, occupancy is disproportionately higher than in the former scenario. The fitness of later time steps should almost never exceed that of earlier time steps because there are fewer points of contention to consider (locations at points in time), however since branching is randomized this is not guaranteed to be the case.

We use two existing (non-collaborative) traffic routing protocols as our comparisons. The first simulates a road network without smart parking and therefore no parking reservation. Each vehicle uses an in-car route planner to find their route to the nearest car park (regardless of whether that car park has spaces available). In our simulation once these vehicles have reached their destination they are removed whether or not spaces are available. This is a best case for this simulation: vehicles which reach their goal are allotted spaces unknown to the original system. These vehicles would usually re-enter the system in search of another car park with an available space.

Our second test run simulates a smart parking system where parking spaces can be booked beforehand. This simulation is run as a single iteration of our algorithm, in which case each agent selects the nearest goal and deducts a space from it. This guarantees each vehicle a space at the end of its route. This test is representative of current smart parking solutions, which to date have not taken account of collaborative path finding. This smart parking approach would reduce congestion of traffic searching for a space among several car parks, which is not factored into the previous comparison.

The third test run represents our own system, combining smart-parking with centralised smart-routing. We selected between optimal computation time and result quality to decide the maximum number of iterations and chose a branching factor slightly higher than 1 to determine the number of vehicles redirected per congested square. We compare the overall congestion as our measure of success. However we also compare total path length to ensure we haven't mitigated congestion at the cost of path length. Our first test run will always have the minimal total path length. A good result would indicate that the additional cost of path length would be comparable to that of

smart parking with no smart routing.

7.6 Results and Evaluation

The experimental results from our scenarios are presented as data sets comparing the congestion of the system with the summation of the total vehicle path lengths. Before informed conclusions can be drawn regarding these factors, they should be explicitly defined.

Total Path Length: Total path length represents the summation of each vehicle's route time from their entry into the simulation until they reach their respective goal. We use total path length as an indicator of the extra distance vehicles will have to travel as a consequence of avoiding congestion. We require the total path length to be comparable to that of our second comparison, smart parking.

Extension: We define extension of capacity as the difference between the activity of a capacity slot and the maximum capacity of that slot. The extension represents how much a particular slot is over capacity. In the case of a slot which is under capacity this value is 0.

Congestion: Total congestion is the summation of extension over the entire map during the entire simulation. This gives us a measure of fitness for our algorithm: the lower the resulting congestion, the better the algorithm has performed. This must be offset against any cost increase in total path length.

Point of Maximum Congestion: The point of maximum congestion is equivalent to the maximum extension, assessed throughout the entirety of the simulation, highlighting the location most vulnerable to congestion. The higher this value, the more likely a given route plan is to exacerbate congestion on a wider scale throughout the road network. This could be considered analogous to choke points causing traffic jams, or grid lock, in particularly busy areas of the road network.

7.6.1 *Standard Traffic Scenario*

The standard traffic scenario models a situation where a large amount of traffic begins within the system, and the traffic linearly increases over the course of the simulation.

Figures 7.1.1 and 7.1.2 illustrate the results obtained from the three test runs of this scenario. It should be noted that the number of vehicles along the x-axis indicates the total number of vehicles present within that iteration of the scenario, and does not represent an incremental increase on vehicle count for a single test. To illustrate, a vehicle count of 20 indicates an test where 10 vehicles were present in the simulation at time step 0, and 10 more were added over the course of the next 9 time steps; a vehicle count of 500 indicates a test where 250 vehicles were present in the simulation at time step 0, and 250 more were added over the course of the next 9 time steps.

In Figure 7.1.1 we note that the total pathing cost of our algorithm is marginally higher than the total pathing cost of both the non-collaborative smart parking and generic car routing solutions for vehicle counts above 100. We recall that the generic

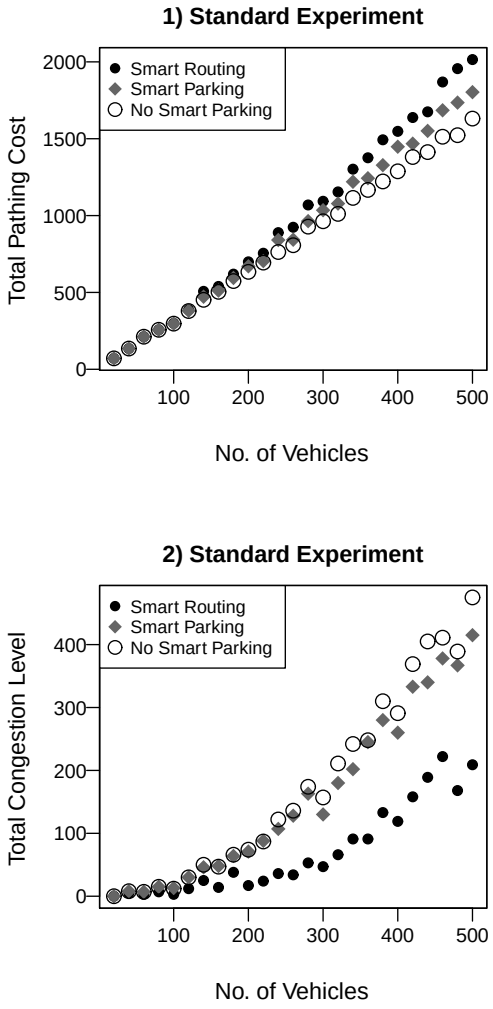


Figure 7.1: Comparison of journey times and congestion level for standard traffic scenario (half of traffic introduced at start of simulation, half introduced linearly as simulation progresses).

car routing solution shall always provide the lowest possible total path cost, as each vehicle plots a route to its destination without consideration to other vehicles, and our percentage comparisons are based upon this.

Taken on average across all vehicle counts, when compared to the car routing approach, the non-collaborative smart parking simulation is 5.55% more expensive. In comparison, our smart routing algorithm is 12.49% more expensive. This means that the minimum possible time for a vehicle to reach its goal is some 12.5% higher under our system than under a system with no smart parking or smart routing, *but* it assumes that congestion plays no role in the time taken to reach a destination.

We know that busy road networks are particularly vulnerable to congestion; the corollary to this is that the more congested a road, the higher the *actual* time taken to reach a destination, whatever the ideal shortest time might be. The reduction of congestion is a key goal in our ongoing research, and Figure 7.1.2 illustrates the performance of our smart routing algorithm in that context.

Statistically, we compare congestion against smart parking, rather than generic car routing. This is a more meaningful and challenging comparison for our algorithm, as smart parking has uniformly lower congestion values than generic car routing. Taking the average across all vehicle counts, the congestion level obtained through the application of smart routing is 42.00% that of the congestion observed when simulating smart parking without collaborative path finding.

At best, observed with 200 vehicles (road network occupancy of approximately 25%), the congestion value obtained through smart routing is 23.08% of that observed through smart parking alone. Considering our points of maximum congestion, smart routing reduces the congestion of these 'gridlock' areas by 25%, on average, and occasionally by as much as 58% (with 420 vehicles in the network).

Taking Figures 7.1.1 and 7.1.2 together, we can also compare the trends in relative performance. As vehicle count increases, there is a very visible performance benefit in terms of congestion level, while a far shallower performance hit in terms of total pathing cost. This invites significant financial benefits to the city as a whole: the road network is able to ferry more vehicles, consistently; the reduction in congestion means that time spent idling in heavy traffic is reduced, beneficial to both the local environment and consumer; commercial districts within a city can encourage a greater throughflow of high street consumers.

7.6.2 *Traffic Stress Point Scenario*

The traffic stress point scenario is designed to present a worst-case environment for our smart routing algorithm, where all vehicles accessing our portion of the road network arrive simultaneously and require collaborate routing en masse. Figures 7.2.1 and 7.2.2 illustrate this scenario's experimental results.

Comparing average total pathing cost increase once again with the best-case, minimum-cost paths provided by generic car routing, non-collaborative smart parking

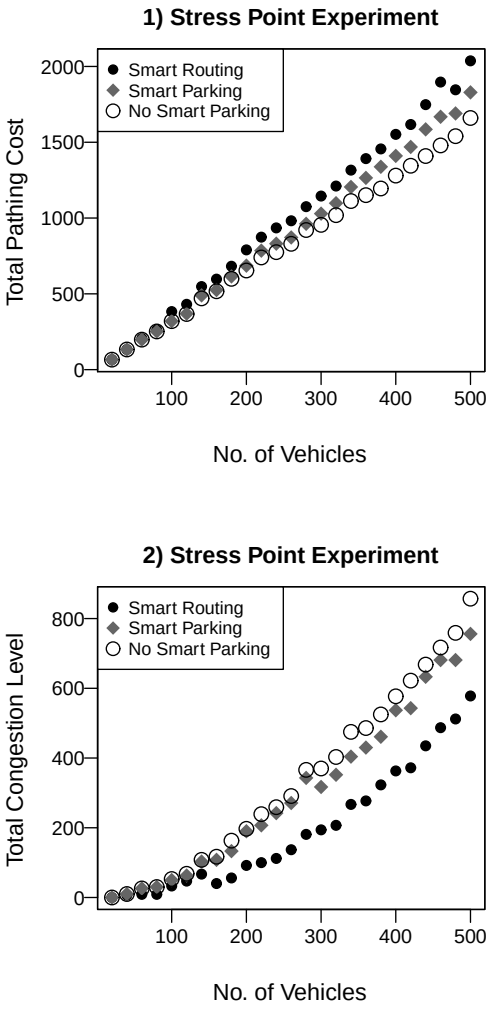


Figure 7.2: Comparison of journey times and congestion level for traffic stress point scenario (high volume of traffic all introduced at start of simulation).

increases total path cost by 5.88%, while smart routing increases total path cost by 16.85%.

In the case of congestion, however, which we again compare to the smart parking case in order to illustrate how collaborative route planning and smart parking can be employed in tandem to greater effect, there are significant performance gains. Taken across all vehicle counts, smart routing reduces congestion caused by traffic using the system to 58.92%, relative to smart parking alone. At the best case, which occurs with road network occupancy of approximately 10%, congestion is reduced by 70%.

The points of maximum congestion across all vehicle counts are lowered by an average of 13.32%, with the best case occurring at network occupancy of approximately 28%, where the worst 'gridlock' point's congestion was reduced by over 46%.

While these figures are not as impressive as those observed in the standard traffic scenario, this is to be expected as the stress point reflects a worst-case situation for our algorithm. The standard traffic scenario, where traffic is added and removed from the system over the course of time, is a better representation of true traffic flow.

Even in this worst case, congestion caused by traffic utilising smart routing is reduced by over 40% relative to smart parking alone. This, combined with the comparatively lower increase in total pathing cost, invites statistically significant financial benefits, both to the commercial districts of cities which might employ smart routing, and to enterprises whose performance hinges upon flowing road networks.

7.7 Conclusions and Future Work

In this work we have introduced the concept of smart routing to the increasingly vital field of smart car parking. We have presented a novel algorithm addressing multi-agent path planning, and applied it to the problem of congestion within major cities. The algorithm has been described in detail, with domain-specific terminology employed where it eases understanding of the underpinning mathematics. Two scenarios have been presented, and the algorithm has been applied to them. Results of those experiments have been provided and discussed in some depth, and shown to be very encouraging.

The results outlined in Section 7.6 make a strong case for the adoption of collaborative route planning, in conjunction with existing smart parking technologies. The computational complexity of the operation is reduced through the algorithmic approach outlined; the relatively small increases in total route length are not indicative of an overall increase in journey time, as reduced congestion on the road network would benefit traffic flow throughout. The reduced congestion caused by drivers searching for spaces has clear implications for city governance, both in terms of increased revenue from parking charges and increased commercial and environmental benefits from better traffic flow in metropolitan areas.

With the advent of GPS systems which communicate through mobile telecommunications networks as a means of relaying real-time traffic data, and the inclusion on many internet-capable smartphones of GPS-based navigation software, much of the

required infrastructure to pursue this technology is already in place. Such systems already have the capability to provide post-hoc assessments of traffic choke points. Smart routing, if employed in conjunction with existing traffic-flow modelling techniques, can provide a deeper insight into road network intersections which are exceptionally vulnerable to congestion, while its commercial implementation would help offset that very vulnerability.

This research aims to encourage the commercial exploration of this potentially beneficial area of information technology. Future work in this area shall explore the application of the prototype engineering solution to large road network segments, drawn from cities noted for their traffic flow issues.

Chapter 8. Critical Review

In this thesis we have constructed an algorithm which solves the Optimal MAPF problem. There are advantages and disadvantages to using this approach over other more general approaches in the area. Also contrasting our specific approach (CIS/CISR) to the state of the art (CBS) we discuss the relative advantages and disadvantages of our techniques.

8.1 Optimal MAPF vs General Multi Agent Approaches

Optimal solvers fully solve the problem that is given to them. Their end goal is a solution which is the most optimal solution of all solutions. This is opposed to optimizing solvers which either start with an unoptimized solution which they continually improve or construct solutions which improve over iteration of the algorithm. Often an optimal solver will have no intermediary stopping point where it can return a suboptimal solution which may meet the requirements of the task.

Optimal solvers also tend to take considerably more time to reach their solution than the suboptimal equivalent. If the only requirement of the task which needs solving is a solution which does not need full optimization then suboptimal solvers would be sufficient. MAPF however is an NP-Hard problem when only considering the completeness of the solution. This can be seen by the relation to the 15-puzzle. This means that even suboptimal solvers can take an inordinate amount of time depending on the complexity of the given configuration of agents on the map.

In contrast to suboptimal and optimizing solvers optimal solvers can guarantee that the solution they return is the most optimal solution. Whereas suboptimal solvers and optimizing solvers may not know the true cost of the most optimal solution. Therefore not knowing when to stop processing the problem. In this manner optimal solutions can be used in critical systems where the cost being optimized is of greater importance than the computational time needed to compute the solution.

