Advances in Pathfinding Algorithms for Games, Route Planning Software, and Automated Warehouses

Bojie Shen

Doctor of Philosophy

Main supervisor: Assoc. Prof. Muhammad Aamir Cheema
Associate supervisor: Prof. Peter J. Stuckey
Associate supervisor: Dr. Daniel D. Harabor

A Thesis Submitted for the Degree of Doctor of Philosophy at
Monash University in 2023
Faculty of Information Technology

Copyright

Abstract

The pathfinding problem is a fundamental challenge in artificial intelligence, with numerous real-world applications across a variety of environments and domains. Many of these applications involve a large number of users, making it crucial to solve the pathfinding problem efficiently in order to ensure scalability. In this dissertation, we focus on the pathfinding problems in three distinct applications: (i) computer games, where the pathfinding task involves navigating an agent in a Euclidean plane with polygonal obstacles, (ii) route planning software, where the objective is to find an efficient path for a user to travel on a road network or a network that considers varying travel time, and (iii) automated warehouses, where multiple agents need to be coordinated simultaneously. Below, we briefly discuss our motivations and contributions made to each of these applications.

In computer games, agents must navigate in a Euclidean space with polygonal obstacles. Leading works in this area all rely on state-space search to find a solution and that search is often an all-or-nothing affair, meaning that no results are returned until the best solution is found. This behaviour may cause first move lag, where a mobile agent must wait for the search to finish completely before taking even the first step towards its destination. To address this undesirable behaviour, we propose efficient algorithms for both optimal and suboptimal pathfinding. Our first algorithm, End Point Search (EPS), finds the optimal path faster than a range of state-of-the-art pathfinding algorithms in the field. Not only is EPS fast, but it also exhibits strong any-time behaviour, enabling it to find feasible solutions before proving optimality. Our second algorithm, Centroid Path Extraction (CPE), finds suboptimal paths within a fixed bound of the optimal solution. Experiments show that CPE is faster than competitors and computed solutions have better path quality than competing suboptimal algorithms.

When it comes to routing planning, our initial focus is static road networks, where the cost on each edge of the network is modelled as a constant value, representing travel time or distance. The state-of-the-art algorithm in this area is Compressed Path Databases (CPD) a powerful database-driven method that forgoes conventional state-space search and instead extracts shortest paths using precomputed first-move data. Yet, CPDs have two main drawbacks: (i) constructing the database requires an offline all-pairs precompute, which can sometimes be prohibitive and; (ii) extracting a path requires a number of database lookups equal to its number of edges, which can be costly in terms of time. To mitigate the disadvantages of CPDs, we investigate connections with another family of successful, but search-based, speedup techniques called Contraction Hierarchies (CH). This leads us to introduce our new technique, CH-CPD, which combines the strengths of the state-of-the-art algorithms CH and CPD. In a range of experiments on road networks, we show that our new CH-CPD can be built cheaper than conventional CPD, and runs faster than a range of existing algorithms, including CH and CPD. Partial CH-CPD further reduces the preprocessing cost while maintaining the competitive runtime.

To better support more accurate route planning, we also explore the pathfinding problem in time-dependent road networks, where the cost of each edge is determined by a piecewise linear function that varies depending on the time of day. Time-dependent road networks provide a more precise representation of road conditions by considering expected traffic congestion. The current state-of-the-art algorithm in this area is called Time-dependent Contraction Hierarchies (TCH). Although fast and optimal, TCH still suffers from two main drawbacks: (i) the usual query process uses bidirectional Dijkstra search to find the shortest path, which can be time-consuming; and (ii) the TCH is constructed w.r.t. the entire time domain T , which complicates the search process for queries q that start and finish in a smaller time period $T_q \subset T$. To further enhance the performance of TCH, we develop several techniques that improve TCH by making use of time-independent heuristics, which speed up optimal search, and by computing TCHs for different subsets of the time domain, which further reduce the size of the search space. Our results indicate substantial improvements in query times compared to the baseline TCH.

Finally, we focus on automated warehouses, addressing the Multi-Agent Pathfinding (MAPF) problem, which asks us to simultaneously plan collision-free paths for groups of moving agents. Among the leading methods for optimal MAPF is Conflict-Based Search (CBS), an algorithmic family which has received intense attention in recent years and for which large advances in efficiency and effectiveness have been reported. Yet all the recent CBS gains come from reasoning over pairs of agents only. We address this problem by demonstrating how CBS can be further enhanced by reasoning about more than two agents at the same time. Our new reasoning techniques, Cluster Heuristic and ByPass (CHBP), allow us to generate stronger lower bound values for CBS and identify additional bypasses (alternative cost-equivalent paths), which can both reduce the number of nodes expanded by CBS. Our experiments show substantial improvements for CBS, especially on dense maps, and we believe that the use of cluster reasoning opens up a promising new research direction.

Amid the swift advances of automated warehouses, MAPF has been receiving increasing attention in recent years. Many works appear on this topic each year, and a large number of substantial advances and performance improvements have been reported. Yet measuring the overall progress in MAPF is difficult: there are many potential competitors, and the computational burden for comprehensive experimentation is prohibitively large. Moreover, detailed data from past experimentation is usually unavailable. To lower the barrier of entry for new researchers and further promote the study of MAPF, we introduce a set of methodological best practices for experimentation and reporting, along with a variety of visualisation tools. These resources can help the community to establish clear indicators for state-of-the-art MAPF performance and facilitate large-scale comparisons between MAPF solvers. We implement these ideas into a publicly available web platform for the benefit of the community.

Our research has yielded efficient and effective algorithms for solving pathfinding problems in a variety of distinct applications. Advancing the state-of-the-art in the field, we are now able to answer pathfinding queries much faster than before, potentially benefiting not only the applications we targeted but also a broader range of other extended applications. Looking forward, we also suggest several promising directions for future research.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature:

Name: Bojie Shen

Date: 20/04/2023

Publications

Below is the list of publications and manuscripts arising from this thesis.

Published Papers

1. **Bojie Shen**, Muhammad Aamir Cheema, Daniel D. Harabor, and Peter J. Stuckey. "Euclidean Pathfinding with Compressed Path Databases." In Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence (IJCAI), pp. 4229-4235. 2021.

2. **Bojie Shen**, Muhammad Aamir Cheema, Daniel D. Harabor, and Peter J. Stuckey. "Fast Optimal and Bounded Suboptimal Euclidean Pathfinding." Artificial Intelligence (AIJ) 302: 103624. 2022

3. **Bojie Shen**, Muhammad Aamir Cheema, Daniel D. Harabor, and Peter J. Stuckey. "Contracting and Compressing Shortest Path Databases." In Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), vol. 31, pp. 322-330. 2021.

4. **Bojie Shen**, Muhammad Aamir Cheema, Daniel D. Harabor, and Peter J. Stuckey. "Improving Time-Dependent Contraction Hierarchies." In Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), vol. 32, pp. 338-347. 2022.

5. **Bojie Shen**, Zhe Chen, Jiaoyang Li, Muhammad Aamir Cheema, Daniel D. Harabor, and Peter J. Stuckey. "Beyond Pairwise Reasoning in Multi-Agent Path Finding." In Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS), pp. 338-347. 2023.

6. **Bojie Shen**, Zhe Chen, Muhammad Aamir Cheema, Daniel D. Harabor, and Peter J. Stuckey. "Tracking Progress in Multi-Agent Path Finding." arXiv preprint arXiv:2305.08446. 2023.

Acknowledgements

Undertaking this Ph.D. journey has been one of the most challenging and rewarding experiences of my life. It would not have been possible without the unwavering guidance, encouragement, and support of my supervisors, colleges and families.

First and foremost, I would like to express my deepest gratitude to my supervisors, Assoc. Prof. Muhammad Aamir Cheema, Dr. Daniel D. Harabor and Prof. Peter J. Stuckey for their invaluable guidance, support, and mentorship throughout my research. In particular, I would like to extend my heartfelt thanks to Aamir for providing me with the scholarship from his funding, which has been instrumental in supporting my Ph.D. studies. I am deeply grateful to Daniel for introducing me to the exciting world of "Pathfinding", which has broadened my horizons and sparked my passion for research in this field. Lastly, I would like to express my utmost appreciation to Peter for teaching me numerous brilliant optimisation tricks that have greatly enhanced the quality and impact of my research work. I am fortunate to have supervisors who are deeply involved in each of my research projects and always available to offer their guidance and assistance. Moreover, I greatly appreciate the freedom they give me to explore various problems that I am interested in. To all of my supervisors, thank you for the tremendous effort you have invested in me, which has led me to where I am today.

I am also grateful to my thesis committee members, Dr. Li Li, Dr. Chakkrit Tantithamthavorn, Assoc. Prof. Michael Wybrow, Dr. Graeme Gange, and Dr. Pierre Le Bodic. Although my thesis committee members underwent some changes, I am truly thankful for their constructive feedback, insightful suggestions, and encouragement during this process. Their expertise and dedication have played a significant role in shaping my research. In addition, I would like to thank my collaborators, Zhe Chen and Dr. Jiaoyang Li. Thank you for introducing me to the multi-agent pathfinding problem and for the immense hard work and dedication you both put into my last research project. Without your invaluable assistance and guidance, I would not have been able to complete my research on this topic in the last year of my candidature. Your contributions have been critical in shaping the outcomes of my work, and I am truly grateful for your support and collaboration.

I would like to extend my appreciation to my fellow researchers and colleagues at OPTIMA lab, who have provided a stimulating and supportive environment for intellectual growth. Special thanks go to Shizhe Zhao, Yue Zhang, and Jayden Filippi for engaging in many insightful discussions with me on the topic of pathfinding. I would also like to thank Hendrik Bierlee, Kelvin Davis, Jinqiang Yu, Jiarong Fan, and other members of the lab for the many enjoyable drinks we shared at Pixel Bar and for the fascinating micro talks we had on Fridays. Furthermore, I would like to express my appreciation to my colleagues at the Urban Computing Lab,

including Faisal Alam, Abdurrahman Beg, Mohammad Sadegh Aslanpour, Xian-Long Lee, and other members of the lab for their continuous support and collaboration. I am grateful for the weekly reading group sessions organised by the lab, which have significantly broadened my knowledge and understanding of the latest research in the field. I would also like to thank them for the memorable party they organised for me on the occasion of my final milestone. Your support and encouragement have played a significant role in motivating me to complete my research journey successfully.

I would also like to express my gratitude to a few of the Ph.D. candidates that I informally co-supervised, including Jinchun Du, Ahmed Fahmin, Faiza Babakano, and Abdallah Abuaisha. I am thankful for the opportunity to be a part of your research projects, and for the valuable learning experiences that I gained from our collaboration. I am deeply appreciative of my supervisors and Dr. Adel Nadjaran Toosi, for providing me with the opportunity to join these research projects and for their continuous support and mentorship. To all of you, I extend my best wishes for your future endeavours. May you complete your Ph.D. studies successfully and continue to pursue your research with passion and dedication. I hope that you achieve great success in both your professional and personal life.

In closing, I want to express my deep gratitude to my parents and my partner for their unending love, unconditional support, and unyielding encouragement throughout my Ph.D. journey. Their belief in my abilities has been a constant source of strength and motivation, even during the most difficult times. I could not have come this far without them, and for that, I am truly grateful.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Pathfinding queries are one of the most ubiquitous practical uses of computing, which involve navigating an agent or groups of agents from the source locations to their destination locations in a shared environment. These queries are widely used in various real-world applications, including path planning in video games [1], route planning in GPS navigation software [2], multi-robot coordination in automated warehouses [3], drone swarm coordination [4], location-based services in indoor spaces [5], relationship reasoning in social networks [6], and so on. Due to the broad range of practical applications and interests, solution methods that tackle the pathfinding queries have been extensively studied by various research communities. These efforts have led to numerous advances and optimisations over the past decades. The majority of attention is on finding paths that: (i) meet domain-specific constraints (e.g., avoiding obstacles, avoiding collisions with other agents, etc.); (ii) are of reasonable quality (e.g., minimal distance, minimal travel time, etc.); and (iii) as efficient as possible in terms of response time and computational resources (e.g., memory and preprocessing time). These criteria play a crucial role in making pathfinding queries more practical and useful in a wide range of applications.

In this dissertation, we investigate pathfinding queries in the context of three prevalent real-world applications: computer games, route planning software and automated warehouses. Section 1.1 elaborates on the applications of pathfinding queries in different domains and our motivation for studying them. Section 1.2 highlights our contributions, including novel algorithms for improving the query performance and reducing preprocessing cost. Lastly, Section 1.3 describes the organisation of this dissertation.

1.1 Motivation

In this section, we provide a summary of our target application settings and the limitations of existing works.

1.1.1 Pathfinding in Computer Games

In computer games, the environment is typically represented as a two-dimensional plane which contains polygonal obstacles that must be avoided by virtual characters. Figure 1.1 shows a typical example from one of the most popular online games, League of Legends (LoL)[1]. The obstacles here are rocks and buildings. During a game, players navigate their characters around the map, which is facilitated by solving a range of pathfinding queries. Pathfinding is also required as part of higher-level decision-making for non-player characters (NPCs), such as build planning and combat planning. In each of these cases, it is desirable to find direct and detour-free paths because virtual characters need to appear intelligent to human observers. Yet this apparently simple task is surprisingly complicated because games occur in real-time, they often involve a large number of player and non-player characters, and because there are only limited CPU and memory resources available for pathfinding (the bulk of computational resources are assigned to other higher-priority tasks such as graphics, networking and so on [7, 8]). Effective algorithms in this space are ones that compute paths which are: (i) obstacle avoiding; (ii) as short as possible; and (iii) computed as fast as possible.

A wide variety of algorithms have been proposed to quickly find paths in game settings. Many of these works model the Euclidean environment in an approximate way using a grid map [9–12]. Grids are simple to build and fast to search. Yet the resulting grid paths are often longer than necessary and agents following grid paths appear unrealistic due to limited movement options (they can only turn at 45 or 90 degrees). An alternative and more precise way of representing a map is to reason over the Euclidean space directly. Paths computed with these types of models are short and agents following these paths appear realistic. Yet there are only limited works in this area [13–15]. All rely on state-space search to find a solution and that search is often an all-or-nothing affair; i.e. until a best solution is found, nothing is returned. This behaviour may be undesirable as it introduces the potential for so-called *first move lag*, where a mobile

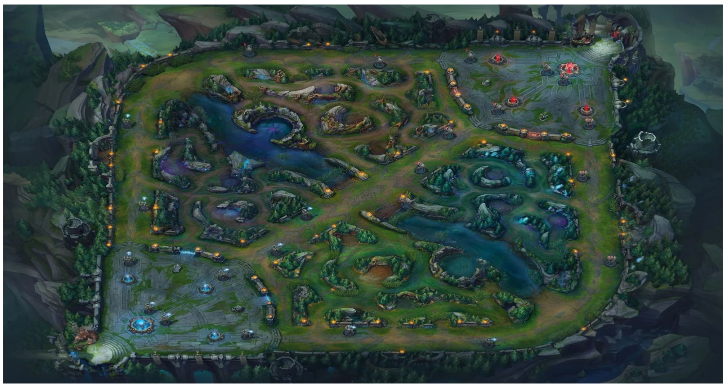


Figure 1.1: An example of online game (League of Legends), where the game engine needs to navigate the player's character by avoiding the obstacles, e.g., rocks, buildings etc.

[1] Downloaded from https://leagueoflegends.com/en-us/

agent must wait for the search to finish completely before it can take even a first step toward its destination. Motivated by these limitations of the existing research, we identify the following research challenge that we set out to address in this dissertation:

Challenge #1. In a Euclidean space that models the environment precisely, can we design an efficient pathfinding algorithm that does not only find the shortest path, but also mitigates the first move lag?

1.1.2 Pathfinding in Route Planning Software

Route planning software, of the type used for in-car navigation, is among the most ubiquitous applications of AI. Here, the environment is modelled as a graph, with each node representing a road intersection and each edge representing a traversable road segment between two adjacent nodes. Figure 1.2 shows an example of route planning from the Monash Clayton campus to the Caulfield campus using Google Maps[2]; the system-recommended path is coloured in blue, and other alternative paths are coloured in grey. From the users' point of view, various types of paths (e.g. shortest distance, fastest travel time, etc.) may be preferred for their daily travel, but the main objective is to make the journey as efficient or economical as possible. From the system perspective, each path recommendation requires a pathfinding query on a large scale road network (e.g., city sized). Every day, millions of requests that need to be processed, making it a challenging task to provide high-quality paths that meet user criteria while maintaining fast response times. Effective algorithms for route planning are ones that: (i) find a feasible path from each start location to each intended destination on the graph; (ii) minimise the user's objective; and (iii) solve each problem as quickly as possible.

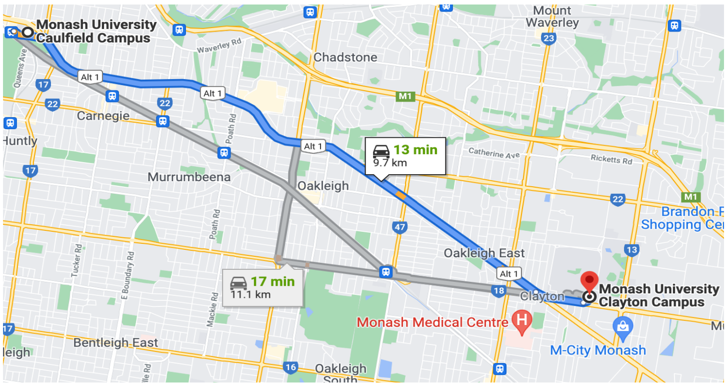


FIGURE 1.2: Route planning on Google Maps, where the user needs to find the fastest path from Monash Clayton campus to Caulfield campus

[2] Downloaded from https://maps.google.com

Static Road Networks is a popular type of route planning model where the cost to travel on each road segment is fixed and unchanging. Many search-based algorithms have been proposed to solve these types of problems. These works tend to reduce the search space by using effective heuristics [16, 17]; bidirectional search [18, 19]; edge labelling [20, 21]; and the idea of shortcuts [22–24]. Yet these algorithms still require substantial search time to find optimal paths, when deployed at scale [25]. Therefore, the state-of-the-art pathfinding algorithms for road networks are often oracle-based approaches [26, 27], a family of techniques that forgo conventional state-space search and instead *extract* the shortest paths using precomputed data. Although the oracle-based approach is fast at finding optimal paths, it requires substantial time and memory to build and store the oracle. These preprocessing costs increase in the size of the input graph, which can be prohibitive in some cases, especially for large input graphs. Motivated by this, we post the following challenge:

Challenge #2**. **In a static road network, can we reduce the preprocessing costs of oracle-based algorithms while still preserving their efficient query performance?

Time-dependent Road Networks is a more sophisticated type of route planning model where travel time on the network are allowed to vary during the day. These models can more accurately represented actual road conditions, since traffic congestion (among other issues) can affect travel times throughout the day. A variety of existing works for static road networks can be extended to the time-dependent scenario, e.g., the idea of heuristic and bidirectional search are extended by [28–30]; edge labelling and shortcuts are generalised by [30–32]. However, unlike the static road network, these extended approaches need to handle the time-dependent data for the entire time domain (usually 24 hours) in order to answer any queries issued. Due to the large period of time domain and complex travel time function used to model changing costs, these approaches often compute solutions much more slowly, and they require substantial time and memory to construct auxiliary data structures that cover the entire time domain. Therefore, we post the challenge below:

Challenge #3**. **In a time-dependent road network, can we design pathfinding algorithms that manage the time-dependent data more efficiently to further accelerate the pathfinding algorithm?

1.1.3 Pathfinding in Automated Warehouses

In automated warehouses, hundreds of robots work together to pick and deliver items simultaneously. To facilitate coordination and synchronisation the problem is often discretised: the environment is mapped onto a 4-connected grid, time is discretised into unit-sized steps and during each timestep robots can only move to an adjacent grid node or wait in their current location. An example of this setup can be seen in Amazon's automated warehouse[3], where each blue driving unit (robot) has been assigned multiple delivery tasks (refer to Figure 1.3). Each individual robot aims to complete its task as quickly as possible, and therefore prefers the shortest path possible. But the robots must also work together, such that there is no collision between robots during the execution of their tasks. A variety of additional considerations appear in such setups. For example, thousands of orders are received daily each of which is further broken down into multiple delivery tasks for robots; the system also needs to continuously synchronise the robots, correct their location, and re-plan their paths; all of this must also occur in real-time. Yet the core challenge can be stated much more simply and abstractly: find a set of paths to move each robot/agent from a fixed source to fixed destination, all while ensuring that: (i) no collisions occur with other agents or obstacles; (ii) the sum of individual path costs is minimised; and (iii) the entire plan is found as quickly as possible. We refer to this type of pathfinding query as classic Multi-Agent PathFinding (MAPF) [3].

Conflict-based Search (CBS) [33] is a state-of-the-art algorithm for MAPF, which can be understood as a best-first search algorithm that routes each agent independently and then resolves conflicts afterwards. In recent years, there has been massive advances in the efficiency and scalability of CBS. These gains have been achieved by: (i) taking into account symmetries that result in the conflicts between two agents [34, 35]; (ii) generating complex admissible heuristics [36, 37]; and (iii) introducing *bypasses* [38], to reduce the number of subproblems that CBS must tackle. Yet CBS timeout failures, even on modest size problems with dozens of agents, are not uncommon. Thus far, the CBS algorithm only reasons about incompatibility between at most two agents at a time. It lazily detects conflicts between *pairs* of agents, resolves those conflicts by adding *pairwise* constraints, and generates heuristics by combining information about the interactions of *pairs* of agents. How to reason with more than two agents

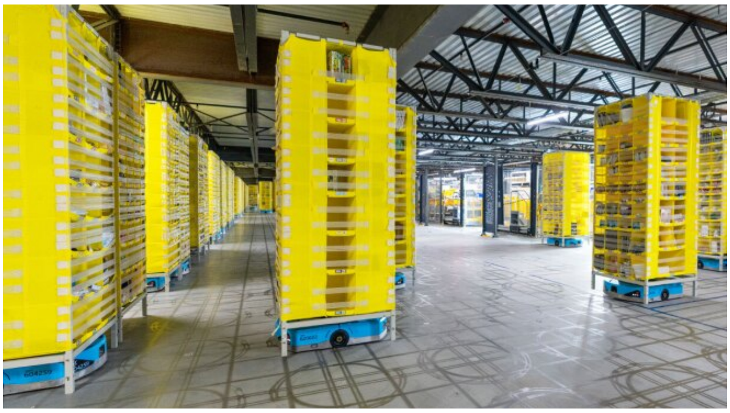


FIGURE 1.3: An example of the automated warehouse in Amazon, where each mobile robot needs to deliver items and coordinate with each other.

[3] Downloaded from https://www.aboutamazon.com/news/operations/10-years-of-amazon-robotics-how-robots-help-sort-packages-move-product-and-improve-safety

at a time is a challenging opening question and is necessary for the CBS algorithm to improve further, thus we identify the following challenge:

Challenge #4**. **In the classic MAPF problems, can we design new algorithmic techniques to capture the incompatibility of more than two agents and utilise this information to further improve the performance of the state-of-the-art optimal MAPF algorithm CBS?

Tracking Progress. As industrial interest continues to grow, the number of publications on the topic of MAPF has exploded [34, 39–43]. Many works now appear, across many different venues, and there have been substantial performance improvements. To track progress in the area, the community has developed a set of standardised MAPF benchmarks [3], which cover a variety of popular application domains and synthetic/pathological test cases. In total, there are more than 1.5 million standard instances with up to thousands of moving agents per instance. Unfortunately, the computational burden associated with running this benchmark is large, which means that most researchers attempt to solve only a limited subset of instances and then only compare against a limited subset of potential competitors. Published works typically only include headline results, such as success rates and total problems solved, they do not mention which specific problems were solved, which were closed, and where the remaining gaps are. Supplementary data, such as concrete plans and best-known bounds, which can allow other researchers to verify claims and build on established results, are seldom available. Thus, despite notable advances, and despite the availability of benchmark problem sets, we do not currently have a clear picture of overall progress in MAPF. We thus identify the following challenge:

Challenge #5**. **In the classic MAPF problems, can we design tools and methodologies to track the progress of MAPF algorithms, lowering the barrier to entry for new researchers and advancing the study of MAPF?

1.2 Contributions

In this section, we outline the contributions made by this dissertation to each challenge of pathfinding problem, including the associated research publications.

1.2.1 Fast Optimal and Bounded Suboptimal Euclidean Pathfinding

To tackle **Challenge #1**, we first develop an optimal Euclidean pathfinding algorithm, End Point Search (EPS), which mitigates first move lag using anytime behaviours [44]. i.e., we aim to compute "good" solutions quickly and we guarantee to return optimal solutions eventually, given sufficient time. In a range of experiments and empirical comparisons we show that: (i) the auxiliary data structures required by EPS are cheap to build and store; (ii) for optimal search, the new algorithm is faster than a range of recent pathfinding algorithms, with speedups ranging from several factors to over one order of magnitude; (iii) for anytime search, where feasible solutions are needed fast, we report even better performance. Our second strategy to address the first move lag is to develop an ultra-fast bounded suboptimal pathfinding algorithm called Centroid Path Extraction (CPE). CPE is completely search-free, simultaneously fast, and returns a path within a fixed bound of the optimal solution. In a range of empirical results, we show that: (i) our approach outperforms a range of optimal and suboptimal pathfinding algorithms proposed in the literature; (ii) our approach demonstrates excellent path quality, better than all existing suboptimal pathfinding algorithms; and (iii) the approach offers flexibility by allowing a trade-off between the preprocessing cost (space and time) and the suboptimality bound.

The optimal algorithm EPS [45] was published in the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI 2020). The suboptimal algorithm CPE [46] was published in the Artificial Intelligence Journal (AIJ 2022) as an extension of our IJCAI 2020 paper.

1.2.2 Contracting and Compressing Shortest Path Databases

To tackle **Challenge #2**, we propose a novel oracle, called CH-CPD, that can efficiently answer the shortest path query in a static road network. CH-CPD mitigates the disadvantages of the leading oracle approach, Compressed Path Databases (CPD), by investigating connections with another family of successful search-based speedup techniques called Contraction Hierarchies (CH). While CH-CPD extracts the shortest path fast, we further propose more advanced techniques to improve the preprocessing time, and a partial CH-CPD search that allows users to trade off between memory cost and query performance. In a range of experiments on road networks, we show that CH-CPD substantially improves on conventional CPDs in terms of preprocessing costs and online performance. We also report convincing query time improvements against a range of methods from the recent literature.

The algorithm CH-CPD [47] was published in the Thirty-First International Conference on Automated Planning and Scheduling (ICAPS 2021).

1.2.3 Improving Time-Dependent Contraction Hierarchies

To tackle **Challenge #3**, we focus on improving the state-of-the-art algorithm, Time dependent Contraction Hierarchies (TCH), which aims to compute the optimal path in a time-dependent road network. Specially, we adapt two admissible lower-bounding functions: landmarks [16] and

CPD heuristics [48]. Both heuristics only require auxiliary data structures built based on the free flow cost, which does not increase in size with the addition of time information. Additionally, we introduce two algorithms called Single-layer TCH (STCH) and Multi-layer TCH (MTCH) that efficiently handle time-dependent data by constructing a series of smaller TCHs, each of which concentrates on a specific part of the time domain T. By choosing the appropriate TCH for each query $q \in T$, we can retain the optimality guarantees of the original algorithm while substantially improving search performance. We evaluate our proposed algorithms on a range of time-dependent road networks, including real-world as well as synthetic datasets. Results show substantial improvement over the baseline TCH method.

This research [49] was published in the Thirty-Second International Conference on Automated Planning and Scheduling (ICAPS 2022).

1.2.4 Beyond Pairwise Reasoning in Multi-Agent Pathfinding

To tackle **Challenge #4**, we focus on improving a leading optimal MAPF algorithm called Conflict-based Search (CBS). Intensively studied, all the recent CBS gains come from reasoning over pairs of agents only, we propose a novel algorithm called Cluster Heuristic and Bypasses (CHBP), which overcomes this limitation by extending CBS heuristics to reason about more than two agents at every node. CHBP does this by exploiting mutex propagation [35], a successful pairwise reasoning technique, which we extend to clusters of more than two agents. CHBP derives stronger bounds for CBS and also generate new kinds of bypasses, where the assigned paths of some agents are replaced to reduce the number of conflicts. In a range of empirical results, CHBP shows substantial improvements for CBS, especially on dense maps.

This research [50] was published in the Thirty-Thrid International Conference on Automated Planning and Scheduling (ICAPS 2023).

1.2.5 Tracking Progress in Multi-Agent Pathfinding

To tackle **Challenge #5**, we introduce a new set of methodological and visualisation tools to facilitate comparisons between a wide range of MAPF methods. We then undertake a large set of experiments, with several currently leading optimal and suboptimal solvers, in an attempt to map the current Pareto frontier. Finally, we propose a new online platform for the MAPF community to track and validate further gains and to improve visibility for and access to existing solvers. We believe that these proposals can help identify the main strengths of existing research and the remaining challenges in the area. They can also be used to track progress on those challenges over time. Finally, we believe that these proposals can help to lower the barrier of entry for new research on the topic of MAPF.

This research [51] was presented as a system demonstration at the Thirty-Thrid International Conference on Automated Planning and Scheduling (ICAPS 2023). The demo can be accessed at the following link: https://icaps23.icaps-conference.org/program/demos. A full-length manuscript is also available at: https://arxiv.org/abs/2305.08446.

1.3 Thesis Organization

This dissertation is organised as follows:

- Chapter 2 reviews the literature that related to pathfinding problem.
- Chapter 3 presents our techniques [45, 46] to efficiently find optimal and suboptimal obstacle-avoiding path in a Euclidean plane.
- Chapter 4 covers our efficient oracle-based approach [47] for retrieving the shortest path in a static road network.
- Chapter 5 describes our enhancement techniques [49] on the state-of-the-art algorithms, TCH, for finding the shortest path in a time-dependent road network.
- Chapter 6 shows our novel cluster reasoning techniques [50] to enhances the performance of the CBS algorithm for solving the classic MAPF problem.
- Chapter 7 describes our proposed platform [51] for continuously monitoring the progress of MAPF algorithms.
- Chapter 8 concludes our research and discusses future works.

Chapter 2

Literature Review

In this chapter, we provide a brief overview of the related work for pathfinding queries in each application domain. More specifically, we provide the related work for pathfinding queries in Euclidean space in Section 2.1. Following by the literature overview of pathfinding queries in road networks in Section 2.2. Section 2.2.4 further discusses the existing work for pathfinding queries in time-dependent road networks. Finally, we present the related work for classic multi-agent pathfinding in Section 2.3.

2.1 Pathfinding in Euclidean Space

In the Euclidean pathfinding problem, we are asked to find point-to-point paths in a continuous 2D workspace which contains polygonal obstacles in fixed positions. Any non-obstacle point from the workspace is a potential source (s) or destination (d) position and the objective is to find an obstacle avoiding, distance minimum path, between pairs of points that are a priori unknown. We next define some necessary terminology.

A polygon is a closed set of edges and a set of points each called a vertex. Each edge is a contiguous interval between two different vertices (i.e., $e = [v_i,v_j]$), where v_i and v_j are the closed ends of e. Polygons can overlap, but only if they share a common edge or vertex. A convex polygon is a polygon where every line drawn between points in the polygon remains within the polygon. Two points are visible if there exists a straight line between this pair that does not intersect with any point from the interior of a polygon. We suppose that a mobile point-sized agent can directly travel between any pair of co-visible points. A path is a sequence of points $P = \langle\ p_1,p_2, \cdots, p_k\ \rangle$ such that $\forall\ p_i,\ p_{i+1} \in P$, p_i and p_{i+1} are co-visible. The cost of a path P is the cumulative Euclidean distance between every successive pair of points; i.e., $\Sigma|P| = \sum_{i=1}^{k-1} ed(p_i, p_{i+1})$ where $ed(p, p')$ is the Euclidean (straight line) distance between p and p'. A path is optimal if its cost is minimum among all paths between its source and destination.

A vertex of a polygon is called a convex vertex if any line between two points "near" the corner stays within the polygon. More formally, if we form the triangle of the vertex with its two neighbours on the polygon, then the triangle is within the polygon. For a path P to be optimal in a Euclidean pathfinding problem with polygonal obstacles, $\forall\, p_i \in P$ except for source and destination, p_i must be a convex vertex [15, 52]. A vertex is a dead-end vertex if it never occurs on an optimal path, unless it is the start or end of the path.

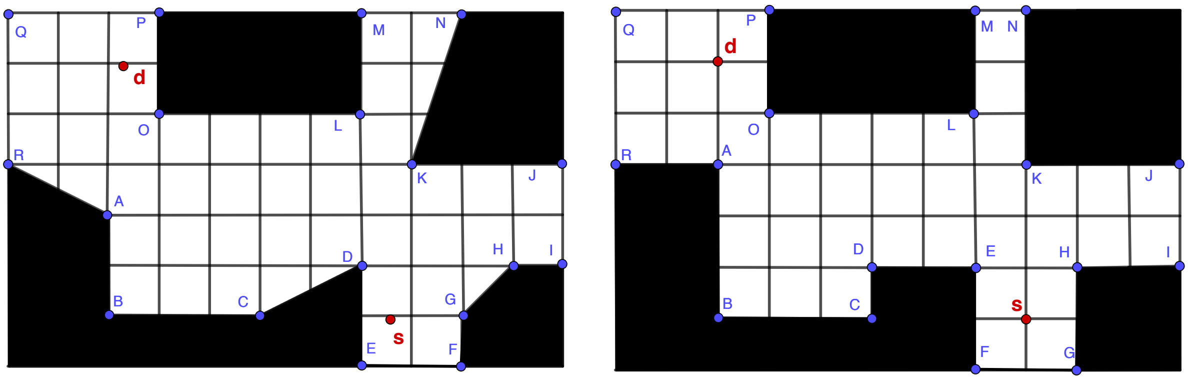
In this section, we categorise related works into four categories: grid-based pathfinding; any-angle pathfinding; visibility graph-based path planning; and mesh-based path planning. Grid-based pathfinding and any-angle pathfinding are designed for grid maps, which involve approximating the environment into a grid of cells. This can result in suboptimal and unrealistic paths. On the other hand, visibility graph-based path planning and mesh-based path planning operate in exact Euclidean space, allowing for a precise representation of the environment and finding the shortest Euclidean path. For each category, we give an overview of existing works.

2.1.1 Grid-based Pathfinding

As mentioned earlier, a grid map is a simple way to represent a two-dimensional space, and the model is commonly used in computer games and robotics. Creating a grid map is a two-step process: (i) Discretise the area into a set of fixed-resolution grids, (ii) Mark each grid as either blocked (if it overlaps with an obstacle) or unblocked (if it's traversable). Figure 2.1 shows an example of creating a grid map, where the Euclidean plane (left) is discretised into a grid map (right). Since each grid is either a block or unblocked cell, the entire grid map can be represented as a Boolean array with linear space cost based on the number of cells. In grid-based pathfinding, an agent is limited to moving only to the adjacent cells in a four-connected grid map. Additionally, it can move diagonally in an eight-connected map. The problem of finding the shortest grid path has been well-researched and can be efficiently solved using various algorithms, many of which instantiate the A* search [17]. We give a brief explanation:

The A* algorithm is one of the most famous algorithms in pathfinding. This algorithm combines the best of both Dijkstra's algorithm [53] and heuristics to determine the optimal path in a grid map. In general, the A* algorithm contains three key components as follows:

- **Search Nodes & Successors:** A search node in the A* algorithm represents the current node being searched. For each search node, the algorithm maintains: (i) a g-value, a cost-incurred value that represents the lowest cost from the source to the node, and (ii) predecessor nodes, a reference to the parent node that allows easy extraction of the path. The successors of a search node are the next set of nodes that can be reached from it. In grid maps, the adjacent grid vertices are considered as the successors of the search node.



(i)

(ii)

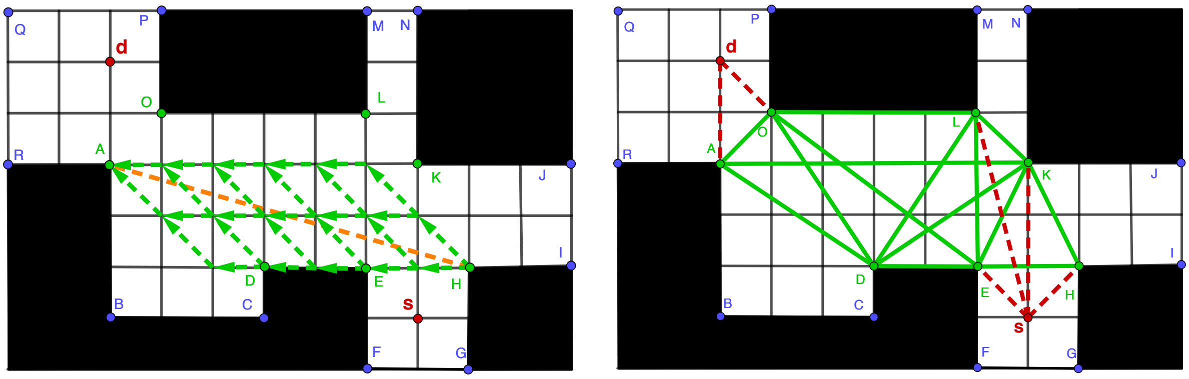
Figure 2.1: An example of creating a grid map, where (i) shows the polygon-shaped obstacles before converting to grid cells and (ii) shows the resultant grid map after discretisation. The source (s) and destination (d) of a shortest path query are shown in red.