Behavioural irregularities can also be avoided when optimal solutions are used. Suboptimal solutions without restrictions on the agents involved can have undesirable effects on the paths computed. Algorithms such as Silvers HCA use reservation tables and partial solving of individual agents to mitigate the time spent computing a solution. However behavioural irregularities can occur due to the partial solving of the problem leading to undesirable results such as oscillating states which make no progress towards the overall solution.

8.2 CIS/CISR vs State of the Art (CBS)

The CIS and CISR algorithms have a similar approach to solving the MAPF problem. Using a two layered approach which branches on collisions between agents the work

done by the algorithms is mitigated to a level related to the number of collisions rather than the branching factor of choices. This restriction of search space however varies between the two algorithms. The search through solution space can introduce more or less collisions depending on how previous collisions had been removed.

The base of our algorithm CIS removes collisions by searching for the first available alternative path before the point of collision. This has the advantage of being a linear operation as opposed to the $O(n\log(n))$ A* operation that CBS uses. However this can reintroduce the collision at a later point if the branch that has been chosen repeats the collision. This can happen in a number of ways, one such process is called *pause migration* where the point at which the algorithm pauses the agent before a collision is shifted back in time but retaining the same path towards collision and hence re-colliding with the other agent.

This problem and other similar problems are solved with the extension to our algorithm called CISR. By analysing simpler paths called Non-Complex paths we remove the recurrent behaviour from the common cases. This approach however can only be used in a planar graph as the algorithm takes advantage of the boundary that is formed by a closed loop of nodes.

The linear nature of CIS and CISR allow for the algorithms to out perform CBS when the problem is sparse in nature. Large maps and few collisions proportionally lead to the linear part of CIS/CISR outperforming the $O(n\log(n))$ lower layer A* search. This can be seen from our results as grids grow larger results are skewed in our favour.

Our approach to proving the validity of our algorithm also has its own advantages. We construct a firm mathematical underpinning deriving base properties of planar graphs and the paths on it. This constructive method of proving our algorithms' validity allows for a greater understanding of the problem and a forms a basis for further extension to our algorithms.

Chapter 9. Conclusion

9.1 Introduction

Computation of optimal and complete solutions to MAPF scenarios is a NP-Hard problem. As such any effort to further the capabilities of the state of the art must focus on a specific domain, and thoroughly explore that domain. This thesis brings forwards a new approach to obtaining such solutions, and improves both the predictability of compute time and its absolute value, in the domain towards which our algorithm is targeted. Our approaches are rooted in the principals of graph theory, and improve our understanding of the nature of the problem.

In this thesis we have constructed two algorithms CIS and CISR. An ordering was constructed and leveraged in order to prove and implement the basis of our technique CIS. Using CIS as a basis and using the Planar condition we were able to reason about the structure of a set of paths contained in a region enclosed by two paths. This region was used to construct a culling algorithm which we called Backtracking. The CIS algorithm was extended into a new algorithm called CISR which included the use of Backtracking. The results of our testing show that CIS and CISR significantly outperform CBS in sparse environments.

9.2 Contributions of the Thesis

The contributions of this thesis can be summarized as:

- A graph theory re-visitation of the underlying theories of optimal MAPF.
  - We have gone back to the first principals of MAPF and focused specifically on problems where collisions are most likely to be an issue, e.g. normalized node graphs. We first approach this from the area of ordering by applying an ordering to the paths of an agent. We then use the planar condition to reason about the motion of Complex and Non-Complex paths. This approach affords us the possibility of a *linear* lower layer to our algorithm and then allows us to cull regions of paths which coincide at the same collision point.
- An algorithm (CIS) based on the ordering of paths which uses a linear lower layer.
  - Inspired by the Conflict Based approach CIS implements a two layer algorithm. The lower layer is a greedy linear search which allows for fast computation of alternative routes. This allows for significant improvement over state of the art, CBS, in sparse environments.
- An algorithm (CISR) which allows the culling of a region paths which coincide at a collision point, through a tracing operation which is itself *linear*.

  - CIS is extended to CISR by including a culling algorithm called Backtracking. The Backtracking algorithm contains all paths which reach a shared point of collision and searches for the first alternative to the collision. The inclusion of the Backtracking algorithm in CISR allows the algorithm to out perform CBS in all cases originally published in their paper.

- New algorithms which provide answers faster than the existing state of the art.
  - Our benchmarks demonstrate the performance improvements over our primary comparator CBS. We show that there are significant improvements in performance when CIS and CISR are applied in sparse environments.

- A more statistically predictable compute time.
  - The predictability of compute time is shown to be a tighter bound as the Inter Quartile Range of compute times is smaller in a significant majority of tested cases.

9.3 Future Work

The CIS algorithm allows for various avenues for development. This thesis has outlined the base algorithm CIS and the extension CISR based on the Backtracking Algorithm. Here we will identify several improvements and extensions for later work:

9.3.1 *Generalized Cost Functions*

Firstly we plan on modifying the Agentverse Data-Structure and Branch search algorithm in order to accommodate more complex cost functions for paths. The current cost function is calculated as the number of time steps and agent spends away from its goal node. The cost function is simple to calculate and also simple to implement within our framework. Small changes would be required to implement similar cost function into the implementation. However cost functions such as the Fuel cost function may require more work. The Fuel cost function eliminates the cost of pause moves allowing the agents to stay on any given square along its solution path for any given time. This will cause the priority driven technique of CIS to never breach the initial cost of the path. Further work can be done to reason about cost functions with zero cost moves.

9.3.2 *MA-CBS and other improvements to CBS*

In future work we plan to investigate the possibility of extending CIS and CISR in the same fashion as MA-CBS extends CBS. Since CBS and CIS/CISR have a similar structure due to the fact that they are both conflict based approaches the possibility of an extension similar to MA-CBS is worth exploring.

Further refinements to the MA-CBS approach are discussed by Boyarski et al[BFSS15, BFS$^+$15], and non-optimal versions of CBS are explored by Barer et al[BSSF14b]. Our current work is compared against the original algorithm[SSFS12a]. In future work we will compare against the refinements[BFSS15] and extensions[SSFS12b].

9.3.3 *Explore the Definition of Non-Complex*

We intend to explore the structure of Non-Complex sub-graphs and the accompanying definition of Non-Complex paths to discover the extent to which the Backtracking Algorithm can be improved and extended. The current definition may have room for improvement and allow the Backtracking algorithm or modifications thereof to be applied more often. For instance removing the restriction that Non-Complex paths must lie on a rudimentary path. This would allow paths to travel laterally by a significant amount outside of the usual Non-Complex sub-graphs. By merging multiple Non-Complex sub-graphs starting from different positions on the map but all ending on the same goal we would be able to reason about the extended definition of Non-Complex paths.

9.3.4 *Explore the Possibility of Graphs Close to Planar*

An exploration of the use of the planar condition in our algorithm may allow us to relax the condition in certain circumstances. First we may consider cases where there is a projection which minimizes the number of intersecting edges or large regions which overlap but do not have complex interconnections. We wish to explore the possibility of several layers of Non-Complex sub-graphs representing the possibilities (over or under the bridge of nodes, etc).

Another possibility to consider is square grids with diagonal movement. This brings edges which cannot be resolved into planar form. However a version of the backtracking algorithm with a dual layer which encompasses a wider channel of nodes may cover the possibilities which would have escaped from the original Backtracking algorithm.

9.3.5 *Profile-Based Preferential Ordering*

It may be possible, by profiling the properties and likely initial directions of agents, to customise the ordering function such that paths of equivalent cost favour those which lead away from other agents. In many ways, this is analogous to the CAT table of CBS. This optimisation would, potentially, reduce the possibility of clashes between agents which theoretically never need meet for any given minimum cost solution, but who might meet under the current schema. If such a profile is consistent throughout a given solution computation, the optimality and correctness of CIS (and, thence, CISR) would be unaffected.

References

[ARS+05] Richard Arnott, Tilmann Rave, Ronnie Schöb, et al. Alleviating urban traffic congestion. *MIT Press Books*, 1, 2005.

[BFS+15] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, Oded Betzalel, David Tolpin, and Eyal Shimony. Icbs: The improved conflict-based search algorithm for multi-agent pathfinding. In *Eighth Annual Symposium on Combinatorial Search*, 2015.

[BFSS15] Eli Boyarski, Ariel Felner, Guni Sharon, and Roni Stern. DonâĂŹt split, try to work it out: Bypassing conflicts in multi-agent pathfinding. In *Twenty-Fifth International Conference on Automated Planning and Scheduling*, 2015.

[BKL10] Subhrajit Bhattacharya, Vijay Kumar, and Maxim Likhachev. Distributed optimization with pairwise constraints and its application to multi-robot path planning. In *Robotics: Science and Systems*, pages 87–94, 2010.

[Bol98] Béla Bollobás. Graduate texts in mathematics: Modern graph theory, 1998.

[BSSF14a] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Seventh Annual Symposium on Combinatorial Search*, 2014.

[BSSF14b] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search*, pages 19–27, 2014.

[CGR96] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[DG97] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66, 1997.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DMC96] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[DP02a] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[DP02b] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[dWtMW14] Boris de Wilde, Adriaan W. ter Mors, and Cees Witteveen. Push and rotate: a complete multi-agent pathfinding algorithm. *Journal of Artificial Intelligence Research*, 51:443–492, 2014.

[EKÖS13] Esra Erdem, Doga Gizem Kisa, Umut Öztok, and Peter Schueller. A general formal framework for pathfinding problems with multiple agents. In *AAAI*, 2013.

[ELP87] Michael Erdmann and Tomas Lozano-Perez. On multiple moving objects. *Algorithmica*, 2(1-4):477–521, 1987.

[FBM12] Mohamad EL Falou, Maroua Bouzid, and Abdel-Illah Mouaddib. Dec-a*: a decentralized multiagent pathfinding algorithm. In *Proceedings of the 2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, pages 516–523, 2012.

[FSBY+04] A. Felner, R. Stern, A. Ben-Yair, S. Kraus, and N. Netanyahu. Pha*: Finding the shortest path with a* in an unknown physical environment. *Journal of Artificial Intelligence Research*, 21:631–670, 2004. cited By 20.

[GC11] Yanfeng Geng and Christos G Cassandras. A new âĂIJsmart parkingâĂİ system based on optimal resource allocation and reservations. In *Intelligent Transportation Systems (ITSC), 2011 14th International IEEE Conference on*, pages 979–984. IEEE, 2011.

[GFS+12] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, and Jonathan Schaeffer. A* variants for optimal multi-agent pathfinding. Technical report, Association for the Advancement of Artificial Intelligence, 2012.

[GFS+14] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan Sturtevant, Robert C Holte, and Jonathan Schaeffer. Enhanced partial expansion a. *Journal of Artificial Intelligence Research*, 50(1):141–187, 2014.

[GMF06] Arnon Gilboa, Amnon Meisels, and Ariel Felner. Distributed navigation in an unknown physical environment. In *Proceedings of the fifth international*

joint conference on Autonomous agents and multiagent systems, pages 553–560. ACM, 2006.

[Gol11] Oded Goldreich. Finding the shortest move-sequence in the graph-generalized 15-puzzle is np-hard. In Oded Goldreich, editor, *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*, volume 6650 of *Lecture Notes in Computer Science*, pages 1–5. Springer Berlin Heidelberg, 2011.

[HBD10] NHHM Hanif, Mohd Hafiz Badiozaman, and H Daud. Smart parking reservation system using short message services (sms). In *Intelligent and Advanced Systems (ICIAS), 2010 International Conference on*, pages 1–5. IEEE, 2010.

[HJW84] John Hopcroft, Deborah Joseph, and Sue Whitesides. Movement problems for 2-dimensional linkages. *SIAM Journal on Computing*, 13(3):610–629, 1984.

[HKI13] So Hashimoto, Ryo Kanamori, and Takayuki Ito. Auction-based parking reservation system with electricity trading. In *Business Informatics (CBI), 2013 IEEE 15th Conference on*, pages 33–40. IEEE, 2013.

[HNR68a] Peter E Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968.

[HNR68b] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[HSS84] J.E. Hopcroft, J. T. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects; pspace hardness of the "warehouseman's problem". *International Journal of Robotics Research*, 3(4):76–88, December 1984.

[JS79] Wm. Woolsey Johnson and William E. Story. Notes on the "15" puzzle. *American Journal of Mathematics*, 2(4):397–404, December 1879.

[KHS11] Mohktar M. Khorshid, Robert C. Holte, and Nathan R. Sturtevant. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *Proceedings, The Fourth International Symposium on Combinatorial Search*, pages 76–83, 2011.

[KMS84] Daniel Kornhauser, Gary Miller, and Paul Spirakis. *Coordinating pebble motion on graphs, the diameter of permutation groups, and applications*. IEEE, 1984.

[LB11a] Ryan Luna and Kostas E Bekris. Efficient and complete centralized multi-robot path planning. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, pages 3268–3275. IEEE, 2011.

[LB11b] Ryan Luna and Kostas E Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI*, pages 294–300, 2011.

[MBCT98] Soraia R. Musse, Christian Babski, Tolga Capin, and Daniel Thalmann. Crowd modelling in collaborative virtual environments. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '98, pages 115–123, New York, NY, USA, 1998. ACM.

[MN09] Ellips Masehian and Azadeh H Nejad. Solvability of multi robot motion planning problems on trees. In *Intelligent Robots and Systems, 2009. IROS 2009. IEEE/RSJ International Conference on*, pages 5936–5941. IEEE, 2009.

[PGT08] Julien Pettré, Helena Grillon, and Daniel Thalmann. Crowds of moving objects: navigation planning and simulation. In *ACM SIGGRAPH 2008 Classes*, pages 54:1–54:7, 2008.

[RBS+14] Callum Rhodes, William Blewitt, Craig Sharp, Gary Ushaw, and Graham Morgan. Smart routing: A novel application of collaborative path-finding to smart parking systems. In *Proceedings of the 2014 IEEE 16th Conference on Business Informatics*, 2014.

[Rey87] Craig W Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM Siggraph Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

[RH12] Gabriele Röger and Malte Helmert. Non-optimal multi-agent pathfinding is solved (since 1984). In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, pages 173–174, 2012.

[RW86] Daniel Ratner and Manfred Warmuth. Finding a shortest solution for the nxn extension of the 15-puzzle is intractable. In *AAAI-86 Proceedings*, pages 168–172, 1986.

[RW90] Daniel Ratner and Manfred Warmuth. The $(n^2 - 1)$-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10(2):111–137, 1990.

[SH00] Thomas Stützle and Holger H Hoos. Max–min ant system. *Future generation computer systems*, 16(8):889–914, 2000.

[Sil05] David Silver. Cooperative pathfinding. In *Proceedings of the First Artificial Intelligence and Interactive Digital Environments Conference*, pages 117–122, 2005.

[Sil06] David Silver. Cooperative pathfinding. In Steve Rabin, editor, *AI Game Programming Wisdom 3*, chapter 2.1, pages 99–112. Charles River Media, Inc., Massachusetts, 2006.

[SK11] Trevor Standley and Richard Korf. Complete algorithms for cooperative pathfinding problems. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, pages 668–673, 2011.

[SSFS12a] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent path finding. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, pages 563–569, 2012.

[SSFS12b] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Meta-agent conflict-based search for optimal multi-agent path finding. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search*, pages 97–104, 2012.

[SSGF11] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. In *Proceedings of the Twenty-Second International Join Conference on Artificial Intelligence*, pages 662–667, July 2011.

[Sta10] Trevor Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 173–178, 2010.

[Stu12] Nathan R. Sturtevant. Benchmarks for grid-based pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):144–148, 2012.

[Sur09] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 3613–3619. IEEE, 2009.

[Sur12a] Pavel Surynek. On propositional encodings of cooperative path-finding. In *Tools with Artificial Intelligence (ICTAI), 2012 IEEE 24th International Conference on*, volume 1, pages 524–531. IEEE, 2012.

[Sur12b] Pavel Surynek. A sat-based approach to cooperative path-finding using all-different constraints. In *SOCS*, 2012.

[Sur12c] Pavel Surynek. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *PRICAI 2012: Trends in Artificial Intelligence*, pages 564–576. Springer, 2012.

[SW02] Penelope Sweetser and Janet Wiles. Current ai in games: a review. *Australian Journal of Intelligent Information Processing Systems*, 8(1):24–42, 2002.

[TL06] Dušan Teodorović and Panta Lučić. Intelligent parking systems. *European Journal of Operational Research*, 175(3):1666–1681, 2006.

[vTCG11] Wouter G. van Toll, Atlas F. Cook, IV, and Roland Garaerts. Navigation meshes for realistic multi-layered environments. In *Proceedings of the 2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3526–3532, 2011.

[vTCG12] Wouter G. van Toll, Atlas F. Cook, IV, and Roland Garaerts. A navigation mesh for dynamic environments. *Computer Animation and Virtual Worlds*, 23(6):535–546, June 2012.

[Wan11] Ko-Hsin Cindy Wang. Tractable massively multi-agent pathfinding with solution quality and completeness guarantees. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, pages 2860–2861, 2011.

[WB11] Ko-Hsin Cindy Wang and Adi Botea. Mapp: a scalable multi-agent path planning algorithm with tractability and completeness guarantees. *Journal of Artificial Intelligence Research*, 42:55–90, 2011.

[WH11] Hongwei Wang and Wenbo He. A reservation-based smart parking system. In *Computer Communications Workshops (INFOCOM WKSHPS), 2011 IEEE Conference on*, pages 690–695. IEEE, 2011.

[Wil74] Richard M Wilson. Graph puzzles, homotopy, and the alternating group. *Journal of Combinatorial Theory, Series B*, 16(1):86–96, 1974.

[YL13] Jingjin Yu and Steven M LaValle. Planning optimal paths for multiple robots on graphs. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 3612–3617. IEEE, 2013.

[YMI00] Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. A* with partial expansion for large branching factor problems. In *AAAI/IAAI*, pages 923–929, 2000.

[YYRO11] Gongjun Yan, Weiming Yang, Danda B Rawat, and Stephan Olariu. Smartparking: a secure and intelligent parking system. *Intelligent Transportation Systems Magazine, IEEE*, 3(1):18–30, 2011.

Symbols List

A The set of all paths on G.

A_i The set of all paths on G which are associated with agent $i \in K$.

α_i The least path $v \in A_i$ with respect to the relation $<_t$.

B The branching set B. A set which can be used as a suitable replacement for a multiverse $\mathbf{v}$ in a working set W. The branching set covers all the solutions that $\mathbf{v}$ does however the multiverse itself is removed.

$\beta_t^f(v)$ The top branch time step. This value represents the latest time step before t that a branch from the path v can be made and that the new path can achieve the cost of f.

$\chi(v, u)$ The collision function. Given two paths $u, v \in A$ then if the two paths collide at any one point in time, including swaps, then the collision function returns that time, otherwise the function returns ∞. The collision function is extended to multiverses $\chi(\mathbf{v})$ by taking the minimum value between all pairwise paths from the multiverse $\mathbf{v}$.

Cx The set of all complex vertices. An agent for context will be assumed.

$\mathrm{CB}(v)$ The time until the complex path v makes its first complex branch.

$\mathrm{Cx}_t(v)$ Indicates that v makes a complex branch at time t.

$[.]$ A conditional value. If the expression in the brackets is false 0 is returned, otherwise if the expression is true then 1 is returned.

$\mathbf{v}_{-i}$ The multiverse $\mathbf{v}$ with the path associated with agent i removed.

$C(v)$ All paths which can be reached using successive iterations of the next algorithm originating from the path v.

$\langle j, k \rangle_n$ Represents an interval of integers which can wrap around from a maximum value. When $j \leq k$ then $\langle j, k \rangle_n = [j, k]$ but when $j > k$ then $\langle j, k \rangle_n = [0, k] \cup [j, n)$.

$D_{\mathbf{y}}(\mathbf{x})$ The shortest distance in nodes from $\mathbf{x} \in V$ to $\mathbf{y} \in V$.

$\mathrm{Edge}(\langle j, v]_n^d)$ Returns the index furthest away from the path that is an element of the winding, i.e. $\mathrm{Edge}(\langle j, v]_n^d) = j$ and $\mathrm{Edge}({}_n^d[v, k\rangle) = k$.

E The set of edges associated with a graph G.

$[v]_t^f$ The set of all paths which are equivalent under the equivalence relation $\sim_t$ and have.

$F(v)$ The cost of a path v. In this thesis cost is calculated as the number of time steps an agent spends away from its goal node.

$F_t(v)$ The truncated cost. An indication of the accumulated additional cost incurred up to time step t. This adds up successive relative costs which indicate whether the agent has been moving towards its goal and making progress or otherwise adding additional distance which will need to be traversed later.

$\mathbf{g}_i$ The goal node/vertex of an agent $i \in K$.

G An abstract construction of vertices V connected by edges from the set E.

$\mathrm{In}(\langle j, v]_n^d)$ The In function calculates the next inwards winding that can be reached from the given winding. The winding travels backwards towards the start node and contracts as far in towards the path v as the neighbours of $\langle j, v]_n^d$ allow.

$\mathrm{Journey}(v)$ The time until path v first reaches its goal node.

$\sim_t$ A set of equivalence relations. Two paths are said to equivalent with respect to the relation $\sim_t$ if they are identical up to the time step t.

$I(\mathbf{x}, \mathbf{y})$ The index of a movement made from $\mathbf{x}$ to $\mathbf{y}$. The index indicates the order in which edges are taken with respect to preference and distance to the goal.

$\mathrm{Inf}(\langle j, v]_n^d)$ The influence function calculates whether new options are available which could not be reacedh backwards from the point of conflict.

$J_d(\mathbf{x})$ The index function $J_d(\mathbf{x})$ assigns an index to every node in slice S_i^d. The assignment is unique by assigning the node on the alpha path as 0 then starting off to the right each vertex is assigned incrementally.

$\mathcal{J}_d(\mathbf{v})$ The slice index function specialized to paths. Returns the slice index of the node at distance d from the goal which lies on the path v. This function is only applicable for non-complex paths as the value is only uniquely defined in such cases.

K The set of agents associated with a MAPF problem.

L_i^f The set of all paths of cost f for agent i.

$L_i^f(\mathbf{v}_{-i})$ A set of representatives of paths from L_i^f which are unique before the first collision of agent i in configuration $\mathbf{v}_{-i}$.

$L_i^f(\mathbf{v}_{-i})[\widetilde{\mathbf{p}}]$ The equivalence class of all paths from the layer configuration $L_i^f(\mathbf{v}_{-i})$ which pass through the space time point $\widetilde{\mathbf{p}}$.

$Left$ An indication of relative positioning to the left assuming that the graph G is projected onto a plane with no edge crossings. The goal is assumed to be in front and edges or nodes to the left side of the point of reference are indicated with this value.

μ A map of the graph G onto the plane such that no edges cross.

$N_t^f(v)$ The next algorithm. Takes the existing path v and produces a path u of cost f while branching before the time step t and minimizing with respect to the relation $<_i$.

$\mathrm{Out}(\langle j, v]_n^d)$ The Out function calculates the next outwards winding that can be reached from the given winding. The winding travels backwards towards the start node and expands as far out from the path v as the neighbours of $\langle j, v]_n^d$ allow.

$P_t(v)$ The position of path $v \in A$ at time $t \in [0, \infty)$.

$\widetilde{\mathbf{P}}$ The space time point of path $v \in A$ at time $t \in [0, \infty)$. i.e. $\widetilde{\mathbf{P}}_t(v) = (P_t(v), t)$.

$\mathrm{Pref}(\mathbf{x}, \mathbf{y})$ The preference function returns the preference priority for taking the edge between $\mathbf{x}$ and $\mathbf{y}$ among all edges which travel away from $\mathbf{x}$. These values can be assigned arbitrarily.

$<_i$ An ordering relation on paths. Paths are ordered lexicographically based on the index function $I_t(v)$.

$P(v)$ The projection of $v \in A$ onto the set of vertices V. i.e. the subset of points from V that the path v visits.

$Q_t(v)$ The relative priority of the movement made at time step t for the path v. This indicates whether a path is moving towards the goal, pausing at equal distance to the goal or moving away from the goal.

$Right$ An indication of relative positioning to the right assuming that the graph G is projected onto a plane with no edge crossings. The goal is assumed to be in front and edges or nodes to the right side of the point of reference are indicated with this value.

R_i The set of all rudimentary paths for agent i.

S The set of all multiverses. Each multiverse $\mathbf{v} \in S$ being a tuple of paths, one for each agent. Each element representing a potential solution of the MAPF problem.

S' The set of true solutions of the given MAPF problem. Each element $\mathbf{v}$ of S' contains no collisions. i.e. $\forall \mathbf{v} \in S', \chi(\mathbf{v}) = \infty$.

$\mathrm{Side}(\langle j, v]_n^d)$ Returns the side of the winding, i.e. a right winding returns $Right$ and a left winding returns $Left$.

$\mathrm{Side}^c(\langle j, v]_n^d)$ Returns the complement or reverse of the side of a winding, i.e. a right winding returns $Left$ and a left winding returns $Right$..

S_i^d The slice at distance d from the goal of agent i. This is the set of all non-complex nodes at distance d from the goal.

$S_i^d(j)$ Suppose $j \in [0, |S_i^{d+1}|)$ is an index from S_i^{d+1} then $S_i^d(j)$ is the subset of S_i^d which are neighbours to $J_{d+1}^{-1}(j)$. And suppose $j \in [0, |S_i^{d-1}|)$ is an index from S_i^{d-1} then $S_i^d(-j)$ is the subset of S_i^d which are neighbours to $J_{d-1}^{-1}(j)$.

$S_i^d(\mathbf{x})$ Suppose $\mathbf{x} \in S_i^{d\pm1}$ is a node from either side of S_i^d then $S_i^d(\mathbf{x})$ is the subset of S_i^d which are neighbours to $\mathbf{x}$.

$\widetilde{\mathbf{p}}$ An arbitrary point in space and time represented by the tuple of the form $\widetilde{\mathbf{p}} = (\mathbf{x}, t)$, where $\mathbf{x} \in V$ and $t \in [0, \infty)$. For convenience we represent the components as $\widetilde{\mathbf{p}}_{\mathbf{x}} = \mathbf{x}$ and $\widetilde{\mathbf{p}}_t = t$.

$\mathbf{s}_i$ The start node/vertex of an agent $i \in K$.

$\mathcal{S}_t^f(v)$ A collection of higher cost branches which are skipped over by the next algorithm. These paths are marked for later exploration when the target cost of the search increases appropriately.

$\mathrm{Sub}_i(\mathbf{v}, B)$ The substitution of of the paths in B into the multiverse $\mathbf{v}$ for the path of agent $i \in K$.

$\mathbf{v}$ A multiverse from the set of potential solutions S. Each multiverse $\mathbf{v}$ being a tuple of paths v_i, each path being associated with the corresponding agent $i \in K$.

V The set of vertices associated with a graph G.

W A set of paths which cover all solutions to a particular MAPF problem.

$\langle j, v]_n^d$ The right winding represents the interval of indices between the search for an alternative option on the right side and the path v. The winding $\langle j, v]_n^d$ is equal to the interval $\langle j, \mathcal{J}_d(v)\rangle_n$.

$\frac{d}{n}[v, k\rangle$ The right winding represents the interval of indices between the search for an alternative option on the right side and the path v. The winding $\frac{d}{n}[v, k\rangle$ is equal to the interval $\langle \mathcal{J}_d(v), k\rangle_n$.

Acronyms

CAT Collision Avoidance Table.

CBS Collision Based Search.

CIS Collaborative Iterative Search.

CISR Collaborative Iterative Search Reasoning.

CT Constraint Tree.

HCA Hierarchical Cooperative A*.

ICTS Iterative Cost Tree Search.

ID Independence Detection.

IQM Inter Quartile Mean.

MA-CBS Meta Agent Conflict Based Search.

MAPF Multi-Agent Path Finding.

MAPP Multi Agent Path Planning.

MDD Mutli-value Decision Diagram.

NC Non-Complex.

OD Operator Decomposition.

RRA* Reverse Resumable A*.