- **Evaluation Function:** The A* algorithm uses an evaluation function f to determine the priority of search nodes. It is calculated as the sum of g-value and h-value, where the h-value is obtained from a heuristic function. Heuristics estimate the cost from the current node to the destination and are important for the algorithm's efficiency. Simple heuristics, such as Euclidean, Manhattan, and Octile, often appear as baselines in the literature.

- **Open and Closed Lists:** The A* algorithm uses two data structures to prioritise the expansion of search nodes: the OPEN list and the CLOSED list. The OPEN list contains the search nodes waiting to be expanded, prioritised by their f-value, while the CLOSED list contains nodes that have already been expanded, avoiding repetition

The A* algorithm searches in a best-first manner by expanding the node with the lowest f-value from the OPEN list. When generating successors, it filters out successors already in CLOSED list and discards successors with higher g-value than the recorded minimum g-value for each vertex. The rest of the successors become new search nodes, with updated g and f values, and are added to the OPEN list. The search ends when the OPEN list is empty or the destination is reached.

Many existing works enhance the A* algorithm from various perspectives. These include enhanced heuristics, such as the differential heuristic [54], which improves distance estimation through the use of the triangle inequality, and the fast map heuristic [55], which calculates a more accurate estimation by utilising a popular dimensionality reduction techniques from machine learning. Additionally, there are symmetry breaking techniques like Jump Point Search (JPS) [9, 56], which reduce the branching factors of the search by selectively expanding only a distinguished set of "jump points" found by scanning ahead grid-based maps. There are also



(i)

(ii)

FIGURE 2.2: (i) shows a pair of h-reachable subgoals `H` and `A` in a grid map, the dashed arrows indicate the shortest octile distance paths from `H` to `A`; (ii) shows an example of subgoal graph considering the subgoals placed on convex vertices of obstacles.

many hierarchical pathfinding algorithms [52, 57] that speed up the A* search by limiting the search to only traverse between different levels of hierarchies.

2.1.2 Any-angle Pathfinding

Typically, grid-based pathfinding results in suboptimal and unrealistic paths as movements are only allowed in certain directions (i.e., horizontal and vertical in a four-connected grid, or in addition diagonal movement in an eight-connected grid). Although the pathfinding process still takes place on a grid map, any-angle pathfinding avoids this restriction by allowing movement in any direction and, consequently, finds paths that are substantially shorter.

The state-of-the-art any-angle pathfinder, ANYA [58], is a fast and online pathfinding algorithm that finds the optimal any-angle path. Anya is an interval-based searching algorithm that instantiates A*. It scans the grid row by row and finds an optimal path by expanding the most promising intervals in the OPEN list. Later on, ANYA has also been extended to Polyanya in order to find the shortest path in Euclidean space, we explain the detail of Polyanya in later of Section 2.1.4. Apart from the optimal algorithm Anya, there is also a range of algorithms that find suboptimal any-angle paths.

The N-level subgoal graph [59] is a state-of-the-art suboptimal any-angle algorithm. The subgoal graph [52] is constructed by placing subgoals on the convex corners of obstacles. Instead of connecting each pair of subgoals that are visible to each other, the subgoal graph only connects a small subset of subgoals that are *direct-h-reachable*. Two subgoals are *h-reachable* if there exists a traversal path equal to the octile distance between them, and they are *direct-h-reachable* iff all shortest trajectories (octile distance paths) are traversable with no subgoals between them. Figure 2.2 (i) shows an example where the subgoal `A` and `H` are *h-reachable*, but not *directly*

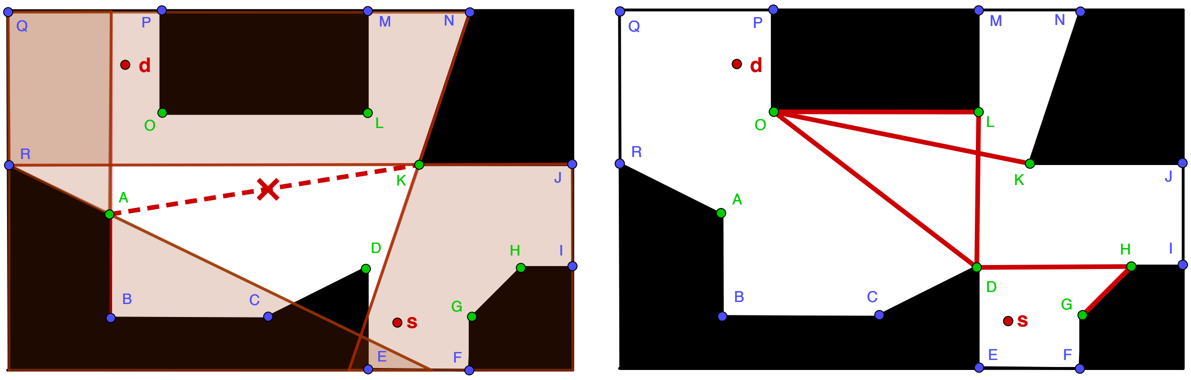
h-reachable as there are other subgoals present in between (e.g. `D` and `E`). Figure 2.2 (ii) shows a complete subgoal graph on the same grid map. The subgoal graph was originally designed to determine the optimal grid path, therefore only *directly h-reachable* edges are added to the graph to guarantee that each edge represents a unique grid path. However, each *directly h-reachable* edge between two subgoals also ensures that they are co-visible, making the subgoal graph also suitable for any-angle pathfinding.

The N-level subgoal graph builds, on top of this simple subgoal graph, a hierarchy that is similar to a contraction hierarchy [22]. Searching in the N-level subgoal graph only requires connecting the source and destination to the corresponding direct-h-reachable subgoals, and then identifying the reachable subgoals from the source to the destination through the ascending edges (edges from a subgoal to its higher-level subgoals). Using the graph consisting of hierarchies of subgoals, one can simply apply an A* algorithm to find a path quickly, or alternatively apply the any angle Theta* [60] algorithm to search for a path with better path quality as the edges in this graph may be non-taut. Other way of improving path quality is to consider the 2^k grid neighbours [12], where the path quality can be controlled using the value of k.

The N-level subgoal graph outperforms a range of A*-like suboptimal algorithms for any-angle pathfinding, such as: Theta* [60] which improves the path quality by performing a line-of-sight check when expanding a search node; Field A* [11] which generalises the ideas from Field D* [61], allowing a straight-line trajectory from a search node to any of vertices on the boundary of its adjacent grids, and updates g-value by using linear interpolation; and Block A* [10] which performs a blocks-based A* search with local distance database, an auxiliary data structure that partitions the grid map into $m \times n$ equal size of blocks and stores the distance of optimal paths between any two boundary vertices inside each block.

2.1.3 Visibility Graphs

A Visibility Graph (VG) [62] is constructed by connecting any pairs of co-visible vertices appearing on the corners of the obstacles. In contrast to approaches that discretise the obstacles using a grid, the visibility graph allows obstacles to be precisely represented in Euclidean space. Hence, it finds the Euclidean optimal path precisely (which is guaranteed to be equal to or shorter than the any-angle path on grid maps). Search in a visibility graph only requires connecting the source and destination to their set of visible vertices before an A* algorithm can be applied to find the Euclidean optimal path. However, the visibility graph suffers from two major issues: (i) it requires high memory consumption because, in the worst case, the size of the graph is quadratic in the number of corners of the obstacles; and (ii) branching factors of nodes in the graph are high and, consequently, the search is unlikely to be efficient.



(i) (ii)

Figure 2.3: (i) shows an example of non-turn edges AK, the shaded regions indicate the potential locations of turning edges relative to A and K; (ii) shows an example of sparse visibility considering the convex vertices (coloured in green) of obstacles.

Sparse Visibility Graphs (SVG) [15] focus on addressing the above-mentioned two major issues and improve over the original visibility graph by removing unnecessary edges (i.e., a non-turn edge $e_{v_i v_j}$ where the angle from v_i to v_j does not allow turning around the polygon defining vertex v_j) and non-convex vertices. Figure 2.3 shows an example of building a sparse visibility graph for the convex vertices of the obstacles. Specifically, Figure 2.3 (i) shows the turning region (i.e., shaded area) of A and K, where the edges must be located in order to turn around the obstacles. The non-turn edge AK that is not within the turning region is pruned. The complete sparse visibility graph is shown in Figure 2.3 (ii). As evident in this toy example, sparse visibility significantly reduces the branching factor on graph nodes, which allows the graph to fit into the memory even for large maps. Due to the smaller size of the graph, the search is also more efficient. Edge N-Level Sparse Visibility Graphs (ENLSVG) [15] build a hierarchy by iteratively removing the non-taut paths. Such a hierarchy partitions the SVG into multiple levels where, in each level, edges only have taut neighbouring paths to the edges in the higher levels. ENLSVG restricts the search to only consider the edges that increase the levels from both ends. This results in searching in a smaller taut-path graph, which improves the performance further.

2.1.4 Mesh-based Planners

Mesh-based planners work by preprocessing the non-obstacle regions of the map into a set of convex polygons, called a *navigation mesh*. In Figure. 2.4, black polygons are obstacles whereas green/white polygons correspond to a navigation mesh. Popular with game developers [63], navigation meshes have several attractive properties: they are easy to compute [64], are cheap to store and update, and guarantee representational completeness (i.e., every traversable point appears in the mesh). Mesh-based path planner combines the strengths of any-angle pathfinding

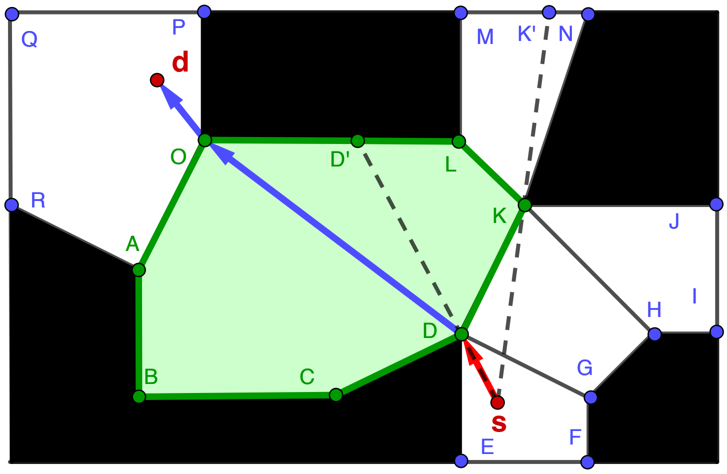


Figure 2.4: Node expansion in Polyanya. When the current node ([D,K],s) is expanded, it generates the observable successors ([D',L],s), and ([L,K],s); and non-observable successors ([D',O],D), ([O,A],D), ([A,B],D) , ([B,C],D), and ([C,D],D).

and visibility graph to find the Euclidean shortest path precisely. The state-of-art algorithm is Polyanya [14], a fast, online, and optimal pathfinding algorithm that extends and generalises Anya. Next, we present the details:

Polyanya search instantiates A* search [17] but on a navigation mesh. The algorithm can therefore be described in the same general way: there exist search nodes which generate successors and these are expanded in the best first order according to some admissible heuristic function. Polyanya differs from A* only in the domain-specific model used for each of these components. We sketch the details below (see Figure 2.4).

- **Search nodes:** A search node is a tuple of the form (I, r) where r is a distinguished vertex called the *root* and I is a contiguous interval of points from an edge of the mesh with every point $p \in I$ being visible from r. The model can be understood as follows: the root r corresponds to the last turning point on the path and I represents all the possible taut continuations of the path, on the way to the destination. The source point s is a special case and defined as $(I = [s], r = s)$. In the example of Figure 2.4, $([\mathtt{D}, \mathtt{K}], s)$ is a search node where the root is s and the contiguous interval visible from s is $[\mathtt{D}, \mathtt{K}]$.

- **Successors:** The successors of node (I, r) are generated by "pushing" the interval I away from r and across the face of an adjacent traversable polygon. There are two types of successors: *observable* and *non-observable*. A successor (I', r) is observable if each point $p' \in I'$ is visible from r. By contrast, a successor $(I', r' \neq r)$ is said to be *non-observable* if each point $p' \in I'$ is not visible from r. Note that observable successors share the same root as the parent. For non-observable successors, the root r' is one of the two endpoints of the parent interval I. Figure 2.4 shows the successors for node $([\mathtt{D}, \mathtt{K}], s)$ in green. Since the edge $[\mathtt{L}, \mathtt{K}]$ is visible from the root s, it is an observable successor and has the same root, i.e., the successor node is $([\mathtt{L}, \mathtt{K}], s)$. In contrast, the edge $[\mathtt{O}, \mathtt{A}]$ is not entirely visible from the root s and it is a non-observable successor with root $\mathtt{D}$, i.e., the successor node is $([\mathtt{O}, \mathtt{A}], \mathtt{D})$. The destination is a special case and can be generated as soon as the search reaches its containing polygon.

- **Evaluation Function:** To prioritise a node $n = (s, r)$ for expansion, Polyanya instantiates the f-value function: $f(n) = g(n) + h(n)$. Here $g(n)$ is the cost of the optimal path from the source node s to the root r. The function h is an admissible lower bound and indicates the cost from r, via some point $p \in I$, to the destination d. The estimate requires only simple geometry. Consider for example the node $n = ([\mathtt{O}, \mathtt{A}], \mathtt{D})$ from Figure 2.4. The g-value (shown in red) is the shortest distance from s to the root $\mathtt{D}$. The h-value (shown in blue) is the minimum Euclidean distance from $\mathtt{D}$ to d that passes through the edge $[\mathtt{O}, \mathtt{A}]$, i.e., $h = ed(\mathtt{D}, \mathtt{O}) + ed(\mathtt{O}, d)$, where $ed()$ is the Euclidean straight line distance. See [14] for more details.

Polyanya terminates when the destination is expanded or when the OPEN list becomes empty. It outperforms a range of pathfinding algorithms that work on Constrained Delaunay Triangulation [65] (CDT), a type of navigation mesh where the non-obstacle regions are modelled as triangles. We discuss a number of these methods below:

Channel search [66] finds the shortest channel between the source and destination from the CDT by using a modified A* algorithm. The search begins from the triangle that contains the source, and always considers the midpoint of the non-constrained edges as neighbours to expand the search nodes. The search terminates when it reaches the destination and applies the funnel algorithm [67] to retrieve a local optimal path within the channel. Channel search is fast and easy to implement, but can return suboptimal paths. Triangulation A* (TA*) [68] works in a way similar to the channel search. However, instead of terminating the search immediately, TA* takes the cost of the best path found so far as an upper bound. It continuously explores the most promising channel and updates the upper bound. TA* terminates when either the search is exhausted or the lower bound (i.e., f-value) of the search becomes greater than this upper bound. Thus, it guarantees finding the Euclidean optimal path for any given source and destination. Triangulation Reduction A* (TRA*) [68] enhances TA* by preprocessing the CDT into an abstract graph which is small, but allows the search to find the most promising channel quickly. Note that both TA* and TRA* are any-time algorithms since these algorithms can return suboptimal paths encountered during the search.

2.2 Pathfinding in Road Network

In road networks, the pathfinding problem asks us to find point-to-point traversable paths within a represented graph of the network. Let us assume the road network is static where each edge has a constant weight. Any two vertices in the graph can be a potential source s and destination d, and the objective is to find the shortest path traversing from s to d while minimising the sum of edge weights. Formally, let $G = (V, E, w)$ be a (directed or undirected) graph, with vertices V, edges $E \subseteq V \times V$ and $W : E \rightarrow \mathbb{R}^{+}$ a weight function that maps each edge $e \in E$ to a non-negative weight $w(e)$, e.g., travel time or distance etc. A path P from s to d in G is a sequence of vertices $\langle v_0, v_1, v_2, \ldots, v_{k-1}, v_k \rangle$, where $k \in \mathbb{N}^{+}$, $v_0 = s$, $v_k = d$, and $e_{v_i v_{i+1}} \in E, 0 \leq i < k$. The length of the path is $\Sigma|P| = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$ and $sd(s, d)$ denotes the length of the shortest path, from s to d in G.

In this section, we first review the literature of pathfinding in a static road network. Specifically, we categorise the existing work into three categories: goal-directed enhancements; hierarchical searches and oracle-based algorithms. Goal-directed and hierarchical approaches are search-based algorithms that typically rely on a search process to complete the pathfinding tasks, whereas the oracle-based approach only requires extraction of the path from the oracle during the query phase. Furthermore, we also consider the time-dependent road network, an extension of the static road network that allows for more reliable route planning, and explain how to extend algorithms accordingly. In this literature review, we focus on the algorithms that find the optimal (i.e., shortest) path. For a static road network, we will assume G is undirected. However, most of the techniques introduced can be trivially extended to directed graphs.

2.2.1 Goal-directed Enhancements

In a static road network, many graph search algorithms can be utilised to find the shortest path. Beginning with the development of the Dijkstra algorithm [53], which iteratively expands search nodes with the lowest cost until the shortest paths from a single source vertex to all other vertices are found. To better find the shortest path between a pair of vertices, the A* algorithm [17] improves upon Dijkstra's algorithm by incorporating a heuristic function using Euclidean estimation. The bidirectional Dijkstra search [18] extends the Dijkstra algorithm by searching simultaneously from the source and the destination. Although these graph search algorithms are online and find the shortest path, they often run slowly and require a significant amount of query time due to the large search space of the road network. One way to overcome this drawback is to precompute information from the input graph, in order to speed up the search during the query phase. Next, we explain two popular goal-directed speed-up techniques, landmarks and arc flags, which direct the search toward the destination by preferring or eliminating edges based on preprocessing data.

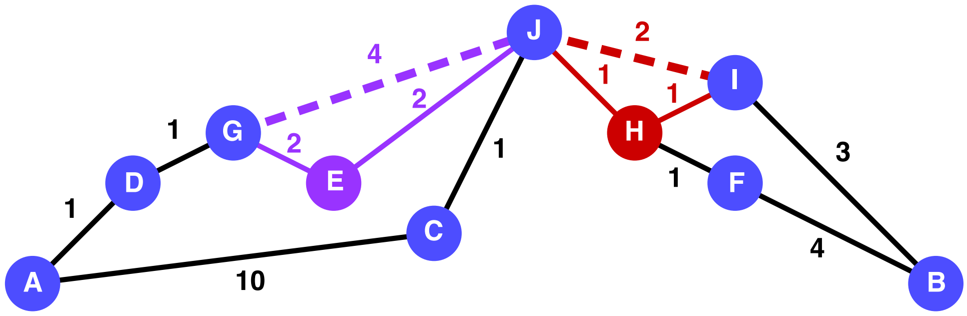


FIGURE 2.5: We show the result of contracting `E` (resp. `H`) in purple (resp. red). Dashed edges indicate shortcut edges.

Ordering	G	D	A	C	J	E	H	F	B	I
$sd(\mathtt{J}, \_)$	4	5	6	1	0	2	1	2	5	2
$sd(\mathtt{A}, \_)$	2	1	0	7	6	4	7	8	11	8
$sd(\mathtt{B}, \_)$	9	10	11	6	5	7	4	4	0	3

TABLE 2.1: Arrays of costs for vertices `J`, `A` and `B` shown in Figure 2.5
.

Landmarks [16] is a method for generating admissible estimates in the shortest path search. For each landmark $l \in L$, the algorithm computes an array during the preprocessing phase that records the optimal distance to every other vertex. The arrays of costs are exploited to lower-bound the true distance, from any vertex v_i to any other vertex v_j:

$$landmark(v_i, v_j) = \max\{|sd(v_i, l) - sd(v_j, l)| \mid l \in L\} \tag{2.1}$$

Table 2.1 shows arrays of costs for three landmarks: `J`, `A` and `B`, where the lower-bound distance from `G` to `F` is $\max\{|4-2|, |2-8|, |9-4|\} = 6$ (i.e., the true distance). The effectiveness of the lower bound depends on the distribution of landmarks, which are often selected on the borders of the graph following the same procedure explained in [54]. Landmarks have been effectively integrated with well-known algorithms such as A* (known as ALT [16]), the early hierarchical approach Reach (known as REAL [69]), the highway hierarchies (known as HH* [70]), and the Core-based routing (known as CALT [71]). Another approach to generate heuristic estimation is called Precomputed Cluster Distance [72] (PCD) which generates a lower-bound estimation by precomputing the optimal distance between all pairs of clusters that partition the input graph. Compared to landmarks, applying PCD does not result in improved query time in practice, but reduces memory cost when the number of clusters is small.

Arc flags [20, 21] is an edge labelling approach that speeds up the shortest path search in a static road network. During preprocessing, the input graph is divided into k cells with roughly the same number of vertices, and each edge is labelled with a k-bit Boolean vector, where the i-th bit indicates whether the arc lies on the shortest path to some vertices of the i-th cell. The precomputed arc flags allow the search algorithm to eliminate edges not in the shortest path to the destination cell, thereby reducing the search space. Similar to landmarks, Bauer et al. [71] have demonstrated that arc flags can be integrated with several efficient pathfinding algorithms, such as ReachFlags (based on Reach [24]), CHASE (based on Contraction Hierarchies [22]), and TNRAF (based on Transit Node Routing [73]). Additionally, SHARC [74] is another successful adaptation of highway hierarchies [23] and arc flags.

2.2.2 Hierarchical Searches

Another family of successful algorithms are hierarchical approaches, which exploit the hierarchical nature of the real-world transportation network. The basis of hierarchical approaches comes from the observation that when travelling between two distant locations, the fastest route always involves transitioning from local roads into highways (i.e., the less important roads merge into more important roads). Reach [24] was the first algorithm that formalise this idea by identifying "important" vertices using a reach score. This score function can also serve as a lower bound to eliminate unimportant vertices during the search phase. Subsequently, Goldberg et al. [69] introduced the concept of shortcuts (i.e., a single edge represents the path of multiple vertices) to lower the reach score and enhance the pruning of nodes during the search. The idea of shortcuts has since become a fundamental component in other hierarchical approaches. Highway hierarchies [23] take the concept of shortcuts and construct k-level hierarchies, with each level being a highway representation of the lower-level graph. This allows the search algorithm to concentrate only on the important higher-level parts of the network during the middle of the route. Later on, this idea has been generalised into the simpler, faster, and widely used hierarchical approach known as contraction hierarchies [22]. Next, we give a detailed description of the algorithm.

A Contraction Hierarchy (CH) is an augmented multi-level graph that can be exploited to speed up pathfinding search. Building a CH is a simple process requiring only the repeated application of a contraction operation to the vertices of the input graph G. In broad strokes:

1. Apply a total lex order $\mathcal{L}$ to the vertices of G.
2. W.r.t. $\mathcal{L}$, choose the least vertex v from the graph that has not been previously selected.
3. (Contraction) Add to G a shortcut edge e_{uw} between *each* pair of in-neighbour u and out-neighbour w of v for which: (i) u and w are both lexically larger than v; and (ii) the shortest path from u to w passes through v. To reduce the number of shortcuts added in G, the subpath $\langle u, v, w\rangle$ should be both unique and optimal. Fewer shortcuts improve query performance but verifying local optimality requires additional pre-processing time.

The step 2-3 repeat until every vertex $v \in V$ has been contracted. Note that the number of shortcuts of CH depends on the lex order $\mathcal{L}$. However, computing an optimal order that minimises the number of shortcuts is NP-hard [75]. Among the many heuristic orders, Nested Dissection orders (ND-orders), is the most popular and efficient as suggested in [76]. Figure 2.5 shows an example of contraction hierarchies, where we contract a toy graph in alphabetical lex order. Note how shortcut edges (dashed) can connect source and destination faster than would otherwise be possible. Without shortcuts, the optimal path from A to B has 7 edges: $\langle \mathtt{A}, \mathtt{D}, \mathtt{G}, \mathtt{E}, \mathtt{J}, \mathtt{H}, \mathtt{I}, \mathtt{B} \rangle$. An equivalent-cost path, with shortcuts, traverses only five edges: $\langle \mathtt{A}, \mathtt{D}, \mathtt{G}, \mathtt{J}, \mathtt{I}, \mathtt{B} \rangle$.

A core idea of contraction hierarchies is that shortcut edges can bypass one or more intermediate vertices in a single step. However, for each shortcut edge euw and each intermediate vertex v we have $f(v) \leq f(w)$; i.e., given a monotonically increasing cost function f, a simple best-first search will usually expand v before w in order to compute an optimal path. To achieve a speedup, authors in [22] divide the set of edges E into two as follows:

- $E_{\uparrow} = \{e_{uv} \in E \mid u <_{\mathcal{L}} v\}$
  (i.e., the set of all "up" edges); and
- $E_{\downarrow} = \{e_{uv} \in E \mid u >_{\mathcal{L}} v\}$
  (i.e., the set of all "down" edges).

The following results, paraphrased here, are due to [22].

Lemma 2.1. ***(ch-path)**: For every optimal path $sp(s, d)$ in E, there is a cost equivalent ch-path whose prefix $\langle s, \ldots k\rangle$ is found in $E_{\uparrow}$ and whose suffix $\langle k \ldots d\rangle$ is found in $E_{\downarrow}$.* □

Corollary 2.2. ***(apex vertex)**: Every ch-path has a vertex k which is lexically largest among all vertices on the path.* □

Following Lemma 2.1, a natural decomposition of the shortest path problem in a contraction hierarchy is the following: first compute a subpath $\langle s, \ldots, k\rangle$ in $E_{\uparrow}$; next, compute a second subpath $\langle k, \ldots, d\rangle$ in $E_{\downarrow}$. All that remains is to identify a suitable vertex k which minimises the total distance. BCH is a variation on the bidirectional Dijkstra search that was developed specifically for solving such problems.

In the forward direction, BCH considers only the outgoing edges in $E_{\uparrow}$. In the reverse direction, BCH considers only the incoming edges in $E_{\downarrow}$.[1] Each meeting point of the two search frontiers corresponds to a tentative shortest path. Unlike standard bidirectional Dijkstra search, which can be terminated as soon as the sum of the minimum f-values on open lists for both

[1] Other edges from E, such as incoming up-edges and outgoing down-edges are safely discarded by BCH to save space.

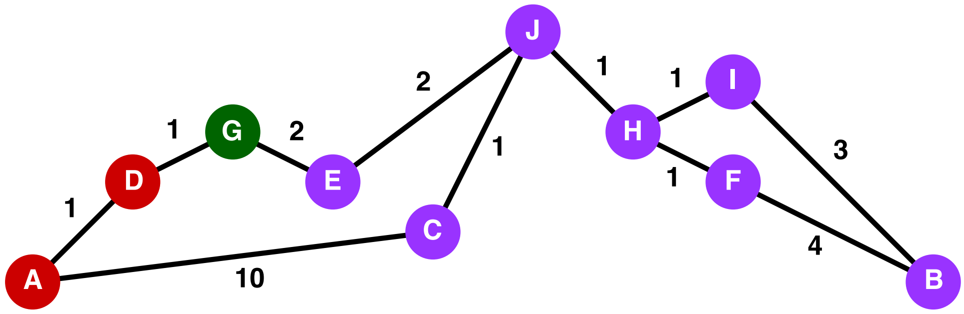


FIGURE 2.6: From the source vertex G, the optimal first move to any vertex coloured red (resp. purple) is D (resp. E).

directions is no less than the length of the best candidate path, BCH continues until it can prove the meeting point k minimises the total distance between s and d, i.e., BCH stops when the minimum f-value on both open lists are at least as large as the best candidate path found so far (or when both lists are empty, if there is no such path). Finally, BCH unpacks the ch-path and returns the shortest path $sp(s, d)$. Though simple, BCH remains state of the art for pathfinding on road networks with millions of vertices [77].

In addition to the contraction hierarchy, the arterial hierarchy [78] improves the worst-case complexity of the query time for CH, though the improvement is not significant in practice. Core-based routing [71], on the other hand, focuses on reducing the space complexity by computing an overlapped core graph that retains only the shortcuts of contracting unimportant vertices. The goal-directed enhancements, such as landmarks or arc flags, are then computed on this core graph to speed up the search when it is restricted to the core graph. Similarly, the transit node routing [73] involves the selection of a subset of transit nodes from the graph and the precomputation of pairwise distances between these transit nodes. While connecting the source and destination to transit nodes enables us to efficiently compute the shortest distance, this approach falls short when it comes to retrieving the complete shortest path.

2.2.3 Oracle-based Algorithms

Preprocessing the input graph can indeed improve the query performance of search-based algorithms, however, these algorithms are still limited by the search process in finding the optimal path, causing a bottleneck in performance. Oracle-based approaches are another family of algorithms that forgo conventional state-space search and instead extract the shortest paths using precomputed data. Although oracle-based methods provide ultra-fast answers to the shortest path queries, they typically require a substantial amount of preprocessing time and memory to build and store such an oracle. Next, we give an overview of state-of-the-art oracle-based algorithms.

Ordering	G	D	A	C	J	E	H	F	B	I
G	*	D	D	E	E	E	E	E	E	E
J	E	E	E	C	*	E	H	H	H	H
I	H	H	H	H	H	H	H	H	B	*

TABLE 2.2: First moves for G, J and I for the example of Figure 2.6

Compressed Path Databases (CPD) [26] are auxiliary data structures that encode and compress the first move (equivalently first arc) on the optimal path from each vertex $s \in V$ to every other vertex $d \in V$. Given a weighted graph G (i.e., an example is shown in Figure 2.6), CPDs are constructed offline using one complete Dijkstra search for each source vertex $s \in V$. The worst-case complexity is therefore $O(|V||E| + |V|^2 \log |V|)$. However, each Dijkstra search can be executed in parallel with a potential speedup depending on the number of processors available on the machine.

- **First-Move Tables:** With only a small modification to the basic Dijkstra algorithm (specifically, for each vertex, we maintain the first outgoing arc on the optimal path from s to this vertex), we compute for each source vertex $s \in V$, a *first move table* where $fm(s, d)$ returns a symbol that tells which of the outgoing arcs of s appear on an optimal path, from s to any $d \in V$. Table 2.2 shows all first moves for source vertices G, J and I in Figure 2.6. Note that each $fm(s, s)$ is assigned a wildcard symbol [79] "*" (i.e., "don't care" symbols) as we never need to look up a move from s to itself.
- **Compression:** The CPD compresses first-move tables using run-length encoding (RLE) [27]. RLE compresses a string of symbols into representative sub-strings, called *runs*. Each run has two values: a start index and a first-move symbol. E.g., the string C; C; C; D; D; D, can be compactly represented as two runs: 1C; 4D. In addition to that, the wildcard symbol "*" is allowed to be compressed with any other preceding or subsequent symbol. For example, row G in Table 2.2 can be compressed into just two runs: 1D; 5E. The effectiveness of RLE compression is dependent on the way the candidate vertices are ordered. Following the suggestion in [80], we use a Depth-First-Search (DFS) ordering of vertices. Specifically, we run a DFS on a randomly selected vertex and the DFS ordering corresponds to the order in which these vertices are accessed by the DFS. This ordering tends to order the vertices that have the same symbol closer to each other which helps with compression. In Table 2.2, the order of the columns is a DFS visit order in Figure 2.6 starting from vertex G.
- **Path Extraction:** CPDs can efficiently retrieve optimal paths for any given source-destination pair within the graph. We denote the function $fm(s, d)$ which extracts from the database a first-move symbol, from s to d. Each extraction operation $fm(s, d)$ requires a binary search on the RLE-encoded first-move table of s to find the first-move symbol

Ordering	G	D	A	C	J	E	H	F	B	I
G	*	G	D	J	E	G	J	H	I	H
J	E	G	D	J	*	J	J	H	I	H
I	E	G	D	J	H	J	I	H	I	*

TABLE 2.3: Reverse First moves for G, J and I for the example of Figure 2.6

to reach d, which runs in $O(\log n)$ where n is the number of symbols in it [27]. Once a first move is extracted, it can be followed to reach a new location. The entire pathfinding process can thus be implemented in a simple recursion: we extract and follow optimal moves until the destination is reached.

Reversed Path Databases (RPD) [25] is another variation of CPD. While a CPD stores first moves, in an RPD, for each $s \in V$, a reverse row $RR(s)$ is computed which stores, for every $d \in V$, the last move on the shortest path from s to d (i.e., the first vertex on the reversed path from d to s). Table 2.3 shows all reverse first moves for source vertices G, J and I in Figure 2.6. Unlike CPD, the compression on $RR(s)$ is not effective. Therefore, RPDs are not compressed. This allows accessing the first move in $O(1)$. The shortest path from s to d can be efficiently obtained by recursively obtaining the $fm(s,d)$ using $RR(d)$, i.e., the shortest path can be extracted using a single row. Note that constructing each reverse row involves conducting a Dijkstra search while taking into account incoming edges of the static road network, they can also be computed in parallel with a speedup linear in the number of processors.

Hub Labelling (HL) [81] is the state-of-the-art approach for computing the shortest distance in road networks. During the preprocessing, hub labelling computes and stores a set of hub labels $H(v_j)$ for each vertex $v_j \in V$. Each hub label is a tuple $(h_i, sd_{ij}) \in H(v_j)$ which contains: (i) a hub vertex $h_i \in V$; and (ii) the short distance sd_{ij} between hub vertex h_i and v_j; The critical importance is to ensure that the hub labels computed satisfy the coverage property, i.e., for every pair of reachable vertices $v_j \in V$ and $v_k \in V$, there must exist a hub vertex h_i in both $H(v_j)$ and $H(v_k)$, such that h_i on the shortest path between v_j to v_k. To achieve this, the HL algorithm repeatedly performs a pruned Dijkstra search on the vertices in graph G following a given lex order $\mathcal{L}$. In each iteration, the pruned Dijkstra search starts from vertex v_j, traversing the road network as a standard Dijkstra search. However, when reaching vertex v_k, the search node is pruned if there is a common hub vertex h_i in both $H(v_j)$ and $H(v_k)$ such that the tentative distance $g(v_j, v_k) \geq sd_{ij} + sd_{ik}$. Otherwise, the search continues, and the hub label $(v_j, sd(v_j, v_k))$ is added to the label set of vertex v_k. Note that, computing the smallest hub labelling while ensuring the coverage property is NP-hard [82]. Therefore, heuristic approaches are often used to compute hub labelling. Given the hub labels computed on an input graph G, the shortest distance between any a pair of (s, d) can be computed as:

Vertex	J	I	H	G	F	E	D	C	B	A
Labels	{J,0}	{J,2}	{J,1}	{J,4}	{J,2}	{J,2}	{J,5}	{J,1}	{J,5}	{J,6}
		{I,0}	{I,1}	{G,0}	{I,2}	{G,2}	{G,1}		{I,2}	{G,2}
			{H,0}		{H,1}		{D,0}		{F,4}	{D,1}
					{F,0}					

Table 2.4: Hub labeling for the graph in Figure 2.6

$$sd(s, d) = \operatorname*{arg\,min}_{h_i \in H(s) \cap H(d)} (sd_{is} + sd_{it}) \tag{2.2}$$

Computing Shortest Distance. Hub labelling sort labels of each vertex based on the hub vertices, the shortest distance between s and d can be calculated by scanning over (similar to the merge phase in sort-merge join) the sorted label set $H(s)$ and $H(d)$, using the equation in Equation 2.2. The complexity is $O(|H(s)|+|H(d)|)$, where $|H(s)|$ denotes the number of labels in $H(s)$. Table 2.4 shows hub labels for the graph in Figure 2.6. To compute $sd(\mathtt{A}, \mathtt{B})$, the hub labels of A and B are scanned and the common hub vertices J is found. Thus $sd(\mathtt{B}, \mathtt{J}) + sd(\mathtt{A}, \mathtt{J}) = 5 + 6$ is returned.

Computing Shortest Path. To enable computing the shortest paths, during the hub labelling computation, an additional successor vertices are stored in the labels. i.e., (h_i, sd_{ij}, s_{ij}) where s_{ij} is the first vertex on the shortest path from v_j to h_i. To find the shortest path between s and d, HL first computes the shortest distance using Equation 2.2 to find the common hub vertex h_i. Then, it retrieves the shortest path from s to h_i by recursively following the successor vertex s_{is} stored in the labels (h_i, sd_{is}, s_{is}). The shortest path from d to h_i is retrieved similarly. Note that each retrieval of the next successor vertex requires a linear scan over the label set of current successor s_{is}. The complexity of this approach is $O(N \times SP)$, where N is the average label size of each vertex and SP is the length of the shortest path.

Pruned Highway Labelling (PHL) [83] is another popular and efficient approach for hub labelling. Unlike traditional methods that maintain vertices as hubs, PHL uses highways (i.e., path segments) as hubs and stores the distances to them. These highways are derived from the shortest paths in the input graph and are identified by using the speed information (i.e., distance divided by travel time) of the road network. However, PHL has two major restrictions: (i) the input graph must be an undirected graph; (ii) the input graph must contain distance and travel data. Recall that the HL is constructed based on the lex order of vertices. Significant path based Hub Pushing (SHP) [84] generalises the idea of PHL by using a heuristic approach for ordering vertices when constructing the HL. Similar to a highway, the shortest path is a "significant path" if it passes by many other shortest paths. To efficiently select vertices on "significant paths" as hubs, SHP ranks the vertices by multiplying the vertex degree and the size difference of its descendants. Though simple, SHP remains the state-of-art ordering scheme for HL techniques [84] and improves PHL by avoiding these restrictions.