SOC Summation Of Costs.

TASS Tree based Agent Swapping Strategy.

Glossary

A* The A* algorithm is a path finding algorithm which uses a heuristic to improve the search time of the algorithm. Using a guestimate of distance called the heuristic the A* selects which nodes to expand next based upon the sum of the heuristic and the distance travelled to that node.

Agent An Agent represents an entity which exists on the vertices and edges of a graph. Each agent has a starting node and a goal node and aims to traverse from one to the other using the edges to move from vertex to vertex.

Agentverse A path associated with a particular agent. Can also refer to the data structure which describes the implementation of a path in our CIS algorithm.

Alpha Path The minimal path of a given agent with respect to the $>_i$ relation.

Ancestor Given a path v the an ancestor of v is a path which can be supplied to the Next algorithm with particular parameters such that v is returned.

Backtracking The act of stepping backwards in time to search for an alternative choice or route around a conflict. The nodes traversed during this process need not be on the original path.

Backwards An edge which leads towards the start nodes.

Boids The term Boids refers to an AI abstraction of the act of flocking. The algorithm developed by Craig Reynolds describes the movement of birds/fish in mathematical terms of Separation, Alignment and Cohesion.

Branching Set A set of multiverses which cover all solutions that a given multiverse $\mathbf{v}$ does, is contained in the cover of $\mathbf{v}$ but does not contain $\mathbf{v}$.

Bypass A region of nodes described between two paths. On a planar graph this region is contained and any path leading out of the region has to pass through one of the nodes on the boundary.

Centralized A Centralized approach is opposed to decentralized approaches. Centralized approaches compute a solution in a single computational unit rather than computing parts of the problem separately and combining the solution later.

Collaborative Iterative Search An iterative approach to solving the MAPF problem. The basis for our approach to MAPF.

Collaborative Iterative Search Reasoning An extension of the CIS method using a backtracking approach to discover alternative routes around collisions.

Collaborative Path Finding Collaborative Path Finding is the generalization of Path Finding to multiple agents. The term Collaborative Path Finding is generally a synonym for MAPF however is used in earlier works on real time Multi Agent problems such as HCA where the completeness of the algorithm is less important than timely results.

Collision An event where two agents try to occupy the same location at the same time. This can happen on a time step on a single vertex or between time steps on an edge between vertices.

Collision Avoidance Table The Collision Avoidance Table (CAT) is a table of locations which lowers the priority at which certain choices can be made helping break tie breakers which could lead to unwanted collisions. The CAT is used in both General A* approaches and the CBS algorithm and only effects arbitrary choices which would lead to paths of equivalent value.

Collision Function The collision function $\chi(u, v)$ returns the first time step the two paths u and v collide or a half time step if they collide between time steps.

Completeness A MAPF solver has completeness if it can always find a correct solution for a MAPF problem when one exists.

Complex Refers to elements of a graph of a complex nature, i.e. a complex edge/point/path.

Complex Branch Function The time step at which a complex path is forced to take its first complex edge.

Complex Edge An edge which does not lie on any non-complex path.

Complex Path A path which does not share a projection set with a rudimentary path.

Complex Point A point which does not lie on any non-complex path.

Configuration Given a particular agent i, a configuraion is a mutiverse with the path associated with agent i removed from the tuple. This structure gives the outer context of a collision by removing the path which needs replacing.

Conflict Based Conflict Based refers to MAPF algorithms which branch search nodes when a conflict is detected between two agents in a potential solution. Conflict based approaches split the computation into two layers to solve the MAPF problem. The upper layer solves the overall problem, each node representing a potential solution. The lower layer solves for paths of individual agents using constraints or specialised search algorithms.

Conflict Based Search A solver for MAPF which searches for conflicts and then adds constraints to a lower layered solver such that the same conflicts does not happen again.

Connection Side When referring to edges of a vertex we can categorize them with respect to an existing path or from the perspective of an agent. Connection side refers to the relative side an edge leads to when considering the goal to be forwards and the start to be behind. The left side would be nodes generally clockwise from the forwards direction, or anti clockwise from the backwards direction, similarly for the right.

Constraint A restricting condition which is imposed on a solver. How a constraint is used depends on the algorithm it is applied in however it generally stops a particlar condition from occurring.

Constraint Tree A Constraint Tree (CT) is a concept used in the CBS algorithm. A constraint tree is a tree which its nodes consist of individual constraints for particular agents. The leaves of the tree represent a list of constraints by tracing the constraints from the leaf to the root.

Correctness Correctness refers to a MAPF solution which contains no collisions.

Cost A heuristic value attached to a path measuring its worth.

Cover The set of paths for which can be reach by successive applications of Next on v.

Crowd Simulation Crowd Simulation is the process of simulating large crowds of entities. The emphasis is moved from the joint behaviour of flocking to the dynamic of large number of entities in a finite space.

Cyclic Interval A generalization of the concept of an interval by allowing for the interval to wrap around in a cyclic manner.

Decentralized Decentralized approaches share computation among several processing units which then communicate to combine the separate results into one solution.

Dijkstra's Algorithm The Dijkstra algorithm is a path finding algorithm. It maintains a list of distances associated with each node. When the algorithm can prove the shortest distance to a given node, because the shortest distance to all nodes leading to it have been calculated, it is labelled as such and can be used to calculate the shortest distance to other nodes. Eventually a path to the goal node is found when the shortest distance can be calculated.

Distance An idea of distance on a graph. Two vertices are of distance n if there exists a path of n edges between the two vertices and no shorter path can be found.

Edge A connection between Vertices on a graph. Edges can be considered lines in *N* dimensional space between vertices if the graph has physical representation. However the abstract notion of relation between two vertices is all that is needed for an entity to be called an edge.

Equivalence A generalized concept of equality. In this thesis we define a number of equivalence relations between elements which share certain properties. An equivalence relation satisfies three properties: Transitivity, Reflexive and Symmetric. i.e. Transitivity: if A relates to B and B relates to C then A relates to C. Reflexivity: A relates to A. Symmetry: if A relates to B then B relates to A.

Equivalence Class The set of all elements which are equivalent to a given representative.

f-value The f-value refers to the sum of the g-value and the h-value of a node giving an estimate of overall fitness for a node during the computation of the A* algorithm.

Face A loop of vertices in the lane with an empty interior, i.e. for the side indicated as the interior no edges cross the face and no vertices exist inside f it.

Face Loop A complete loop which travels through all the vertices in a particular slice. The loop does not cross itself and contains all vertices in the slice. The loop does not cross any edges of the non-complex sub graph as it passes only through the faces of the graph.

First Option The first half of the Backtracking algorithm which searches for available alternative routes.

Flocking Flocking is a general term for behaviour based on the collective movement of a group of animals which travel in packs/schools/flocks.

Forwards An edge which leads towards the goal node.

g-value The g-value refers to the distance travelled to a node from the start during the computation of the A* algorithm.

Goal The end location for a particular agent.

Goal Oriented Action Planning Goal Oriented Action Planning refers to the problem of assigning tasks and resources such that a larger system can function correctly. These resources may have interdependency and mutual exclusivity depending on the nature of the problem. GOAP shares similarity to MAPF and in some cases can be rewritten in terms of the navigation of multiple agents in a graph.

Graph An abstract concept of connecting elements. A graph is made up of vertices which are point entities. These Vertices are connected together by a line called an Edge. The edge can be curved or straight. The configuration of connections between vertices is the meaning of a graph and graphs are considered equivalent depending on the configuration of edges and vertices irrespective of representation in space.

h-value The h-value refers to the heuristic value of a node during the computation of the A* algorithm.

Heuristic A Heuristic is a guestimate of the distance from a node in a graph to the goal node. For correctness in A* the heuristic must always be an underestimate of the distance.

Hierarchical Cooperative A* Hierachical Cooperative A* is a MAPF solver aimed at real time computation. Agents take turns to fill a reservation table of planned moves which informs other agents where they can move to during which time steps.

Independence Detection Independence Detection (ID) is a MAPF technique by which the problem can be split into independent groups for separate computation to reduce overhead. The MAPF solver is run independently for small groups of known interacting agents and then collisions are searched for in the results. If collisions are found the groups are merged and the process is begun again..

Index A value assigned to a choice made during the traversal of a path. 0 indicates the most preferred option incrementally increasing through less preferred options. Given a fixed set of preferences index depends on the context of the agent and the direction of their goal node.

Indexing Function The function which assigns an index to each choice along a path. Starting from 0 for the most preferred path and iterating through the choices including the pause choice. Each choice takes acount of the relative cost of the move with choices which have the smallest relative cost coming first.

Influence Function A function which indicates whether a winding has a valid option for escaping a given conflict. The influence function calculates whether new options are available which could not be reached backwards from the point of conflict.

Iterative Cost Tree Search Iterative Cost Tree Search is a MAPF solver which uses a different approach than the general A* method. Agents are assigned target costs and a data structure called a Multi-value Decision Diagram (MDD) is constructed which represents all paths of the given cost. An upper layer to the algorithm

constructs configurations of these target costs for each agent and searches for the most optimal combination which has a solution as indicated by the MDD.

Journey The time until a given path reaches the goal for the first time.

Layer All paths of a given cost for a particular agent.

Layer Configuration The layer configuration is a restriction on the layer set such that only one path exists to represent each unique path before the time of collision with a given configuration.

Layer Equivalence Layer Equivalence is an equivalence class for each space time point for each layer configuration. This equivalence represents the fact that all paths which lead to the same point of collision need to be removed from consideration and can be considered the same.

Left Given a connection leading forwards the next edge clockwise around the given vertex is considered to the left. Given a connection leading backwards the next edge anti-clockwise around the given vertex is also considered to the left.

Lexicographic Ordering Lexicographical Ordering is an ordering which applies to compound structures. The ordering is applied to each element of the structure in turn. The most common usage of this ordering is alphabetical ordering.

Local Maximum A vertex on a face which neighbours two nodes which are nearer the goal.

Local Minimum A vertex on a face which neighbours two nodes which are nearer the start.

Meta Agent Conflict Based Search Meta Agent Conflict Based Search (MA-CBS) is an extension to the CBS algorithm. Agents can be combined into Meta Agents which are solved separately using another technique such as Generalized A*. The main algorithm treats these agents as singular agents in a manner similar to CBS however if two Agents (including meta agents) collide too frequently then they can be combined into a meta agent.

Minimal Connection The second half of the Backtracking algorithm which connects the alternative route to the existing path in as small a distance as possible.

Multi-Agent Path Finding The extension of Path Finding to multiple agents. A number of agents occupy a given graph G. These agents attempt to traverse the graph travelling from their start node to the goal node over a number of time steps. If two agents occupy a given node at the same time or attempt to traverse the same edge at the same time this is considered to be a collision. The aim of

Multi-Agent Path Finding is to find a consistent set of paths through time for each agent such that no two agents collide and each agent reaches its goal node.

Multi-Value Decision Diagram Mult-value Decision Diagram (MDD) is a directed graph which represents all paths from a start node to the goal node of a given cost. Each node is a space time point and connects to each possible choice that will lead to a solution of the target cost..

Multiverse A collection of Agentverses. One for each agent. The Multiverse may include colliding paths, however the Multiverse is considered a potential solution during computation and will be modified to remove any collisions found.

Next The Next algorithm is an algorithm which increments a path along the ordering given by the $>_t$ relation. The Next algorithm is the basis of the CIS algorithm.

Node Another name for a vertex on the graph G.

Non-Complex Refers to elements of a graph of a non-complex nature, i.e. a non-complex edge/point/path.

Non-Complex Edge An edge which exists along some non-complex path.

Non-Complex Path A path which projects onto the same set of nodes as a rudimentary path.

Non-Complex Point A point which exists along some non-complex path.

Non-Complex Sub-Graph A subgraph of the map of all the non-complex points and non-complex edges with respect to a given agent.

Non-Optimal Non-Optimal solvers will find a solution but will not have any guarantees on the cost of the solution.

NP-Hard NP-Hardness is a class of Computer Science problems which are provably at least as hard as the hardest NP problem. This gives a lower bound on the time complexity of the problem and generally shows that the problem can at least grow exponentially in computation time given certain input.

Operator Decomposition Operator Decomposition is an optimisation technique for the generalized approach to MAPF. Operator decomposition splits the processing of each time step into a sub time step for each agent reducing the branching factor of each time step.

Optimal Optimal solvers find solutions with a maximized or minimized cost out of all possible solutions.

Partial Expansion A* Partial Expansion A* reduces the problem of large branching factors by splitting the groups of branches into groups and merging them into nodes which can be explored later.

Path A set of space time points describing a journey on the graph. The path includes one and only one point for each time step. Each two consecutive space time points along the path either have neighbouring vertices (via an edge between the two) or the share the same vertex.

Path Finding The act of finding a path of connected nodes through a graph from a start node to a goal node.

Path Selection Algorithm The algorithm which combines the application of next for the complex paths and backtracking for the non-complex paths in the CISR algorithm. This is also where pause migration is removed.

Pause Migration Pause Migration is the recurrent behaviour of moving the time at which an agent pauses further back in time but retaining the same path. If this is attempted when trying to solve a collision in a later time step the process of pause migration is wasteful and therefore a recurrent behaviour which needs removal.

Planar A graph which can be drawn in the 2D plane without two of its edges from intersecting.

Planar Projection A mapping for a particular graph which draws it within the plane without any of its edges intersecting.

Point Representative A single path which represents all paths which travel through this point in space and time.

Preference An arbitrary value assigned to each edge leading away from a given node. These values are used to order equivalent choices when choosing paths of the same length or cost.

Preferential Ordering An arbitrary ordering we apply to all connections in a graph. This ordering resolves conflicts in equivalent choices and allows us to order the set of all paths.

Priority Priority is the value given to a connection given the context of a particular agent. Edges which lead to the goal are given a higher priority and are chosen first above edges which are preferred but have lower priority.

Projection The set of locations visited by a path. This set does not include the associated time step and is solely a subset of the vertex set of G.

PSPACE-Hard PSPACE-Hard describes a set of problems which grow in memory storage at a polynomial rate as the size of the problem increases.