2.2.4 Time-Dependent Scenarios

The time-dependent road network extends the static road network by considering the time-varying factors that impact road transportation, such as traffic congestion, road closures etc. Unlike the static road network, the time-dependent road network uses a piecewise-linear function to model the changing travel times of each edge, allowing for a more accurate representation of road conditions. Given a source (s), destination (d) and departure time (t), the actual travel time of each edge depending on its corresponding arrival time, the pathfinding queries find the fastest path traversing from s to d at departure time t. We formalise the time-dependent road network as follows:

Let $G = (V, E, F, T)$ be a directed graph, with vertices V, edges $E \subseteq V \times V$ and $f \in F$ maps each edge $e \in E$ to a Travel Time Function (TTF) which returns the non-negative travel time $f(t)$ needed to travel through the edge e for a given specific start time t in the time domain T. Each directed edge $e_{v_iv_j} \in E$ with its corresponding TTF $f_{v_iv_j}$ represents the edge that connects vertex v_i to v_j. In a road network, we naturally assume that the network G satisfies the FIFO property (i.e., $f_{v_iv_j}(t') + t' \geq f_{v_iv_j}(t) + t \mid \forall e_{v_iv_j} \in E$ and $\forall t' > t \in T$), that is departing later or waiting at an intermediate vertex cannot result in arriving earlier. Similar to many existing works [32, 85], we model the TTF as a continuous piece-wise linear function with the time domain of 24 hours. Figure 2.7 shows an example of a time-dependent network. For exposition only, we assume the graph G is undirected and only the edges that are shown in red have non-constant TTF with $T = [0, 180)$. Next, we explain some of the important operations of TTF:

- **Evaluation** of $f(t) \in F$ normally requires a binary search over the sorted array of interpolate points, and runs in $O(log(|f|))$ where $|f|$ denotes the number of interpolate points of f. However, we use the bucket-implementation [85], which evaluates $f(t)$ by scanning only the interpolate points inside the bucket corresponding to t.
- **Chaining** computes the TTF of a path $e_{v_iv_j} \rightarrow e_{v_jv_k}$ as $f_{v_iv_k}(t) = f_{v_jv_k}(t + f_{v_iv_j}(t))$. We use $f_{v_iv_k} = f_{v_iv_j} \circ f_{v_jv_k}$ to denote the chaining. Since chaining two piece-wise linear functions can only result in a piece-wise linear function, the operation can be computed in linear time (i.e., $O(|f_{v_iv_j}| + |f_{v_jv_k}|)$). The resultant function $f_{v_iv_k}$ has the number of interpolate points $|f_{v_iv_k}| \leq |f_{v_iv_j}| + |f_{v_jv_k}|$ with the lower bound $min(f_{v_iv_k}) \geq min(f_{v_iv_j}) + min(f_{v_jv_k})$.

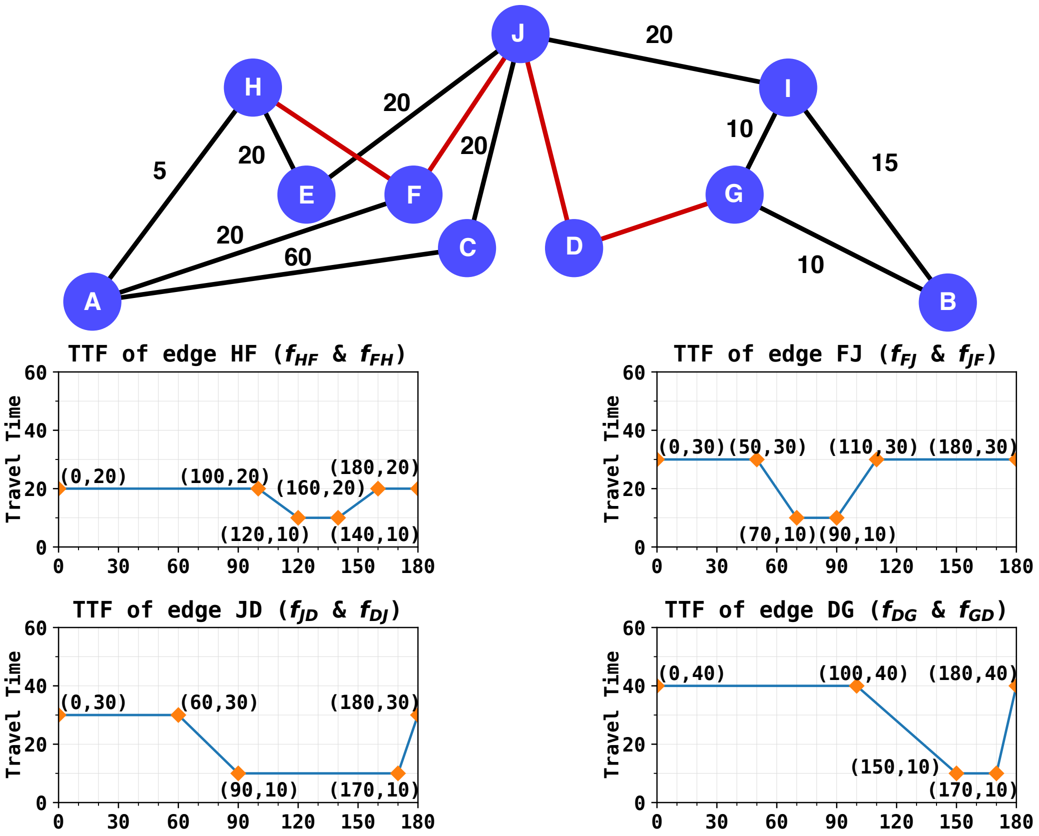


FIGURE 2.7: An example of an undirected time-dependent graph. TTFs of the red edges are shown below the graph, and the travel cost of the other edges are constant.

- **Merging** minimises the TTFs $f'_{v_iv_j}$ and $f''_{v_iv_j}$ on two parallel edges of $e_{v_iv_j}$ while preserving all the shortest paths in G. The operation $f_{v_iv_j} = min(f'_{v_iv_j}, f''_{v_iv_j})$ is defined as $f_{v_iv_j}(t) = min\{f'_{v_iv_j}(t), f''_{v_iv_j}(t)\} \mid \forall t \in T$. Similar to chaining, this operation also runs in $O(|f'_{v_iv_j}| + |f''_{v_iv_j}|)$ and results in a piece-wise linear TTF.

A path P from source s and destination d is a sequence of vertices $\langle v_0, v_1, v_2, \ldots, v_{k-1}, v_k\rangle$, where $k \in \mathbb{N}^+$, $v_0 = s$, $v_k = d$, and $e_{v_iv_{i+1}} \in E$ for $0 \leq i < k$. In a time-dependent road network, the length (travel cost) of path $\Sigma|P|$ depends on the departure time $t \in T$ and $\Sigma|P| = f_{v_0v_k}(t)$, where $f_{v_0v_k}(t) = f_{v_0v_1} \circ f_{v_1v_2} \ldots \circ f_{v_{k-1}v_k}$. Given a departure time $t \in T$, $sp(s, d, t)$ denotes the shortest path from s to d.

2.2.4.1 Adapting Algorithms for Time-Dependent Road Networks

In a time-dependent road network, we assume the network G satisfies the FIFO property. Therefore, algorithms such as Dijkstra and A* can be applied directly, with the main difference being the evaluation of the travel time function during generating the successors. Among the limited existing research, Nannicini et al. [29] extend the bidirectional A* search with landmarks

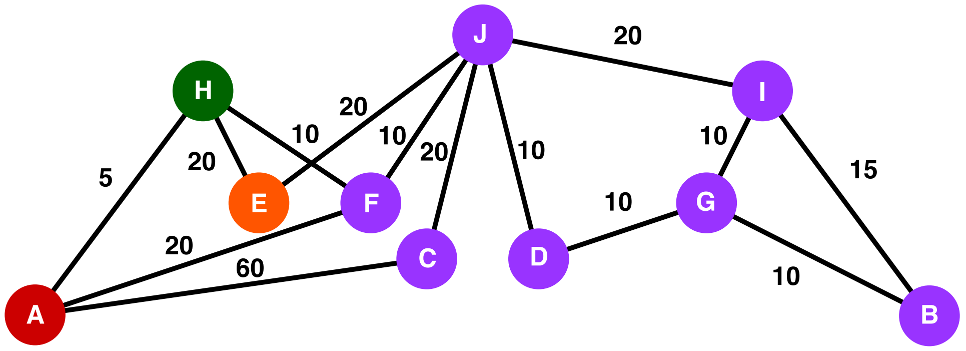


FIGURE 2.8: From the source vertex H, the first move on the optimal path to any node are A (red), E(orange) and F (purple).

Ordering	H	A	F	J	I	B	G	D	C	E
H	∗	A	F	F	F	F	F	F	F	E
A	H	∗	H	H	H	H	H	H	H	H
J	F	F	F	∗	I	D	D	D	C	E

(i): First Move Table

Ordering	H	A	F	J	I	B	G	D	C	E
H	H	H	H	F	J	G	D	J	J	H
A	A	A	H	F	J	G	D	J	J	H
J	F	H	J	J	J	G	D	J	J	J

(ii): Reverse First Move Table

TABLE 2.5: (i) First moves and (ii) Reverse first moves for H, A and J for the example of Figure 2.8

to time-dependent scenarios (extending landmarks will be discussed later on). Delling and Nannicini [30] extend the core-based routing by introducing the idea of contracting edges with travel time functions. In addition, Delling [31] extends SHARC, a combination of highway hierarchies and arc flags, to time-dependent scenarios. However, in this case, the arc flags are calculated to indicate whether an edge appears on the fastest path at any time within the domain T. Next, we provide the details of algorithms that are highly relevant to our research:

Landmarks [28] have also been extended to generate the admissible heuristic in a time-dependent road network. Unlike static road networks, a time-dependent graph is a directed graph with changing travel time, therefore, this requires slight modifications: (i) For each landmark $l \in L$, we use the minimum travel time of each edge to compute an array that records a pair $(sd(l,v), sd(v,l))$ on each $v \in V$, where $sd(l,v)$ denotes the shortest travel time from l to v. (ii) The array of tuples is exploited to be the lower bound of $sp(v_i, v_j, t)$ for $\forall t \in T$, from any v_i to any v_j:

$$landmark(v_i, v_j) = \max_{l \in L}\{max(sd(v_i, l) - sd(v_j, l), sd(l, v_j) - sd(l, v_i))\} \tag{2.3}$$

The shortest travel time pairs in (i) are stored to support the directed graph. The minimum travel time is considered to lower-bound the actual travel time of each edge, leading to equation 2.3 based on the triangle inequality. Similar to the static road network, the landmarks are selected on the border of the graph to provide a stronger estimation.

Path databases heuristics [48] are another state-of-the-art method that extends the path oracles to generate admissible heuristics in dynamic environment settings. Similarly, given a time-dependent graph G, we consider the minimal travel time on each edge of G and build a CPD or RPD following the same procedure as explained in section 2.2.3. Let $cpd(s, d)$ denote the optimal path extracted from the CPD between s and d. Then, this path must be no larger than the shortest travel time paths on G (i.e., $\Sigma|cpd(s, d)| \leq \Sigma|sp(s, d, t)|$ for $\forall(s, d) \in G$ and $\forall t \in T$). Therefore, the first-move oracle defines an admissible heuristic. Figure 2.8 shows an example, where we construct a CPD (resp. RPD) on a graph with minimal edge weight shown in Figure 2.7 (i) (resp. Figure 2.7 (ii)). The shortest path extracted from CPD between `H` to `J` is a valid lower bound, i.e., $(\Sigma|cpd(\texttt{H}, \texttt{J})| = 20) \leq (\{\Sigma|sp(\texttt{H}, \texttt{J}, t)| \mid \forall t \in T\} = [30,40])$, similarly for RPD. Note that, when a path database is used as a heuristic, it needs to continuously extract the path $cpd(s, d)$ at each node expansion of s. Therefore, a caching strategy is often applied to cache the distance from every intermediate vertex v to d during the path extraction of $cpd(s, d)$. This strategy helps avoid repetitive first-move extraction.

Time-dependent Contraction Hierarchies (TCH) [32, 85] extends the Contraction Hierarchies to find the fastest path in a time-dependent road network. While many of the concepts are similar, we provide a complete explanation of TCH for the sake of comprehensiveness. Given a graph G, a TCH can be built by repeatedly applying a contraction operation to $v \in V$. In broad strokes:

1. Apply a total lex order $\mathcal{L}$ to the vertices V of G.
2. W.r.t. $\mathcal{L}$, choose the least vertex $v \in V$ that has not been previously contracted.
3. (Contraction) Add to G a shortcut edge e_{uw} between *each* pair of in-neighbour u and out-neighbour w of v for which: 1) the lex order u and w are larger than v; and 2) $\langle u, v, w\rangle$ is the shortest path between u and w at *some time* in T. When adding the shortcut edge e_{uw}, the TTF is computed as $f_{uw} = f_{uv} \circ f_{vw}$. However, the parallel edges can exist, and in this case, we merge existing TTF f'_{uw} as $f_{uw} = min(f'_{uw}, f_{uv} \circ f_{vw})$ and maintain a *middle vertex profile* to track the intermediate vertex for the corresponding interval in TTF.

Fewer shortcuts improve query performance, but computing a lex order $\mathcal{L}$ that minimises the number of shortcuts is NP-hard [75]. Thus, a heuristic order is suggested in [85]. Note that the contraction operation requires verifying local optimality and can be costly in time-dependent scenarios, therefore the steps 2-3 are parallelised. For more details of the parallelisation, we refer the reader to the paper [85]. Consider Figure 2.9 as an example, where we contract the time-dependent graph shown in Figure 2.7 in alphabetical lex order. The TTF of the shortcut edge $e_{\texttt{HJ}}$ is computed as $f_{\texttt{HJ}} = min(f_{\texttt{HE}} \circ f_{\texttt{EJ}}, f_{\texttt{HF}} \circ f_{\texttt{FJ}})$ and the corresponding middle vertex profile is $\{(0, \texttt{E}), (40, \texttt{F}), (80, \texttt{E})\}$ which indicates the middle vertices are `E`, `F` and `E` for the time

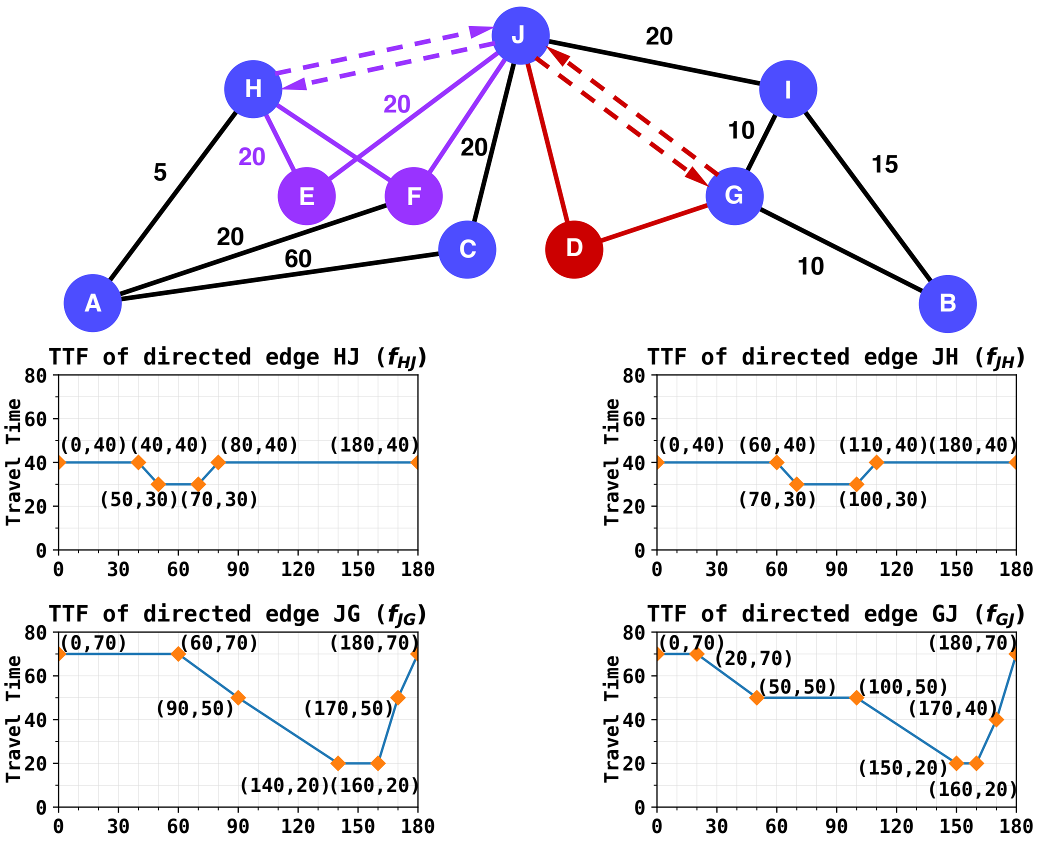


FIGURE 2.9: We show the result for contracting vertices E and F in purple, and D in red. Dashed edges are the shortcut edges and their corresponding TTFs are shown in the figure below.

period $[0, 40)$,$[40, 80)$ and $[80, 180)$, respectively. Also note, without shortcut edges (dashed), the optimal path from A to B at time 150 has 6 edges: $\langle$A, H, E, J, D, G, B$\rangle$. However, with shortcut edges, we traverse only four edges: $\langle$A, H, J, G, B$\rangle$.

Recall that TCH adds a shortcut edge e_{uw} iff both u and w are lexically larger than the intermediate vertex v, and e_{uw} is optimal and equivalent to the path $\langle u, v, w\rangle$. Therefore, for every pair of edges (e_{uv}, e_{vw}), there must exist a cost equivalent *up edge* $e_{uw} \uparrow$ (i.e, $u <_{\mathcal{L}} w$) or *down edge* $e_{uw} \downarrow$ (i.e, $u >_{\mathcal{L}} w$), if $\langle u, v, w\rangle$ is $sp(u, w, t)$ for $t \in T$. Thus, we have the following:

Lemma 2.3. ***(tch-path)***: *For every optimal path $sp(s, d, t)$ in G, there is a cost equivalent tch-path $\langle s, \dots k \dots d\rangle$ whose prefix $\langle s, \dots k\rangle$ is an up path (i.e., $s <_{\mathcal{L}} s+1 <_{\mathcal{L}} k$ and suffix $\langle k \dots d\rangle$ is a down path (i.e., $k >_{\mathcal{L}} k+1 >_{\mathcal{L}} d$)* □

Corollary 2.4. ***(apex vertex)***: *Every tch-path has a vertex k which is lexically largest among all vertices on the path.* □

The key idea of TCH is that the shortcut edges can bypass one or more intermediate vertices in a single step. To achieve a speedup, authors in [85] develop the Bidirectional TCH search (BTCH) to efficiently find the tch-path, following Lemma 2.3. To support the bidirectional search, BTCH divides the set of edges E into two as follows:

- $E_{\uparrow} = \{e_{uv} \in E \mid u <_{\mathcal{L}} v\}$
  (i.e., the set of all "up" edges); and

- $E_{\downarrow} = \{e_{uv} \in E \mid u >_{\mathcal{L}} v\}$
  (i.e., the set of all "down" edges).

Given a source-destination pair (s, d) and departure time t, the main challenge of the bidirectional search is that the backward search is prohibitive in time-dependent scenarios (i.e., we can not search backward without knowing the arrival time t'). Therefore, BTCH runs in two phases:

(i) **Bidirectional Search:** In the forward direction, BTCH runs a time-dependent Dijkstra search from s, considering only the outgoing edges in $E_{\uparrow}$. Similar to the static graph, this Dijkstra search differs mainly in that it considers the TTFs when it generates successors. In the reverse direction, BTCH performs the backward exploration from d using a static Dijkstra search, and considers only the lower-bound edge weights of the incoming edges in $E_{\downarrow}$. We also maintain U (initially infinity) which is the smallest upper-bound distance of any path from s to d seen so far by the algorithm. During the backward Dijkstra search, we store each traversed edge in E_{trv} and compute the upper bound by taking the maximal value on each edge in $E_{\downarrow}$. Whenever the two searches meet at an apex vertex k, we: (i) update U if the upper-bound distance from s to d via k is smaller than current U; and (ii) obtain a lower bound $lb(s, k, d)$ for the path from s to d via k. If $lb(s, k, d) \geq U$, we discontinue the search on both sides from k. Every time the search meets, we also record the apex vertex k in K. The bidirectional search continues until the minimum f-value on both open lists are no less than U (or when both lists are empty, if no such path).

(ii) **Forward Search:** Once the bidirectional search stops, for each recorded apex vertex $k \in K$ and $lb(s, k, d) \leq U$, we continue a forward time-dependent Dijkstra search on the edges E_{trv} by iteratively inserting k into the queue. This forward search is funnelled into d and considers only the edges that are previously traversed by the backward search. The search terminates if d is expanded.

Finally, BTCH unpacks the tch-path using the middle vertex profile maintained on the corresponding shortcut edges, and the optimal path $sp(s, d, t)$ is returned. BTCH remains state-of-the-art for time-dependent routing [86].

2.3 Classic Multi-Agent Pathfinding

The previous sections cover pathfinding in two-dimensional spaces and road networks. Despite the differences in environments, the main focus remains on finding a path for a single agent to travel between two locations, often disregarding other agents that may operate in the same environment (though not always, as time-dependent road networks consider the effect between agents as a traffic pattern for each edge). Multi-Agent Pathfinding (MAPF) takes interactions between agents into account and plans the paths that avoid collisions with other agents and obstacles. In this section, we consider the classic MAPF problem where the underlying workspace is represented by a four-connected grid map as explained in section 2.1.1. Next, we formalise the classic MAPF problem.

Following Stern et al. [3], a MAPF instance consists of (i) an input grid map, where each cell connects to only orthogonal neighbours, and (ii) a set of k agents $A = \{a_1, \dots, a_k\}$. We represent the grid map as an undirected and unweighted graph $G = (V, E)$ with nodes V and edges $E \subseteq V \times V$. Each agent $a_i \in A$ has a unique source ($s_i \in V$) and destination ($d_i \in V$). Time is discretised into unit-sized timesteps, and, at each timestep, agents are allowed to move to an adjacent vertex or else wait at their current location. A path of agent a_i is a sequence of vertices $P = \langle s_i, \dots, d_i \rangle$, indicating the location of a_i at each timestep. An agent has reached its destination if it permanently waits at its destination location and never has to move off to make way for another agent. The cost of a path P is the number of timesteps (i.e., $|P| - 1$) required for an agent to reach the destination location from its source (ignoring wait costs after reaching). The paths of two agents a_i and a_j can conflict in two ways: (i) a vertex conflict $\langle a_i, a_j, v_i, t \rangle$ when agent a_i and a_j reach the same vertex $v_i \in V$ at the same timestep t, and (ii) an edge conflict $\langle a_i, a_j, v_i, v_j, t \rangle$ when two agents a_i and a_j traverse the same edge $e_{v_i v_j} \in E$ from the opposite directions at the same timestep t. A solution is a set of conflict-free paths, one for each agent. Our objective is to find an optimal solution that minimises the sum of the individual costs (SIC) of the paths.

In this section, we discuss the existing works related to the classic MAPF problem by categorising them into three categories: optimal MAPF algorithms, bounded-suboptimal MAPF algorithms and unbounded-suboptimal MAPF algorithms. We begin by providing an overview of each type of MAPF algorithm. Given that one of our research objectives is to improve the state-of-the-art optimal search-based algorithm, Conflict-Based Search (CBS), we will then present a detailed explanation of the algorithm, along with its recent improvements.

2.3.1 Optimal MAPF Algorithms

The MAPF problem is challenging because it requires consideration of potential collisions that may occur between multiple agents. Even more challenging is finding an optimal solution for all agents that optimises the SIC, as this problem is NP-hard [87]. Optimal Multi-Agent Pathfinding (MAPF) algorithms can be categorised into search-based and compilation-based algorithms. Search-based algorithms utilise approaches such as joint-state A* search, increasing cost tree search, and conflict-based search. On the other hand, compilation-based algorithms transform the problem into a combinatorial optimisation problem, which is then solved using an off-the-shelf solver. Next, we present an overview of each type of search-based algorithms, including their variations, as well as compilation-based algorithms.

Joint-space A* search is a straightforward approach to find the optimal solution for MAPF by utilising A* search algorithm with a joint-state space. In a joint-state space, each agent corresponds to a different location among a total of $|V|$ vertices on a grid map. Therefore, the successors of the A* search correspond to all potential non-collision combinations of actions the agents can take (i.e., for a four-connected grid map, there are 4^k number of successors in the worst case where k is the number of agents in a MAPF instance). The g-value and h-value of the A* search is the sum of g-value and h-value for each agent in a search node. Since the joint-state space grows exponentially with the number of agents, various techniques have been proposed to eliminate the search space of joint-state A* search. For example, Operator Decomposition (OD) [88] modifies the joint-state A* search by only changing the position of one agent at a time. Though this reduces the branching factor, it also increases the search tree depth substantially. Independence Detection (ID) [88] divides the agents into disjoint subgroups and solves each subgroup independently. However, it may not work well when the number of agents is large and partitioning is difficult. Enhanced Partial Expansion (EPEA*) [89] focuses on reducing the surplus nodes (i.e., the node with the cost larger than optimal cost) of the joint-state A* search by only generating the successors with $f(n) = f'(n)$ where $f'(n)$ is the minimal f-value in the OPEN list. Sub-dimensional expansion (M*) [90] changes the joint-state A* search, such that it initially runs an A* search on each individual agent and only locally increases the dimensionality (i.e., merges the search space) when the search space of one agent is conflicting with another.

Increasing Cost Tree Search (ICTS) [91] is a two-level search algorithm that improves the joint-state A* search by limiting the joint-state space to only consider a combination of fixed cost agents. In the high-level, ICTS searches in an increasing cost tree where each node contains a set of costs, one for each agent, and a parent node connects to k number of child nodes, each of which increases the cost of one of k agents by one. When expanding a high-level node, the lower-level search verifies whether a solution exists in the joint-state space that combines the potential locations of each agent with its corresponding cost of the high-level node. Sharon et al. [91] also propose several pruning techniques to quickly skip the high-level nodes that do not contain a solution, leading to better performance than the joint-state A* search.

Conflict-Based Search (CBS) [33] is another two-level search algorithm that resolves conflicts by adding constraints and replanning the paths that satisfy the imposed constraints. We give the details of the CBS algorithm and its recent enhancement in Section 2.3.4. In addition to that, CBS has two variations that solve the MAPF optimally. Iterative Deepening CBS (ID-CBS) [92] instantiates the iterative deepening A* [93] on the high-level search of CBS. This adaptation allows CBS search to continue running without memory-out failure due to a large number of high-level search nodes. Meta-Agent CBS (MA-CBS) [94] inherits a similar idea from independence detection, which groups multiple agents into a sub-problem in the high-level search of CBS. In general, this approach helps to reduce the branching factor of the CBS search, but how to more effectively subgroup agents is a challenging task. Unlike the joint-state search algorithms, the efficiency of CBS algorithms is dependent on the specific conflicts that occurred in the current plan. Although there are various recent enhancements to joint-state search algorithms, the CBS family algorithms remain state-of-the-art for search-based MAPF algorithms.

Compilation-based algorithms are another family of algorithms that solve the MAPF problem optimally. These algorithms formalise the MAPF problem as a traditional optimisation problem such as Integer Linear Programming (ILP) problem, Boolean Satisfiability (SAT) problem, Constraint Programming (CP) problem etc, and efficiently solve them by using a well-implemented optimisation solver. For example, ILP has been used to encode a MAPF problem as a network flow problem with multiple commodities [95] and solved it with an ILP solver. Branch-and-Cut-and-Price (BCP) [39, 96] is another algorithm based on a widely-used ILP technique, branch and price, but generalises the idea of CBS to use a search algorithm to plan a single-agent path on the lower level and ILP solver to assign the paths and resolve the conflicts on the high level. In addition, MAPF can also be mapped to a SAT problem, such as MDD-SAT [97], which involves using Boolean variables to represent whether each agent is at each location at each timestep. SMT-CBS [98] further utilises the satisfiability modulo theories (SMT) to combine the resultant SAT-based algorithm with CBS. Finally, Lazy-CBS [41] replaces the high-level of CBS with a depth-first search and adapts the CP approach with no-good learning in order to avoid redundant searching efforts across different branches of CT. Overall, while some compilation-based approaches [95, 97] can outperform search-based algorithms on small maps, they can not scale well to larger maps due to the large number of Boolean variables required by their solvers. Other compilation-based approaches [39, 41, 96, 98] have improved performance, but many of them rely on or generalise the idea of CBS, making them non-dominated alternatives. Therefore, we focus on the search-based algorithm CBS in this research.

2.3.2 Bounded Suboptimal MAPF Algorithms

Bounded-suboptimal MAPF algorithms aim to find the solutions within a fixed bound of the optimal solution, which allows the trade-off between the solutions' quality and the query performance of the algorithms. Bounded-suboptimal MAPF algorithms often focus on the relative bound of the suboptimality, that is $\mathcal{BS} = \mathcal{O} \times (1 + \epsilon)$, where ϵ is the error bound, $\mathcal{BS}$ and $\mathcal{O}$ are the SIC of bounded-suboptimal and optimal solutions, respectively. In the existing literature, bounded-suboptimal MAPF algorithms are usually variants of optimal MAPF algorithms, we briefly review them below.

For joint-space A* search algorithms, Barer et al. [99] extends OD and EPEA* by incorporating a weighted A* search [100]. The evaluation function $f = g+h$ is modified to $f = g+h\times(1+\epsilon)$. By introducing a weight factor to the heuristic value h, the search can potentially bypass large sections of the state space that are less likely to yield feasible solutions (i.e., search nodes with low g-values but high h-values). Similar strategies are also employed in inflated M* [90] and inflated ODrM* [101], where ODrM* is a combination of M* and MA-CBS. For ICTS-based approaches, Aljalaud and Sturtevant [102] replace the best-first search at the high level with alternative strategies, such as All-Agent-Costs, where the approach increments the cost of all agents by one instead of increasing the cost of just one agent. Although the paper lacks experimental results, it highlights an intriguing method for finding suboptimal solutions within a fixed absolute bound (i.e., $\mathcal{BS} = \mathcal{O} + \epsilon$). For CBS-based approaches, Weighted CBS (WCBS) [99] extends the low-level A* search of CBS by using a weighted A* search to increase the heuristic value. Bounded CBS (BCBS) and Enhanced CBS (ECBS) [99] further improve both lower and high-level searches of CBS by employing Focal Search [103]. The focal search is a variation of A* search that aims to find bounded-suboptimal solutions. It utilises two lists of nodes: OPEN and FOCAL. The OPEN list is the same as in A* search, while the FOCAL list contains a subset of nodes from the OPEN list. Focal search uses two functions, f_1 and f_2, where the FOCAL list includes nodes from the OPEN list such that $f_1(n) \leq \epsilon \times f_{1_{min}}$ and f_2 is utilised to select which nodes in the FOCAL list to expand. By choosing an appropriate f_2, the search preferentially expands nodes with solutions within the error bound guaranteed by f_1. In addition to ECBS, Explicit Estimation CBS (EECBS) [104] replaces the high-level focal search of ECBS with Explicit Estimation Search [105], another bounded-suboptimal A* framework that better incorporates inadmissible heuristics. Flexible EECBS (FEECBS) [106] further improves the low-level focal search of EECBS by allowing the overall cost (instead of each agent's cost) of the paths to be bounded-suboptimal, providing flexibility in distributing the cost differently according to their needs. Lastly, aside from search-based MAPF algorithms, eSMT-CBS [107] and eMDD-SAT [108] extend the compilation-based algorithms SMT-CBS and MDD-SAT to return bounded suboptimal solutions.

2.3.3 Unbounded Suboptimal MAPF Algorithms

Unbounded suboptimal MAPF algorithms typically pay less attention to solution quality, but instead focus on finding feasible solutions fast. Popular ones in this area are prioritised planning algorithms, rule-based algorithms, and large neighbour search-based algorithms. We give more details on each type of algorithm in the following sections.

Prioritised planning algorithms, as introduced by [109], offer a collision-free solution for multi-agent pathfinding (MAPF) problems by assigning a specific order to prioritise agents. By adhering to this order, each agent plans a collision-free path while avoiding conflicts with the previously planned paths of higher-priority agents. The effectiveness of prioritised planning algorithms relies heavily on the priority order designated to each agent. This priority order can be pre-assigned utilising heuristic approaches [110], constraint programming (CP) solvers [111], and machine learning methods [112]. Alternatively, the priority order can be determined during execution using search-based algorithms [113], or partially planned for distinct sub-segments of paths [114]. Though prioritised planning algorithms often run efficiently and scale effectively to MAPF instances with a large number of agents, a significant drawback of this approach is the lack of completeness guarantees, i.e., prioritised planning algorithms may not be able to find a solution for a MAPF instance, even when one exists.

Rule-based algorithms typically solve the MAPF problems in near polynomial time by establishing pre-determined movement rules for agents. One popular way is to convert the map into a graph with predefined routes between start and destination of all agents, ensuring a MAPF solution is explored only within this reduced graph. Numerous approaches have been proposed, demonstrating that such a graph can be constructed as an acyclic graph (i.e., tree) [115], a bi-connected graph containing two more vertices than the number of agents [116], and a graph representing the special topology of the maps [117]. On the other hand, MAPF instances can also be solved using predefined agent operations, such as “Slide” [118] (under certain assumptions) and “Push and Swap” [119, 120]. Variants of “Push and Rotate” include ”Push and Rotate” [121] and “Push, Swap and Wait” [122]. Due to the popularity and superiority of “Push and Swap”, this technique has been further enhanced by combining it with prioritised planning in a method called Priority Inheritance with Backtracking (PIBT) [42, 123, 124]. More recently, Lazy Constraints Addition Search (LaCAM) [40] has improved upon PIBT by employing it to guide exhaustive searches. In summary, rule-based algorithms can run extremely fast (near polynomial time) on even very challenging MAPF instances. However, the solutions they find are often substantially inferior compared to other MAPF algorithms.

Large Neighbour Search-based algorithms employ a well-established optimisation strategy known as Large Neighbour Search (LNS). Within the context of the MAPF problem, LNS can be understood as a framework in which algorithms begin with a current plan containing paths for all agents. In each iteration, a subgroup of agents is selected to replan their paths to improve the current plan. This process is repeated until a timeout occurs. The effectiveness of LNS algorithms relies on the subgroup of agents selected in each iteration, which can often be achieved using heuristic approaches [43, 125] or machine learning approaches [126]. The overall LNS strategy can be applied to enhance solution quality [125], given a feasible solution provided by the suboptimal MAPF algorithms introduced. Alternatively, it can be adapted to repair an infeasible plan [43], given any unfinished plan returned by any MAPF algorithms introduced.

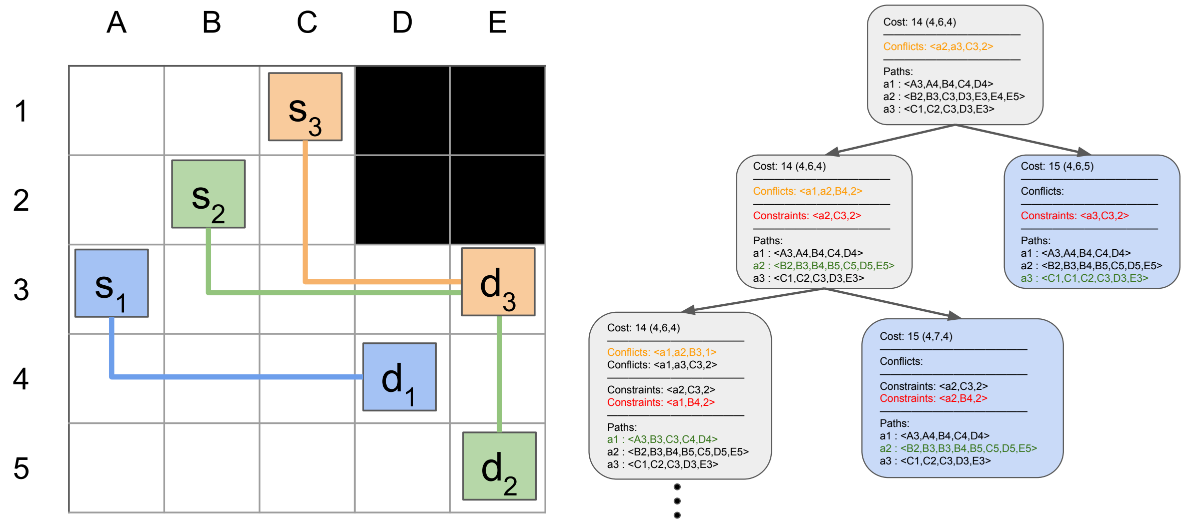
The combination of both approaches led to the development of a solver [127], winning the 2020 Flatland challenge [128]. Not only is LNS efficient and effective, but it also provides the flexibility to be combined with state-of-the-art MAPF algorithms in the field.

2.3.4 Conflict-Based Search and Its Enhancements

Conflict-Based Search (CBS) [33] is a state-of-the-art optimal algorithm for solving MAPF. CBS runs a two-level search. The high level of CBS focuses on a pair of agents that have at least one conflict with each other and resolves the conflict by adding constraints. This process involves building a binary tree called the Constraint Tree (CT). Each high-level CBS node N is a CT node, which contains:

- a set of constraints $N.constraints$, in which each constraint $\langle a_i, v_i, t\rangle$ (resp. $\langle a_i, v_i, v_j, t\rangle$) prohibits agent a_i from visiting vertex v_i (resp. edge e_{v_i,v_j}) at timestep t;
- a set of paths $N.\mathcal{P}$ (one for each agent), in which each path $N.\mathcal{P}(a_i)$ is a cost-minimal path for agent a_i that satisfies $N.constraints$ without considering other agents;
- a set of conflicts $N.conflicts$, where each conflict is either a vertex $\langle a_i, a_j, v_i, t\rangle$ or edge conflict $\langle a_i, a_j, v_i, v_j, t\rangle$ between $N.\mathcal{P}(a_i)$ and $N.\mathcal{P}(a_j)$; and
- a cost $N.cost$, which is the SIC of $N.\mathcal{P}$.

To find a conflict-free solution that minimises the SIC, CBS searches in a best-first-search manner and maintains a queue to prioritise the CT nodes using their costs $N.cost$. Initially, the priority queue contains a root CT node with an empty set of constraints, and each path $P \in N.\mathcal{P}$ is an optimal path while ignoring other agents. Whenever CBS expands a CT node N, it selects a conflict between a_i and a_j from $N.conflicts$ and resolves it by splitting N into two child CT nodes. In each of the child CT nodes, CBS adds an additional constraint that prohibits one of the agents from visiting the contested vertex or edge at timestep t. Since the path of a_i (or a_j) no longer satisfies the constraints of the child CT node, CBS calls a low-level solver to replan the path by using a time-space A* search [114]. Once replanned, the conflicts of the child CT node are updated, and all other paths in $N.\mathcal{P}$ remain the same. The search continues by inserting the child CT nodes to the queue and terminates when it selects for expansion a CT node N that has no conflicts (i.e., $N.conflicts = \emptyset$). The current $N.\mathcal{P}$ is a cost-minimal solution, as CBS guarantees to explore both ways of resolving each conflict. Figure 2.10 (i) shows an example of a MAPF instance. We show a CT of the CBS search in Figure 2.10 (ii), where the root CT node contains the shortest path for each agent while ignoring the paths of the other agents (as shown in Figure 2.10 (i)). In each of the CT nodes, we show the selected conflict in yellow, as well as the constraints added in red to resolve the conflict of the parent



(i)

(ii)

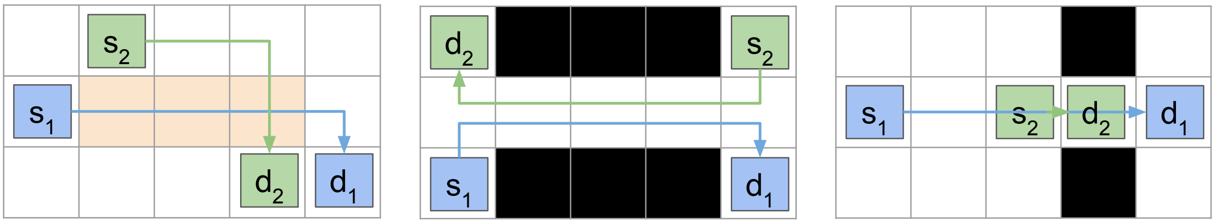
FIGURE 2.10: (i) A MAPF instance with three agents. (ii) An example of a CT for the MAPF instance in (i), In each CT node, we show the selected conflict (yellow), the newly added constraints (red), and the modified path (green). The CT nodes that contain feasible solutions are shown in blue.

CT node. Additionally, we highlight the replanned path resulting from the new constraints in green. Note that, although the CBS search generates CT nodes with conflict-free solutions as shown in blue, it needs to keep splitting the CT node located at the bottom left in order to prove optimality (i.e., until the minimal value in the OPEN list is no less than the best solution cost found so far).

Prioritising and Bypassing Conflicts: Recall that CBS needs to select a conflict from $N.conflicts$ when generating the child CT nodes. Each split operation is costly and increases the branching factor of the CBS search. In order to select the most promising conflict to resolve, Boyarski et al. [129] propose to *Prioritise conflicts* by sorting $N.conflicts$ into three categories:

- Cardinal conflict, iff replanning the path for any agent in the conflict increases $N.cost$;
- Semi-cardinal conflict, iff replanning the path for one involved agent increases $N.cost$ while others do not;
- Non-cardinal conflict, iff replanning the path for any agent involved in the conflict does not increase the $N.cost$;

CBS builds MDDs to classify conflicts. A Multi-value Decision Diagram (MDD) [91] MDD_i for an agent a_i in a CT node N is a Directed Acyclic Graph (DAG) which compactly stores all cost-minimal paths that satisfy the constraints $N.constraints$. Let us assume the cost of $N.\mathcal{P}(a_i)$ is c_i, the MDD_i has $c_i + 1$ levels. For each level t from 0 to c_i, MDD_i contains nodes that correspond to all possible locations of agent a_i at timestep t when agent a_i follows a path of



(i): Rectangle Conflicts (ii): Corridor Conflicts (iii): Target Conflicts

FIGURE 2.11: Examples of three types of symmetric conflicts between agents a_1 and a_2.

cost c_i that satisfies $N.constraints$. If MDD_i has only one MDD node (v, t) at level t, we call this node a *singleton* and all shortest paths of agent a_i must go through vertex v at timestep t. e.g., the source and destination are singletons, indicating that agent a_i occupies the vertex s_i and d_i at timestep 0 and c_i, respectively. Building MDD_i for an agent a_i is simple. We run a breadth-first search from the source s_i to explore the nodes that satisfy the constraints $N.constraints$ within cost c_i. Once the search is finished, MDD_i only records the partial DAG that reaches destination d_i. To classify the conflicts, a vertex conflict $\langle a_i, a_j, v_i, t\rangle$ is cardinal iff the MDDs of a_i and a_j have singletons at depth t, and an edge conflict $\langle a_i, a_j, v_i, v_j, t\rangle$ is cardinal iff both MDDs have singleton at depth t and $t - 1$. The semi/non-cardinal conflicts can be identified analogously. CBS is a best-first search that finds optimal solutions by progressively pushing the minimal lower bound. Therefore, the conflicts that increase the $N.cost$ are preferably resolved, i.e., CBS prefers to select a cardinal conflict first, then a semi-cardinal conflict and finally a non-cardinal conflict.

During node splitting in the CBS algorithm, another way to improve the search efficiency is to *bypass conflicts* by modifying the chosen path of one of the agents [129]. Given a CT node N and its constraints $N.constraints$, a path P_i is a valid bypass for agent a_i, iff (i) P_i has the same source and destination of a_i; (ii) P_i is a cost-equivalent path of $N.\mathcal{P}(a_i)$ which satisfies $N.constraints$; and (iii) replacing $N.\mathcal{P}(a_i)$ with P_i reduces the total number of conflicts (i.e.,$|N.conflicts|$) of N. CBS finds bypasses when generating child CT nodes. Recall that CBS selects a conflict between two agents a_i and a_j, and each child CT node replans a path P_i (resp. P_j) for agent a_i (resp. a_j) to resolve the conflict. If the replanned path P_i (or P_j) is a valid bypass, we replace the path of a_i (i.e., $N.\mathcal{P}(a_i)$) with P_i and remove the generated child CT nodes without splitting N. Identifying a bypass can resolve a conflict without branching, which reduces the size of CT.

Symmetry Reasoning: For a pair of agents that are currently in conflict, it is possible that all cost-minimal paths of two agents are colliding in the same area. Although adding the standard (vertex or edge) constraints will eventually resolve the conflict, the size of CT can grow exponentially, which leads to timeout or memory out failures of CBS. We refer to this type of conflict as symmetric conflict. Symmetry reasoning is a family technique that resolves

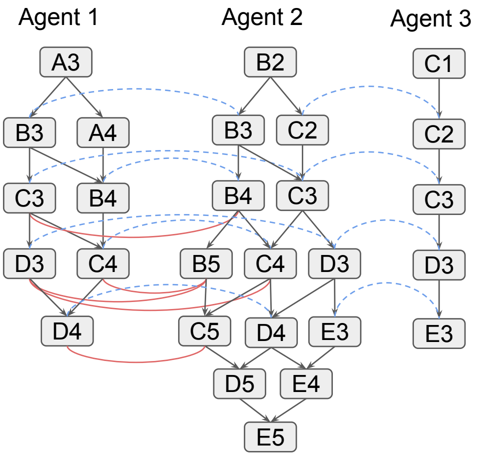


Figure 2.12: Examples of MDDs for three agents shown in Figure 2.10 (i), as well as the results of mutex propagation between agent a_2 with agent a_1 and a_3. The initial and propagated mutexes are shown in dashed blue arcs and red solid arcs, respectively.

symmetric conflicts in a single split, which dramatically improves the success rate of CBS. Li et al. [130, 131] propose a handcrafted approach that categorises the symmetric conflicts into threes types:

- Rectangle conflicts, where all cost-minimal paths of two agents collide in a rectangle area. Fig 2.11 (i) shows an example of a rectangle conflict, vanilla CBS resolves this conflict by, at least, expanding 3 nodes. However, with the width and length of the rectangular area increasing, the number of nodes expanded by CBS grows exponentially.
- Corridor conflicts, where two agents traverse through a corridor from opposite directions. Fig 2.11 (ii) shows an example of a corridor conflict with a length equal to 4, vanilla CBS resolves this conflict by expanding $2^{4+1} = 32$ nodes.
- Target conflicts, where one agent traverses through the other agent's target vertex. Fig 2.11 (iii) shows an example of a target conflict, where the distance between s_1 and g_2 equal to 3, vanilla CBS resolves the conflict by expanding 3 nodes.

For each type of symmetric conflict, Li et al. [130, 131] propose a customised algorithm to efficiently detect and classify it into three categories discussed before (e.g., cardinal, semi-cardinal and non-cardinal). In addition, a symmetry-breaking constraint is designed to enable CBS to resolve the conflict in a single split while preserving optimality. Later, Li et al. [34] extend the concept of symmetric conflicts to include (i) the irregular-shaped rectangle conflict, which is a generalisation of the rectangle conflict, and (ii) the pseudo-corridor conflict, which occurs in a non-corridor region but behaves like a corridor conflict, as well as (iii) the target corridor conflict, which arises when the target of an agent is located inside a corridor. For more details, we refer the reader to the paper [34].

Mutex[2] propagation [35, 132] is another approach that automatically identifies and resolves symmetric conflicts. Originally, mutex propagation is a popular technique used in AI planning, such as planning graph [133], state-space planner [134], and improving SAT-based planner [135]. Like many constraint propagation techniques, mutex propagation finds incompatible nodes between the MDDs of two agents. Given MDDs for two agents, mutex propagation finds two types of mutexes:

- Initial mutexes: a pair of MDD nodes/edges is an initial mutex iff these two MDD nodes/edges correspond to a vertex/edge conflict at the same level t.
- Propagated mutexes: a pair of MDD nodes (resp. edges) is a propagated mutex iff they are at the same level t and all pairs of their parent MDD edges (resp. nodes) are either initial mutex or propagated mutex.

A pair of MDD nodes is *mutex* if they are either initial or propagated mutex. In general, the initial mutexes are detected first and then propagated through MDD to find the propagated mutexes. Many existing algorithms [35, 136] can detect mutexes between MDDs. We omit the details of such algorithms. Given two MDD nodes at the same level that are mutex, we have the following:

Property 2.5. *Iff two nodes from different MDDs at the same level are mutex, there exists no pair of conflict-free paths that traverse through the two nodes and reach their destination locations at their individual minimum cost [35].*

Given the MDDs of two agents, Zhang et al. [35] utilise mutex propagation to identify cardinal symmetric conflicts. Specifically, a pair of agents has a cardinal symmetric conflict if all pairs of MDD nodes are mutex at the same level (i.e., there exist no pair of conflict-free paths for both agents to reach destinations). In this case, the algorithm returns two sets of MDD nodes to generate symmetric-breaking constraints. Each set consists of the MDD nodes of one agent that are mutex with the nodes of the other agent. Although mutex propagation can automatically identify and resolve symmetric conflicts, it detects only cardinal conflicts and requires higher runtime overhead than handcrafted approaches. Figure 2.12 shows an example of mutex propagation of MDD_2 with MDD_1 and MDD_3 for three agents shown in the Figure 2.10 (i). The initial mutexes and propagated mutexes are shown in blue dotted arcs and solid red arcs, respectively. Although the current path of a_2 and a_3 are in conflict, the mutex propagation can not identify any symmetric conflict, because there exists a pair of conflict-free paths between a_2 and a_3.

High-level Heuristic (WDG): So far, CBS prioritises the CT nodes using $N.cost$. However, like many other A* searches, the performance of CBS can be significantly improved by using an

[2] Mutex is a short term for mutual exclusion.

admissible heuristic h, which prioritises CT nodes based on $f = N.cost + h$. The first high-level heuristic of CBS is introduced by Felner et al. [36] which focused on the pairs of agents with cardinal conflicts (i.e., resolving such conflicts must increase the costs of child CT nodes). Later, Li et al. [37] improved and extended this heuristic considering all pairs of agents that are in conflict. Among many heuristics proposed in [37], we explain the leading heuristic, Weighted Pairwise Dependence Graph (WDG) heuristic.

In order to compute the heuristic for a CT node N, WDG considers all pairs of agents that are currently in conflict. For each such pair of agents (a_i, a_j), WDG takes the paths and constraints of a_i and a_j in a CT node N and runs a sub-CBS search to solve them as a sub-instance. Completely solving the sub-instance may be costly and easily dominates runtime. Therefore, each sub-CBS solver is given a node limit $|N|$, which only allows the solver to expand at most $|N|$ CT nodes. When the sub-CBS search concludes, it is easy to see that the increase of the minimal f-value in the OPEN list Δ_{ij} is a valid lower bound for agent pairs (a_i, a_j). To further consider the intersection of pairs of agents, WDG builds a weighted pairwise dependency graph $G_D = (V_D, E_D, W_D)$ for these agent pairs whose $\Delta_{ij} > 0$. Each vertex $v_i \in V_D$ indexes an agent a_i, each edge $e_{v_i v_j} \in E_D$ corresponds to an agent pair (a_i, a_j), and $W_D : E \rightarrow \mathcal{D}$ is a weight function that maps each edge $e_{v_i v_j} \in E_D$ to $\Delta_{ij} \in \mathcal{D}$ as edge weight. The graph G_D is used to create an integer program to minimise $\sum_i x_i$ subject to $\wedge_{ij} x_i + x_j \geq \Delta_{ij}$, where each x_i represents the increase in length of the current path for agent a_i. The optimal value of this integer program is an admissible heuristic for CT node N. Although computing WDG requires building G_D for each node expanded, most of the edges in G_D can be inherited from the parent CT node.

Though fast and effective, WDG as well as other existing heuristics [36, 137] (except for the one introduced below) compute the heuristics only by considering the pairs of agents that are in conflict. Recently, Mogali et al. [138] proposed a Lagrangian-Relax-and-cut-based (LR) heuristic that reasons about conflicts among groups of three agents. It shows promise that reasoning beyond pairs of agents can generate better heuristics. However, due to the large runtime overhead of the LR heuristic, (i) they apply the LR heuristic only at the root CT node, (ii) they have to limit the maximum cost of the paths, and (iii) the overall speedup is very limited (e.g., within the given runtime limit, they do not solve more instances than the existing algorithm).

Chapter 3

Fast Optimal and Bounded Suboptimal Euclidean Pathfinding

3.1 Overview

In this chapter, we present our techniques for improving the pathfinding queries in Euclidean space. Our approach combines the strengths of two recent pathfinding techniques: Polyanya [14], an online mesh-based planner, and Compressed Path Databases (CPDs) [26, 27], a family of preprocessing-intensive speedup techniques developed for grids and spatial networks. We provide two versions, one designed for finding optimal paths, and a second for bounded suboptimal paths. Like many Euclidean pathfinding algorithms, both use a two-step approach involving offline preprocessing followed by online search. In broad strokes:

- During the offline phase, we preprocess the input mesh to extract a graph of "interesting" points. We then preprocess the graph to create a CPD: an auxiliary data structure that stores compressed all-pairs data and, which can be used to efficiently extract optimal paths between any pair of "interesting points" p_i and p_j.
- During the online phase, we connect the source and destination points to the "interesting points" graph. We use the CPD to identify candidate paths. In the optimal algorithm, we consider paths from each of the $|V_s|$ outgoing successors of the source s to each of the $|V_d|$ incoming successors of the destination d. In the suboptimal algorithm, we only consider paths from the nearest "interesting point" to the source to the nearest "interesting point" to the destination.

For the optimal approach, because each candidate path is a feasible solution, our approach can provide strong anytime performance and it guarantees to return the optimal path after

Symbol	Description
s	The source of a pathfinding query.
d	The destination of a pathfinding query.
p_i	A point in arbitrary Euclidean space.
v_i	A vertex on a polygonal obstacle.
I	A contiguous interval (i.e., segment) of an edge of the mesh
c_i	The centroid of a circle.
δ	The radius of a circle.
ϵ	The error bound for bounded suboptimal search.
P	A path that consists of a sequence of points $\langle p_1$,$p_2, \cdots, p_k\rangle$.
$\Sigma\|P\|$	The cost of a path P.
$\oplus$	A concatenation operator that concatenates two paths P_i and P_j.
$sp(s, d)$	The shortest path from s to d.
$ed(p_i, p_j)$	The Euclidean distance between two points p_i and p_j.
$fm(p_i, p_j)$	A function that extracts the *first-move* on $sp(p_i, p_j)$ using CPD where p_i and $p_j \in$ CPD.
$cpd(p_i, p_j)$	The shortest path $sp(p_i, p_j)$ extracted using CPD where p_i and $p_j \in$ CPD.
$lb(p_i, p_j)$	A lower bound on the shortest distance between p_i and p_j, i.e., $lb(p_i, p_j) \leq \Sigma\|sp(p_i, p_j)\|$.
$bsp(s, d)$	A bounded-suboptimal path between s and d, i.e., $\Sigma\|bsp(s, d)\| \leq \Sigma\|sp(s, d)\| + \epsilon$.

Table 3.1: Summary of the notations used in this chapter

considering at most $|V_s| \times |V_d|$ possible paths, where $|V_s|$ and $|V_d|$ correspond to the number of convex vertices visible from source and destination, respectively. For the suboptimal approach, the whole pathfinding process is completed by considering only one path instead of $|V_s| \times |V_d|$ paths, so it is very fast.

We give a complete description of the new algorithms and a number of additional enhancements that can speed up the optimal search. We then demonstrate effectiveness in a range of experiments: on maps from real games and in comparison to a range of leading Euclidean pathfinding algorithms, both optimal and bounded suboptimal, appearing in the recent literature. For optimal path construction, we show that the new method can be substantially faster: from a few factors to over one order of magnitude. For computing fast anytime solutions, and for solutions with bounded suboptimal costs, we show that the speed gains are even larger.

3.2 Preliminaries

In the Euclidean pathfinding problem, we are asked to find point-to-point paths in a continuous 2D workspace which contains polygonal obstacles in fixed positions. Any non-obstacle point from the workspace is a potential source or destination position, and the objective is to find an obstacle avoiding, distance minimum path, between pairs of points that are priori unknown. In this chapter, we follow the definitions and terminologies introduced in Section 2.1, and Table 3.1 summarises the symbols used in this chapter.

As mentioned earlier, our techniques combine the strength of two recent pathfinding techniques: Polyanya [14] and Compressed Path Databases (CPD) [26]. The details of the online pathfinding algorithm Polyanya as well as its underlying navigation mesh have been discussed in Section 2.1.4. On the other hand, CPD is an oracle-based algorithm that was originally designed for pathfinding in road networks, we have given the description of the algorithm in Section 2.2.3.

3.3 Optimal Search

Our first contribution is an algorithm for quickly finding optimal Euclidean paths. We examine the two components of offline preprocessing followed by path extraction.

3.3.1 Offline Preprocessing

We now describe the auxiliary data structures required by our new algorithm and the offline preprocessing step that constructs them. There are two main steps: constructing a graph of co-visible convex vertices and building a corresponding CPD. This phase takes as input a navigation mesh which can be constructed as described in [14].

3.3.1.1 Identifying Co-Visible Vertices

A variety of methods exist for generating a graph of co-visible vertices. All have worst-case upper bounds of $O(n^2)$ where n is the number of vertices in the planar environment. Faster performance can be achieved in practice by only considering and connecting convex vertices. Variations of this idea appear many times in the literature and under different names; e.g. Tangent Graphs [139], Silhouette Points [140] and Sparse Visibility Graphs [15].

We now propose a new efficient algorithm for computing such a Visibility Graph, in two dimensions, using the Polyanya path planner. The vertex set V of the visibility graph consists of all convex vertices of the obstacles. In Fig. 3.1, $\{\texttt{A}, \texttt{D}, \texttt{G}, \texttt{H}, \texttt{K}, \texttt{L}, \texttt{O}\}$ are convex vertices. Other obstacle vertices (e.g., `C`) cannot appear on any optimal path, and are dead-end vertices. Next, for each $v \in V$, we run a modified Polyanya search to find convex visible vertices from v. Specifically, we modify the Polyanya such that it only generates visible successors and the search runs in a depth-first search manner without using any heuristic. If a successor's interval contains a convex vertex v', we add an edge $e_{vv'} \in E$, where initially $E = \emptyset$. The cost of this edge is $ed(v, v')$. This algorithm has the quadratic worst-case but in practice runs much faster.

Consider Fig. 3.1 as an example and assume that the source node is `A`. The search starts by generating all the visible successors for the two adjacent polygons that contain `A`. It cannot expand further for the successors that are on the obstacles or map boundary (e.g., $([\texttt{A}, \texttt{B}], \texttt{A})$,

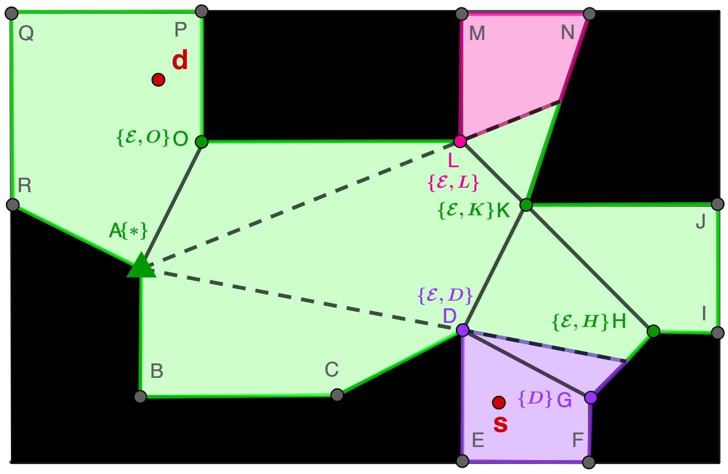


FIGURE 3.1: Green area corresponds to the area visible from the source node A. The first move on the optimal path from A to any node in the purple (resp. pink) area is D (resp. L).

Ordering	A	D	G	H	K	L	O
A	*	{$\mathcal{E}$,D}	D	{$\mathcal{E}$,H}	{$\mathcal{E}$,K}	{$\mathcal{E}$,L}	{$\mathcal{E}$,O}
D	{$\mathcal{E}$,A}	*	{$\mathcal{E}$,G}	{$\mathcal{E}$,H}	{$\mathcal{E}$,K}	{$\mathcal{E}$,L}	{$\mathcal{E}$,O}
G	D	{$\mathcal{E}$,D}	*	{$\mathcal{E}$,H}	{$\mathcal{E}$,K}	{$\mathcal{E}$,L}	{$\mathcal{E}$,O}

TABLE 3.2: First moves for A, D and G for the example of Fig. 3.1.

([B, C], A), ([P, O], A) and ([P, Q], A) etc.). Thus, the remaining visible successors are ([D, K], A), ([K, L], A) and ([A, O], A). When we expand the successor ([D, K], A), it finds two visible convex vertices D and K, and generates the visible successor ([H, K], A). Since it is a depth-first search, the successor ([H, K], A) is expanded which finds the visible convex vertex H, and generates the successors ([H, I], A), ([I, J], A) and ([K, J], A) (which are all ignored because they are either on obstacles or on the map boundary. Similarly, the successors ([K, L], A) and ([A, O], A) are processed and two visible convex vertices L and O are found by them, respectively. The search terminates after exploring the area visible from A (green area shown in Fig. 3.1). Thus, this Polyanya search adds edges from A to each of the convex visible vertices {D, K, H, L, O} into E along with their corresponding Euclidean distances.

We remark that a previous work [15] used a similar approach to find co-visible vertices but their searches are conducted using Anya [58]: an optimal any-angle path planner where polygonal obstacles are rasterised using a grid. Anya searches on a grid map and generates the successors by considering the adjacent grid row. The search space is explored row-by-row. On the other hand, Polyanya extends Anya in Euclidean space which searches on the convex polygons of mesh and explores polygon-by-polygon, hence improves the node expansions by a few factors and achieves up to one order of magnitude speed up [14]. In experiments, we compare against this method and we improve it using our more general mesh-based approach.

3.3.1.2 Building the CPD

Given the graph of co-visible nodes, we construct a corresponding CPD [26]: an all-pairs data structure that encodes the first move (equivalent first arc) on the optimal path from each node $s \in V$ to every other node $d \in V$.

First-Move Tables: As stated earlier, the first-move table of s stores the first-move symbol $fm(s, d)$ for every $d \in V$. When s and d are co-visible (i.e., $fm(s, d) = d$), in addition to storing the first-move d, we also store a redundant symbol $\mathcal{E}$ which represents that s and d are co-visible (i.e., a direct path from s to d exists). For example, for the first-move from A to D, we store D as well as $\mathcal{E}$. Another special symbol is (wildcard) "*" symbol which we add for table entries where $s = d$, since these entries will never be retrieved. We include the redundant and wildcards symbols because they substantially improve compression as shown in [79] and explained shortly. Table 3.2 shows all first moves for source vertices A, D and G in Fig. 3.1.

Compression: We compress first-move tables using run-length encoding (RLE) [27]. To improve RLE compression we apply several known enhancements [79]. First, we allow the wildcard symbol "*" to be compressed with any other preceding or subsequent symbol. Secondly, for table entries with redundant symbols, we choose the one that produces a longer run. For example, row A in Table 3.2 can compress into just two runs: 1D; 4$\mathcal{E}$ (cf. 3 runs if we choose $\mathcal{E}$ as the symbol for column D).

We use a Depth-First-Search (DFS) ordering of columns as suggested in [80]. In Table 3.2, the order of the columns is a DFS traversal order of the convex vertices appearing in Fig. 3.1 starting from A.

3.3.2 Online Search

CPDs can efficiently retrieve optimal paths when both source s and destination d are the vertices of the co-visible graph as discussed in section 3.3.1.2. One of the main challenges for pathfinding in Euclidean space is that s and d can be arbitrary (i.e., a priori unknown) locations on the map. To handle such cases we propose to first identify all graph vertices visible from s, denoted V_s, and all graph vertices visible from d, denoted V_d. We then extract a set of paths, from each $v_s \in V_s$ to each $v_d \in V_d$. Let $cpd(v_i, v_j)$ denote an optimal path from v_i and v_j (extracted via the CPD). If s and d are not visible to each other, the shortest path (i.e., the one with the shortest distance) sp from s to d is then

$$sp(s, d) = \text{argmin}\{\Sigma|\langle s, v_s\rangle \oplus cpd(v_s, v_d) \oplus \langle v_d, d\rangle| \mid v_s \in V_s, v_d \in V_d\} \tag{3.1}$$

Algorithm 1: End Point Search (EPS)

```
   Input: $s$:source, $d$:destination, CPD: compressed-path-database
   Output: an optimal path from $s$ to $d$
   Initialisation: $V_s \leftarrow \emptyset$, $V_d \leftarrow \emptyset$, $sp \leftarrow \langle\rangle$, $\Sigma|sp| \leftarrow \infty$
 1 $cur \leftarrow s$; $opp \leftarrow d$;
 2 while $search_s$ and $search_d$ are not exhausted do
 3     Let $v$ be the next visible vertex returned by $search_{cur}$;
 4     if $v = s$ or $v = d$ then // this implies $s$ and $d$ are co-visible
 5         return $\langle s, d\rangle$;
 6     if $v \neq \phi$ then // $search_{cur}$ is not exhausted
 7         for each $v' \in V_{opp}$ do
 8             $P \leftarrow \langle opp, v'\rangle \oplus cpd(v', v) \oplus \langle v, cur\rangle$;
 9             if $\Sigma|P| \leq \Sigma|sp|$ then
10                 $sp \leftarrow P$;
11                 set $\Sigma|sp|$ as the search bound for both $search_s$ and $search_d$;
12         $V_{cur} \leftarrow V_{cur} \cup v$;
13     $cur, opp \leftarrow opp, cur$;
14 return $sp$;
```

In Fig. 3.1, $V_s = \{\mathtt{D}, \mathtt{G}, \mathtt{H}, \mathtt{K}, \mathtt{L}\}$ and $V_d = \{\mathtt{A}, \mathtt{O}\}$ and the optimal path from s to d can be obtained by computing the pair-wise optimal paths for each $v_s \in V_s$, $v_d \in V_d$. As evident from Eq. (3.1), this basic algorithm extracts at most $|V_s| \times |V_d|$ candidate paths using the CPD and guarantees to return an optimal solution.

3.3.2.1 Incremental Exploration

We now consider a more sophisticated algorithm, End Point Search (EPS), that improves performance by reducing the number of pair-wise optimal paths that must be examined before guaranteeing optimality. Algorithm 1 provides an overview of the algorithm. Additional pruning rules and optimisations are discussed in Sections 3.3.2.2 and 3.3.2.3.

The key idea of Algorithm 1 is to incrementally explore the visible area from each of s and d, discovering visible vertices for s and d one by one. We propose to execute two best-first Polyanya searches, denoted $search_s$ and $search_d$, each of which is resumable, generates only visible successors at every expansion step and returns visible vertices as they are found. V_s and V_d record the visible vertices returned so far by $search_s$ and $search_d$, respectively. The shortest path sp and its length $\Sigma|sp|$ are initialised to be empty and infinity, respectively (we use $\leftarrow$ as an assignment operator and $=$ as an equality condition in the pseudocode).

The algorithm iteratively expands nodes from $search_s$ and $search_d$ in turn until both searches are exhausted (line 2). We use cur (resp. opp) to denote the current (resp. opposite) direction in which the search is expanding (line 1); i.e., if cur is source s then opp is destination d and vice versa. During each iteration, the algorithm incrementally progresses the relevant Polyanya search which returns the next visible vertex v (line 3). If the returned vertex is s or d, the search terminates because s and d are visible from each other and the optimal path is $\langle s, d \rangle$ (line 5). If the search is not exhausted (i.e., v is not empty), the algorithm updates the shortest path sp by considering all paths from visible vertices at the opposite end V_{opp} to this new vertex v. Specifically, for each $v' \in V_{opp}$, the algorithm uses the CPD to get the optimal path from v' to v and updates sp if the new path P is shorter than sp (lines 7 to 10). The search bound for both searches $search_s$ and $search_d$ is updated to be the shortest distance $\Sigma|sp|$ found so far (line 11). The new vertex v is added to the corresponding visible set V_{cur}. The two ends cur and opp are then swapped so that the search is alternated between $search_s$ and $search_d$ (line 13). When the while loop concludes, the algorithm returns the best found path sp.

Note that EPS is a bidirectional path extraction algorithm. In traditional bidirectional *search* algorithms [19], the search is guaranteed to meet in the middle and the challenge is to balance the searching effort between the two sides. In contrast, EPS is a bidirectional *path extraction* algorithm that only requires a bidirectional insertion to connect with the CPD nodes. Here, the main challenge is to avoid $|V_s| \times |V_d|$ total path extractions.

3.3.2.2 Pruning Candidate Paths

Recall that, in each iteration, the algorithm obtains a vertex v visible from cur (line 3). We can immediately discount *dead-end* vertices, and *non-turn* [141] vertices. A vertex is called a dead-end vertex if it cannot lead to anywhere else in the map, e.g., in Fig. 3.2, `E` is a dead-end vertex for the source s. A vertex v on a polygon ρ is a non-turn vertex for cur (e.g., s or d) if v is visible from cur and the ray shot from cur to v enters ρ from v – the non-turn vertex v does not allow turning *around* such obstacle. In Fig. 3.2, vertex `G` is visible from `s` but there is no turning point possible since the ray `sG` continues into the obstacle polygon. In contrast, `H` is not a non-turn vertex because the ray from s to it does not enter the polygon (and we can turn around this obstacle from `H`).

We can also prune a vertex v which cannot lead to a shorter path than the current bound, e.g. where $ed(s, v) + ed(v, d) \geq \Sigma|sp|$. For example in Fig. 3.2, we can safely ignore the vertex `K` as $ed(\mathtt{s}, \mathtt{K}) + ed(\mathtt{K}, \mathtt{d}) > \Sigma|sp|$, where $\Sigma|sp|$ is the length of the optimal path found so far (highlighted as red). Finally, $search_{cur}$ can terminate when the top of the OPEN list has an f value greater than $\Sigma|sp|$, since no path using this entry can be shorter than $\Sigma|sp|$.

We can avoid extracting paths for pairs ($v_s \in V_s$, $v_d \in V_d$) if they cannot lead to a shorter path than the current bound, i.e., $ed(s, v_s) + ed(v_s, v_d) + ed(v_d, d) > \Sigma|sp|$ since $ed(v_s, v_d)$ is a lower bound on the shortest path distance $\Sigma|cpd(v_s, v_d)|$. Similarly, we can prune vertex pairs (v_s, v_d) where the first move from either end is non-taut, i.e., string pulling results in a shorter path. For example, let w be the first move on the shortest path from v_s to v_d. If $\langle s, v_s, w \rangle$ is

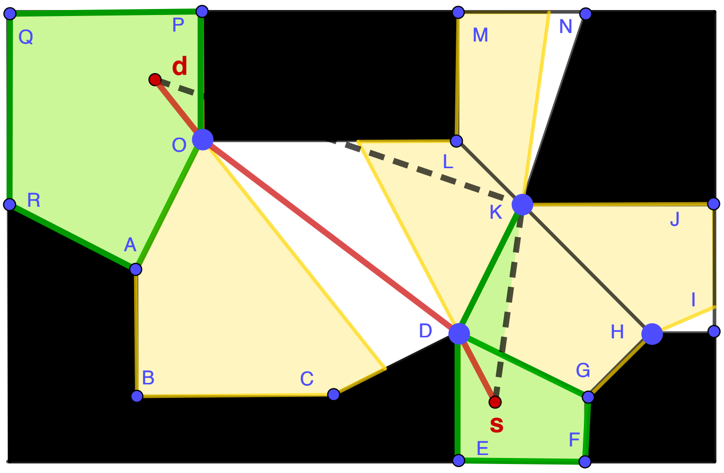


FIGURE 3.2: An example of End Point Search. The red lines show the optimal path. The area shown green or yellow corresponds to the space visible from s and d. The green area shows the space incrementally explored by Polyanya when $search_s$ and $search_d$ are both exhausted.

non-taut then it cannot be part of a shortest path. Similarly, we can prune the vertex pair if $\langle d, v_d, w' \rangle$ is not taut where w' is the first move on the shortest path from v_d to v_s. Consider the example in Fig. 3.2, the first move from `H` to `O` is `O` but $\langle$`s`, `H`, `O`$\rangle$ is non-taut, so we do not need to consider the pair (`H`,`O`) further.

3.3.2.3 CPD Cost Caching

In each iteration of the while loop, the algorithm uses the CPD to extract the paths between a vertex v and every $v' \in V_{opp}$ (line 8). We use the CPD to extract the optimal path *from* v' *to* v and, for each vertex v_k on the extracted path, we cache $\Sigma|cpd(v_k, v)|$, the shortest path distance from v_k to v. For a subsequent CPD path extraction, if the optimal path *from* v'' *to* v reaches the vertex v_k for which $\Sigma|cpd(v_k, v)|$ is cached, we can use the cached distance to get the path length from v'' to v. This simple caching strategy avoids unnecessarily using the CPD to extract the path that is already cached. Although the algorithm can cache $\Sigma|cpd(v_k, v)|$ for every $v \in V_s \cup V_d$, in our implementation, we only cache $\Sigma|cpd(v_k, v)|$ for the vertex v in the current iteration of the while loop and reuse the space in each iteration for the new v. This ensures that the caching uses $O(1)$ space for each vertex, i.e., the total space used by the caching is $O(|V|)$ where $|V|$ is the number of nodes in the co-visible graph. Moreover, we observed that this caching is already quite effective and caching the distance for each $v \in V_s \cup V_d$ does not result in a significant further improvement in query performance.

3.3.2.4 Putting it All Together

End Point Search gives us an incremental exploration of the pairs of endpoints on the CPD, which is reduced by pruning and improved by caching CPD distances, eventually leading to an optimal path. Next, we prove the correctness of our approach.