Recurrent Behaviour A recurrent behaviour is a repeating pattern of work done by an algorithm which can be removed when the pattern is recognised. Recurrent behaviour is an informal name for these patterns which we have identified.

Region A contiguous set of nodes on a graph or an enclosed area in the plane.

Relative Priority Relative priority is a measure of the gain in cost a particular edge will add to a path. Edges which lead towards the goal are given relative priority 0, edges which lead to a node equidistant to the goal are given the value (along with the pause action which leaves the agent in place), and the value 2 is given to edges which lead away from the goal.

Reverse Resumable A* The A* algorithm searches for the shortest path from a starting node to a goal node. While computing this path a value called the g-value is maintained for each node representing the distance travelled from the start node. Reversing the direction of this algorithm and processing the path from the goal node to the start node this algorithm can be used to calculate the distance from any node to the goal node as long as the algorithm is run long enough to process the node.

Right Given a connection leading forwards the next edge anti-clockwise around the given vertex is considered to the left. Given a connection leading backwards the next edge clockwise around the given vertex is also considered to the left.

Rudimentary Path A path of minimal cost from the start node to the goal node.

Search Space The set of all potential solutions to a given MAPF problem. This will include elements which have collisions between agents.

Slice The set of all vertices from a non-complex sub graph of a given distance from the goal.

Slice Index An index assigned to each vertex in a face loop in order. The assignment is unique by assigning the node on the alpha path as 0 then starting off to the right each vertex is assigned incrementally.

Smart Parking Smart Parking is the use of navigation systems and AI processing to optimise the use of parking in an urban environment.

Solution A set of paths for each agent which is included in the MAPF problem such that no two paths collide and each agent reaches its destination from its start node.

Solution Space The set of all solutions to a given MAPF problem.

Space Time Point A Space Time Point is a location on the graph, i.e. a vertex, with an associated time. This structure represents an intermediary point in time and space along the path of an agent.

Space-Time Point Equivalence The equivalence relation which equates all paths which travel through a given space time point as the same.

Start The starting location for a particular agent.

Stationary Segment The portion of a path after the journey segment. This portion may or may not contain additional movement of the agent as other agents attempt to pass through its goal square.

Stem The Stem is a set of alternative branches which have been skipped over by the Next algorithm because their cost was too high to be the next considered alternative.

Sub-Graph A selection of Vertices and Edges selected from an existing graph which constructs a complete graph, i.e. each of the edges have ends which are a part of the selected subset of vertices.

Summation Of Costs A heuristic for MAPF which sums all the costs of each individual path of a composite solution, i.e. the sum of all paths included in an element of the search space.

Swap A particular type of collision where two agents try to cross a single edge at the same time from opposite directions.

Time Complexity Time Complexity refers to the asymptotic behaviour of the computation time of an algorithm. Time complexity is described by a single expression which describes the general long term behaviour of the algorithm. Slower growing terms of the expression are ignored and constant multiples are removed.

Time Step A Time Step represents an individual unit of time. Between two time steps an agent can move across one edge from one vertex to an adjacent vertex. On a time step each agent will occupy a single vertex.

Top Branch Top Branch is the latest alternative branch which can be taken along a path that will given a path of a particular cost.

Truncated Cost Truncated cost is the cost accumulated up to a particular time step. This is equivalent to the sum of the relative priorities up to the given time step.

Verse Set The set of all Agentverses for a particular agent.

Vertex A point representing a single entity of a graph.

Winding A cyclic interval which is anchored to a path at one of its ends. The winding can be to the left or the right and indicates itself as such.

Working Set A set of multiverses which cover all solutions of a given MAPF problem.