Theorem 3.1. *Algorithm 1 returns an optimal path from s to d*

Proof. If s and d are visible to each other, one of the two Polyanya searches $search_s$ and $search_d$ will discover this returning $\langle s, d \rangle$ at line 5. Otherwise, the shortest path must contain at least one vertex v_s visible from s (i.e., $v_s \in V_s$) and at least one vertex v_d visible from d (i.e., $v_d \in V_d$). Algorithm 1 explores all paths through $v_s \in V_s$ and $v_d \in V_d$ (see Equation (3.1) at the beginning of Section 3.3.2) except: (i) those vertices that are dead-end, non-turn or have f-values bigger than current distance $\Sigma|sp|$ (thus can never be part of the optimal path); (ii) and the vertex pairs (v_s, v_d) where the shortest possible path through them, $\langle s, v_s \rangle \oplus cpd(v_s, v_d) \oplus \langle v_d, d \rangle$, is either non-taut or longer than the current distance $\Sigma|sp|$. Hence, the returned path is optimal. □

Example 3.1. Fig. 3.2 gives an example of the algorithm in action. End Point Search (EPS) starts the $search_s$ and returns a visible vertex `D` (line 3). The non-turn vertex ($\{\texttt{G}\}$) and dead-end vertices ($\{\texttt{E}, \texttt{F}\}$) of the polygon containing s are visited by Polyanya but are pruned as explained in Section 3.3.2.2. Then, the search is swapped and the $search_d$ returns a visible vertex `O` after filtering $\{\texttt{A}\}$ (a non-turn vertex) and $\{\texttt{P}, \texttt{Q}, \texttt{R}\}$ (dead-end vertices). The CPD is used to extract the path from `D` to `O`, and the shortest path sp (shown red in Fig. 3.2) and search bound $\Sigma|sp|$ are updated accordingly (lines 7 - 11). In the next iteration, $search_s$ expands the successor ([`D`,`K`],s) and returns the next visible vertex `K`. However, the vertex `K` can be safely ignored by our distance pruning approach introduced earlier because the path through `K` (shown in broken lines) is longer than the current shortest path sp, i.e., $ed(s, \texttt{K}) + ed(\texttt{K}, d) > \Sigma|sp|$. After that, the $search_s$ and $search_d$ are both exhausted as the f-values of the rest of the successors in their respective queues are bigger than the search bound $\Sigma|sp|$, e.g., the successor ([`K`,`H`],`s`) is never explored by $search_s$ because its f-value $ed(s, \texttt{K}) + ed(\texttt{K}, d) > \Sigma|sp|$. Thus, the algorithm terminates and returns sp (the path shown in red).

3.3.3 Experiments

We run experiments on a variety of grid map benchmarks which are described in [142], including 373 game maps from four sets of maps: DAO (156), DA (67), BG (75), SC (75). All benchmarks are available from the HOG2 online repository.[1] We compare our algorithm with a range of competitors detailed below:

- **Polyanya**: [14] is a fast, optimal, online pathfinding algorithm on navigation meshes. The source code of Polyanya and input navigation mesh are from the publicly available repository.[2]

[1] https://github.com/nathansttt/hog2

[2] https://bitbucket.org/mlcui1

	#M	#Q	#V	#CV	Build Time		Raw Memory		CPD\\$\mathcal{E}$ Memory		CPD Memory	
					Avg	Max	Avg	Max	Avg	Max	Avg	Max
`DAO`	156	159k	1727.6	926.5	0.033	0.831	8.012	134.977	0.310	4.605	0.207	3.640
`DA`	67	68k	1182.9	610.8	0.006	0.048	2.244	20.611	0.115	0.484	0.063	0.254
`BG`	75	93k	1294.4	667.7	0.011	0.233	3.887	66.064	0.188	1.828	0.119	1.366
`SC`	75	198k	11487.5	5792.7	0.711	8.463	190.38	2202.23	3.615	19.493	2.325	14.075

Table 3.3: Total number of Maps (#M) and Queries (#Q), average number of vertices (#V) and convex vertices (#CV) in the maps, average and maximum building time in minutes, and average and maximum memory before compression (Raw memory), after compression without using the $\mathcal{E}$ symbol (CPD\\$\mathcal{E}$ memory) and after compression (CPD memory) in MB for the four benchmark suites.

- **ENLSVG**: (Edge-N-Level Spare Visibility Graph) [15] is an optimal, off-line pathfinding algorithm. The implementation of ENLSVG is taken from an online repository.[3]

- **Poly-ENLSVG**: is an improvement of the original ENLSVG algorithm which we improve by applying our Polyanya-based visible vertex retrieval approach (see Section 2.1) for the insertion phase of ENLSVG. Here, we prune the dead-end and non-turn vertices to further improve the performance.

- **SUB-N-T**: (N-level Subgoal graph) [59] is a suboptimal, off-line pathfinding algorithm. We run Theta-A* [60] on top of N-level subgoal graph, using the publicly available implementation.[4]

- **TRA***: (Triangulation Reduction A*) [68] is a anytime, optimal pathfinding algorithm that runs on navigation meshes.

For more details of the competitors, see section 2.1. All algorithms (including the competitor algorithms) are implemented in C++ and compiled with -O3 flag. The experiments are performed on a 2.6 GHz Intel Core i7 machine with 16GB of RAM and running OSX 10.14.6.

3.3.3.1 CPD Statistics

Table 3.3 shows the average and maximal size of CPD, and building time for the four benchmarks suites. Clearly, our CPDs are memory efficient and the compression reduces the size of first-move tables by up to two orders of magnitude. The tables have very small numbers of runs per entry and hence very fast lookup times. In terms of compression, we also observed that the $\mathcal{E}$ symbol helps to reduce the size of CPDs by 20% – 50% on average and, for some of the maps, we observed that the $\mathcal{E}$ symbol reduces the size by a few factors. In games, it is common to treat all the maps for one game together. Although the raw memory of the first move table of one

[3] `https://github.com/Ohohcakester`

[4] `http://idm-lab.org/anyangle`

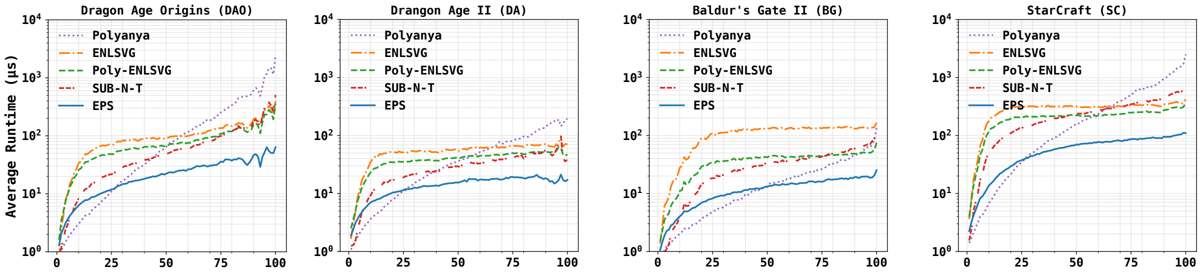


Figure 3.3: Runtime comparison on the four benchmarks. The x-axis shows the percentile ranks of queries in number of node expansions needed by A* search to solve them.

	Total		Poly-ENLSVG		EPS			
	$\lvert V_s \rvert$	$\lvert V_d \rvert$	$\lvert V_s \rvert$	$\lvert V_d \rvert$	$\lvert V_s \rvert$	$\lvert V_d \rvert$	#Paths	#FM
DAO	69.324	71.495	19.778	19.987	15.108	14.955	5.094	329.463
DA	46.171	45.707	13.202	12.922	10.779	10.755	2.470	144.872
BG	51.926	49.175	15.629	14.335	9.445	9.226	2.707	98.955
SC	180.013	178.874	45.889	45.356	29.819	29.707	4.028	976.194

Table 3.4: $|V_s|$ (resp. $|V_d|$) denotes the average number of vertices visible from s (resp. d) considered by an algorithm to obtain the results. Total includes all visible vertices for s or d without any pruning. For EPS, we also show the average number of path extractions (#Paths) and first move extractions (#FM) from the CPD.

map may be affordable, it may not be feasible to store the raw tables of all the maps of a game in the main memory (e.g., the sum of raw memory for SC benchmark maps is around 14GB in total). Our CPDs are cheap to build, and for most of the maps can be computed within a few minutes. Note that the CPDs are built on a 12 core Macbook Pro laptop and the performance would be better/worse if more/less processors are available.

3.3.3.2 Query Processing Time

In Fig. 3.3, we compare the query processing time for our approach against the competitors. We sort the queries by the number of node expansions required by the standard A* search to solve them (which is a proxy for how challenging a query is) and the x-axis corresponds to the percentile ranks of queries in this order. Fig. 3.3 shows that EPS significantly outperforms the competitors on all four benchmarks especially when the queries are more challenging. Note that the y-axis scale is logarithmic. EPS is around 2-4 times faster than SUB-N-T (which does not guarantee optimal solutions) and 2-5 times faster than Poly-ENSLVG. Polyanya is faster than EPS for the less challenging queries because, for such queries, s and d are close (and often visible from each other) and the dominant cost for EPS is the two incremental Polyanya searches from s and d. For challenging queries, EPS is more than an order of magnitude faster than Polyanya.

Table 3.4 reports the average number of the vertices visible from s and d expanded by Poly-ENLSVG and EPS after pruning non-turn and dead-end vertices. Both algorithms significantly

Algorithm	EPS		No Caching & Pruning		Cost Caching Only		Pruning Only		Visible First	
map	#FM	Time	#FM	Time	#FM	Time	#FM	Time	#FM	Time
DA0	329.46	23.90	23029.61	309.09	1981.00	50.34	363.92	24.51	335.01	29.67
DA2	144.87	14.18	6830.88	93.20	874.53	24.64	149.62	14.97	148.14	16.91
BG	98.95	12.29	2684.69	48.73	499.16	20.41	100.19	12.66	110.23	17.85
SC1	976.19	61.32	85617.37	1232.71	6298.06	177.68	994.95	62.37	1084.39	82.89

Table 3.5: Average query processing time (μs) and number of first move extractions (#FM) of our proposed EPS approach. We report results for each different pruning strategy.

reduce the number of visible vertices expanded. Since EPS makes use of the search bound $\Sigma|sp|$ to restrict the Polyanya search, it expands a smaller number of visible vertices than Poly-ENLSVG especially for BG and SC benchmarks. Also, note that the number of path extractions by EPS is much smaller than $|V_s| \times |V_d|$ since path pruning can avoid considering many of them. We remark that most of the first-move extractions (#FM) are incurred for non-taut pruning, e.g., to determine whether a path connecting $v' \in V_{opp}$ to $v \in V_{cur}$ is non-taut from the opposite end *opp*, we need to get the first-move v_i from v' to v and then check whether $\langle opp, v', v_i \rangle$ is non-taut or not. Similarly, to check whether the path is non-taut from the *cur* end, we need to extract the first-move v_j from v to v' and check whether $\langle cur, v, v_j \rangle$ is non-taut or not. Although this non-taut pruning requires first-move extractions, overall it improves the performance as it reduces the number of paths extracted (#Paths) by EPS (which in turn reduces the overall number of first-move extractions needed to extract the paths). Also, we found that our previous implementation of EPS reported in [45] applied non-taut pruning only from the opposite end (*opp*). In this extended version, we updated the implementation by also applying non-taut pruning from the current end (*cur*) which significantly reduced both the number of paths and first-moves considered by EPS, resulting in an improved performance.

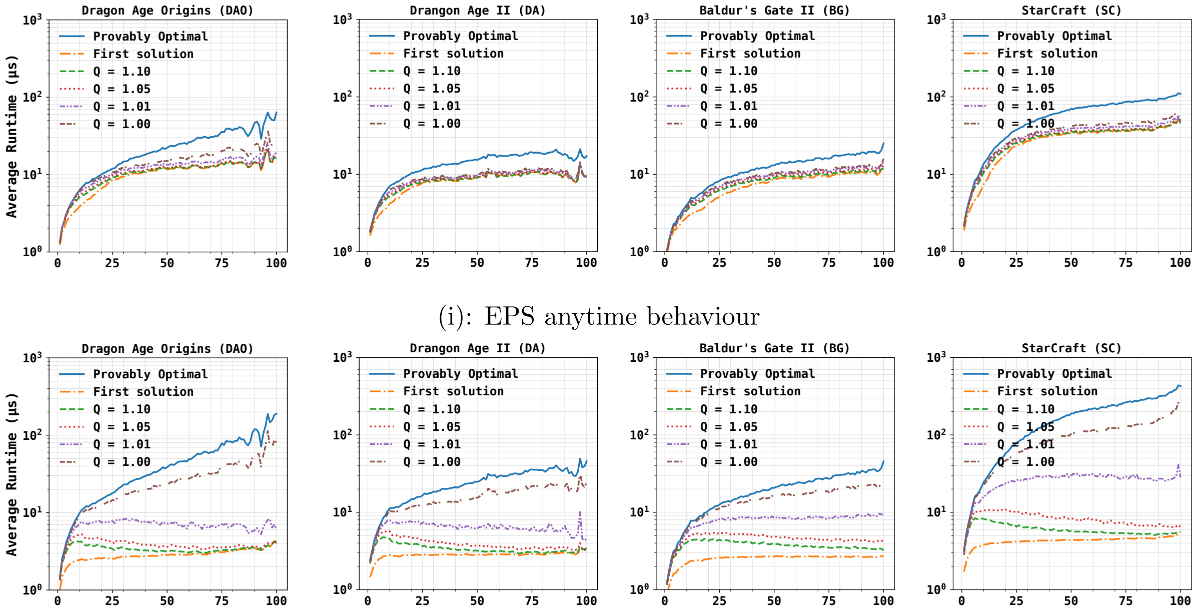
3.3.3.3 Pruning Strategies

In Table 3.5, we show the average number of first move extractions and query processing time over four benchmark suites for: our final algorithm (EPS); we omit both pruning and caching (No Caching & Pruning); we use only cost caching but not pruning (Cost Caching Only); we use only pruning but not cost caching (Pruning Only); and we apply both pruning and caching but we first retrieve all visible vertices w.r.t. s and d (Visible First) instead of incrementally exploring them. Clearly the reduction in search is significant and Pruning is the most important enhancement. Among the many ingredients of the Pruning techniques, we also observe that the non-taut path pruning is the most significant one. By performing non-taut path pruning on both ends, it allows the search to filter out most suboptimal candidate paths and quickly leads to the optimal one. On the other hand, Cost Caching Only essentially considers all the possible solutions and achieves a speedup vs. the baseline (i.e., No Caching & Pruning) by a few factors. Visible First requires the full insertions (i.e., finding all the visible vertices) which is time-consuming, thus runs slower than EPS.

3.3.3.4 Anytime Search

In time-constrained applications (e.g., computer games), anytime pathfinding is often desirable which returns a valid but potentially suboptimal path as soon as possible before progressively optimising it until an optimal path is found. This motivates us to consider EPS as an anytime search algorithm. We begin with evaluating the anytime behaviour of EPS. In Fig. 3.4 (i), we show the average runtime of EPS to find: the first valid path (shown as First solution); the first path with length within a certain factor Q of the optimal path length (shown as $Q = 1.xx$); and the guaranteed optimal path (i.e., when EPS terminates). Here, $Q = 1.00$ is the time taken by EPS until it happens to discover the optimal path (although it cannot terminate because it cannot yet guarantee the optimality of this path). Recall that EPS makes use of pruning techniques to eliminate the candidate paths that are found to be non-optimal. Although this improves the runtime of EPS to find the optimal path, it adversely affects the performance of EPS for anytime search (e.g., the first valid path is found later as several candidate paths may have been pruned). This further motivates us to consider another variation of EPS for anytime search, where we consider all candidate paths and only use cost caching to improve the efficiency. We denote this as EPS (Cost Caching Only). Fig. 3.4 (ii) shows anytime search for EPS (Cost Caching Only). It is clear that EPS (Cost Caching Only) demonstrates excellent anytime behaviour, e.g., it finds the first valid path within $5\mu s$ and a path with $Q = 1.10$ (a path with length at most 10% longer than that of the optimal path) within $10\mu s$. However, note that EPS (Cost Caching Only) understandably runs slower than EPS for the optimal path search (i.e., Provably Optimal) which demonstrates that the pruning rules provide a tradeoff between optimal search and anytime search.

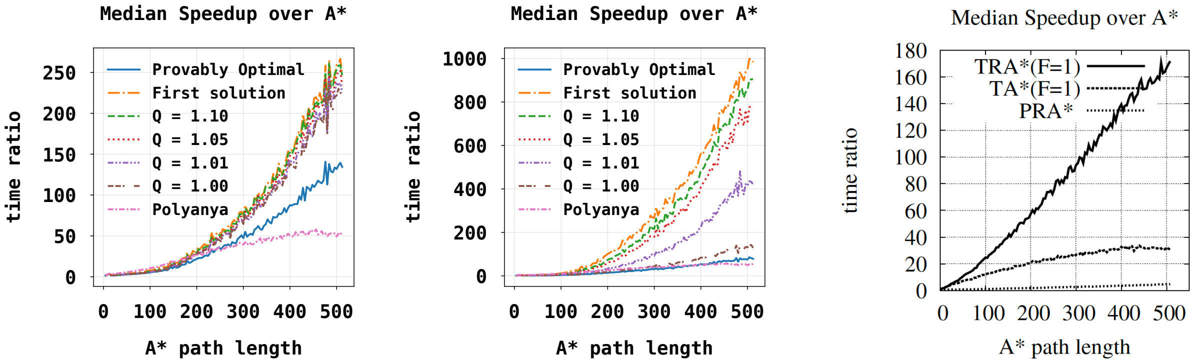
In Fig. 3.5 (i) and (ii), we show the speedup of anytime search compared to A* search for EPS and EPS (Cost Caching Only), respectively. Fig. 3.5 (iii) shows a graph reproduced from [68] showing similar comparison for TRA* anytime search, a popular mesh-based planner, which aims at finding the first solution fast. It can be seen that the speedup over A* search provided by EPS(Cost Caching Only) is significantly bigger than those achieved by TRA*. For example, to find the first solution for queries with path length around 500, EPS(Cost Caching Only) is around 1000 times faster than A* whereas TRA* is around 180 times faster than A*. EPS provides a bigger speed up over A* compared to TRA* for queries with longer paths, but a smaller speed up for the shorter queries. This is mainly because EPS focuses on finding the optimal solution fast and prunes many candidate paths which may result in a delay in finding the first solution.



(i): EPS anytime behaviour

(ii): EPS (Cost Caching Only) anytime behaviour

FIGURE 3.4: (i) EPS and (ii) EPS (Cost Caching Only) anytime behaviour. The x-axis is the same as in Fig. 3.3. The y-axis shows the average runtime when EPS finds the first path with length within a certain factor Q of optimal path length (i.e., 1.00, 1.01, 1.05 and 1.10). Q = 1.0 is the time when EPS happens to discover the optimal path, but cannot guarantee its optimality. The provably optimal path is the guaranteed optimal path at termination.



(i): EPS

(ii): EPS (Cost Caching Only)

(iii): TRA*

FIGURE 3.5: Speedup of (i) EPS and (ii) EPS (Cost Caching Only) over A* search for finding solutions of different quality on benchmark suite BG, and a reproduced graph for the same experiment for (iii) TRA* where ($F = 1$) represents the query performance that TRA* finds the first solution

.

3.4 Bounded Suboptimal Search

In this section, we present techniques to find a bounded suboptimal path. We focus on an absolute bound on the suboptimality, that is, $\Sigma|bsp(s,d)| \leq \Sigma|sp(s,d)| + \epsilon$ where ϵ is the error bound, $\Sigma|bsp(s,d)|$ is the length of the bounded suboptimal path between s and d returned by our algorithm and $\Sigma|sp(s,d)|$ is the shortest path distance between s and d. First, we show how to modify EPS to opportunistically terminate early when it finds a bounded suboptimal path. We then describe a search free method that builds a bounded suboptimal path using a larger CPD.

3.4.1 Bounded Suboptimal EPS Search

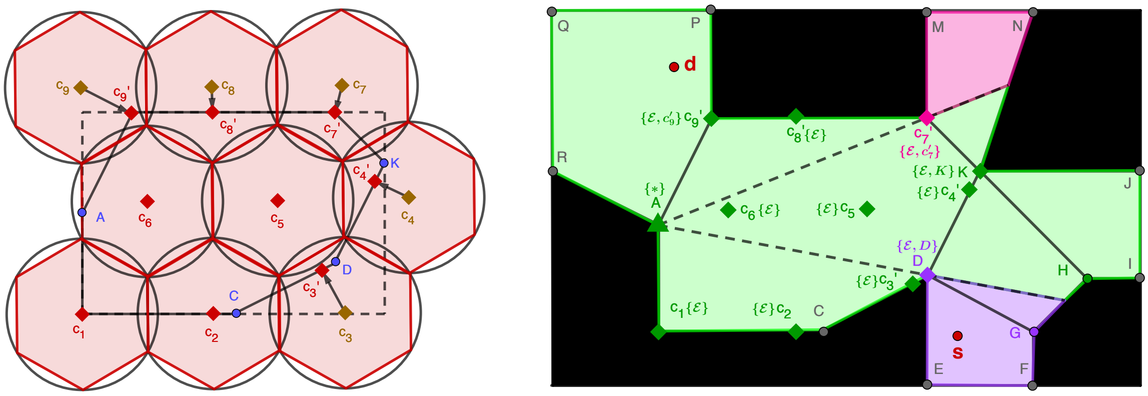
For efficient suboptimal search, it is important that the search can be terminated as soon as we find a candidate path that is within the bound $\Sigma|sp(s,d)| + \epsilon$. Therefore, each candidate path found by the EPS becomes important which motives us to consider EPS (Cost Caching Only) for the suboptimal search; i.e., we consider all possible solutions found by EPS without applying our pruning rules and apply only the cost caching to improve the performance. Let $lb(s,d)$ be a lower bound on the length of the shortest path between s and d. EPS can terminate as soon as it finds a path $bsp(s,d)$ such that $\Sigma|bsp(s,d)| \leq lb(s,d) + \epsilon$. Note that, given the lower bound, EPS can also be used to obtain a path with a relative bound by terminating EPS when $\Sigma|bsp(s,d)| \leq lb(s,d) \times (1+\epsilon)$. One can use the Euclidean distance to obtain $lb(s,d)$ but it is not likely to be very effective. We propose a more effective lower bound as described below.

Recall that EPS uses CPD to compute the shortest distance between $v_s \in V_s$ and $v_d \in V_d$ denoted as $\Sigma|cpd(v_s,v_d)|$ where v_s is visible from s and v_d is visible from d. Using the triangle inequality, it is easy to show that $\Sigma|cpd(v_s,v_d)| - ed(s,v_s) - ed(d,v_d) \leq \Sigma|sp(s,d)|$, i.e., $\Sigma|cpd(v_s,v_d)| - ed(s,v_s) - ed(d,v_d)$ is a valid lower bound on the shortest distance. Whenever EPS uses CPD to compute the shortest distance between a pair of vertices (at line 8 in Algorithm 1), we compute the lower bound for the pair. The algorithm maintains $lb(s,d)$ to be the largest lower bound computed so far. Specifically, we initialise the $lb(s,d)$ to be the Euclidean distance $ed(s,d)$, let V_{sd} denotes the set of (v_s, v_d) pairs for which EPS has computed the shortest distance using CPD. Then, $lb(s,d) = max(lb(s,d), argmax_{(v_s,v_d)\in V_{sd}}\ \Sigma|cpd(v_s,v_d)| - ed(s,v_s) - ed(d,v_d))$.

Consider the example of Fig. 3.6 (ii) and assume that EPS uses CPD to compute the shortest distances between the pair of vertices ($\mathtt{D}$,c'_9) and ($\mathtt{E}$,$\mathtt{P}$). Then, $lb(s,d) = \Sigma|cpd(\mathtt{P},\mathtt{E})| - ed(s,\mathtt{E}) - ed(d,\mathtt{P})$ because the pair ($\mathtt{E}$,$\mathtt{P}$) generates a better (larger) lower bound than that of ($\mathtt{D}$,c'_9). For the rest of the paper, we denote the bounded (S)uboptimal EPS as SEPS.

3.4.2 Centroid-based Path Extraction

Although our SEPS may terminate earlier to find a path within a given bound ϵ, it still requires searching for the endpoints and computing distances between multiple pairs of vertices using CPD, which may be inefficient. This motivates the Centroid-based Path Extraction (CPE) algorithm, a search-free algorithm that provides a bounded suboptimal path by using the CPD



(i)

(ii)

FIGURE 3.6: (i) shows an example of our hexagonal tiling approach for a given polygon, where c_1 - c'_9 (shown in red) are the centroids selected for building CPD; and (ii) shows a partial example with centroids created for central polygon according to (i). Green area corresponds to the area visible from the source node `A`. The first move on the optimal path from `A` to any node in the purple (resp. pink) area is `D` (resp. c'_7).

to extract the path between only a single pair of points. The key idea of this algorithm is to fill in the navigation mesh with a set of candidate nodes, called centroids, such that for every point p in the space there exists a centroid visible from p and within distance $\delta = \epsilon/4$. Then, first-move tables are created using these centroids which are compressed to obtain a centroid-based CPD. To compute the shortest path, we find a centroid c_s (resp. c_d) that is within distance δ from s (resp. d) and use the CPD to compute the shortest path between c_s and c_d. We show that the path $\langle s, c_s, c_d, d\rangle$ is within the absolute bound ϵ. The path is further refined using string pulling.

3.4.2.1 Building the Centroid-based CPD

We need to fill each polygon of the navigation mesh with centroids such that the circles centred at these centroids with radii δ cover every point in the polygon. The main objective here is to cover the polygon with a minimal number of such circles. This problem belongs to the class of covering problems, many of which are shown to be NP-Complete [143]. Next, we describe a greedy algorithm (Algorithm 2) based on hexagon tiling.

First, we create a set of candidate nodes CN which will be used for CPD construction. CN is initially empty and the algorithm iteratively accesses each traversable polygon ρ of the navigation mesh (line 1) to populate CN as follows. Let R be the minimum bounding rectangle of the polygon. We use hexagonal tiling to cover the rectangle R such that the circumradius of each hexagon (i.e., the radius of its circumscribing circle) is δ, e.g., see the hexagon tiling in Fig. 3.6 (i). Since these hexagons completely cover the polygon, their circumscribing circles also completely cover the polygon. For each hexagon, we insert its centroid c_i in CN if the centroid

```
Algorithm 2: Centroid-based CPD Construction
Input          : Navigation mesh
Output         : A Centroid-based CPD
Initialisation: $CN \leftarrow \emptyset$
1  for each traversable polygon $\rho$ of the mesh do
2      $R \leftarrow$ minimum bounding rectangle of $\rho$;
3      for each hexagon of the hexagonal tiling of $R$ do
4          $c_i \leftarrow$ centroid of the hexagon;
5          if $c_i$ is inside $\rho$ then
6              $CN \leftarrow CN \cup c_i$;
7          if the hexagon overlaps $\rho$ but $c_i$ is outside $\rho$ then
8              $c'_i \leftarrow$ closest point of $\rho$ from $c_i$;
9              $CN \leftarrow CN \cup c'_i$;
10     insert convex vertices of $\rho$ in $CN$ if not already present;
11 construct a CPD using $CN$ as candidate nodes and return;
```

is within the polygon ρ (line 6), e.g., in Fig. 3.6 (i), c_1, c_2, c_5, and c_6 are inside ρ and inserted in CN. If the hexagon overlaps ρ but its centre c_i is outside ρ, we find the closest point from c_i on ρ (denoted c'_i) and insert it in CN, e.g., c'_3, c'_4, c'_7, c'_8 and c'_9 in Fig. 3.6 (i) are inserted in CN. It is easy to show that the circles centred at these moved centroids still completely cover the polygon. Finally, all convex vertices of ρ are inserted in CN if not already present (line 10). In Fig. 3.6 (ii), `A`, `D` and `K` are inserted.

Once all traversable polygons are processed as described above, a CPD is constructed using the set of candidate nodes CN (line 11) by following the same procedure as described in section 3.3.1.2. E.g., in Fig. 3.6 (ii) $CN = \{c_1, c_2, c'_3, c'_4, c_5, c_6, c'_7, c'_8, c'_9, \mathtt{A}, \mathtt{D}, \mathtt{K}\}$, the vertex `C` is excluded as it is a non-convex vertex. Hereafter, we use "centroid" to refer to candidate nodes in CN. Note that this CPD stores first moves from each centroid to every other centroid in CN, and only the convex vertices can be the first moves between any pair of centroids. Also, if two centroids are visible from each other, $\mathcal{E}$ may be used as the first move. As shown later in our experimental study (Table 3.6), this results in a pretty good compression and reduces the space by up to two orders of magnitude. This is because the symbols in the first move tables are either convex vertices of the navigation mesh or $\mathcal{E}$.

3.4.2.2 Online Search

Algorithm 3 outline the pseudocode of our CPE algorithm. First, we find the centroids c_s and c_d which are the closest centroids to s and d, respectively. Different approaches may be used to achieve this. We implement a grid-based fetching approach. Specifically, during pre-processing, we use a grid for each polygon in the navigation mesh where each cell of the grid stores every centroid of the polygon whose circle overlaps it. During the online search, we identify the grid

Algorithm 3: Centroid-based Path Extraction

```
Input  : $s$: source, $d$: destination, $CPD$: compressed-path-database
Output : a bounded suboptimal path from $s$ to $d$
$c_s \leftarrow$ getClosestCentroid($s$);
$c_d \leftarrow$ getClosestCentroid($d$);
$sp(c_s, c_d) \leftarrow cpd(c_s, c_d)$;
refine the path $sp(s, c_s) \oplus sp(c_s, c_d) \oplus sp(c_d, d)$ and return
```

cell containing s (resp. d) and find the closest centroid recorded in this cell, which is in the same polygon as s (resp. d). The lookup time is linear in the number of centroids that overlap this grid cell.

Once c_s and c_d are obtained, we compute the shortest path between c_s and c_d using the CPD. Since both c_s and c_d are nodes in CPD, the number of CPD look ups required to extract this path is linear in the number of vertices on the shortest path between c_s and c_d. Then, we obtain $sp(s, c_s) \oplus sp(c_s, c_d) \oplus sp(c_d, d)$ where $sp(s, d)$ denote the shortest path from s to d and $\oplus$ is a concatenation operator.

Theorem 3.2. *If the centroid-based CPD is constructed using $\delta = \epsilon/4$ then length of $sp(s, c_s) \oplus sp(c_s, c_d) \oplus sp(c_d, d)$ is at most $\Sigma|sp(s, d)| + \epsilon$.*

Proof. We show that the length of the path, $\Sigma|bsp(s, d)| = ed(s, c_s) + \Sigma|cpd(c_s, c_d)| + ed(c_d, d)$, is at most $\Sigma|sp(s, d)| + \epsilon$. Due to the triangle inequality, we have $\Sigma|cpd(c_s, c_d)| \leq ed(c_s, s) + \Sigma|sp(s, d)| + ed(d, c_d)$. Therefore, we have $\Sigma|bsp(s, d)| \leq 2 \cdot ed(s, c_s) + \Sigma|sp(s, d)| + 2 \cdot ed(c_d, d)$. The hexagon tiling ensures that the circles centred at all centroids with radii δ cover every point in the map. Hence, for any points p, its closest centroid is no further than δ, i.e., $ed(s, c_s) \leq \delta$ and $ed(c_d, d) \leq \delta$. Therefore, we have $\Sigma|bsp(s, d)| \leq \Sigma|sp(s, d)| + 4\delta \leq \Sigma|sp(s, d)| + \epsilon$. □

Consider the example of Fig. 3.7. We first get the closest centroids c_{18} and c_{13} from s and d, respectively. Then, the centroid-based CPD is used to extract an optimal path from c_{18} to c_{13} (i.e., $\langle c_{18}, \texttt{0}, c_{13} \rangle$). The bounded suboptimal path is then $\langle s, c_{18}, \texttt{0}, c_{13}, d \rangle$ (shown using blue lines in the figure).

Path Refinement: Although the path generated by the approach described earlier is bounded, we can further improve the path quality by string pulling. We describe how to refine the path from the source end. Let v be the first vertex after c_s on the unrefined path $sp(s, c_s) \oplus sp(c_s, c_d) \oplus sp(c_d, d)$. We use string pulling to refine the subpath $\langle s, c_s, v \rangle$. To implement the string pulling, we use a combination of the funnel algorithm [67] and our CPD. Specifically, we use the funnel algorithm to find the first turning point p on the refined path from s to v. Since p must be a vertex in the CPD, we can use the CPD to obtain the optimal path from p to v. Thus, the

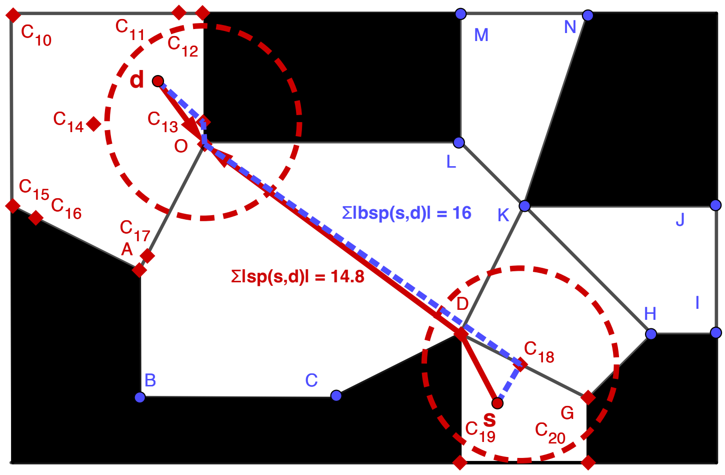


FIGURE 3.7: An example of Centroid Path Extraction (CPE) that runs on a centroid-based CPD with $\delta = 3$. The bounded suboptimal path found by CPE is shown as blue. The final refined path of CPE, as well as the optimal path between s to d, are shown as red.

subpath $\langle s, c_s, v\rangle$ is refined as $sp(s,p) \oplus sp(p,v)$. After the path is refined, we use a similar approach to refine the path from the destination end.

To find the first turning point p from s to v, the funnel algorithm iteratively accesses the mesh edges that intersect with the subpath $\langle s, c_s, v\rangle$ in this order (from s to v). It also maintains a maximal visible interval $I = [v_l, v_r]$ where v_l and v_r correspond to the left and right most vertices, respectively, visible from s through the accessed edge. The algorithm terminates when the accessed edge is completely invisible from s via I and returns v_l (resp. v_r) if the invisible edge is on the left (resp. right) of $I = [v_l, v_r]$.

In the example of Fig. 3.7, the blue path is refined as follows. The first vertex on the blue path after c_{18} is $\mathtt{O}$. Therefore, we use string pulling to refine the subpath $\langle s, c_{18}, \mathtt{O}\rangle$. The funnel algorithm follows the subpath $\langle s, c_{18}, \mathtt{O}\rangle$ and finds intersected mesh edges $\mathtt{DG}$, $\mathtt{DK}$ and $\mathtt{AO}$ in this order. The algorithm initialises the maximal visible (from s) interval I as $[\mathtt{D}, \mathtt{G}]$. When it processes the next intersecting edge $\mathtt{DK}$, it updates I to be $[\mathtt{D}, \mathtt{K}]$ since the whole edge is visible from s. It then proceeds to process the edge $\mathtt{AO}$. Since $\mathtt{AO}$ is completely invisible from s via $I = [\mathtt{D}, \mathtt{K}]$ and $\mathtt{AO}$ is towards the left of I (i.e., node $\mathtt{D}$), the funnel algorithm returns D as the first turning point on the path from s to $\mathtt{O}$. We use the CPD to compute the path from $\mathtt{D}$ to $\mathtt{O}$ which happens to be $\langle \mathtt{D}, \mathtt{O}\rangle$. Therefore, the subpath $\langle s, c_{18}, \mathtt{O}\rangle$ is refined to $\langle s, \mathtt{D}, \mathtt{O}\rangle$. Similarly, the subpath from the destination end (i.e., $\langle d, c_{13}, \mathtt{O}\rangle$) is refined to $\langle d, \mathtt{O}\rangle$. Thus, the final refined path is $\langle s, \mathtt{D}, \mathtt{O}, d\rangle$ (shown using red lines in the figure).

3.4.3 Experiments

We run experiments on the same benchmarks and machine as described in section 3.3.3. We compare our suboptimal algorithms (SEPS and CPE) with SUB-N-T [59] (briefly described in

Section 3.3.3). We also compare with SUB-N-A [59] which is the same as SUB-N-T except that it uses A* instead of Theta-A*. For our SEPS, we refine the path by using the same path refinement technique of CPE. i.e., given a path $sp(s, v_s) \oplus sp(v_s, v_d) \oplus sp(v_d, d)$ returned by SEPS and let v be the first turning vertex after v_s, we refine the subpath $\langle s, v_s, v\rangle$ if it is not taut. We refine the path in a similar way for the destination end. For both SUB-N-T and SUB-N-A, as suggested in [59], we use path smoothing to improve the path quality.

3.4.3.1 Centroid-based CPD Statistics

In order to construct them faster, our centroid-based CPDs were built on 32 cores Nectar research cloud with 64GB of RAM and running Ubuntu 18.04 LTS (Bionic) amd64. Table 3.6 shows the average and maximal number of centroids, the raw size of the first move tables (e.g., without compression), as well as the size of the CPD with (i.e., M) or without $\mathcal{E}$ (i.e., $M \setminus \mathcal{E}$) symbols, and its building time for the four benchmark suites. Clearly, our centroid-based CPDs are cheap to build and memory efficient for large radii (i.e., δ = 4, 8). On the other hand, when the radius is small (i.e., δ = 1, 2), our centroid-based CPDs require more pre-processing time and space. In later experiments, we will show that the variation of radius in the centroid-based CPDs has only a minor effect on the running time of our CPE algorithm, but may result in worse path quality for larger radius. Therefore, there is a trade-off between the sub-optimality requirements and the memory limitations and a decision regarding the radius can be made considering the specific requirements of the application.

Compared with the raw memory, our CPD compression reduces the size of the first move tables by up to five hundred times, and allows CPDs even with a small radius (i.e., δ = 1 or 2) to fit into the main memory of modern devices. In addition, we also observe that the $\mathcal{E}$ symbols significantly improve the size by a few factors on average, especially when the radius is small (i.e., δ = 1 or 2), because there exists many centroids that are co-visible and without the $\mathcal{E}$ symbols, they do not compress well.

3.4.3.2 Query Processing Time

In Table 3.7, we compare the average query processing time for our CPE algorithm using different radii (i.e., δ = 1, 2, 4, and 8) against the competitors. Table 3.7 shows that CPE significantly outperforms all competitors in all settings. Specifically, CPE is around one order of magnitude faster than SUB-N-T, SUB-N-A and optimal EPS, and is several times faster than the suboptimal EPS, i.e., SEPS(ϵ=32), SEPS($\epsilon = 8\%$). The SEPS with relative bound $\epsilon = 8\%$ matches the same sub-optimality with SUB-N-T and SUB-N-A, because the shortest grid paths are at most 8% longer than the optimal any-angle path [13]. We also observe that the query processing time of CPE is not significantly affected with the change in δ. This is because CPE

δ		CPE(δ=1)		CPE(δ=2)		CPE(δ=4)		CPE(δ=8)	
map	stat	Avg	Max	Avg	Max	Avg	Max	Avg	Max
DAO	C	14294.705	87898	5577.500	34404	3129.910	19059	2341.621	14079
	T	1.109	29.474	0.178	4.662	0.061	1.486	0.036	0.873
	R	1961.839	30904.233	293.112	4734.540	90.923	1452.981	50.574	792.872
	M$\backslash\mathcal{E}$	122.745	1900.786	15.885	208.670	3.831	53.326	1.688	23.585
	M	27.318	536.143	6.467	120.143	2.308	38.942	1.213	19.043
DA2	C	10557.328	29008	4094.597	11701	2211.462	7343	1608.164	5959
	T	0.251	2.120	0.042	0.431	0.014	0.149	0.009	0.130
	R	651.578	3365.856	97.233	547.653	28.752	215.678	15.664	142.038
	M$\backslash\mathcal{E}$	40.726	189.284	5.561	21.793	1.382	5.593	0.627	2.768
	M	7.921	30.164	2.033	7.231	0.762	3.018	0.418	1.780
BG	C	36570.453	94750	11590.760	31316	4511.653	15998	2449.186	11423
	T	2.763	26.548	0.250	2.080	0.047	0.589	0.019	0.332
	R	6100.325	35910.250	619.045	3922.767	102.595	1023.744	37.330	521.939
	M$\backslash\mathcal{E}$	723.109	2049.922	142.178	1943.864	15.081	153.247	2.858	14.774
	M	74.253	442.378	14.857	67.309	3.807	19.536	1.390	10.161
SC	C	149261.360	469291	52441.946	173836	25505.546	92309	17263.480	66775
	T	130.440	1013.990	13.255	98.862	2.836	25.950	1.284	13.265
	R	114718.511	880936.170	14302.496	120875.819	3476.542	34083.805	1638.570	17835.602
	M$\backslash\mathcal{E}$	1061.652	2099.440	502.036	2017.235	92.047	397.907	31.269	133.774
	M	635.166	3966.987	143.700	850.968	45.803	257.136	21.087	105.182

TABLE 3.6: Number of (C)entroids, building (T)ime in minutes, (R)aw memory (first-move tables without compression) in MB, and (M)emory usage without/with ($\mathcal{E}$) symbol in MB, for different radius δ over four benchmarks.

needs to extract only one path using the CPD and the size of the CPD does not significantly affect the performance because the number of first move extractions is small. Also, the query time of CPE is dominated by the cost of getting the closest centroids from s and d. Note that the path refinement is cheap for both CPE and SEPS which is mainly because, we only use string pulling to find the first turning vertex, and make use of CPDs to quickly recover the rest of the path. The refinement time for CPE is even cheaper because, in most of the cases, the first vertex after c_s (resp. before c_d) is visible from s (resp. d) resulting in cheap string pulling that behaves like a simple line-of-sight check.

In Fig. 3.8, we extend the comparison to show the cactus plots for each competing algorithm over four different benchmarks. Note that for our CPE and SEPS algorithms, we also include the path refinement time. Fig. 3.8 shows that both CPE and SEPS are significantly faster than EPS, SUB-N-T and SUB-N-A. CPE is the best performing algorithm and scales really well with the increasing difficulty of the queries.

3.4.3.3 Path Quality

In this experiment, we compare the path quality for CPE and the suboptimal EPS against the competitors. Our evaluation is based on the following three measures: (i) The (O)ptimal ratio

	CPE		CPE		CPE		CPE		SEPS		SEPS		EPS	SUB-N-T	SUB-N-A
par	(δ=1)		(δ=2)		(δ=4)		(δ=8)		(ϵ=32)		(ϵ=8%)		-	-	-
map	Q	R	Q	R	Q	R	Q	R	Q	R	Q	R	Q	Q	Q
DAO	2.24	0.76	2.13	0.75	2.13	0.78	2.16	0.80	4.53	1.45	5.14	1.41	23.90	81.51	31.82
DA2	2.21	0.65	2.08	0.64	2.04	0.65	2.04	0.68	4.12	1.28	4.97	1.21	14.18	31.29	15.45
BG	1.97	0.69	1.91	0.73	1.88	0.69	1.84	0.69	4.05	1.08	4.92	1.02	12.29	33.83	14.32
SC	3.98	1.50	3.52	1.37	3.45	1.54	3.46	1.61	22.04	1.73	14.93	1.76	61.32	252.96	83.37

Table 3.7: Runtime comparison on the four benchmarks: we show the average (Q)uery processing time (µs) and path (R)efinement time (µs) for Centroid-based Path Extraction (CPE), and compare with: the Suboptimal EPS (SEPS) with an absolute bound ϵ=32 (the same sub-optimality bound as $\delta = \epsilon/4 = 8$) and with a relative bound $\epsilon = 8\%$ (the same sub-optimality bound as SUB-N-T and SUB-N-A); the optimal EPS; and two subgoal graph methods SUB-N-T and SUB-N-A.

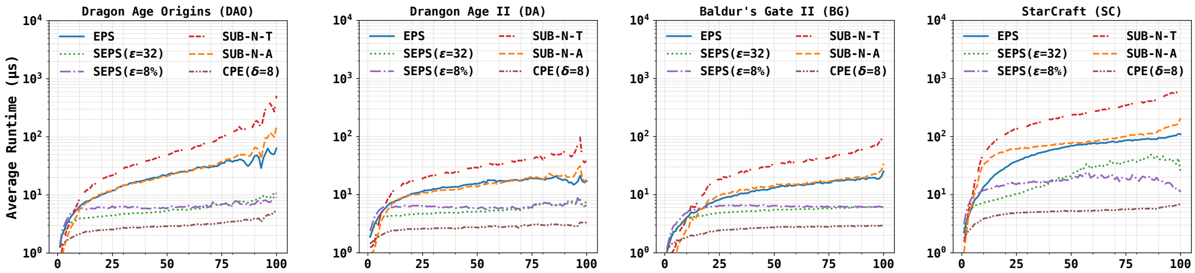


Figure 3.8: Runtime comparison on the four benchmarks. The x-axis shows the percentile ranks of queries in number of node expansions needed by A* search to solve them.

(%) that corresponds to the percentage of instances when an algorithm retrieved optimal paths, i.e., number of queries for which an algorithm returns an optimal path × 100 /total number of queries; (ii) The (S)uboptimality ratio (%) which is the difference between the length of the retrieved path and the optimal path divided by the length of optimal path (i.e., (Σ|retrieved path| $-$ Σ|optimal path|) × 100 / Σ|optimal path|); (iii) The path (D)ifference which is the difference between the length of the retrieved path and optimal path. (i.e., Σ|retrieved path| $-$ Σ|optimal path|).

Effectiveness of path refinement in CPE and SEPS: In Table 3.8, we compare the path quality before and after path refinement for our CPE and SEPS algorithms. Clearly, our path refinement strategy significantly improves the path quality on all three measures. Before the path refinement, the optimality ratio for both CPE and SEPS are low because: the retrieved path by CPE can never be optimal unless the closest centroids of both s and d are either turning points on the path or s and d lie on centroids; and SEPS essentially terminates as soon as it satisfies the suboptimality constraint. However, it is clear that the optimality ratio is significantly improved after the path refinement and, in fact, most of the paths returned by CPE are optimal. In addition, we observe that the path refinement also significantly reduces the average suboptimality ratio (S) and difference (D). Thus, the path refinement is cheap (see

		CPE(δ=1)		CPE(δ=2)		CPE(δ=4)		CPE(δ=8)		SEPS(ϵ=32)		SEPS(ϵ=8%)	
map	stat	Before	After	Before	After	Before	After	Before	After	Before	After	Before	After
DAO	O(%)	3.110	95.291	3.140	89.409	3.061	80.893	3.077	70.393	3.931	39.283	9.067	44.413
	S(%)	1.021	0.001	1.953	0.008	3.398	0.029	5.235	0.072	14.332	1.431	2.888	0.366
	D	1.030	0.002	1.990	0.009	3.565	0.031	5.744	0.088	12.771	1.525	10.241	1.126
DA2	O(%)	2.315	96.486	2.342	91.925	2.253	84.061	2.335	74.797	5.625	47.126	12.013	54.842
	S(%)	1.036	0.001	1.982	0.005	3.597	0.022	5.661	0.066	14.563	1.340	2.672	0.272
	D	1.059	0.001	2.053	0.006	3.714	0.024	5.974	0.074	12.339	1.198	6.750	0.663
BG	O(%)	6.053	97.988	6.053	95.392	6.047	91.105	6.080	85.045	9.153	49.024	16.868	56.305
	S(%)	0.861	<0.001	1.671	0.002	3.159	0.008	5.352	0.030	12.964	1.292	3.191	0.383
	D	1.082	0.001	2.105	0.003	4.020	0.015	7.020	0.052	15.131	1.899	8.478	1.058
SC	O(%)	2.161	97.570	2.171	94.275	2.170	88.506	2.169	79.565	7.272	26.844	8.252	27.222
	S(%)	0.473	<0.001	0.915	0.001	1.715	0.005	3.088	0.020	6.856	1.447	3.205	0.703
	D	1.117	0.001	2.166	0.004	4.104	0.016	7.372	0.059	13.842	3.073	16.921	3.457

Table 3.8: Our Centroid-based Path Extraction (CPE) and Suboptimal EPS (SEPS), before and after path refinement. We show (O)ptimal ratio (%): i.e., # optimal path × 100 / # queries; average (S)uboptimality ratio (%): i.e., (cost(retrieved path) − cost(optimal path)) × 100 / cost(optimal path) and (D)ifference: i.e., cost(retrieved path) − cost(optimal path).

Table 3.7) and very effective in improving the path quality. Finally, note that the average path difference of our CPE both before and after path repair is much smaller than the theoretical bound (i.e., 4 × δ). Similarly for SEPS, the theoretical error bound ϵ is also larger than the average suboptimality ratio and path difference reported.

Comparison of different algorithms: Table 3.9 compares the algorithms on the optimal ratio (O) as well as average and maximum suboptimality ratio (S) and path difference (D). Clearly, CPE demonstrates excellent path quality for all different benchmarks on all measures, e.g., the optimal ratio, average suboptimality ratio and average path difference are better than those of SUB-N-T, SUB-N-A, SEPS(ϵ = 32) and SEPS(ϵ = 8%) for different values of δ and remarkably better for smaller δ. SUB-N-A and SUB-N-T are better in terms of *maximum* suboptimality ratio. Since CPE returns a path within an absolute bound, the path quality in terms of suboptimality ratio may be very poor when the source and destination are close to each other but δ is large (i.e., 4 or 8). On the other hand, CPE usually has a smaller *maximum* path difference (D) compared to SUB-N-T for δ = 1, 2 or 4. For the SEPS with an absolute bound, SEPS(ϵ = 32), we see that the maximum path difference is always close to the theoretical bound ϵ as our SEPS greedily terminates the search as soon as it explores a path that satisfies the constraint (note that the refinement does not improve the path quality if the returned path is taut on both ends). This also causes the suboptimality ratio of the absolute bounded SEPS to be large, especially in terms of maximum suboptimality ratio (which is mainly due to the queries when source and destination are close to each other). The SEPS with a relative bound, SEPS(ϵ = 8%), has the advantage that it allows the search to control the suboptimality ratio within a certain percentage. However, in this case, it results in large absolute differences in the worst-case (for queries when source and destination are far from each other).

		CPE)		CPE		CPE		CPE		SEPS		SEPS		SUB-N-T		SUB-N-A	
	par	(δ=1)		(δ=2)		(δ=4)		(δ=8)		(ϵ=32)		(ϵ=8%)		-		-	
map	stat	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
`DAO`	O(%)	95.291		89.409		80.893		70.393		39.283		44.413		56.303		35.767	
	S(%)	0.001	12.75	0.008	19.73	0.029	141.42	0.072	89.44	1.431	1623	0.366	7.92	0.093	4.87	0.155	4.94
	D	0.002	1.81	0.009	4.00	0.031	6.87	0.088	11.81	1.525	31.83	1.126	53.10	0.504	7.24	0.732	9.64
`DA2`	O(%)	96.486		91.925		84.061		74.797		47.126		54.842		63.949		48.312	
	S(%)	0.001	7.70	0.005	6.89	0.022	20.89	0.066	78.04	1.340	470	0.272	7.97	0.096	3.76	0.129	5.47
	D	0.001	1.92	0.006	2.98	0.024	6.91	0.074	10.44	1.198	30.49	0.633	63.28	0.354	6.01	0.438	7.29
`BG`	O(%)	97.988		95.392		91.105		85.045		49.024		56.305		82.958		66.343	
	S(%)	<0.001	14.41	0.002	14.41	0.008	14.41	0.030	29.72	1.292	570	0.383	7.79	0.061	5.58	0.115	5.58
	D	0.001	1.34	0.003	5.46	0.015	6.43	0.052	15.44	1.899	31.62	1.058	35.11	0.188	14.1	0.348	16.2
`SC`	O(%)	97.570		94.275		88.506		79.565		26.844		27.222		54.780		27.952	
	S(%)	<0.001	4.69	0.001	8.27	0.005	17.08	0.020	49.99	1.447	554	0.703	7.84	0.076	5.15	0.175	5.15
	D	0.001	2.57	0.004	3.94	0.016	10.12	0.059	18.06	3.073	30.93	3.457	109.3	0.526	17.5	1.120	19.8

TABLE 3.9: Comparing CPE, SEPS, SUB-N-T and SUB-N-A on (O)ptimal ratio (%) as well as average/maximum (S)uboptimality ratio (%) and (D)ifference.

3.5 Discussion

We introduce new approaches to Euclidean path finding based on Compressed Path Databases (CPD). Our optimal algorithm, End Point Search (EPS), substantially improves the state-of-the-art for optimal Euclidean shortest paths and also has impressive anytime behaviour. It makes use of powerful CPD approaches to handle path finding on the visibility graph, and an efficient incremental attachment of the end points to this graph, to quickly find high quality solutions, and prove optimality fast. The bounded suboptimal variant, Centroid-based Path Extraction (CPE), is several times faster than EPS for finding (absolute) bounded suboptimal paths. It allows us to trade off the suboptimality bound versus the size of the resulting CPD. In practice, its behaviour is much better than the theoretical bound, with ≈90% of paths found being optimal for $\delta = 2$.

Chapter 4

Contracting and Compressing Shortest Path Databases

4.1 Overview

Compressed Path Databases (CPD) are powerful database-driven methods for shortest path extraction in static road networks. A CPD can be understood as an oracle $cpd(s, d)$ which, given a source-destination pair, resp. s and d, tells the identity of the first edge on the optimal path: from s toward d. Using a simple recursive procedure, CPDs can extract the entire paths at ultra-fast speed [144]. There are however two drawbacks: (i) the time complexity for building the database is quadratic in the size of the input graph, which can be prohibitive in some cases; (ii) the query time performance grows supra-linearly with the number of edges in the shortest path,[1] which affects performance when there are many edges to extract.

In this chapter, we mitigate the disadvantages of CPDs by investigating connections with another family of successful, but search-based, speedup techniques called Contraction Hierarchies (CH) [22, 77]. The CH method can be understood as a type of embedded graph abstraction [145]. During preprocessing, additional "shortcut" edges are added to the graph. During online search, these shortcuts help the search to bypass many "unimportant" vertices which would otherwise need to be expanded. Moreover, the total number of edges on a path, from source to destination, is reduced. For this reason, we consider contracted graphs in combination with CPDs. First, CH graphs help to improve CPD online performance, by reducing the number of lookups we need to perform when extracting a path. Second, CH graphs help to reduce CPD offline costs, by allowing us to speed up the many Dijkstra searches required to construct first-move data. In broad strokes, our strategy is as follows:

[1] Complexity per lookup is logarithmic in the size of the compressed data w.r.t the corresponding row.

Symbol	Description
G	A weighted input graph.
V	A set of vertices in the input graph G.
E	A set of edges in the input graph G.
W	A set of weights that maps each edge in G to a non-negative weight.
s	The source of a pathfinding query and $s \in V$.
d	The destination of a pathfinding query and $d \in V$.
v_i	A vertex in the input graph G and $v_i \in V$.
P	A path that consists of a sequence of vertices $\langle v_1,v_2, \cdots, v_k \rangle$.
$\Sigma\|P\|$	The cost (i.e., length) of a path P.
$\mathcal{C}$	A selected subset of vertices that are used to cache the distance and $\mathcal{C} \subseteq V$.
$\mathcal{T}$	A selected subset of vertices that are used to build partial CH-CPD and $\mathcal{T} \subseteq V$.
$g(v_i, v_j)$	The tentative distance of a search node from source v_i to current vertex v_j.
$sd(v_i, v_j)$	The shortest distance (i.e., sum of cost) between v_i and v_j.
$landmark(v_i, v_j)$	The lower-bound heuristic value returned by landmarks between v_i and v_j.

Table 4.1: Summary of the notations used in this chapter

- We compute *distance tables* for a small number of important CH vertices. We show that these tables can be used to provide bounds for, and therefore can help to speed up, each of the many Dijkstra searches necessary for computing first-move data.
- We also compute first-move data for only a selected subset of CH vertices. This further reduces the overall time needed for CPD precomputation and also lowers the storage cost.
- We develop a new bidirectional query algorithm, which combines online search in a CH with CPD path extraction. This allows us to compute the shortest paths substantially faster than either CH or CPD.

We compare our approach on several well-known road network benchmarks. Our principal points of comparison are SHP [84] and PHL [83]: two recent and also database-driven query algorithms. We show that, for computing shortest paths, CH-CPD offers substantially better performance and has overall smaller storage costs.

4.2 Preliminaries

In a static road network, the network is represented as a weighted graph where each edge assigns a non-negative weight indicating distance, travel time etc. The pathfinding query asked us to find point-to-point traversable paths within the weighted graph. Any two nodes in the graph can be a potential source s and destination d, and the objective is to find the shortest path traversing from s to d while minimising the sum of edge weights. In this chapter, we follow the terminologies introduced in Section 2.2, and Table 3.1 shows the symbols used in this chapter.

As discussed in the overview, our algorithms utilise the Contraction hierarchies (CH) [22] to mitigate the disadvantage of the state-of-the-art algorithms Compressed Path Databases

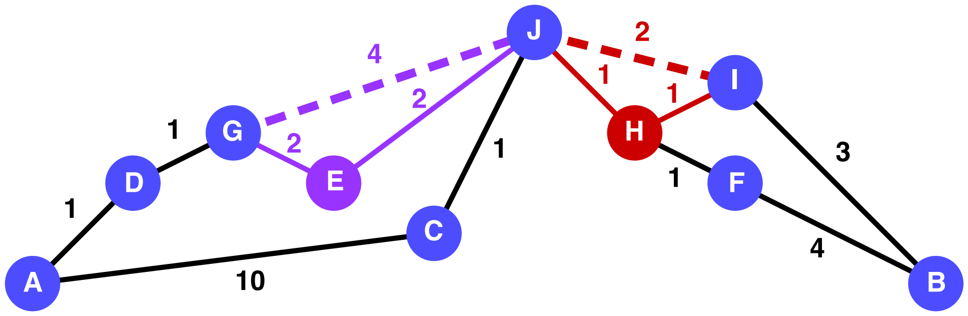


FIGURE 4.1: Same as Figure 2.5, where we show the result of contracting `E` (resp. `H`) in purple (resp. red). Dashed edges indicate shortcut edges.

(CPD) [26]. We have given a full description of CH and CPD in Section 2.2.2 and Section 2.2.3, respectively. In addition, we adapt the landmark heuristic to further improve the performance of bidirectional search (BCH) in CH, the details of landmark heuristic can be found in Section 2.2.1. For the ease of presenting our techniques, we redraw an example of CH in Figure 4.1, where the dashed edges show the shortcuts added by contracting the vertex E (resp. H) shown in purple (resp. red).

4.3 Combining CH and CPD

Given a weighted graph G, we first construct a Contraction Hierarchy. With this contracted graph in hand, a CPD can now be constructed by following the same general procedures already described. However, we introduce three changes: (i) we modify the successor generating function of the Dijkstra algorithm so that every vertex is reached along a ch-path; (ii) we enhance the basic Dijkstra algorithm using precomputed *distance tables*, which speeds up the computation of first-move data; (iii) we store compressed data for only a subset of all graph vertices.

4.3.1 CH-Paths in Dijkstra Search

Recall a ch-path always has an apex vertex which is lexically larger than all the other vertices on the path. For a given source s and destination d, deconstructing the ch-path gives the following three cases:

1. Up ch-path: d is an apex vertex (i.e., $d >_{\mathcal{L}} v$ for $v \in \langle s, \ldots d-1 \rangle$)

2. Up-Down ch-path: an intermediate vertex k is an apex vertex (i.e., $k >_{\mathcal{L}} v$ for $v \in \langle s, \ldots k-1 \rangle \mid \langle k+1, \ldots d \rangle$)

3. Down ch-path: s is an apex vertex (i.e., $s >_{\mathcal{L}} v$ for $v \in \langle s+1, \ldots d \rangle$)

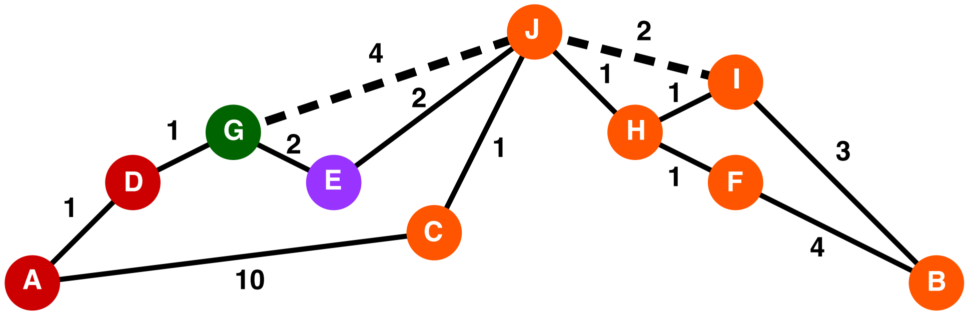


Figure 4.2: Constructing CPD on top of CH. The source vertex `G` is highlighted as green. The first move on the optimal path from source vertex to any vertex are `D`, `E` and `J` shown as red, purple and orange, respectively.

In other words, before the apex vertex, every subsequent vertex on a ch-path is lexically larger than the previous. After the apex, every subsequent vertex on a ch-path is lexically smaller than the previous. We modify Dijkstra search to only consider ch-paths by way of a simple neighbour pruning rule called UTD (Up-Then-Down) [146]. The idea is simple: (i) if the predecessor of the current vertex is lexically larger than the current vertex we prune all up successors (this covers case 3); (ii) if the predecessor is lexically smaller than the current vertex we generate all successors, up or down (this covers case 1 and 2). The only paths disallowed by UTD involve edges to lexically smaller vertices followed by edges to lexically larger vertices; i.e., non ch-paths.

4.3.2 Distance Tables Enhancement

In a CH, vertices with high lex values often appear as the apex vertex along many shortest ch-paths. We exploit this observation to reduce the cost of first-move preprocessing, as follows:

Caching: For a given contracted graph G, we first select a set of vertices $\mathcal{C}$ that have the largest lex values. For each vertex $v_c \in \mathcal{C}$, we run our modified single-source Dijkstra search and store not only a first move table but also a table of shortest distances, from v_c to every other vertex in the graph.

Pruning: For each remaining vertex $v \notin \mathcal{C}$, we start a single-source Dijkstra search as usual but we never generate any successors for any cached vertex $v_c \in \mathcal{C}$. When expanding a cached vertex we instead refer to the cached distances and attempt to relax the tentative estimate $g(v, v')$ for each vertex $v' \in G$. We perform the relaxation if $g(v, v') > sd(v, v_c) + sd(v_c, v')$ where $sd(v_c, v')$ is the cached distance. The first move table of v is also relaxed accordingly using the first move on the current optimal path $P(v, v_c)$. Simultaneously relaxing all tentative distances gives tight upper bounds sooner and helps the search to terminate faster. First, we never expand successors for any vertex $v_c \in \mathcal{C}$. Second, all distance information usually propagated by such vertices is copied into the first move table of v. That means we also never expand any vertex $v_n \notin \mathcal{C}$ where the apex of the optimal ch-path from v resides in $\mathcal{C}$. It is easy to see this approach limits the search space and allows for faster termination.

On Demand Reading: Storing all distance tables in RAM requires $O(|\mathcal{C}||V|)$ space and can be prohibitive as $|\mathcal{C}|$ grows large. We thus store the tables to disk and retrieve them on demand: whenever the search expands a vertex $v_c \in \mathcal{C}$. When loading, distance data is mapped into virtual memory at once, which avoids unnecessary I/O operations.

Example 4.1. *Consider building the CPD for the graph of Figure 4.2. Assume we have already constructed $\mathcal{C} = \{J\}$. We begin a Dijkstra search from `G` and update tentative distances and first move tables when expanding `J`. The search terminates after expanding `D`, `A`, `E` and `J`, without exploring the rest of the graph. The first moves to `C`, `H`, `I`, `F`, and `B` are the same as for `J` since the shortest ch-path from `G` to each of these vertices is via `J`.*

4.3.3 Partial CH-CPD

CH and cost caching help to speed up each Dijkstra search, which means computing first-move data is faster in practice compared to the original baseline. However, the worst-case time complexity is unchanged: $O(|V||E| + |V|^2 \log |V|)$. To further improve preprocessing costs, we propose to compute and store first-move data for only a selected subset of the full CH graph (i.e., $\mathcal{T} \subseteq G$).

There are many ways to choose $\mathcal{T}$, but an important requirement is that the subgraph is closed, in other words, for each pair of vertices, $s, d \in \mathcal{T}$, the shortest ch-path must also belong to $\mathcal{T}$. Since every ch-path is always an Up-Down path, we can therefore select any subset of vertices which is lexicographically upwards closed, i.e., $\mathcal{T}(v_l) = \{v_l \in V \mid v_l >_{\mathcal{L}} v \text{ for } v \in V \setminus \mathcal{T}\}$. Clearly, all Up-Down paths between vertices in $\mathcal{T}(v_l)$ only make use of vertices in $\mathcal{T}(v_l)$.

With $\mathcal{T}$ selected, we now compute a partial CPD for some upper part of the contraction hierarchy. However, we also need to define a new shortest path algorithm, to support queries for arbitrary pairs of source s and destination d. Here, we combine the BCH query method with CPD path extraction. The approach is similar to End Point Search (EPS) [45] in that, when a vertex $v \in \mathcal{T}$ is expanded, we use the CPD to extract candidate the shortest paths: from v to all other vertices $v' \in \mathcal{T}$ that have been expanded in the opposite direction. We give a detailed description in the next section.

4.4 Bidirectional CPD Search

Our search algorithm is similar to BCH, but with some fundamental differences. Firstly, we employ bidirectional A* search instead of bidirectional Dijkstra. In particular, for each s and d

Algorithm 4: Bidirectional CPD Search

```
   Input: $s$: source, $d$: destination, CPD: for vertices in the set $\mathcal{T}$
   Output: an optimal path from $s$ to $d$
   Initialisation: $V_s = \emptyset$, $V_d = \emptyset$, $P = \langle\rangle$, $\Sigma|P| = \infty$, $R_s = \emptyset$, $R_d = \emptyset$
 1 $cur = s$; $opp = d$;
 2 $Q_s = \{s\}$; $Q_d = \{d\}$
 3 while both A* searches are not exhausted do
 4     $v =$ pop($Q_{cur}$);
 5     if $v \in \mathcal{T}$ then // $v$ is a CPD vertex
 6         $V_{cur} \leftarrow V_{cur} \cup \{v\}$
 7         for each $v' \in V_{opp}$ do
 8             if $g(cur, v) + sd(v, v') + g(v', opp) < \Sigma|P|$ then
 9                 $P \leftarrow \langle cur, v, v', opp\rangle$
10     else
11         if $v \in R_{opp}$ then // $v$ reached by $opp$ search
12             if $g(cur, v) + g(v, opp) < \Sigma|P|$ then
13                 $P \leftarrow \langle cur, v, opp\rangle$
14         A*Expand($v, Q_{cur}, R_{cur}$)
15     if $Q_{opp}$ is not empty then
16         $cur, opp \leftarrow opp, cur$;
17 return $P$ after unpacking it;
```

query, our search is guided by a landmark heuristic; i.e., the f-value for a vertex v is $g(s, v) + landmark(v, d)$ with $g(s, v) \geq sd(s, v)$ being a tentative upper bound for the optimal distance from s to v. Secondly, when the search expands a CPD vertex (i.e., a vertex that exists in the partial CPD), we do not generate any successors. Instead, the partial CPD is used to extract paths/distances to all CPD vertices expanded in the other direction. We also reduce the number of first move extractions using pruning rules discussed later. Similar to BCH, our approach considers only "up" edges and employs the well-known and performance-improving technique called *stall-on-demand* [22]. This technique allows us to stall search nodes that cannot be part of the shortest path, optimising the search process efficiently.

The pseudo-code of our approach is shown in Algorithm 4. We start the bidirectional search from s and d with separate queues Q_s and Q_d. We use *cur* (resp. *opp*) to denote the current (resp. opposite) direction in which the search is expanding; i.e., if *cur* is source then *opp* is destination and vice versa. The optimal path P and the optimal path length $\Sigma|P|$ are initialised to be empty and infinity, respectively.

In each iteration, we pop the vertex v with the smallest f-value from the current queue Q_{cur} to expand (line 4). If this vertex is in the CPD (i.e., $v \in \mathcal{T}$), we add this to V_{cur} to record that this CPD vertex is expanded by the search from *cur* (line 5 - 6). Then, we use the CPD to efficiently compute the shortest path/distance from v to each CPD vertex $v' \in V_{opp}$ found by the search from the opposite side *opp*. If, for any $v' \in V_{opp}$, $g(cur, v) + sd(v, v') + g(v', opp)$ is smaller than $\Sigma|P|$ (length of the current optimal path P), we update the optimal path to be

$\langle cur, v, v', opp\rangle$ (line 7 - 9). Note that this only records two intermediate CPD vertices on the path P. The complete optimal path is recovered once at the end of the algorithm. Also, note that $g(cur, v)$ and $g(v', opp)$ are already known due to the two A* searches from *cur* and *opp*, respectively, whereas $sd(v, v')$ is efficiently extracted using the CPD.

If the vertex v is not a CPD vertex, we follow bidirectional search. For vertex v that we have met from the other direction (i.e., $v \in R_{opp}$), we calculate the path length from s to d via v and replace P with this path if it is better than the current P (line 11 - 13). We then expand v, adding its neighbours to the "reached vertices" from the current direction R_{cur} and the priority queue Q_{cur}, pruning as appropriate using the current incumbent path length (line 14). Finally, we swap the search and proceed in the other direction (assuming *opp* is not already exhausted). Note that the search never expands beyond a CPD vertex, i.e., only non-CPD vertices can generate successors.

The loop terminates when both of the A* searches exhaust. We say that an A* search exhausts if either the queue becomes empty or the top vertex v has f-value at least equal to $\Sigma|P|$ (line 3). When the loop terminates, we unpack P to obtain the complete path and return it (line 17). First, the Up-Down path in the contraction hierarchy is recovered from P by using the predecessor for each vertex (recorded either during the two A* searches or the CPD path extraction as discussed shortly). Then, the shortcuts in this Up-Down path are unpacked to obtain the optimal path in the original graph. Next, we discuss pruning to speed up the algorithm.

4.4.1 Pruning Enhancement

Recall, at line 8, we compute $sd(v, v')$ for every $v' \in V_{opp}$ using the CPD. This involves recursively obtaining first moves to identify a complete path from v to v'. In some cases, we can avoid computing $sd(v, v')$ by checking if $g(cur, v) + landmark(v, v') + g(v', opp) \geq \Sigma|P|$. For each pair v, v' that cannot be pruned in this way, we employ a caching scheme that can reduce unnecessary first move extractions. Specifically, whenever we extract a first move $u \in V$ on the shortest path from v to v', we also record $g(cur, u)$ which corresponds to the shortest distance from *cur* to u seen so far. We also maintain the predecessor of u which helps in path recovery as discussed earlier. Later, when extracting a path from any $v \in V_{cur}$ to any $v' \in V_{opp}$, we can terminate the recursion early if we reach a vertex u for which $g(cur, u) < g(cur, v) + sd(v, u)$. This is because we already have explored a path to u which is shorter than the current path to u via v. Notice that our caching strategy maintains only tentative distances. As such, it requires no additional memory overheads beyond what is typically allocated for bidirectional search. Though simple, this approach substantially improves the performance of CH-CPD search.

Theorem 4.1. *Algorithm 4 returns an optimal path.*

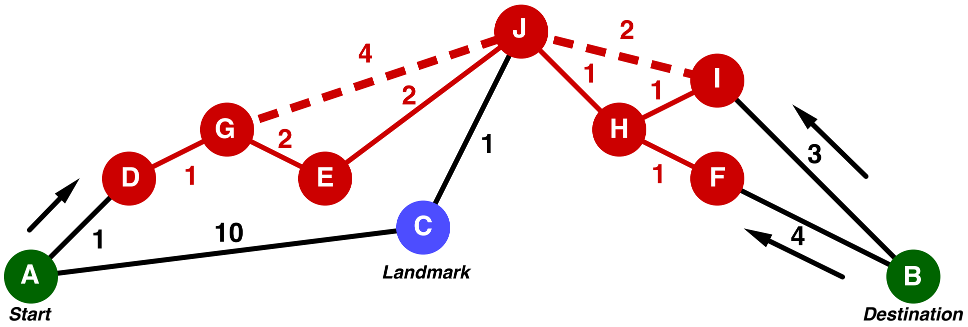


FIGURE 4.3: A* search from source (resp. destination) expands `D` (resp. `I` and `F`). `C` is pruned by landmark heuristic.

Proof. Clearly Algorithm 4 examines all the paths that either meet, or connect via a pair of CPD vertices (v_s, v_d). But avoids exploring the vertices that have the f-values bigger than the cost of current shortest path $\Sigma|P|$ (thus can never be part of optimal path), and vertices pairs (v_s, v_d) of CPD where $sd(s, v_s) + sd(v_s, v_d) + sd(v_d, d) \geq \Sigma|P|$. □

Example 4.2. *Consider the example in Figure 4.3 and assume that* `D` *to* `J` *are CPD vertices (shown in red) and* `C` *is a landmark (shown in blue). The A* search from source* `A` *first expands the CPD vertex* `D`*. Then, the A* search from destination* `B` *expands the CPD vertex* `I`*. CPD is used to extract the path from* `I` *to* `D` *and the distances from* `B` *to each vertex on the path are cached. The optimal path P is updated to be* $\langle B, I, D, A\rangle$ *with length* 11*. A* search from* `A` *prunes the vertex* `C` *using the landmark heuristic because* $g(A, C) + landmark(C, B) = 10 + 6 = 16 > 11$*. This A* search exhausts. The A* search from the destination expands* `F`*. It needs to extract a path from* `F` *to* `D` *using the CPD. First moves are extracted and when the vertex* `J` *is reached, the path extraction stops. This is because the cached distance* $g(B, J) = 5$ *is smaller than* $g(B, F) + sd(F, J) = 6$*. A* search from the destination is also exhausted. The path* $\langle B, I, D, A\rangle$ *is unpacked and returned as* $\langle B, I, H, J, E, G, D, A\rangle$

4.5 Experiments

We test our proposed algorithms against baseline implementations of our two main ingredients, CH and CPD, and against other state-of-the-art methods from the recent literature. By **CPD**, we refer to Compressed Path Databases, as represented by SRC [27] and implemented by original authors.[2] **CH** refers to Contraction Hierarchies [22, 77], as implemented in RoutingKit.[3] **CH+L** (CH + Landmarks) is a customised variant similar to CH where we replace bidirectional Dijkstra search with bidirectional A* and Landmark heuristics. We show CH+L performs significantly better than CH. Our approach meanwhile is denoted **CH-CPD (x%)**

[2] https://bitbucket.org/dharabor/pathfinding

[3] https://github.com/RoutingKit/RoutingKit

where $x\%$ means that a CPD is constructed for the top $x\%$ of the vertices in the contraction hierarchy. Both CH-CPD and CH+L use 4 landmarks for distance estimates (adding more did not improve performance).

For further comparison, we also consider two recent hub-labelling algorithms, PHL and SHP, as described in [84] and using implementations from those authors[4]. **PHL** (Pruned Highway labelling) [83] is a popular and efficient hub labelling method for shortest distance queries on road networks. While most hub labelling algorithms use vertices as hubs, PHL differs mainly in that it uses highway paths as hubs and the distances are maintained to these highways. **SHP** (Significant path based Hub Pushing) [84] is another state-of-the-art hub labelling approach for road networks. It employs ideas similar to PHL but considers the vertices on "significant paths" as hubs. Such vertices are simply ranked by the multiplication of vertex degree and descendant size difference. To efficiently recover the shortest paths, both PHL and SHP store predecessor vertex along with each hub label as suggested in [84]. While this increases the index size, it significantly speeds up the shortest path recovery time.