Appendices

Appendix A. Maps

A.1 Map Key

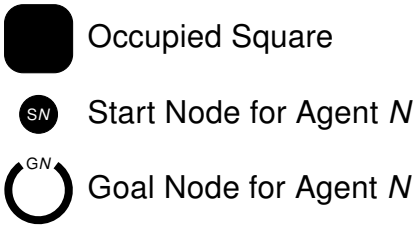


Figure A.1: Map Key

A.2 Permute Maps

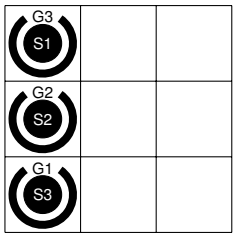


Figure A.2: Permute 1

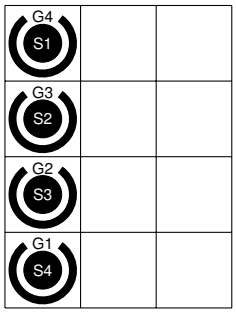


Figure A.3: Permute 2

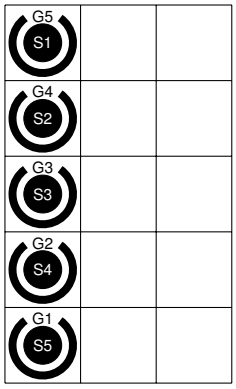


Figure A.4: Permute 3

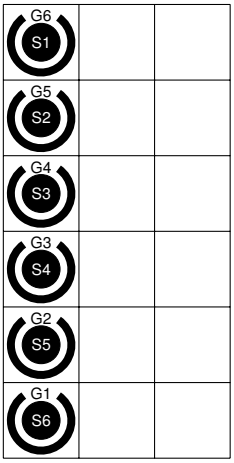


Figure A.5: Permute 4

A.3 Outline Maps

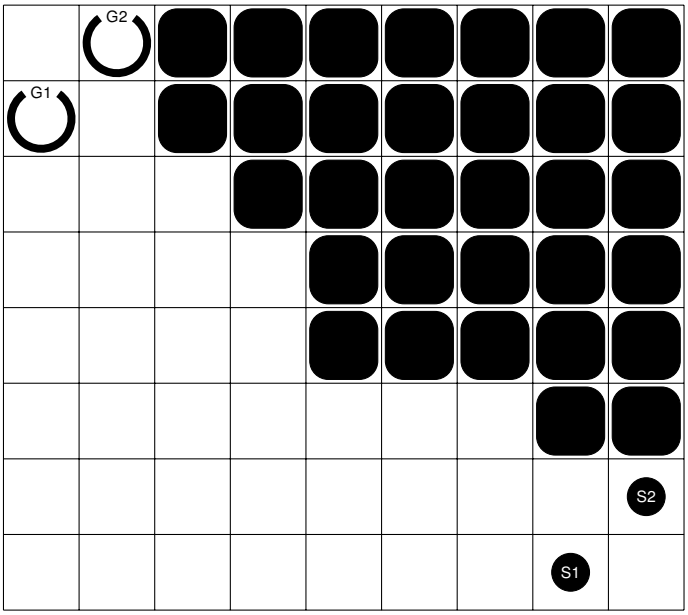


Figure A.6: Outline

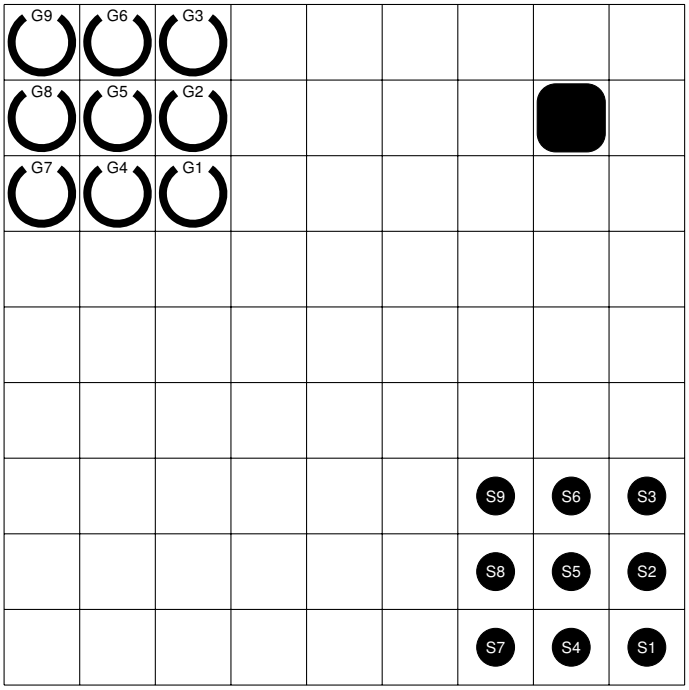


Figure A.7: Outline Grid

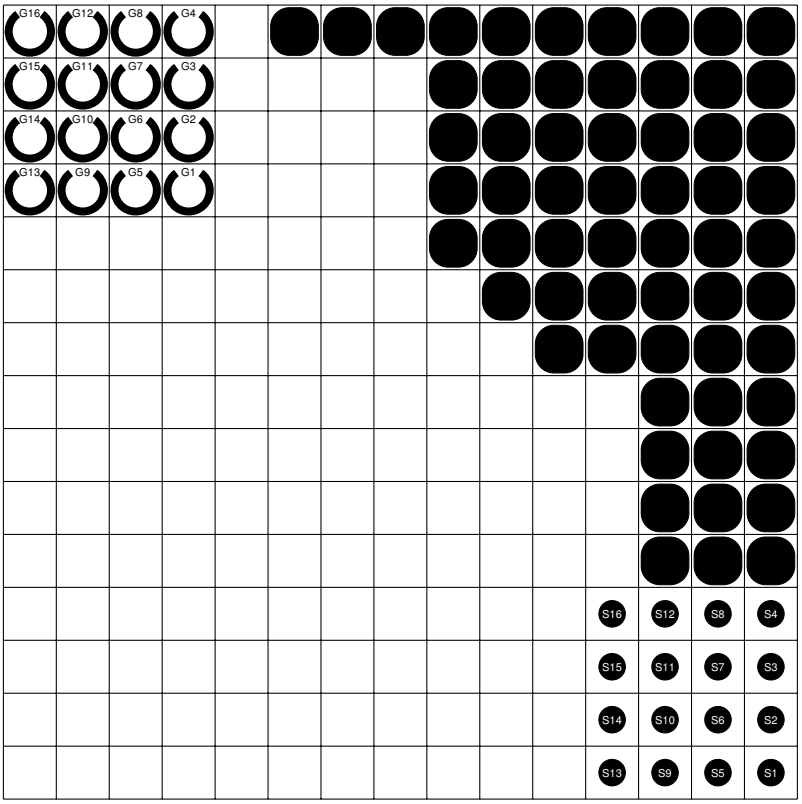


Figure A.8: Outline Grid 2

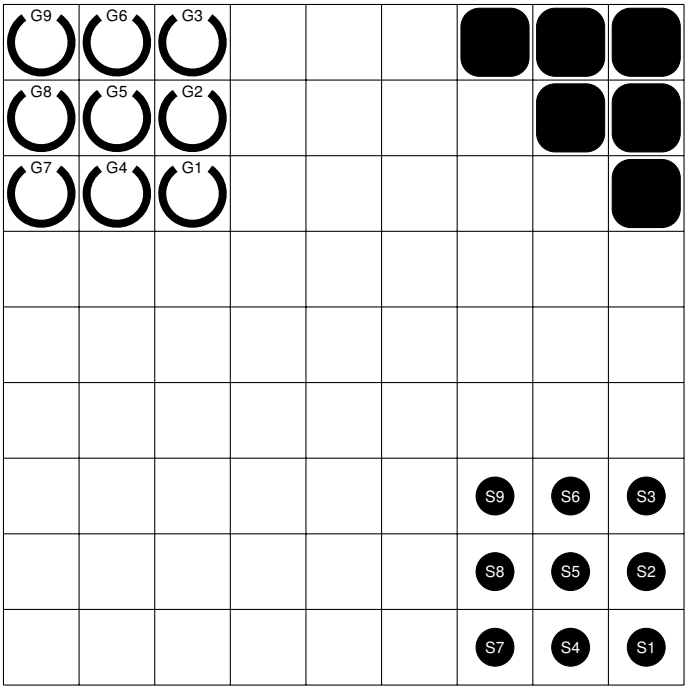


Figure A.9: Outline Grid 3

A.4 Maps with Geometry

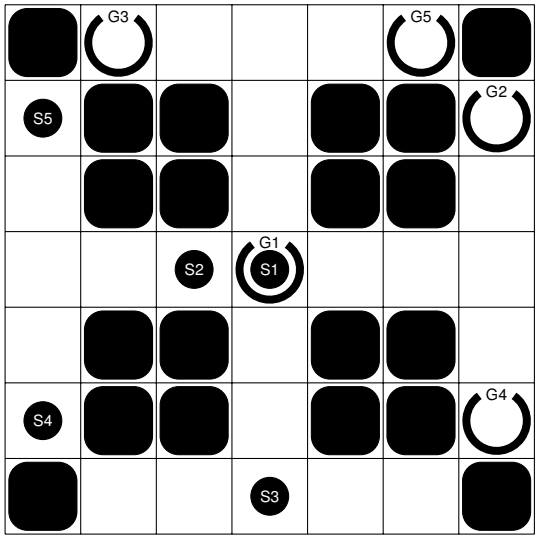


Figure A.10: Crossroad 1

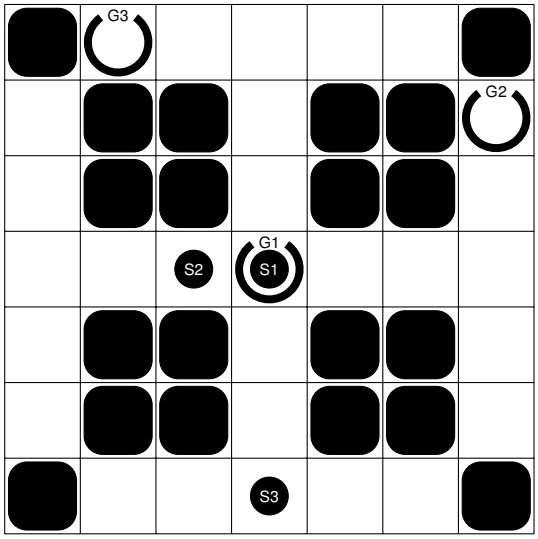


Figure A.11: Crossroad 2

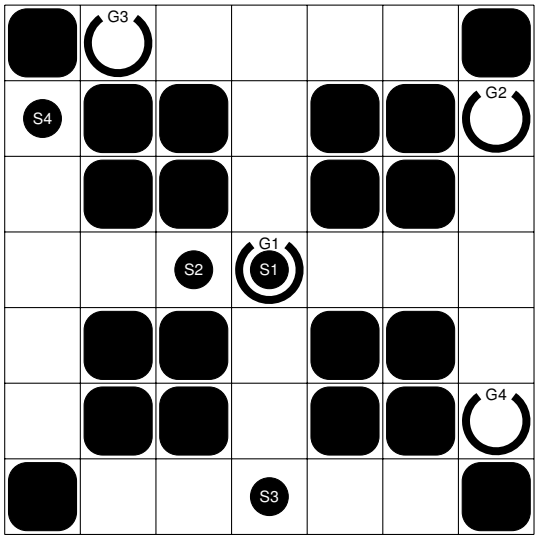


Figure A.12: Crossroad 3

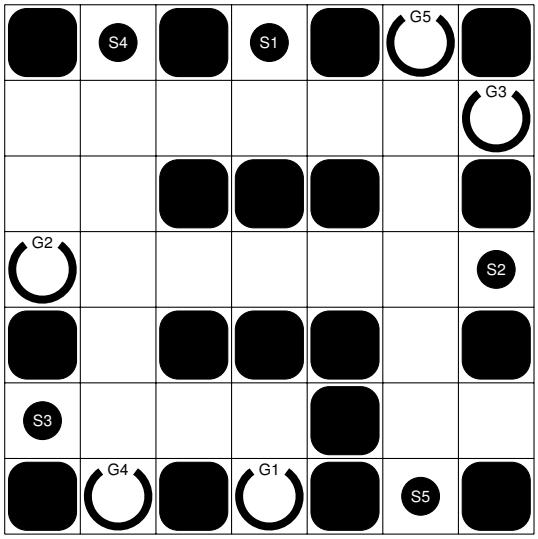


Figure A.13: Geometry 2

A.5 Swapping Maps

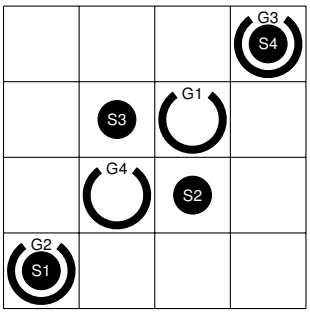


Figure A.14: Bypass

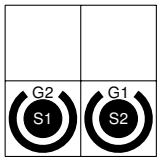


Figure A.15: Swap Test

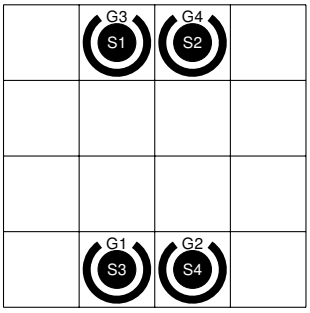


Figure A.16: Swap Test 2

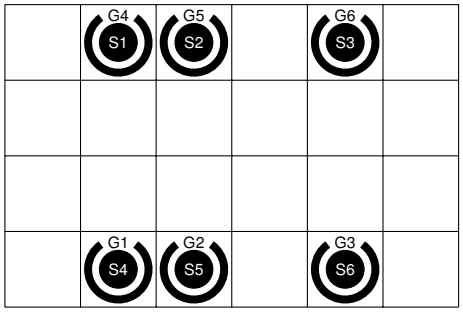


Figure A.17: Swap Test 2.5

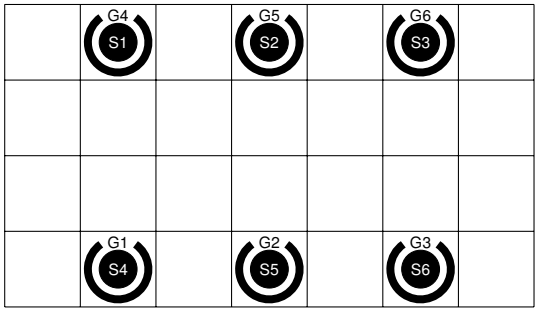


Figure A.18: Swap Test 2.7

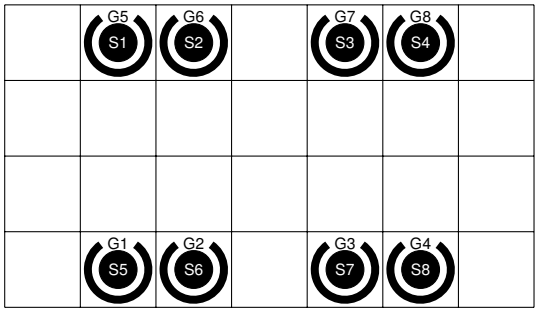


Figure A.19: Swap Test 3

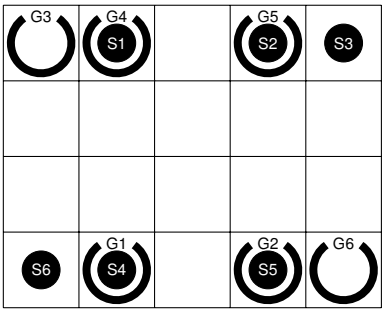


Figure A.20: Swap Test 4

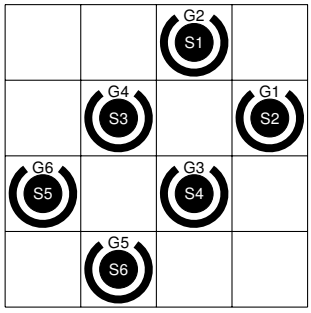


Figure A.21: Pass

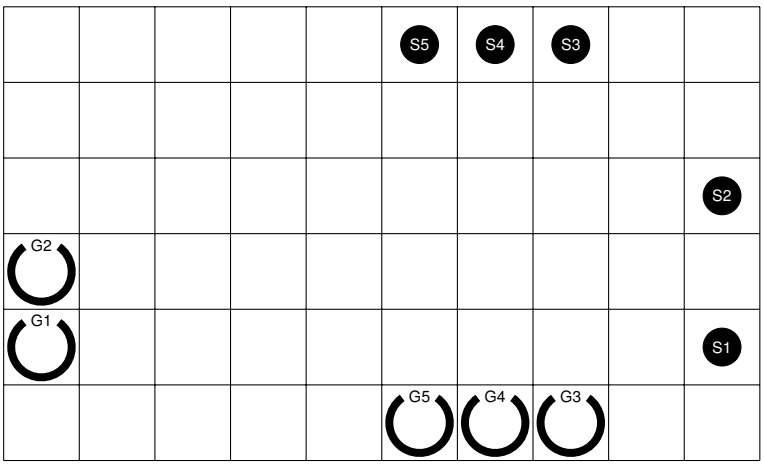


Figure A.22: Pass 1

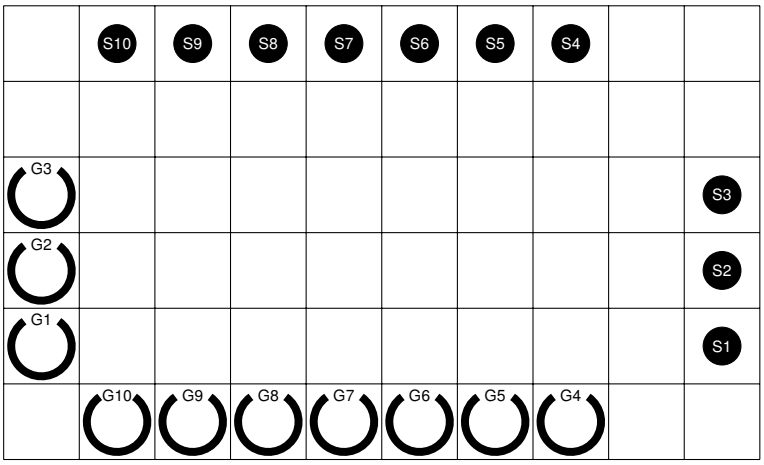


Figure A.23: Pass 2

Appendix B. Pseudo Code

Agentverse `getNext`(i, v, t, s)**:**
  Data: Agent: i, Agentverse: v, Collision Time: t, Target Cost (Stem): s
  Let r be the target relative cost $s - v.cost$;
  if $r > 2$ **then**
    return no solution
  end
  Let b be the result of `getBranch`$(i,\ v,\ t,\ r)$;
  if b *is a valid branch* **then**
    return A new Agentverse at b;
  else
    return `getNext`$(i, v, t, s + 1)$;
  end
end

Algorithm 3: The *Next* Algorithm.

unsigned int `getBranch`$(i,\ v,\ t,\ r)$**:**
  Data: Agent: i, Agentverse: v, Collision Time: t, Target Relative Cost: r
  if *Relative cost* $r > 2$ **then return** no solution;
  if *Time of collision* $t < 0$ **then return** no solution;
  if *Time of collision* $t \leq v.startTime$ **then**
    if *Branch from parent to child is a valid location to branch from and has the correct change in priority* **then**
      return $v.startTime - 1$;
    else
      return `getBranch`$(i,\ v.parent,\ \min(t, v.startTime - 1),$
       $r + v.cost - v.parent.cost)$;
    end
  end
  if *The available priority at* $t - 1$ *matches the target relative priority* r **then**
    return $t - 1$;
  else
    Let s be the step of time step $t - 1$, then use the time step given in $s.priorities$ which matches the target relative cost r;
    return time of alternative given by $s.priorities$;
  end
end

Algorithm 4: Calculate Branch Step.

void `makeBranch(`i, m, t`)`**:**
  Data: Agent: i, Multiverse: m, Collision Time: t
  Let vu be the VerseUnit $m.verses[i]$;
  Let $next$ be the result of `getNext(`i, $vu.verse$, t, $vu.stem$`)`;
  if $next$ *is not valid* **then**
    return
  end
  Let $curr$ be the Multiverse with $next$ replacing the verse for agent i;
  The stem for agent i is reset too $next.cost$;
  if cs *does not contain* $curr$ **then**
    Insert $curr$ into cs;
    Push $curr$ onto pq;
    Create a stem $stem$ from m by incrementing the $vu.stem$ of agent i;
    Push $stem$ onto pq;
  end
end

Algorithm 5: Make a single branch using *Next*.

Multiverse `search()`**:**
  Initialize priority queue pq;
  Initialize closed set cs;
  Push initial Multiverse $curr$ onto pq;
  Insert $curr$ into cs;
  while pq *not empty* **do**
    Pop a Multiverse from pq and store in $curr$;
    if $curr.stemType = STEM$ **then**
      `makeBranch(`$curr.stemAgent$, $curr$, $curr.stemTime$`)`;
      continue;
    end
    Search $curr$ for a clash;
    if *No clash* **then**
      return $curr$
    Let i, j be the agents invloved in the clash and t be the time of the clash;
    `makeBranch(`i, $curr$, t`)`;
    `makeBranch(`j, $curr$, t`)`;
  end
  return no solution;
end

Algorithm 6: Main Algorithm loop, without backtracking.

void `backtrack`(i, $curr$, t, $swap$, $side$)**:**
    Data: Agent: i, Multiverse: $curr$, Collision Time: t, Swap: $swap$, Side: $side$
    initialize scanner sc;
    initialize solver sv;
    if $swap$ **then**
        find next node just before swap collision;
    do
        step scanner;
        step solver (including any initial corrections for swap collisions);
        find potential solutions along solver;
    while *tracker not at start and tracker has no solutions*;
    if *tracker has solution and solutions isn't one explored before* **then**
        push new tracker solution onto priority queue;
    end
    create complex stem along boundary traced by the tracker;
    create non-complex stem along original path before the time of the collision;
end

Algorithm 7: The *Backtracking* Algorithm.

unsigned int `scanBack`(i, v, t)**:**
    Data: Agent: i, Agentverse: v, Current Time: t
    Let r initially be 0;
    do
        Add to r the value $Q_t(v) - 1$;
        decrement t;
    while $r \neq -1$;
    return t;
end

Algorithm 8: Scanner scanBack.

unsigned int `nextStep`(i, v, s)**:**
    Data:
    if $stick$ **then**
        $next \leftarrow id$;
    else
        call stepBackwards routine;
    end
    push current node onto boundary path;
    if *going inwards* **then**
        push current node onto solution path;
    end
    $forwards \leftarrow$ direction dictated by $next, curr$;
    $curr \leftarrow next$;
end

Algorithm 9: Solver nextStep.

```
void preprocess(i, v):
    Data: Agent: i, Agentverse: v
    if v.parent = null then
        v.complex ← false;
        return;
    if v.parent.complex And v.startTime > v.parent.cb then
        v.cb ← v.parent.cb;
        v.complex ← true;
        return;
    if v.startTime ≥ Journey(v) then
        v.cb ← v.startTime;
        v.complex ← true;
        return;
    end
    dist ← D^G(P_{v.startTime}(v));
    if dist > D^G(Start) then
        v.cb ← v.startTime;
        v.complex ← true;
        return;
    end
    calcExtAndComplexity(i, v, dist);
end
```

Algorithm 10: Pre-process Extent and Complex Branch.

void `calcExtAndComplexity`$(i, v, dist)$**:**
Data: Agent: i, Agentverse: v, Staring Distance: $dist$
Let u initially be equal to $v.parent$;
Let b initially be the value $invalidIndex$;
foreach u *such that the extent* $u.ext \leq dist$ **do**
Let $\mathbf{x}_u$ be the first position in Agentverse u;
Let $\mathbf{x}_v$ be the last position along v at distance $D^G(\mathbf{x}_u)$ from the goal;
if $\mathbf{x}_u \neq \mathbf{x}_v$ **then**
Let b be redefined as $\min(b, v.startTime + dist - D^G(\mathbf{x}_u))$;
end
end
if $u \neq null$ **And** $F(u) \neq F(v)$ **then**
Let $\mathbf{x}_v$ be the position $P_{v.startTime}(v)$;
Let $\mathbf{x}_u$ be the last position along u at distance $D^G(\mathbf{x}_v)$ from the goal;
if $\mathbf{x}_v \neq \mathbf{x}_u$ **then**
$v.cb \leftarrow v.startTime$;
$v.complex \leftarrow true$;
return;
end
if $b \neq invalidIndex$ **then**
$v.cb \leftarrow b$;
$v.complex \leftarrow true$;
return;
end
$v.complex \leftarrow false$;
end