Queries: For experiments, we consider a variety of road networks taken from the 9th DIMACS challenge.[5] We generate queries as suggested in [147]: each road network is discretised into a 1024 × 1024 grid with cell side length l. We randomly generate ten groups of queries such that i-th group contains 1000 (s, d) pairs with Euclidean distance between them within $2^{i-1} \times l$ to $2^i \times l$, thus 10,000 queries in total. In discussions, we distinguish between *path queries*, which asks for a shortest (uncontracted) path from source to destination, and *distance queries*, which ask only for the length. Individual queries are run 10 times; we omit the best and worst run and average all the rest.

All algorithms (including the competitor algorithms) are implemented in C++ and compiled with -O3 flag. We use a 32 cores Nectar research cloud with 64GB of RAM and Ubuntu 18.04 LTS (Bionic) amd64.

4.5.1 Preprocessing Cost and Space

We first compare various methods for shortest path query processing in terms of the preprocessing time required and the size of the data structures required to support the method. Table 4.2 compares the CH-CPD approaches where the CPD is produced on the top 20%, 40%, 60%, 80%, or all the vertices versus other techniques. Note that the costs for CH-CPD include all the costs for constructing and storing the contraction hierarchy. We make use of the distance tables enhancement introduced earlier to speed up the CH-CPD preprocessing by caching the top 0.5% of the vertices.

[4] http://degroup.cis.umac.mo/sspexp

[5] http://users.diag.uniroma1.it/challenge9/

Type			Distance						Travel Time					
Name			NY	BAY	COL	FLA	NW	NE	NY	BAY	COL	FLA	NW	NE
#V			264k	321k	435k	1070k	1207k	1524k	264k	321k	435k	1070k	1207k	1524k
#E			733k	800k	1057k	2712k	2840k	3897k	733k	800k	1057k	2712k	2840k	3897k
Build Time (Mins)	CH-CPD	20%	0.36	0.23	0.33	3.11	2.00	5.28	0.27	0.15	0.21	2.41	1.20	3.21
		40%	0.73	0.47	0.79	8.41	10.54	24.14	0.96	0.30	0.50	5.10	5.55	12.60
		60%	1.18	0.86	2.21	19.78	25.10	56.44	1.79	0.61	1.26	10.88	13.80	27.29
		80%	1.87	1.73	4.38	36.01	46.13	108.30	2.24	1.10	2.51	21.38	26.45	52.75
		100%	2.95	3.13	7.46	56.24	73.63	167.40	3.00	1.83	4.24	33.97	42.56	85.10
	Competitors	CPD	8.76	13.37	23.07	148.57	204.57	342.81	11.03	13.84	25.38	166.38	259.31	436.75
		CH	0.24	0.14	0.19	0.44	0.43	1.18	0.16	0.09	0.10	0.29	0.32	0.59
		PHL	0.60	0.35	0.71	1.85	2.62	7.89	0.18	0.12	0.19	0.71	0.71	1.44
		SHP	0.44	0.31	0.59	1.71	2.55	7.05	0.17	0.14	0.22	0.75	0.86	1.51
Memory (MB)	CH-CPD	20%	70	60	86	189	205	436	63	50	67	152	168	325
		40%	104	80	155	328	424	836	88	63	117	261	346	602
		60%	183	132	217	542	665	1438	156	102	158	405	509	1037
		80%	271	183	268	755	868	2160	222	137	193	567	631	1599
		100%	338	213	337	984	1100	2708	277	157	241	706	789	2028
	Competitors	CPD	219	144	239	692	818	1998	188	106	174	514	586	1597
		CH	29	29	38	97	100	149	28	28	36	92	98	141
		PHL	411	302	495	1325	1515	3453	161	116	180	526	568	995
		SHP	449	359	593	1586	2008	4434	198	180	250	727	835	1350

TABLE 4.2: Number of vertices (#V) and edges (#E) in maps, build time in Mins, and memory in MB for CH-CPD and competitors.

We use road maps from the DIMACS challenge using either the distance weights or the travel time weights (which we will see are surprisingly different). Unsurprisingly, the contraction hierarchy is the cheapest approach to both compute and store. Both PHL and SHP are more expensive to compute, but only by a few factors. The CPD approaches are the most expensive to compute; what is interesting is that the CPD on the original graph is *more expensive* to compute than on the contraction hierarchy, since for the CH-CPD we can restrict to Up-Down paths and make use of caching in the preprocessing. The full CH-CPD is around 50% larger than the CPD on the original graph, even though it is cheaper to compute. Partial CH-CPDs are significantly cheaper to both compute and store than the full CH-CPD, reaching the same ballpark compute times as PHL and SHP at 20%. What is somewhat surprising is that the full CH-CPD storage costs are significantly smaller than both PHL and SHP on the Distance maps, while in the Travel Time maps they tend to be larger than PHL and around the same size as SHP.

Overall, we see that the storage requirements for CH-CPDs are not overwhelming, and indeed can be smaller than the competitors. The use of partial CH-CPDs means we can trade off

Type			Distance				Travel Time			
Name			NY		NE		NY		NE	
Algorithm		#C	#CE	TT	#CE	TT	#CE	TT	#CE	TT
CH-CPD	20%	0%	0	0.4	0	11	0	0.4	0	13
		0.25%	13.48	0.12	15.64	4.39	4.97	0.12	6.04	2.37
		0.50%	12.83	0.12	15.64	4.10	5.05	0.12	6.00	2.62
		1.00%	12.60	0.12	15.09	4.42	4.97	0.13	5.99	2.52
	40%	0%	0	1.5	0	49	0	1.6	0	60
		0.25%	12.35	0.51	17.90	26.08	6.04	0.80	5.86	12.01
		0.50%	12.24	0.49	17.70	22.96	6.01	0.81	5.85	12.01
		1.00%	12.20	0.56	17.69	24.44	6.00	0.78	5.76	12.03
	60%	0%	0	3.2	0	126	0	4.0	0	156
		0.25%	12.70	0.94	16.58	57.16	5.88	1.56	5.61	24.82
		0.50%	12.62	0.94	16.55	55.26	5.87	1.63	5.55	26.71
		1.00%	12.65	1.00	16.59	56.82	5.88	1.67	5.55	28.02
	80%	0%	0	6.0	0	255	0	7.4	0	307
		0.25%	13.22	1.57	16.27	98.66	5.38	1.98	5.61	48.41
		0.50%	13.20	1.63	16.26	107.12	5.38	2.08	5.55	52.17
		1.00%	13.24	1.71	16.31	99.02	5.39	2.22	5.56	52.88
	100%	0%	0	9.9	0	382	0	12	0	489
		0.25%	12.45	2.56	16.28	150.07	5.26	2.68	5.58	81.97
		0.50%	12.45	2.71	16.28	166.22	5.26	2.84	5.58	84.52
		1.00%	12.49	2.81	16.32	148.02	5.26	2.91	5.57	82.65

Table 4.3: Average number of cache vertices expanded (#CE) and total build time (TT) in minutes for constructing CPD for various proportions of the contraction hierarchy. We show the number of cached vertices (#C) in range of 0 to 1% of the total vertices.

storage requirements with query time.

Table 4.3 shows some results about the effectiveness of the caching preprocessing enhancement. The table shows the average number of cached vertices expanded when building CPDs for various proportions of the contraction hierarchy, as well as the total build time to build the (partial) CPD, for the smallest and largest map (not including the CH build time as in Table 4.2). We vary the number of cached vertices to be 0% (no distance tables enhancement), and 0.25%, 0.5% and 1% of the total number of vertices in the map. Clearly the caching preprocessing enhancement is highly effective, reducing construction times to one third, and by more on larger maps. The percentage of cached vertices does not make that much difference, as clearly the number of expanded cached vertices hardly changes. The results on the other maps are similar. Note that this approach can be adapted to improve the preprocessing of traditional CPDs as well. Although not reported here, we observed improvement by a few factors for constructing traditional CPDs by caching the distances on the same top-n% of vertices. But the choice of which vertices to cache is important to the success of the approach. With randomly chosen cache vertices, we observed a slowdown as it incurs a large number of distance updates in the order of $|CE| \times |E|$. Our optimisation is not limited by certain ordering and we believe that other intuitive lexical orderings (e.g., vertices having high ”reach”) may further improve preprocessing of traditional CPDs.

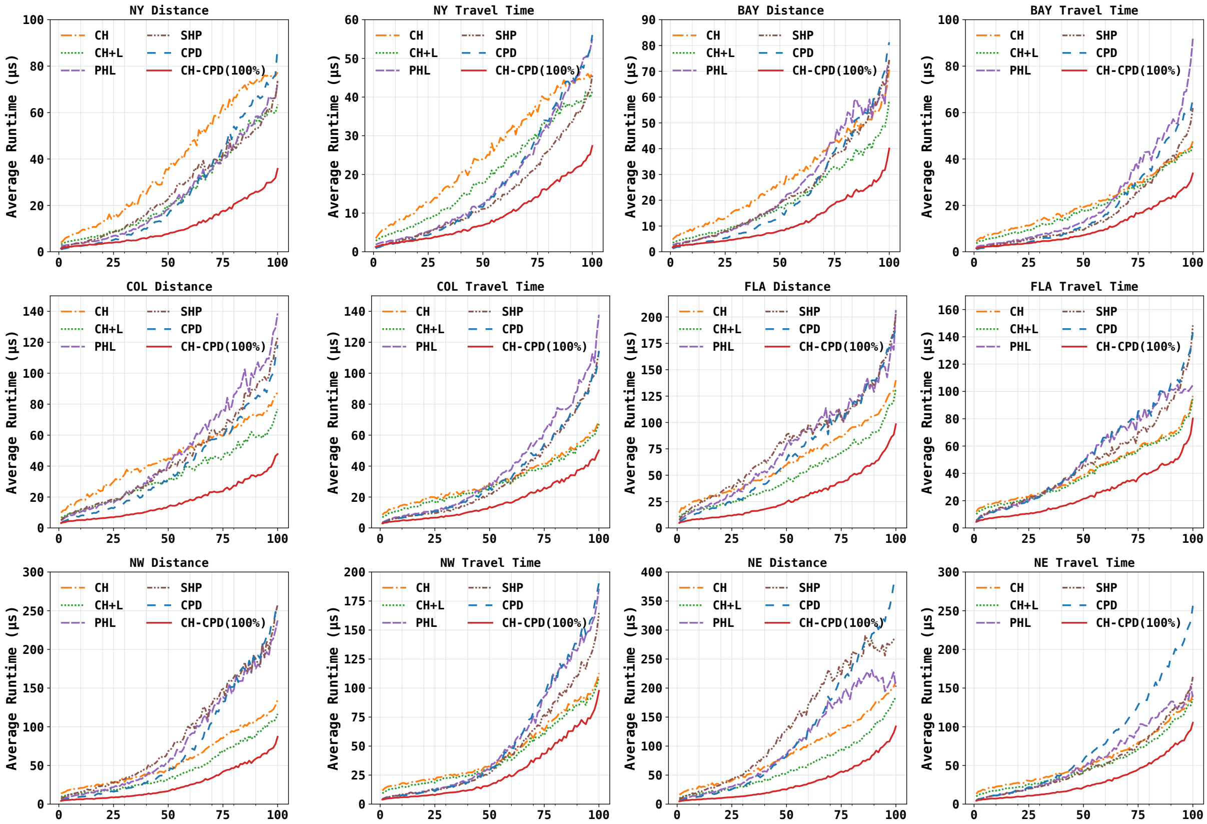


Figure 4.4: Runtime comparisons on the six road network Distance and Travel Time graphs. The x-axis shows the percentile ranks of path queries sorted based on actual distances between source and destination.

4.5.2 Query Processing Time

Next, we compare path query processing times of the various methods. Figure 4.4 shows cactus plots for each competing algorithm on 12 different graphs. Note that for each CH-based method, the path query time includes path unpacking. Examining the results we can see that in terms of worst case performance contraction hierarchies are particularly important. The top three methods in the larger maps are CH-CPD, CH+L and CH. Other methods can be faster for some queries, since they avoid path unpacking, but by the time we reach 100% of queries solved, these methods dominate. Interestingly, Landmarks are significantly beneficial for CH on the Distance maps while making almost no difference on the Travel Time maps. Basic CPD is very good for queries with short paths but becomes less competitive as more extractions are required. CH-CPD is faster than all other approaches on all maps; the combination of no search together with few lookups finds optimal paths very quickly.

In Table 4.4, we extend the comparison to consider the shortest distance queries and also include the partial CH-CPDs. Unsurprisingly, the distance based methods PHL and SHP are far superior for distance queries, since the other methods essentially find the shortest path and then calculate its length (although they can avoid path unpacking). We can see that the partial

Type			Distance						Travel Time					
		Name	NY	BAY	COL	FLA	NW	NE	NY	BAY	COL	FLA	NW	NE
Distance Queries	CH-CPD	20%	12.83	10.83	12.96	18.91	17.53	25.97	10.53	9.15	10.28	14.31	15.83	19.59
		40%	10.99	9.01	10.18	15.66	14.87	19.06	9.77	7.79	8.56	12.52	13.34	15.39
		60%	7.57	6.69	8.43	11.69	11.41	14.46	6.81	5.77	6.05	10.15	10.29	11.33
		80%	5.32	4.50	5.80	7.91	7.68	8.91	4.72	3.75	4.89	6.78	7.25	7.63
		100%	3.23	2.79	3.64	5.02	4.99	6.44	2.55	2.31	3.01	4.19	4.19	5.51
	Competitors	CH	28.33	17.99	29.43	31.10	30.90	53.54	16.53	12.23	14.63	17.43	18.79	27.10
		CH+L	15.64	11.59	17.28	21.50	19.45	30.97	11.83	10.55	12.50	15.33	16.58	21.65
		PHL	0.92	0.69	0.94	1.19	0.98	1.82	0.55	0.50	0.54	0.59	0.57	0.70
		SHP	1.01	0.72	0.87	1.04	1.05	1.76	0.57	0.49	0.51	0.58	0.55	0.65
Path Queries	CH-CPD	20%	23.17	19.78	29.06	49.12	41.87	62.34	19.16	19.06	26.22	38.40	41.39	49.71
		40%	20.83	18.90	26.33	44.66	38.82	51.88	18.82	17.36	24.48	35.68	39.32	43.22
		60%	17.19	16.67	23.37	39.61	35.05	47.27	14.83	15.11	20.85	32.73	36.03	37.28
		80%	13.67	13.75	20.18	34.63	29.84	40.31	12.85	12.54	19.97	29.11	32.22	32.09
		100%	11.42	11.78	17.01	30.62	26.28	37.70	9.53	10.61	17.26	25.52	27.95	31.22
	Competitors	CPD	26.38	22.85	39.75	70.86	76.62	124.37	18.23	19.15	34.76	54.15	55.38	79.94
		CH	38.64	28.69	46.00	63.63	57.42	90.85	25.27	21.64	31.54	42.21	44.97	58.04
		CH+L	25.58	20.85	33.96	50.68	44.28	65.57	20.06	19.97	29.43	40.44	41.71	51.55
		PHL	25.36	25.88	49.81	76.54	81.40	105.20	18.32	22.86	39.85	51.90	54.71	59.43
		SHP	26.76	23.90	45.15	80.53	89.20	138.86	14.93	16.79	33.47	49.21	46.61	53.16

TABLE 4.4: Running time comparison for distance and path queries. We report average time (µs) between CH-CPD and competitors.

CH-CPDs roughly double the query time when moving to a 20% CH-CPD, where the query times are roughly still slightly ahead of CH+L and still significantly better than PHL and SHP for path retrieval.

Table 4.5 provides more insights. CH+L performs significantly better than CH due to the smaller number of vertices generated and expanded using the landmark heuristic. As shown by CPD Usage, CH-CPD does not always need to use the CPD (when the optimal path does not pass through CPD vertices). For such queries, CH-CPD essentially is the same as CH+L. Note that the numbers of CPD vertices expanded from source ($|V_s|$) and destination ($|V_d|$) are pretty small. Furthermore, the number of paths extracted using CPD is also significantly smaller than $|V_s| \times |V_d|$ which shows the effectiveness of our pruning rules that also help significantly reduce the number of first move extractions. Compared with the CPD on the original graph, the full CH-CPD requires a significantly smaller number of first move extractions which explains its superior performance.

Type	Name	Stat	CH-CPD					Competitors		
			20%	40%	60%	80%	100%	CPD	CH	CH+L
Distance	NY	#Generated	62.63	52.10	18.99	6.98	-	-	301.53	123.44
		#Expanded	22.05	17.86	7.42	3.01	-	-	101.30	41.47
		CPD Usage	68.4%	75.0%	90.6%	95.5%	100%	100%	-	-
		$\|V_s\|$	3.30	2.59	2.14	1.38	1	1	-	-
		$\|V_d\|$	3.29	2.57	2.09	1.39	1	1	-	-
		#Path	2.91	1.96	1.54	1.15	1	1	-	-
		#FirstMove	25.53	18.56	15.46	11.16	9.59	191.74	-	-
	NE	#Generated	87.95	51.54	21.33	1.16	-	-	503.16	208.47
		#Expanded	29.72	18.14	7.84	0.63	-	-	149.77	62.80
		CPD Usage	81.5%	90.0%	96.9%	99.9%	100%	100%	-	-
		$\|V_s\|$	4.32	3.17	2.28	1.21	1	1	-	-
		$\|V_d\|$	4.37	3.19	2.24	1.23	1	1	-	-
		#Path	4.00	2.44	1.65	1.09	1	1	-	-
		#FirstMove	41.31	26.51	19.67	13.55	12.43	607.93	-	-
Travel Time	NY	#Generated	57.17	45.27	20.22	7.46	-	-	161.92	93.25
		#Expanded	24.08	18.55	8.81	3.56	-	-	71.82	40.29
		CPD Usage	66.3%	74.7%	89.9%	95.1%	100%	100%	-	-
		$\|V_s\|$	2.51	2.09	2.00	1.32	1	1	-	-
		$\|V_d\|$	2.48	2.06	1.97	1.34	1	1	-	-
		#Path	2.05	1.65	1.58	1.17	1	1	-	-
		#FirstMove	15.33	14.36	15.04	11.52	9.95	168.59	-	-
	NE	#Generated	73.96	42.74	19.22	1.15	-	-	197.16	121.69
		#Expanded	32.07	19.02	8.71	0.64	-	-	89.64	54.05
		CPD Usage	77.9%	89.5%	96.2%	99.9%	100%	100%	-	-
		$\|V_s\|$	2.74	2.35	1.88	1.22	1	1	-	-
		$\|V_d\|$	2.76	2.35	1.88	1.23	1	1	-	-
		#Path	2.29	1.84	1.48	1.12	1	1	-	-
		#FirstMove	20.56	19.47	17.76	14.44	12.96	520.37	-	-

TABLE 4.5: Average number of vertices #Generated and #Expanded by each algorithm. CPD Usage corresponds to % of queries for which both A* searches expand at least one CPD vertex (and hence end up using CPD). For queries that use CPD, we report average $|V_s|$ (resp. $|V_d|$) that denote # of CPD vertices expanded from s (resp. d), and average #Path and #Firstmove extractions.

4.6 Discussion

We show how to use Compressed Path Databases and Contraction Hierarchies to generate the fastest shortest path query retrieval method we are aware of. The use of Contraction Hierarchies also allows us to cache information for the CPD construction that actually makes CPD construction significantly faster. We also show how we can tradeoff preprocessing time and space with path retrieval time by building partial CPDs. While path retrieval now requires search it is still highly competitive with other methods.

Chapter 5

Improving Time-Dependent Contraction Hierarchies

5.1 Overview

Time-dependent Contraction Hierarchies (TCH) [32, 85] is the state-of-the-art algorithm in time-dependent road networks. TCH is a family of successful speed-up techniques that embeds the road network into a hierarchical graph (see section 2.2.2). There are however two drawbacks: (i) TCH inherits the bidirectional Dijkstra search from static Contraction Hierarchies [76]. This approach does not rely on any lower-bounding heuristics for guidance, although such methods are known to improve performance. (ii) In time-dependent road networks, TCH is built by considering the entire time domain T, in order to answer all queries $q \in T$. However, each individual query only corresponds to a trip within a limited time period T_q, such that $T_q \subset T$. Embedding the travel time metric for the entire time domain T can increase the size of TCH search space, which again affects query performance. In this chapter, we investigate how to improve the TCH algorithm. In broad strokes, our strategy is as follows:

- We show the traditional bidirectional Dijkstra search on TCH can be extended to a bidirectional A* search using landmark heuristics [16].
- We revise the bidirectional search on TCH to a forward A* search, this allows us to combine the search with a path databases heuristic [48] (i.e., CPDs and RPDs). We also consider the contracted graphs in combination with CPDs, which further improve the lower bound and the performance for extracting such a bound.
- We propose to build a set of smaller TCHs, each of which focuses on a subset of the time domain. By choosing the appropriate TCH for each query $q \in T$, we can retain

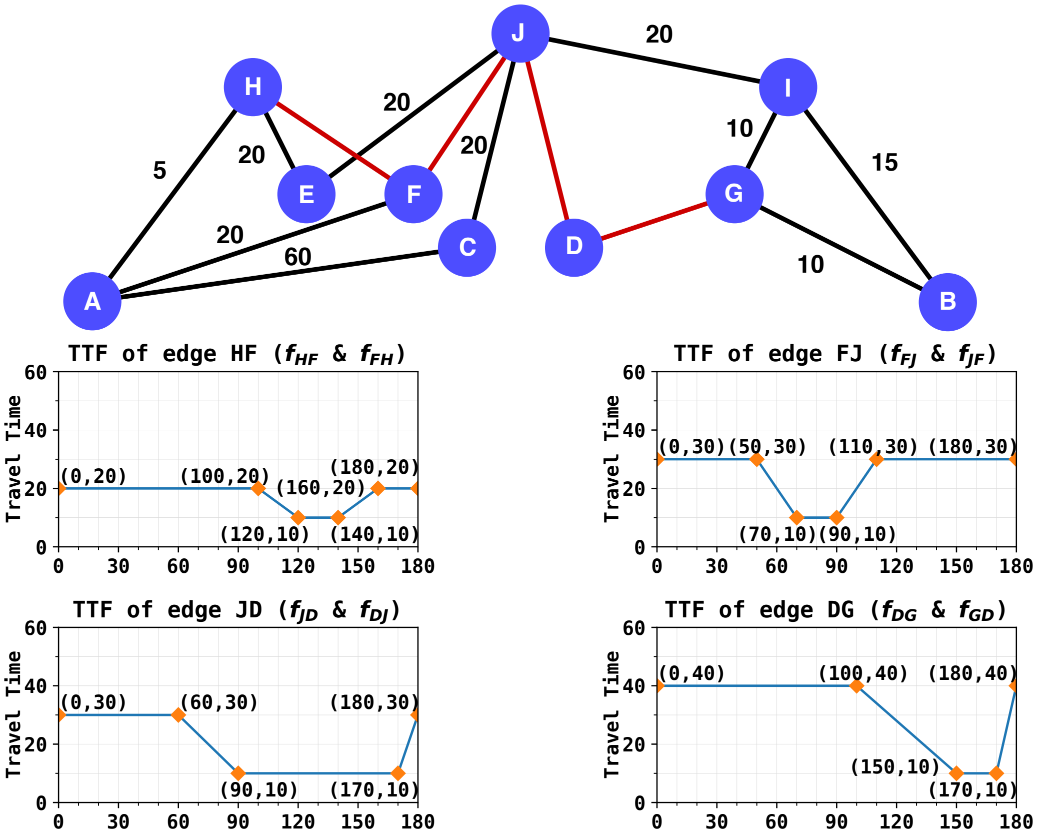


FIGURE 5.1: Same as Figure 2.7, where we show an example of an undirected time-dependent graph. TTFs of the red edges are shown below the graph, and the travel cost of the other edges are constant.

the optimality guarantees of the original algorithm while substantially improving search performance.

We give a complete description of the new algorithms, and evaluate them on a range of road networks, including real-world as well as synthetic datasets. Results show substantial improvement over the baseline TCH method.

5.2 Preliminaries

In a time-dependent road network, the network is modelled as a directed graph, where each edge is associated with a piece-wise linear function that represents the varying travel time of traversing the edge over the time period T (usually 24 hours). For the ease of presenting our techniques in this chapter, we redraw the Figure 5.1 to show an example of a time-dependent network, where we assume the graph G is undirected and only the edges that are highlighted in red have non-constant Travel Time Function (TTF) with $T = [0, 180)$. Any two locations in the time-dependent network can be the source s and destination d. Given a departure time $t \in T$,

Symbol	Description
G	A time-dependent graph.
V	A set of vertices in the input graph G.
E	A set of edges in the input graph G.
F	A set of functions that map each edge in G to a travel time function.
T	The time domain of the input graph G.
s	The source of a pathfinding query and $s \in V$.
d	The destination of a pathfinding query and $d \in V$.
t	The departure time of a pathfinding query and $t \in T$.
q	A pathfinding query q, i.e., $q = (s, d, t)$.
v_i	A vertex in the input graph G and $v_i \in V$.
$e_{v_i v_j}$	A directed edge that connects v_i to v_j in the input graph G and $e_{v_i v_j} \in E$.
$f_{v_i v_j}$	The travel time function of the directed edge $e_{v_i v_j}$ and $f_{v_i v_j} \in F$.
$f_{v_i v_j}(t)$	Evaluating the travel time function $f_{v_i v_j}$ at the departure time t.
$f_{v_i v_j} \circ f_{v_j v_k}$	Chaining the travel time functions $f_{v_i v_j}$ and $f_{v_j v_k}$ of two edges $e_{v_i v_j}$ and $e_{v_j v_k}$.
$min(f'_{v_i v_j}, f''_{v_i v_j})$	Merging the two travel time functions $f'_{v_i v_j}$ and $f''_{v_i v_j}$ of a same edge $e_{v_i v_j}$.
P	A path that consists of a sequence of vertices $\langle v_1, v_2, \cdots, v_k \rangle$.
$\Sigma\lvert P\rvert$	The cost (i.e., length) of a path P i.e., $\Sigma\lvert P\rvert = f_{v_0 v_k}(t) = f_{v_0 v_1} \circ f_{v_1 v_2} \ldots \circ f_{v_{k-1} v_k}$.
$g(v_i, v_j)$	The tentative distance of a search node from source v_i to current vertex v_j.
$sp(v_i, v_j, t)$	The shortest path from v_i to v_j at the departure time t.
$fm(v_i, v_j)$	A function that extracts the *first-move* between v_i and v_j using path databases.
$landmark(v_i, v_j)$	The lower-bound heuristic value returned by landmarks between v_i and v_j.

TABLE 5.1: Summary of the notations used in this chapter

the objective is to find the fastest (i.e., shortest) path traversing from s to d at departure time t. In this chapter, we follow the definitions and terminologies discussed in the Section 2.2.4, and Table 5.1 shows a summary of the symbols used in this chapter.

In this chapter, we focus on improving the Time-dependent Contraction Hierarchies (TCH) [32, 85], where the details of TCH can be found in Section 2.2.4.1. Figure 5.2 shows an example of TCH by contracting the time-dependent graph shown in Figure 5.1. The dashed edges indicate the shortcuts added by contracting the vertex E and F in purple and D in red. Our techniques also involve two important ingredients: landmarks heuristic [28] and Path databases heuristic [48], we give the description of both algorithms in Section 2.2.4.1 and the detail of constructing the Compressed Path Databases (CPD) and Reverse Path Databases(RPD) can be found in Section 2.2.3.

5.3 Improving Search on TCH

Given a time dependent graph G, we construct the Time-dependent Contraction Hierarchy. For a given pair of source and destination and its departure time t, the traditional Bidirectional TCH search (BTCH) is a successful and efficient method to solve the pathfinding problem. However, we show the search algorithm can be further improved (i) by combining the BTCH with landmark heuristics; (ii) by changing the search to a forward search and combining with

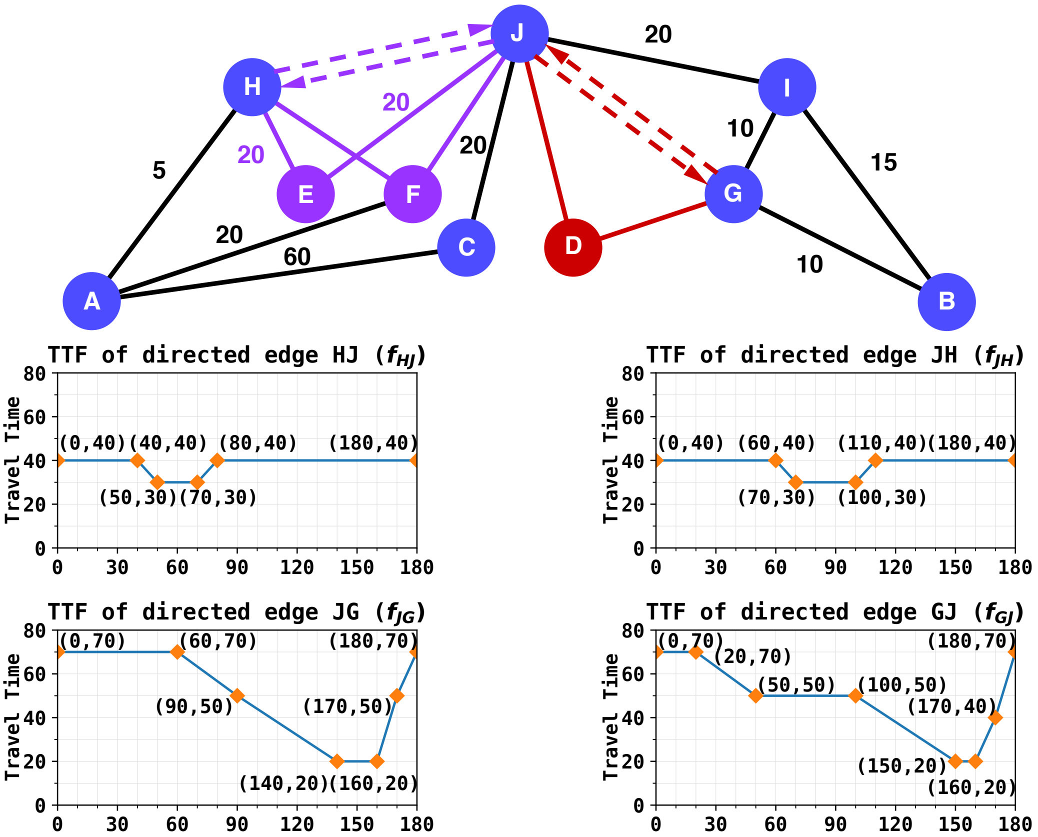


FIGURE 5.2: Same as Figure 2.9, where we show the result for contracting nodes E and F in purple, and D in red. Dashed edge are the shortcut edges and their corresponding TTFs are shown in the figure below.

a more advanced CPD-based heuristic: TCPD heuristic; and (iii) by splitting the time domain T into multiple T' (i.e., $T' \subseteq T$), such that each query in T' can be efficiently solved by TCH constructed for T' only.

5.3.1 Combining BTCH with Landmarks

In time-dependent road networks, landmarks have been widely used in many existing techniques, such as bidirectional A* search [29], core routing [30] and SHARC routing [31]. Following the success of these algorithms, we extend BTCH using landmarks:

(i) During the *Bidirectional Search*: We employ the bidirectional A* search instead of bidirectional Dijkstra search, and the search on each side is guided by the landmark heuristic, i.e., the f-value for a vertex v is $g(s, v) + landmark(v, d)$ with $g(s, v) \geq \Sigma|sp(s, v, t)|$ being a tentative upper bound for the shortest path from s to v. We also apply the pruning rule used in the BTCH and mark the edges that are traversed by the backward search as E_{trv}. The search terminates when the minimum f-value for both A* searches are larger or equal to the smallest upper bound U so far.

(ii) During the *Forward Search*: Recall that the backward search in the bidirectional phase computes the lower bound $lb(v, d)$ on each vertex v using lower-bound edge weights of the incoming edges in $E_{\downarrow}$ and records the edges that have been traversed as E_{trv}. We argue that the lower bound $lb(v, d)$ can be directly used as a heuristic, because $lb(v, d) \leq \Sigma|sp(v, d, t)|$ for $\forall t \in T$. $lb(v, d)$ is also more effective than the landmarks as this lower bound computes the true minimum distance in $E_{\downarrow}$ (i.e., $lb(v, d) \geq landmark(v, d)$).

Combining the landmarks heuristic with bidirectional search speeds up the search in both directions, and results in a smaller subset E_{trv} compared with the Dijkstra search. Although the forward search is funnelled into the direction of d and is usually cheap to run, directly reusing the previously computed lower bound further improves the query performance, as we show later in the experiments section.

5.3.2 Forward TCH Search with CPD-based Heuristic

Due to the fact that the travel cost on a TTF can not be computed without a known departure time t, forward search seems to be a natural way to solve the pathfinding problem in a time-dependent road network. This motivates us to revise the search on TCH to be a forward search. In addition to landmarks, we consider a more sophisticated goal-directed heuristic, called the TCPD heuristic, and we show this heuristic (i) requires fewer first move extractions; and (ii) provides a more effective lower bound compared to the CPD heuristic. To further improve the query performance, we also propose several pruning rules and optimisations.

5.3.2.1 Forward TCH Search (F-TCH):

For a given source s and destination d, suppose we expand a search node n with its predecessor $pred(n)$, a time-dependent Dijkstra search on the TCH typically generate successors $succ(n)$, which fall into one of the following types:

1. Up-Up successors: $pred(n) <_{\mathcal{L}} n$ and $n <_{\mathcal{L}} succ(n)$;
2. Up-Down successors: $pred(n) <_{\mathcal{L}} n$ and $n >_{\mathcal{L}} succ(n)$;
3. Down-Down successors: $pred(n) >_{\mathcal{L}} n$ and $n >_{\mathcal{L}} succ(n)$;
4. Down-Up successors: $pred(n) >_{\mathcal{L}} n$ and $n <_{\mathcal{L}} succ(n)$;

Given a departure time t, although this uni-directional Dijkstra search finds the shortest path $sp(s, d, t)$, the search is unlikely to be efficient without avoiding the non tch-paths. Therefore, we modify the search to consider only the tch-paths by a simple pruning rule called UTD (Up-Then-Down) [146]. Recall that a tch-path is always an UP-Down path with an apex vertex which is lexically larger than all the other vertices on the path. The F-TCH needs to generate (i) the type 1 and 3 successors, which covers the Up or Down tch-paths (i.e., s or d is the apex vertex) by continuously moving up or down; (ii) the type 2 successors, which covers the Up-Down tch-paths (i.e., an intermediate vertex k is an apex vertex and $k >_{\mathcal{L}} v$ for $v \in \langle s, \ldots k-1\rangle \mid \langle k+1, \ldots d\rangle$) by switching the direction at the apex vertex. Therefore, the only type of successors that is disallowed by UTD is the Down-Up successors, which leads to a non tch-path.

5.3.2.2 TCH-CPD (TCPD) heuristic:

Given a time-dependent graph G, we construct a CPD using the minimal edge cost of each edge, and use this CPD in a forward TCH search as a heuristic to further improve the search. However, path extraction for CPD requires a number of lookups (i.e.,$fm(s,d)$) equal to the number of edges on the optimal path, thus can be costly when used as a heuristic. To mitigate such defects, we propose to combine the CPD with TCH. Our idea is motivated by the previous Chapter 4 that shows combining CPD with contraction hierarchies in a static graph reduces the first move extractions significantly. However, combining CPD with TCH is more complicated.

Given a TCH constructed on a time-dependent graph G, we obtain a contracted graph by taking the lower-bound value of each TTF. In order to build the CPD, we further modify F-TCH to compute the first moves on the optimal tch-path, from each source vertex s toward $\forall d \in V$. For a given source vertex s, we divide the search node v with its parents $pred(v)$ into two types: (i) Up-reach search node v_u: $pred(v_u) <_{\mathcal{L}} v_u$. (ii) Down-reach search node v_d: $pred(v_d) >_{\mathcal{L}} v_d$. At each search node v, we allow the search to independently expand each type of search nodes once with $g(s, v_u)$ and $g(s, v_d)$ being the best tentative distance respectively. The search terminates when the OPEN list becomes empty, and for each vertex v, we have (i) $g(s, v_u)$ which computes the shortest up tch-paths; and (ii) $g(s, v_d)$ which covers all the shortest up-down or down tch-paths. Therefore, the first move from s to any v on the optimal tch-paths can be easily obtained from the path $min(g(s, v_u), g(s, v_d))$.