Algorithm 11: Calculate Extent and Complex Branch.

```
void searchWithBacktracking():
    Initialize priority queue pq;
    Initialize closed set cs;
    Push initial Multiverse curr onto pq;
    Insert curr into cs;
    while pq not empty do
        Pop a Multiverse from pq and store in curr;
        if curr.stemType = STEM_ONCOMPLEX then
            makeOnComplexBranch(curr.stemAgent, curr, curr.stemTime);
            makeNonComplexBranch(curr.stemAgent, curr, curr.stemTime);
            continue;
        if curr.stemType = STEM_BACKTRACKING then
            makeNonComplexBranch(curr.stemAgent, curr, curr.stemTime);
            continue;
        if curr.stemType = STEM_COMPLEX then
            makeComplexBranch(curr.stemAgent, curr, curr.stemTime);
            continue;
        end
        Search curr for a clash;
        if No clash then return curr;
        Let i, j be the agents invloved in the clash, t be the time of the clash and
          swap be a boolean indicating whether a swap occured;
        makeReasoningBranch(i, curr, t, swap);
        makeReasoningBranch(j, curr, t, swap);
    end
    return no solution;
end
```

Algorithm 12: Main Algorithm loop, with the Backtracking Algorithm.

```
void makeReasoningBranch(i, curr, t, swap):
    Data: Agent: i, Multiverse: curr, Collision Time: t, Swap: swap
    Let v be the agentverse curr.verses[i].verse;
    if (v.complex And v.cb ≤ t) Or Journey(v) < t then
        makeOnComplexBranch(curr.stemAgent, curr, curr.stemTime);
        return;
    end
    backtrack(i, curr, t, swap, LEFT);
    backtrack(i, curr, t, swap, RIGHT);
end
```

Algorithm 13: Case responsible for using the Backtracking Algoritm.

void `makeComplexBranch(`i, $curr$, t`)`**:**
- **Data:** Agent: i, Multiverse: $curr$, Collision Time: t
- Let vu be the verse unit $curr.verses[i]$;
- **foreach** *Time step* $s < t$ **And** $F_s(vu.verse) + 2 \geq F(vu.verse)$ **do**
  - Let r be the maximum relative cost;
  - **foreach** *Direction* d *from* $P_s(v)$ **do**
    - Create a path u diverging from v at time s in direction d. **if** *If the relative cost* $F(u) - F(v) = r$ **then**
      - Let m be the Multiverse with u replacing the verse for agent i in Multiverse $curr$;
      - Insert m into cs;
      - Push m onto pq;
  - **end**
- **end**
- **if** *The stem value* $vu.stem < vu.verse.cost + 2$ **then**
  - Let $stem$ be the Complex stem of $curr$ with the stem of agent i incremented by 1;
  - Push $stem$ onto pq;

end

Algorithm 14: Case responsible for constructing Complex branches.

void `makeNonComplexBranch(`i, m, t`)`**:**
- **Data:** Agent: i, Multiverse: m, Collision Time: t
- Let vu be the VerseUnit $m.verses[i]$;
- Let $next$ be the result of `getNext(`i, $vu.verse$, t, $vu.stem$`)` restricted too the subgraph of the Non-Complex portion of $vu.verse$ extrapolated to the goal;
- **if** $next$ *is not valid* **then**
  - **return**
- **end**
- Let $curr$ be the Multiverse with $next$ replacing the verse for agent i;
- The stem for agent i is reset too $next.cost$;
- **if** cs *does not contain* $curr$ **then**
  - Insert $curr$ into cs;
  - Push $curr$ onto pq;
  - Create a stem $stem$ from m by incrementing the $vu.stem$ of agent i;
  - Push $stem$ onto pq;
- **end**

end

Algorithm 15: Case responsible for solutions along a Non-Complex path.

void `makeOnComplexBranch`(i, m, t)**:**
    Data: Agent: i, Multiverse: m, Collision Time: t
    Let vu be the VerseUnit $m.verses[i]$;
    Let $next$ be the result of `getNext`(i, $vu.verse$, t, $vu.stem$) restricted too after the complex branch point $vu.cb$;
    if $next$ *is not valid* **then**
        return
    end
    Let $curr$ be the Multiverse with $next$ replacing the verse for agent i;
    The stem for agent i is reset too $next.cost$;
    if cs *does not contain* $curr$ **then**
        Insert $curr$ into cs;
        Push $curr$ onto pq;
        Create a stem $stem$ from m by incrementing the $vu.stem$ of agent i;
        Push $stem$ onto pq;
    end
end

Algorithm 16: Case responsible for solutions after a Complex branch.

Appendix C. Tables

Agents	CBS	CISR	Draw
4	19	1981	0
5	38	1961	1
6	70	1928	2
7	102	1895	3
8	169	1824	7
9	282	1707	11
10	365	1619	16
11	523	1438	39
12	626	1305	69
13	711	1116	173
14	803	882	315
15	820	662	518
16	707	478	815

Table C.1: Comparison between CBS and CISR on 8x8 grids.

Agents	CBS	CISR	Draw
4	11	1985	4
5	13	1979	8
6	14	1975	11
7	22	1958	20
8	21	1957	22
9	42	1923	35
10	58	1885	57
11	79	1851	70
12	90	1834	76
13	107	1787	106
14	137	1708	155
15	134	1684	182
16	158	1606	236
17	186	1529	285
18	198	1457	345
19	210	1377	413
20	187	1285	528
21	185	1170	645
22	186	1070	744
23	103	1029	868
24	99	935	966
25	82	815	1103

Table C.2: Comparison between CBS and CISR on 16x16 grids.

Agents	CBS	CISR	Draw
4	22	1971	7
5	13	1976	11
6	17	1971	12
7	12	1961	27
8	8	1960	32
9	14	1957	29
10	13	1941	46
11	20	1912	68
12	24	1875	101
13	19	1868	113
14	21	1880	99
15	19	1837	144
16	43	1795	162
17	36	1788	176
18	32	1741	227
19	29	1689	282
20	44	1643	313
21	45	1607	348
22	43	1596	361
23	37	1558	405
24	43	1494	463
25	46	1440	514
26	43	1433	524
27	35	1323	642
28	38	1300	662
29	26	1279	695
30	22	1198	780
31	20	1151	829
32	22	1108	870
33	16	1065	919
34	7	971	1022
35	10	928	1062

Table C.3: Comparison between CBS and CISR on 32x32 grids.

Map	Agents	CBS	CISR	Draw
brc202d	4	144	1813	43
brc202d	6	184	1592	224
brc202d	8	138	1359	503
brc202d	16	4	495	1501
brc501d	4	3	1980	17
brc501d	6	5	1930	65
brc501d	8	9	1872	119
brc501d	16	2	1226	772
den012d	4	41	1955	4
den012d	6	103	1882	15
den012d	8	115	1804	81
den012d	16	105	1033	862
den520d	4	2	1996	2
den520d	6	5	1987	8
den520d	8	5	1952	43
den520d	16	6	1729	265
hrt201d	4	8	1984	8
hrt201d	6	32	1954	14
hrt201d	8	48	1912	40
hrt201d	16	65	1596	339
lak103d	4	28	1968	4
lak103d	6	98	1877	25
lak103d	8	263	1669	68
lak103d	16	278	519	1203
lak202d	4	10	1976	14
lak202d	6	31	1936	33
lak202d	8	49	1890	61
lak202d	16	131	1427	442
lak303d	4	58	1928	14
lak303d	6	116	1817	67
lak303d	8	122	1625	253
lak303d	16	20	522	1458

Table C.4: Comparison between CBS and CISR on selected maps. (Part 1)

Map	Agents	CBS	CISR	Draw
lak401d	4	117	1863	20
lak401d	6	197	1711	92
lak401d	8	170	1527	303
lak401d	16	18	499	1483
orz900d	4	17	1973	10
orz900d	6	19	1939	42
orz900d	8	16	1890	94
orz900d	16	3	1389	608
orz999d	4	5	1995	0
orz999d	6	9	1976	15
orz999d	8	14	1934	52
orz999d	16	3	1498	499
ost003d	4	15	1976	9
ost003d	6	30	1924	46
ost003d	8	46	1821	133
ost003d	16	10	1054	936
ost102d	4	18	1977	5
ost102d	6	56	1929	15
ost102d	8	120	1847	33
ost102d	16	394	981	625
oth000d	4	95	1890	15
oth000d	6	123	1780	97
oth000d	8	103	1675	222
oth000d	16	19	756	1225
rmtst	4	7	1981	12
rmtst	6	10	1966	24
rmtst	8	22	1944	34
rmtst	16	55	1741	204

Table C.5: Comparison between CBS and CISR on selected maps. (Part 2)

Size	Agents	CIS	CISR	Draw
8x8	5	1408	550	42
8x8	6	1439	521	40
8x8	7	1321	664	15
8x8	8	1176	803	21
8x8	9	1008	948	44
8x8	10	778	1161	61
8x8	11	616	1224	160
8x8	12	458	1280	262
8x8	13	424	1165	411
8x8	14	280	1035	685
16x16	5	1247	722	31
16x16	6	1286	685	29
16x16	7	1224	746	30
16x16	8	1154	819	27
16x16	9	1022	933	45
16x16	10	846	1081	73
16x16	11	683	1222	95
16x16	12	553	1342	105
16x16	13	388	1464	148
16x16	14	324	1475	201
32x32	7	1176	789	35
32x32	8	1125	838	37
32x32	9	1109	856	35
32x32	10	985	966	49
32x32	11	967	962	71
32x32	12	816	1080	104
32x32	13	751	1133	116
32x32	14	645	1247	108
32x32	15	525	1325	150
32x32	16	401	1430	169

Table C.6: Comparison between CIS and CISR on selected agent counts and grids.

Map	Agents	CIS	CISR	Draw
ost102d	4	1158	790	52
ost102d	6	1125	837	38
ost102d	8	870	1070	60
ost102d	16	26	1077	897
lak103d	4	1052	926	22
lak103d	6	837	1093	70
lak103d	8	541	1215	244
lak103d	16	13	565	1422
lak202d	4	1039	945	16
lak202d	6	994	968	38
lak202d	8	832	1090	78
lak202d	16	225	1243	532
lak303d	4	1257	705	38
lak303d	6	1274	608	118
lak303d	8	1069	609	322
lak303d	16	118	411	1471
den012d	4	1118	870	12
den012d	6	1092	858	50
den012d	8	1062	800	138
den012d	16	426	642	932
den520d	4	1202	795	3
den520d	6	1272	717	11
den520d	8	1269	686	45
den520d	16	1002	729	269
lak401d	4	1086	868	46
lak401d	6	1026	831	143
lak401d	8	920	708	372
lak401d	16	172	336	1492
brc501d	4	1153	828	19
brc501d	6	1270	662	68
brc501d	8	1289	584	127
brc501d	16	693	534	773

Table C.7: Comparison between CIS and CISR on selected maps.

Map	CIS	CISR	CBS
bypass	0.000257	0.000537	0.000410
crossroad1	0.070840	0.038931	0.072242
crossroad2	0.000777	0.000985	0.001837
crossroad3	0.003388	0.003674	0.012078
geometry2	0.049006	0.058314	0.016839
outline	0.001155	0.001377	0.001307
outlinegrid	0.003747	0.003621	0.021882
outlinegrid2	100.000000	0.041100	0.075587
outlinegrid3	0.057366	0.004901	0.042139
pass	0.002789	0.012415	0.012094
pass1	0.001182	0.005318	0.005070
pass2	0.002621	0.009488	0.008415
permute1	0.000511	0.000981	0.002367
permute2	0.005454	0.014725	0.035659
permute3	0.145791	0.273409	1.954335
permute4	1.275205	0.845847	11.363638
swaptest	0.000163	0.000517	0.000204
swaptest2	0.005527	0.007893	0.003968
swaptest25	0.083173	0.113126	0.018209
swaptest27	0.055442	0.067693	0.018347
swaptest3	1.756666	2.295569	0.175644
swaptest4	0.006844	0.009989	0.009977

Table C.8: IQM data for bespoke map selection.

Appendix D. Graphs

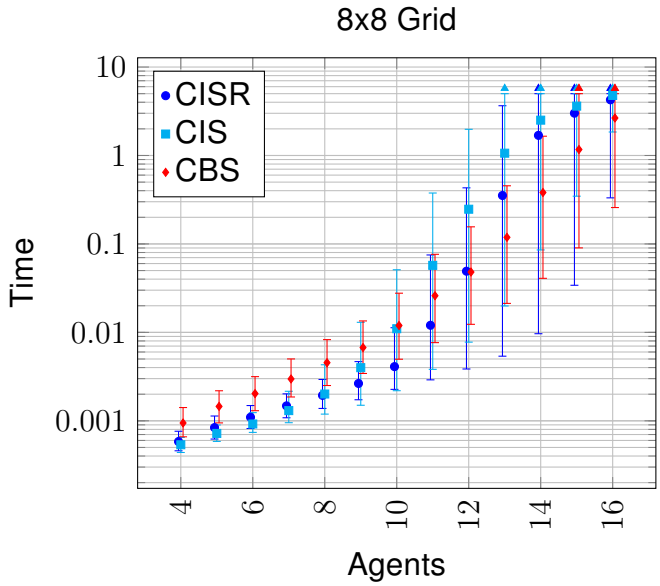


Figure D.1: Interquartile Mean for the 8x8 Grid.

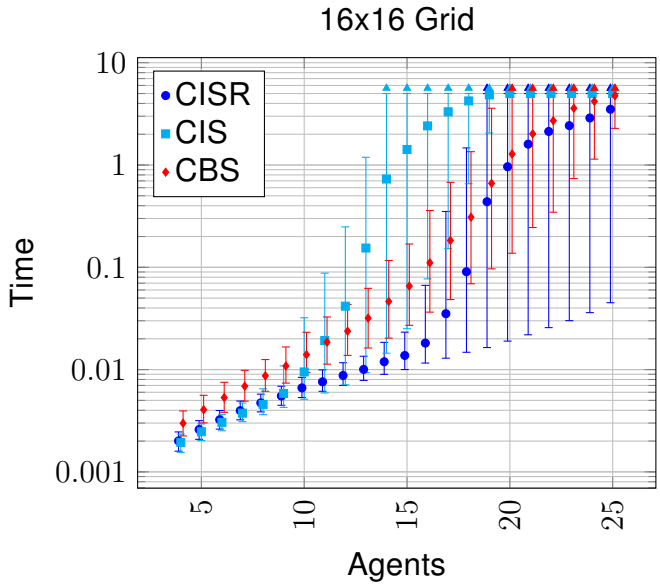


Figure D.2: Interquartile Mean for the 16x16 Grid.

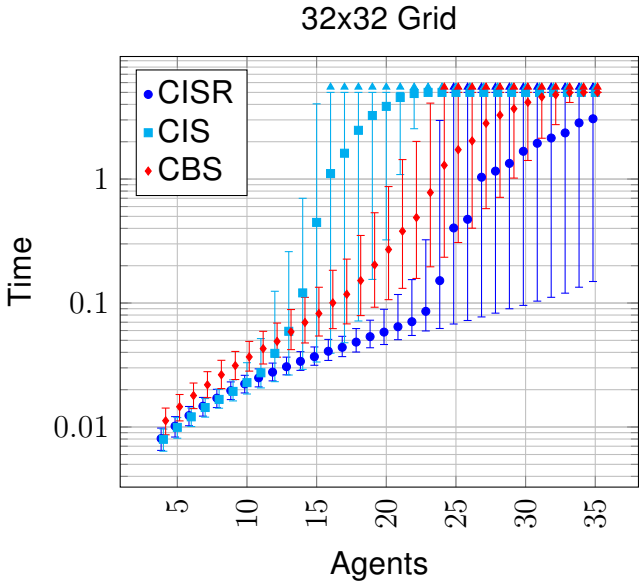


Figure D.3: Interquartile Mean for the 32x32 Grid.

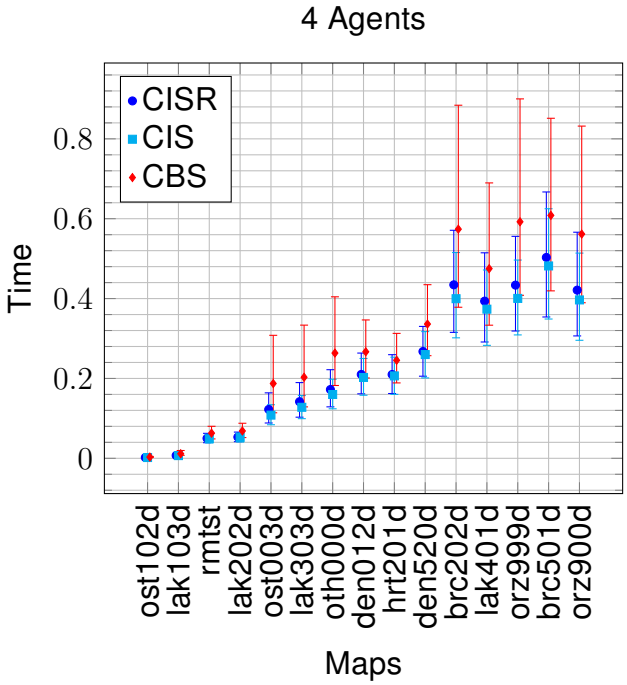


Figure D.4: Interquartile Mean for the 4 agents case.

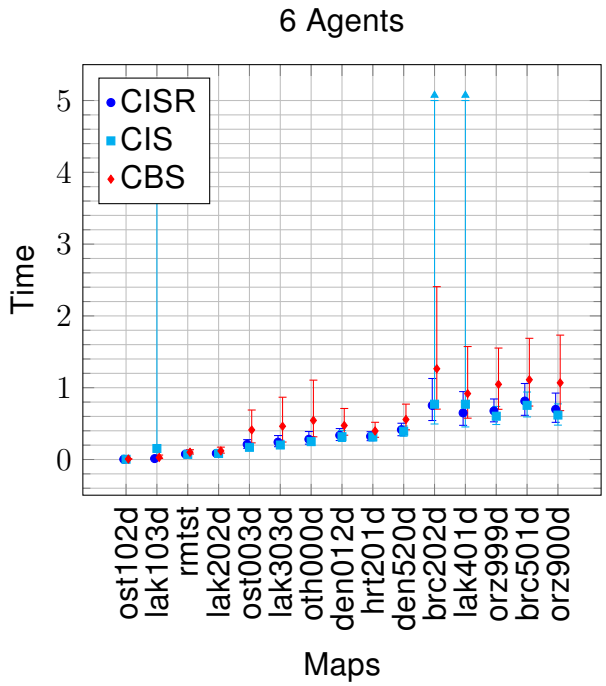


Figure D.5: Interquartile Mean for the 6 agents case.

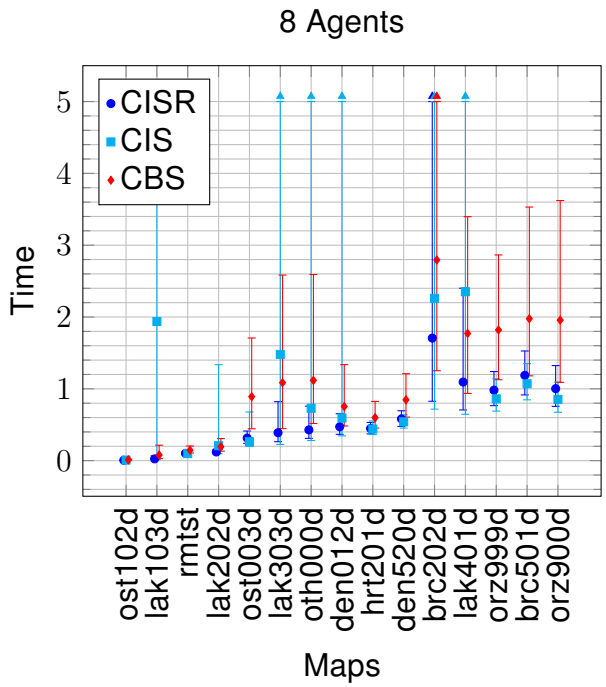


Figure D.6: Interquartile Mean for the 8 agents case.

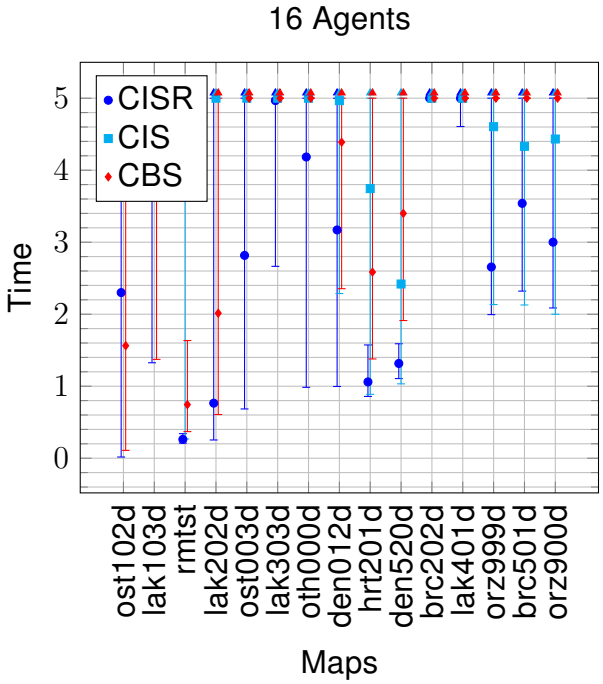


Figure D.7: Interquartile Mean for the 16 agents case.

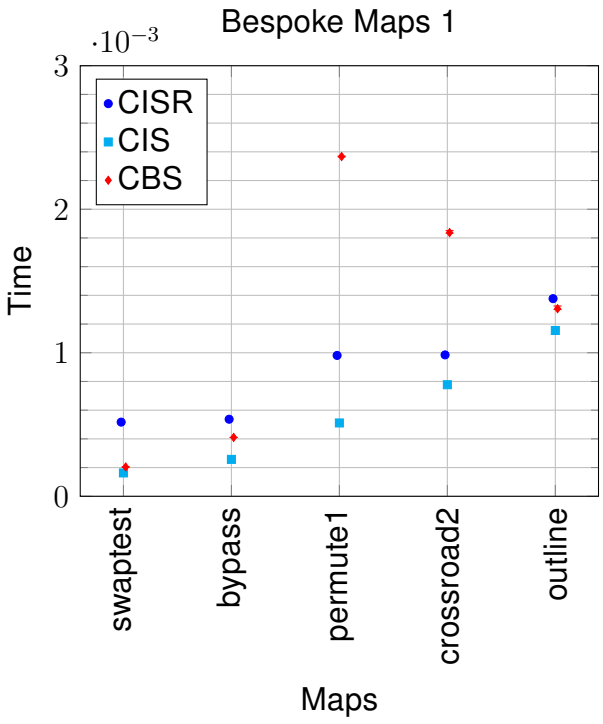


Figure D.8: Interquartile data for bespoke maps.

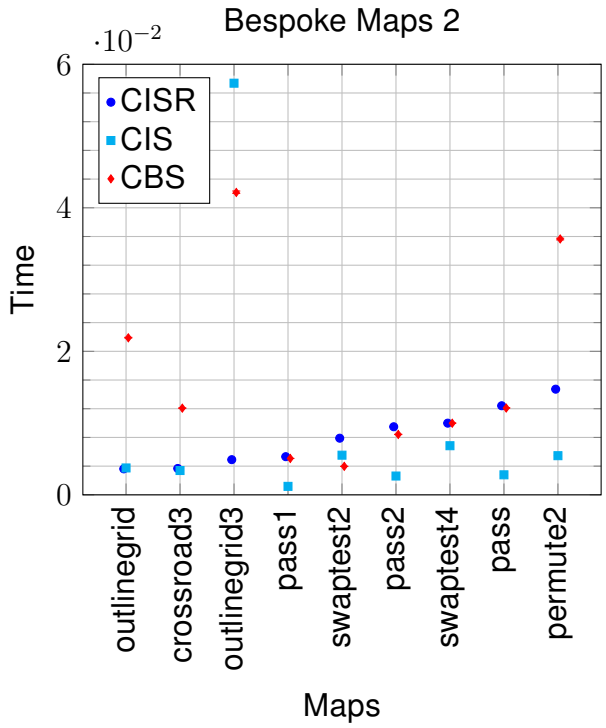


Figure D.9: Interquartile data for bespoke maps.

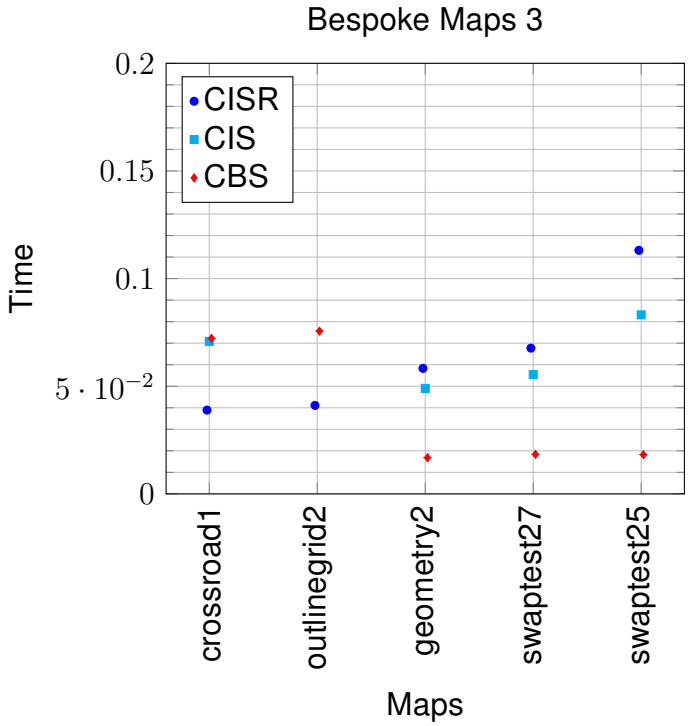


Figure D.10: Interquartile data for bespoke maps.

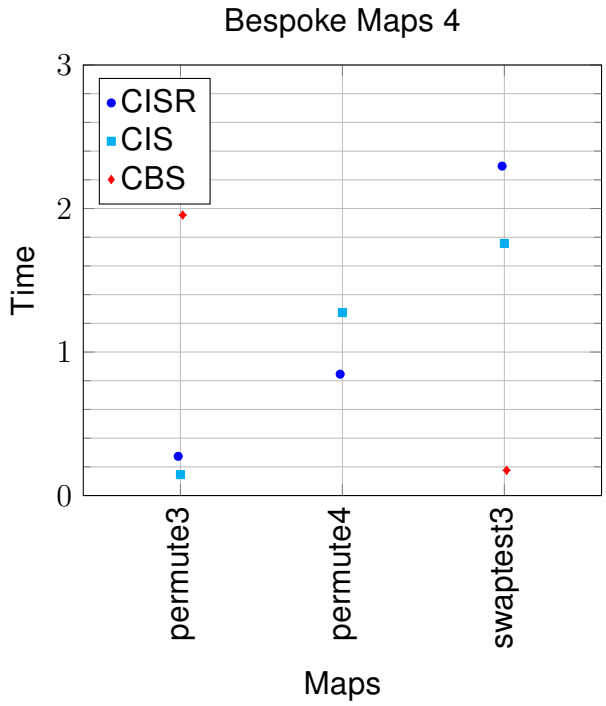


Figure D.11: Interquartile data for bespoke maps.