Example 5.1. *In Figure 5.3, the contracted graph is taken from the TCH in Figure 5.2. Assume we compute the first move row on the source vertex* `A`*. A simple F-TCH would falsely prune the successor* `J` *when expanding search node* `F` *because the best tentative solution from* `A` *to* `F` *is* $\langle A, H, F\rangle$ *and the successor* `J` *is a down-up successor (i.e.,* $H >_{\mathcal{L}} F <_{\mathcal{L}} J$*). However, the modified F-TCH computes the optimal path* $\langle A, F, J\rangle$ *correctly as it manages up and down search nodes separately, i.e., the down-up successor pruning is only performed when expanding a down-reachable search node. Also note the first moves from source node* `H` *are now on the optimal tch-path.*

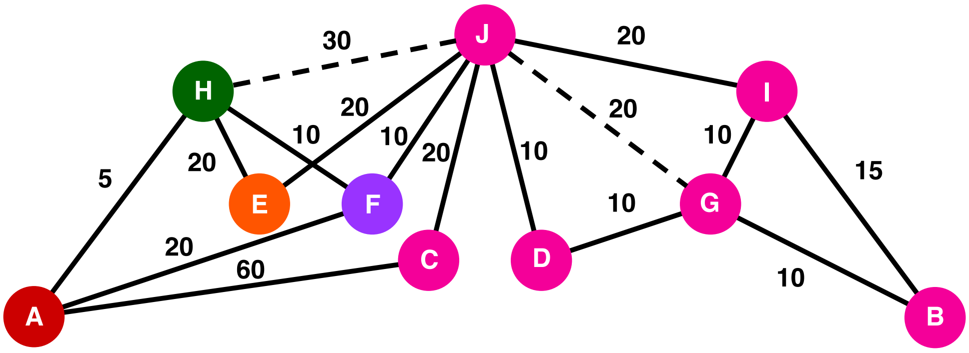


FIGURE 5.3: From the source vertex `H`, the optimal first move to any vertex are `A` (red), `E`(orange), `F` (purple) and `J` (pink)

With the modified F-TCH search, we now construct the TCPD following the same general procedures already described. Given a source and destination, we denote the path extracted from TCPD as $tcpd(s, d)$.

Theorem 5.1. *Given a pair of* (s, d) *and departure time* $t \in T$ *in a TCH,* $\Sigma|tcpd(s, d)|$ *is a lower bound for* $\Sigma|sp(s, d, t)|$.

Proof. Given a TCH constructed in G, there must exist a cost equivalent tch-path for every $sp(s, d, t)$ and $t \in T$. In a lower-bound graph of TCH, TCPD computes and encode the first move on the optimal path, from any s to any d by examining all possible (i) up; (ii) up-down; and (iii) down tch-path. Thus, $\Sigma|tcpd(s, d)|$ defines a lower bound, i.e., $\Sigma|tcpd(s, d)| \leq \Sigma|sp(s, d, t)|$ for $\forall t \in T$. □

Due to Theorem 5.1, $\Sigma|tcpd(s, d)|$ defines an admissible heuristic and can be easily combined with forward TCH search. Recall that the lower bound on a shortcut edge $min(f_{uw}) \geq min(f_{uv}) + min(f_{vw})$, thus $\Sigma|tcpd(s, d)|$ is a tighter lower bound, i.e., $\Sigma|tcpd(s, d)| \geq \Sigma|cpd(s, d)|$. Next, we introduce several optimisation techniques to further improve the query performance.

5.3.2.3 Downward Successors Pruning:

Although the F-TCH prunes the up-successors, when expanding the node n, it has to generate every down-successor v no matter whether $pred(n) >_{\mathcal{L}} n$ or $pred(n) <_{\mathcal{L}} n$. However, not every down-successor v can lead to a path $\langle n, \ldots d\rangle$ that reaches the destination d from the apex vertex n. Therefore, we reuse the concept of CPD and propose a *reachability oracle*, called *Reach*, that tells whether there is a down path $\langle s, \ldots d\rangle$, from a given vertex s to any vertex d. Whenever the F-TCH generates the down-successors n, we use $Reach(n, d)$ to prune non-reachable successors. Eliminating the down-successors early helps the search to reduce the branching factor. Next, we discuss the construction of *Reach*.

To build *Reach*, we first compute a down-reachability table (DT) by running a Depth First Search (DFS) on each source vertex s that only generates the down successors (i.e., $succ(n) >_{\mathcal{L}} n$). Although DT is a simple truth table, without a better ordering, it does not compress well. But note that the TCH is a hierarchical data structure that is rooted from one or more important vertices, and these vertices are assigned to the largest lex order possible. Therefore, we run a DFS from the vertex that has the largest lex order. Similarly, the search only visits the down-successors, not earlier expanded by a vertex with larger lex order. We order the columns of DT according to the order of vertices accessed by this DFS, and compress it using RLE. With this novel ordering, *Reach* can be compressed effectively (e.g., in a graph with millions of vertices, *Reach* only needs a few MB to store), and each binary search $Reach(s,d)$ runs in near constant time.

5.3.2.4 Cost Caching:

When TCPD is used as a heuristic, it needs to continuously extract the path $tcpd(s,d)$ at each node expansion of s. To further reduce the number of $fm(s,d)$, we apply a simple cost caching strategy. When we extract a path from s to d, we cache distance $dist(v,d) = \Sigma|tcpd(v,d)|$ for each vertex v on the extracted path $tcpd(s,d)$. Later on, for a subsequent path extraction $tcpd(s',d)$, we terminate early if the path extraction reaches the vertex v such that $dist(v,d)$ is cached, and we use the cached distance to get the path length from s' to d. Similar cost caching strategies have also appeared in CPD-based search [45], and CPD-heuristics [48], since they significantly reduce the first move extractions.

5.3.2.5 Reverse TCH Path Database:

Another way to improve the path extraction of $tcpd(s,d)$ is to speed up each $fm(s,d)$ by building a reversed tch path database (RTPD) that is similar to the RPD as discussed earlier. Recall that a RPD computes a reverse first move table RR that records the first move on an optimal path from any s to a given destination vertex d. Similarly, on each source vertex s, we compute $RR(s)$ in TCH using a modified F-TCH which considers only the incoming edges of TCH. RTPD and TCPD essentially compute the same first moves, but only store in a different way, thus all the properties of TCPD preserved.

5.3.3 Splitting the Time-domain

In order to handle any arbitrary query that is issued at $t \in T$, a TCH is usually built by considering the entire time domain T. However, each individual query only requires us to consider a much shorter time period $T_q = [t, t']$, where t' is the optimal arrival time when travelling from s to d at t. Therefore, the TCH has two drawbacks: (i) In each edge, the TTF stores all interpolate points w.r.t T, thus requires more time to evaluate the travel time for t, and results in a looser lower and upper bounds than T_q. (ii) For each shortcut edge e_{uw}, e_{uw} is added if $\exists T' \subseteq T$ such that $\langle u, v, w \rangle$ is an optimal path from u to w during T', such a shortcut edge may be unnecessary for T_q. To mitigate these disadvantages of TCH, we propose to split T into n number of equal-sized time buckets T_i (i.e., $T_i = [t_i, t_{i+1}] = |T|/n$ for $0 \leq i \leq n-1$) for processing start times $t \in [t_i, t_{i+1})$. For each bucket T_i, we build a TCH to cover all $\Sigma|sp(s, d, t)| \leq t_u$ by adding an upper bound t_u, and building TCH$_i(T)$ over time range $[t_i, t_{i+1} + t_u]$. First, we show how to choose the upper bound t_u to form a single-layer TCH, denoted as STCH, such that $\exists$ TCH$_i \in$ STCH and TCH$_i(T) \supseteq T_q$ for all queries $q \in T$. We then describe a multi-layer TCH which combines STCHs with customised t_u. For each individual query $q \in T$, we show q can be improved using TCH built in STCH with minimal t_u.

5.3.3.1 Single-layer TCH

In the time-dependent shortest path problem, highly accurate solutions are needed, especially when a user needs to plan a short-term trip. Therefore, our main focus is on city-sized road networks. From point to point in a city-sized graph, the shortest path $sp(s, d, t)$ does not typically take a large amount of time (e.g., travel within Melbourne almost always requires less than four hours). Suppose $\Sigma|sp(s, d, t)| < t_u$ for every possible start s, destination d and start time t. We build a TCH for each start time bucket $T_i = [t_i, t_{i+1}]$ which only considers travel in the time $[t_i, t_{i+1} + t_u]$. Then we can answer a shortest path query $sp(s, d, t)$ correctly by using TCH$_i$ built for time bucket T_i where $t \in T_i$ if travel time is no more than t_u.

Example 5.2. *Consider the example in Figure 5.2. Assume we split the time domain $T = [0, 180]$ into 6 time buckets, i.e., $|T_i| = 30$ for $0 \leq i \leq 5$. Since $\forall sp(s, d, t) \leq 90$ for $t \in T$ and $(s, d) \in G$, we set the upper bound $t_u = 90$. Notice that contracting the vertex $\mathtt{D}$ for T_0 (i.e., $T_0 = [0, 120]$) does not add the shortcut $e_{\mathtt{JG}}$, as $\langle \mathtt{J}, \mathtt{I}, \mathtt{G} \rangle$ is a shorter path than $\langle \mathtt{J}, \mathtt{D}, \mathtt{G} \rangle$ (i.e., $20 + 10 \leq min(f_{\mathtt{JD}} \circ f_{\mathtt{DG}})$).*

5.3.3.2 Multi-layer TCH

Although we can predict an upper bound t_u that is large enough for solving all queries, the upper bound t_u may not be efficient, because many, if not most travel time of queries $q \in T$ may be much smaller than t_u. In order to solve q using a TCH such that TCH(T) is as small as possible, we propose to form multiple STCHs with different t_u into a Multi-layer TCH (MTCH). In the *offline* phase, we build MTCH as the following:

1. At the root R of MTCH, we construct a TCH w.r.t the entire time domain T. In addition, we build a TCPD as discussed earlier.
2. For each lower layer j of R, we build STCH_j by splitting T into n time buckets T_i, and the upper bound t_u can be set to any customised value, but only needs to be less than t'_u, where t'_u is the upper bound used in STCH_{j-1}.

During the *online* phase of MTCH, given a query $sp(s, d, t)$ with $t \in T_i$, we first obtain an upper bound $U(s, d)$ using the maximal travel cost for the path extracted from TCPD between s and d. Therefore, we know the query can be solved in $T_q = [t, t + U(s, d)]$. For TCHs built in time bucket T_i, we check each layer of MTCH via a top-down scan from STCH_0 to STCH_j. The scan terminates when $\text{TCH}_i(T) \not\supseteq T_q$, and returns TCH_i with minimal $\text{TCH}_i(T)$ that covers T_q. If TCH_i is found, we answer the query using the TCH_i. Otherwise, we answer the query by running a forward TCH search in R, with TCPD guiding as a heuristic. MTCH is guaranteed to solve all queries $q \in T$ in R, and each $q \in T_i$ is safely improved by using the TCHs maintained in lower layers.

5.4 Experiments

We test our proposed algorithms against the baseline implementation of Time-dependent Contraction Hierarchies (**TCH**), taken from the repository[1] of original authors [85]. The implementation is also known as KaTCH. In a recent study [86], it was shown that KaTCH is the state-of-the-art algorithm and outperforms a range of optimal algorithms, including CATCHUp [86], TD-CALT [30], and TD-SHARC [31]. Meanwhile, our approaches: (i) the Single-layer TCH (**STCH**) is built by splitting the time domain into 24 hourly buckets and the upper bound t_u is set to 4 hours as no trip in our tested maps exceeds this limit. (ii) The Multi-layer TCH (**MTCH**) is built by adding three layers of STCH under the root R. For each layer, we maintain 24 hourly time buckets and set t_u to 4h, 2h and 1h correspondingly.

For the heuristics, we further compare our approaches with Compressed Path Databases [27] and take the implementation from the publicly available repository.[2] By **CPD**, we mean the Compressed Path Databases that is built on the contracted graph of TCH. On the other hand, our approach **L12** indicates using 12 landmarks for travel time estimation (we varied the number of landmarks from 4 to 16, and 12 appears to be the best). **TCPD** and **RTPD** refer to TCH-based Compressed Path Databases and Reverse TCH Path Database, respectively. We also use the letter **B** and **F** to denote the Bidirectional search and Forward search, respectively. For example, **B-TCH(L12)** denotes our algorithm bidirectional TCH search with landmarks

[1] https://github.com/GVeitBatz/KaTCH

[2] https://bitbucket.org/dharabor/pathfinding

heuristic, and **F-TCH(L12)** denotes the forward TCH search with landmarks heuristic, while applying all optimisations introduced.

Dataset: For experiments, we consider the real-world dataset taken from the public repository.[3] The dataset contains the road network for New York (NY) and the historical travel time that is estimated every hour during the entire 2013 year. Note that the NY dataset used in this chapter is distinct from Chapter 4, as they originate from different repositories. In order to compute TTF for each edge, we take the travel time data from Tuesday to Thursday following [85] and average them for each hour after filtering out the data by two standard deviation. Overall, the NY dataset has 12.59% of edges that are time-dependent with time domain $[0h, 24h)$. Since there are not many time-dependent datasets available online, we also create a few synthetic datasets to evaluate our algorithms: (i) In order to simulate the data on other cities, we take the road networks for Gold Coast (GC), Sydney (SYD), and Melbourne (MEL) from the OpenStreetMap[4] and use the traffic pattern taken from NY dataset, to assign each type of road the same percentage of time-dependent edges. (ii) In order to simulate a more accurate TTF, we change the TTF of NY dataset by taking the 5 mins data points on a cubic spline created using the original data. We denote this dataset as NY-5.

Queries: We generate queries following the same method as in [147]. For each road network, we discretise the map into a 1024 × 1024 grid with cell side length l. Then, we randomly generate ten groups of queries such that i-th group contains 1000 (s, d) pairs with Euclidean distance between them within $2^{i-1} \times l$ to $2^i \times l$, i.e., 10,000 queries in total. During each hour in the time-domain $[0h, 24h)$, we report the performance for each algorithm to determine the length of the shortest path, without outputting the complete description of the paths. Individual queries are run 10 times; we omit the best and worst runs and average the rest.

All algorithms (including the competitor algorithms) are implemented in C++ and compiled with -O3 flag. We use a 2.6 GHz Intel Core i7 machine with 16GB of RAM and running OSX 10.14.6. For reproducibility, all of our implementations are available online.[5]

5.4.1 Preprocessing Cost and Space

All indexes were built on a 32 core Nectar research cloud with 128GB of RAM running Ubuntu 18.04.4 LTS (Bionic Beaver). Table 5.2 shows the build time and memory required for TCH, STCH and MTCH, as well as constructing heuristic CPD, TCPD and RTPD. Note that for STCH and MTCH, TCPDs are constructed on top of TCH in each time bucket T_i. All CPD-based heuristics include the costs for constructing and storing the underlying hierarchical data

[3] https://uofi.app.box.com/v/NYC-traffic-estimates

[4] https://www.openstreetmap.org

[5] https://github.com/bshen95/Improving-TCH

Name			NY	GC	SYD	MEL	NY-5
#V			96k	39k	192k	314k	96k
#E			260k	81k	421k	690k	260k
Build Time (Mins)	TCH	-	1.72	0.03	0.37	0.62	30.76
		CPD	2.92	0.18	5.17	15.55	31.94
		TCPD	2.94	0.17	4.99	14.65	31.96
		RTPD	3.21	0.18	6.19	17.93	32.30
	STCH	-	2.99	0.25	1.88	3.50	22.15
		TCPD	31.87	3.56	111.99	342.74	50.68
	MTCH	-	9.04	2.51	11.01	24.52	37.18
		TCPD	95.12	12.46	341.03	1030.67	122.48
Memory (MB)	TCH	-	269	32	214	417	4242
		CPD	346	36	290	587	4319
		TCPD	353	36	294	609	4325
		RTPD	9596	1614	37168	99237	13569
	STCH	-	1279	231	1421	2710	12254
		TCPD	3286	334	3254	7132	14262
	MTCH	-	3193	829	3711	6815	23289
		TCPD	9198	1138	9147	19927	29297

Table 5.2: Total number of vertices (#V) and edges (#E) in maps, build time in Mins, and memory in MB for TCH-based data structure with different heuristics.

structure and the reachability oracle *Reach* (for all datasets, *Reach* takes <10 secs to build and <5MB to store).

For heuristics of the TCH, our proposed TCPDs are built in similar time to CPDs, but take slightly more space to store. On the other hand, RTPDs shows higher build time and space-consumption than both CPDs and TCPDs. Comparing with TCH, both STCH and MTCH are generally more expensive to build and store. However, interestingly, we see STCH is cheaper to compute than TCH in the NY-5 dataset. This is because the TTF of NY-5 has more interpolate points (i.e., every 5 mins) than the other datasets. During TCH construction, adding shortcut edges requires search to verify the local optimality, which can adversely affect the performance when considering the entire time domain T. The memory of STCH and MTCH can be large after including the TCPDs heuristics. However, during the query time of the entire day, STCH and MTCH roll over the TCHs built for each hourly bucket, so the actual memory required in RAM is only 1/24 of the memory reported in Table 5.2.

5.4.2 Query Processing Time

In this experiment, we compare the average query processing time for our algorithms against the baseline implementation of TCH. During the entire time domain [0h, 24h), we evaluate all queries by setting the departure time t at each hour. We report the average runtime of each algorithm.

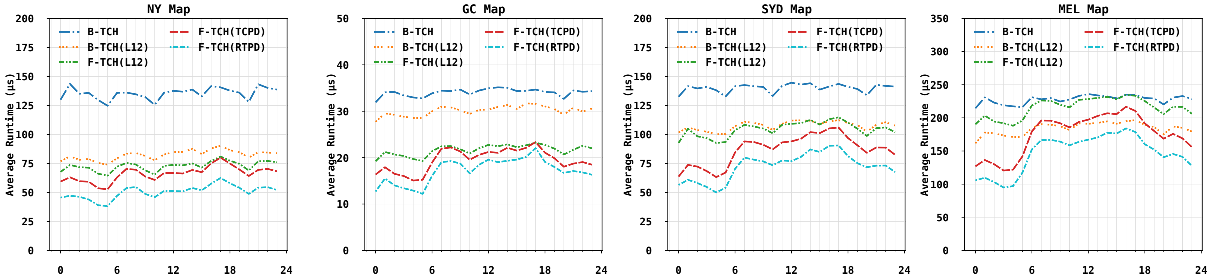


FIGURE 5.4: Runtime comparisons between TCH and different heuristic searches on TCH. The x-axis represents every hour during the time domain [0h, 24h). The y-axis shows the average runtime of different algorithms in μs.

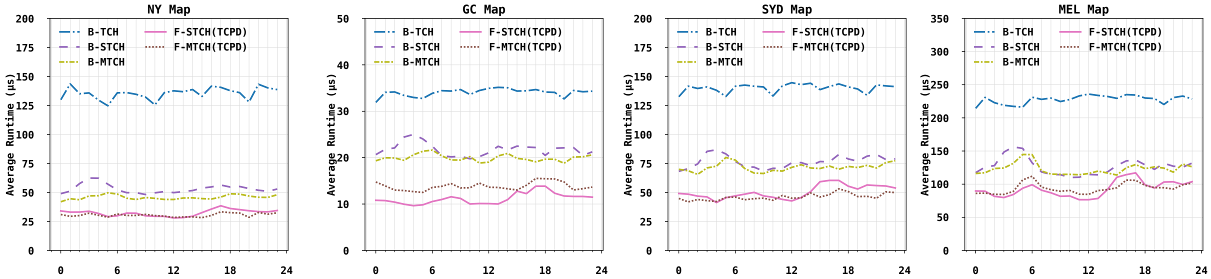


FIGURE 5.5: Runtime comparisons between the baseline implementation of TCH, and our algorithms STCH and MTCH with/without TCPD heuristics. The x-axis and y-axis are same as in Figure 5.4.

5.4.2.1 Heuristic Search

Figure 5.4 compares the query performance between the bidirectional TCH search (B-TCH) and our proposed heuristic searches on TCH. Examining the results, we see that all heuristic searches are substantially faster than the baseline implementation of TCH. For bidirectional TCH search, our proposed algorithm B-TCH(L12) improves B-TCH by around 30% of time. Moving to forward TCH search, the landmark heuristic F-TCH(L12) becomes faster than B-TCH(L12) for most datasets (e.g., NY, GC and SYD). In addition, the TCPD heuristic outperforms the landmark heuristics for forward TCH search in all datasets. Using RTPD heuristics dominates all methods. We also compared against CPD heuristics, the performance of F-TCH(CPD) is however significantly slower than other approaches (see Table 5.3).

5.4.2.2 Splitting the Time Domain

Figure 5.5 compares B-TCH against our algorithms. From the results, it is clear that B-STCH significantly outperforms B-TCH, and B-MTCH further improves the query performance. We also see that TCPD heuristics improve both algorithms, but F-MTCH(TCPD) costs additional time to switch to the corresponding TCH, thus, only shows competitive results with F-STCH(TCPD). Furthermore, unlike the TCH-based algorithms, the STCH and MTCH-based

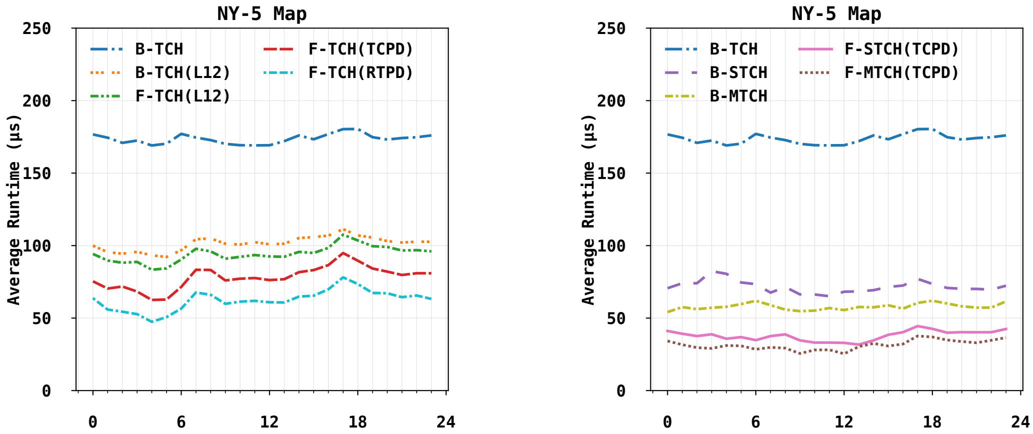


FIGURE 5.6: Runtime comparisons on NY-5 datasets, we show the results for different heuristics on TCH (left), STCH and MTCH with/without TCPD heuristics (right).

approaches do not exhibit a similar trend with the departure time variant. This disparity arises from the fact that each query with a different departure time is answered by a sub-TCH of STCH/MTCH constructed within a distinct time domain, and these sub-TCHs often incorporate different shortcuts. One can build RTPDs in STCH and MTCH by storing each of RTPD with same amount of space as shown in Table 5.2. However, the improvement is only a few microseconds, thus omitted. We also remark that the MTCH can be extended to speed up only certain period of the time domain (e.g., during the peak hours: 6 - 10 AM or 16 - 19 PM) or further improved by adding more layers with smaller t_u. However, how to efficiently choose upper bound t_u and split the time domain is left as future work.

5.4.2.3 Tolerance of More Accurate TTF

In Figure 5.6, we reproduce the experiments on NY-5 datasets, where the travel times are evaluated every 5 mins. Clearly, our proposed algorithms are tolerant to the more accurate TTF. In addition, unlike the NY dataset, both F-MTCH and F-MTCH(TCPD) outperform the STCH approaches. This is because the TCH constructed for $t_u = 1h$ and $2h$ in MTCH can be more efficiently evaluated than STCH (i.e., $t_u = 4h$).

5.4.2.4 Query Statistics

Table 5.3 provides more insights. For the searches that are conducted on TCH, B-TCH(L12) significantly improves B-TCH due to the smaller number of nodes generated and expanded using the landmark heuristic. Although F-TCH(CPD) requires smaller number of nodes generated and expanded, it still performs worse than B-TCH as the number of first move extractions is large, even after applying the cost caching discussed earlier. On the other hand, F-TCH(TCPD) retrieves the first moves on the optimal tch-paths which significantly reduces #FirstMove. Also,

Map	Stat	B-TCH		F-TCH				B-STCH	F-STCH	B-MTCH	F-MTCH
		-	L12	L12	CPD	TCPD	RTPD	-	TCPD	-	TCPD
NY	#Gen	409.030	238.549	194.885	146.910	128.333		231.202	73.821	191.762	60.718
	#Exp	188.190	87.927	52.343	33.512	26.761		127.227	16.281	111.311	13.217
	#FM	-	-	-	437.900	104.098		-	72.327	8.354	71.835
	#RT	-	-	469.593	320.587	263.588		-	155.519	-	126.272
	Time	135.462	82.297	72.773	143.242	65.966	50.428	53.158	32.372	45.806	30.484
GC	#Gen	157.091	125.512	89.900	75.883	68.356		115.409	48.613	102.002	41.311
	#Exp	98.240	72.178	43.035	33.852	28.400		78.561	19.883	71.528	16.536
	#FM	-	-	-	287.394	61.682		-	49.488	10.170	54.765
	#RT	-	-	265.317	202.391	168.863		-	113.855	-	91.703
	Time	34.048	30.106	21.637	36.205	19.509	17.188	21.747	11.182	19.940	13.822
SYD	#Gen	396.275	283.144	243.200	200.175	163.915		258.679	104.310	219.491	84.920
	#Exp	203.671	123.877	81.389	60.884	43.175		151.042	28.513	134.567	22.991
	#FM	-	-	-	857.854	130.283		-	98.494	11.558	97.879
	#RT	-	-	823.346	624.071	447.878		-	274.051	-	218.711
	Time	140.094	107.453	104.462	235.888	86.982	72.609	76.488	50.362	71.604	46.152
MEL	#Gen	558.484	420.292	394.894	333.293	262.745		374.386	161.660	322.278	133.948
	#Exp	293.440	189.117	139.651	108.987	73.811		219.413	43.616	196.001	36.044
	#FM	-	-	-	1630.162	193.499		-	143.565	13.761	139.118
	#RT	-	-	1592.497	1283.166	909.075		-	520.653	-	428.705
	Time	227.828	184.182	215.427	570.812	175.058	146.730	126.825	92.807	122.892	93.952
NY-5	#Gen	427.587	248.241	201.013	151.863	133.485		238.443	75.512	195.478	61.123
	#Exp	191.516	88.954	51.879	33.236	26.944		129.906	16.548	112.625	13.321
	#FM	-	-	-	354.327	106.424		-	73.479	8.428	72.187
	#RT	-	-	475.553	325.475	270.539		-	157.568	-	126.871
	Time	173.661	101.496	94.273	148.735	78.185	62.465	71.651	37.887	57.855	31.426

Table 5.3: Average runtime (Time) in μs and number of nodes generated (#Gen) and expanded (#Exp) by each algorithm. For forward search, we report the number of reachability tests (#RT) and first move extractions (#FM) performed using *Reach* and different CPDs.

#Generated and #Expanded are also reduced because TCPD computes tighter lower bounds. Overall, F-TCH(TCPD) outperforms the other heuristics in terms of average runtime for all maps, and RTPD further improves the performance by extracting each first move in constant time. For the searches that are conducted on STCH and MTCH, both B-STCH and B-MTCH outperform B-TCH, because building the TCHs for small time period (i) has smaller number of shortcut edges; (ii) stores tighter lower and upper bound; and (iii) maintains much smaller number of interpolate points on each TTF. In addition, TCPD heuristic further improves the performance for STCH and MTCH, and achieves speed up against baseline algorithm B-TCH by 3-6 factors.

5.5 Discussion

We show how search in time-dependent road networks can be substantially improved by applying several heuristics. Forward search using heuristics based on landmarks or CPDs is able to improve upon the usual bidirectional search. We also show that we can improve TCHs by building a set of TCHs to be used depending on the start time, and a given upper bound on the shortest path. The resulting TCHs allow for faster query processing, using a heuristic search to quickly find an upper bound, and then choosing the appropriate TCH to answer the query.

Regarding bounded-suboptimal search, there exist many techniques that compute solutions faster at the cost of slightly suboptimal paths, such as TD-CRP [148], TD-CFLAT [149], and ATCH [85]. The CPDs can also be used to find bounded-suboptimal paths [48], e.g., whenever we compute a lower bound using CPDs, we can find a solution by following the path extracted from CPDs. We believe the TCPDs and RTPDs should have the same advantage and can be easily extended for bounded suboptimal search, which we leave as future work.

Chapter 6

Beyond Pairwise Reasoning in Multi-Agent Pathfinding

6.1 Overview

Conflict-Based Search (CBS) [33] is a state-of-the-art algorithm for multi-agent pathfinding in the discretised grid map. Thus far, the CBS algorithm and its enhancements only reason about incompatibility between at most two agents at a time. In this chapter, we extend CBS heuristics, applicable at every node, to more than two agents. Our proposed cluster reasoning utilises mutex propagation to reason about clusters of more than two agents, which not only derive stronger bounds for CBS, but also generate new kinds of bypasses that help CBS to reduce the number of conflicts. In addition to that, our cluster reasoning technique can seamlessly integrate with existing enhancements. In broad strokes, our strategies are:

- For a given CT node of CBS, we compute the WDG heuristic h as explained in Section 2.3.4. For the agents that are not considered in the WDG heuristic, we select the most "important" agent a_i to detect a cluster or bypass.
- For a select agent a_i, we consider the current path P_i as guidance and iteratively perform mutex propagation between a_i and other agents a_j in conflict with P_i in order to detect clusters or bypasses.
- For each cluster found, we compute and increase the heuristic value h. Alternatively, we replace the path of a_i with P_i if a bypass of a_i is found. The above steps repeat until all agents have been selected.

Symbol	Description
G	A four-connected grid map of a MAPF instance.
V	A set of vertices in the input graph G.
E	A set of edges in the input graph G.
A	A set of k agents of a MAPF instance and $A = \{a_1, \ldots, a_k\}$.
a_i	An agent in a MAPF instance and $a_i \in A$.
s_i	The source location of agent a_i in a MAPF instance and $s_i \in V$.
d_i	The destination location of agent a_i in a MAPF instance and $d_i \in V$.
P_i	A path that consists of a sequence of vertices $\langle s_i, \cdots, d_i \rangle$ for agent a_i.
$\Sigma\|P_i\|$	The cost (i.e., length) of a path P_i i.e., $\Sigma\|P_i\| = \|P_i\| - 1$.
t	The discretised timestep in a MAPF instance.
N	A search/CT node in the CBS.
$N.constraints$	A set of constraints in a CT node N.
$N.\mathcal{P}$	A set of cost-minimal paths (one for each agent) that satisfies $N.constraints$
	without considering other agents in a CT node N.
$N.\mathcal{P}(a_i)$	The path for agent a_i in the current paths $N.\mathcal{P}$ of a CT node N.
$N.conflicts$	A set of conflicts in the current paths $N.\mathcal{P}$ of a CT node N.
$N.cost$	The SIC of the current paths $N.\mathcal{P}$ of a CT node N.
MDD_i	The Multi-value Decision Diagram for an agent a_i in a CT node N.
C	A conflict cluster in the current paths $N.\mathcal{P}$ of a CT node N and $C \subseteq A$.

Table 6.1: Summary of the notations used in this chapter

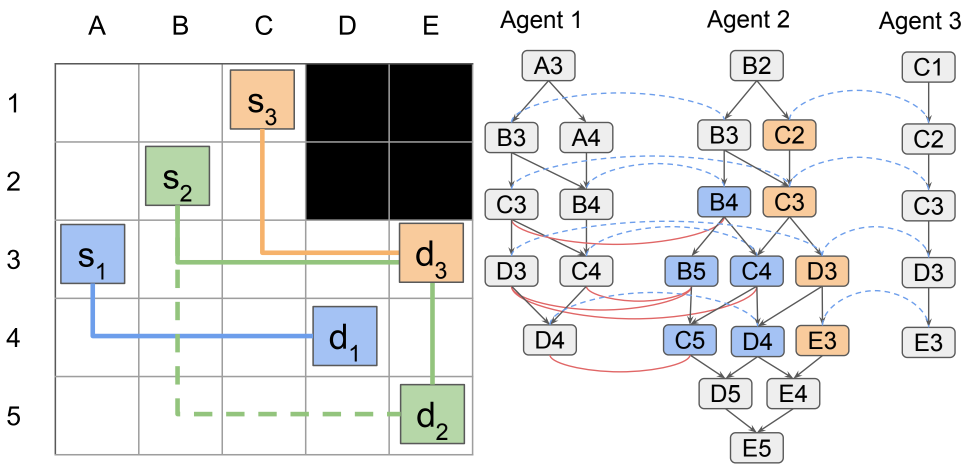
We give a complete description of our cluster reasoning algorithm, and then evaluate on popular MAPF benchmarks using the same instances published on MovingAI repository.[1] Our principal point of comparison is CBSH2-RTC, the current state-of-the-art variation of CBS. Through a range of experimental results, we demonstrate that our approach significantly outperforms CBSH2-RTC in terms of runtime, CT node expansions, and the number of instances solved, particularly on dense maps.

6.2 Preliminaries

In the classic Multi-Agent PathFinding (MAPF) problem, we consider the underlying workspace as a four-connected grid map. Time is also discretised into unit-sized timesteps and at each timestep agents are allowed to move to an adjacent vertex, or else wait, at their current location. Given a set of agents with source s and destination d, the objective is to find a set of collision-free paths, one for each agent, that minimises the sum of individual costs. In this chapter, we follow the definitions and terminologies defined in Section 2.3. Additionally, the symbols used in this chapter are shown in Table 6.1.

This chapter focuses on enhancing Conflict-Based Search (CBS) [33] by extending its heuristics to consider the incompatibility of more than two agents. Our techniques exploit *mutex propagation* [35], a successful pairwise reasoning technique that we extend to clusters of more

[1] https://movingai.com/benchmarks/mapf



(i) (ii)

FIGURE 6.1: (i) A MAPF instance with three agents. (ii) Examples of MDDs for three agents and the results of mutex propagation between agent a_2 with agent a_1 and a_3. The initial and propagated mutexes are shown in dashed blue arcs and solid red arcs, respectively. The incompatible nodes between a_2 and a_1 (resp. a_3) are coloured in blue (resp. orange). All paths of a_2 have incompatible nodes and thus must collide with either a_1 or a_3.

than two agents. A detailed discussion on mutex propagation, the CBS algorithm, and its recent enhancements can be found in Section 2.3.4

6.3 Our Approach

While the best heuristic for CBS is quite sophisticated, it only ever reasons about the interactions of pairs of agents. In this work, we detect and make use of interactions between three or more agents to improve heuristics and find bypasses.

Definition 6.1 (Conflict Cluster)**.** Given a CT node N, a conflict cluster C is a set of agents such that, considering every agent $a \in C$ with a set of cost-minimal paths that satisfy $N.constraints$, there exist no conflict-free assignments of paths for these agents.

Example 6.1. *Figure 6.1 shows a conflict cluster containing three agents, where the current paths (shown in solid lines) of a_2 and a_3 collide at $C3$. Although switching a_2 to another cost-minimal path (e.g., the path shown in the dashed line) avoids the conflict with a_3, it conflicts with another agent a_1. In fact, there exists at least one conflict between two or more agents, no matter what cost-minimal paths the agents choose. Thus, the optimal solution requires at least one agent to wait for at least 1 timestep.*

The critically important feature of conflict clusters is that, if a CT node N has a conflict cluster C, then the SIC of any collision-free paths that satisfies $N.constraints$ is guaranteed to be larger than the cost of N because some pair of agents in C must conflict, i.e., the cost must

Algorithm 5: Compute Heuristic and Bypass

```
   Input: $N$: a current CT node of CBS.
   Output: Heuristic value $h$ for a CT node.
 1 $h_p, EA_p \leftarrow$ computeWDGHeuristic($N$);
 2 $h_c, EA_c \leftarrow$ inheritClusterFromParent($N$);
 3 $EA \leftarrow EA_p \cup EA_c$;
 4 $SG \leftarrow$ buildConflictStateGraph($N$);
 5 while $a_m \leftarrow getMaxConflictAgent(SG, EA)$ do
 6     $R \leftarrow$ findClusterOrBypass($a_m, EA, N$);
 7     if $R \equiv$ ConflictCluster(C) then
 8         appendCluster($N$, $C$);
 9         $EA \leftarrow EA \cup C$;
10         $h_c$ ++;
11     else if $R \equiv$ Bypass($P_m$) then
12         updatePathAndConflict($N, a_m, P_m$);
13         updateConflictStateGraph($N, SG$);
14 return $h \leftarrow h_p + h_c$;
```

increase by at least 1. However, the WDG heuristic fails to capture this case, since conflict-free paths exist for any pair of agents while ignoring the other agents.

6.3.1 Computing Heuristic and Bypass

Our approach iteratively detects the conflict clusters for a CT node N. Whenever our algorithm finds a conflict cluster, we increment the heuristic value by one and exclude the agents in this cluster to ensure the clusters detected are independent of each other. As a byproduct, our approach may also explore a bypass, a cost-equivalent path that satisfies the constraints $N.constraints$ and reduces the total number of conflicts $N.conflicts$. Here, we explain the high-level idea of computing an admissible heuristic by integrating the best pairwise heuristic WDG and the heuristic value of conflict clusters found, as well as adapting the bypasses based on the CBS framework. The details of detecting conflict clusters and bypasses will be explained later.

The pseudo-code of our approach for computing an improved heuristic or bypass is shown in Algorithm 5. Similar to CBS, given a CT node N, our algorithm first computes the WDG heuristic following [37] and returns the pairwise heuristic value h_p and a set of agents EA_p considered in WDG (i.e., agents in the dependency graph G_D) (line 1). Our algorithm uses $N.clusters$ to store a set of conflict clusters detected in N. Before detecting the new conflict clusters, our algorithm inherits the conflict clusters EA_c from the parent CT node of N and their heuristic value h_c (explained later in the section) in order to avoid recomputation (line 2). Both EA_p and EA_c are appended to a set EA, which maintains the set of excluded agents (line 3). This is to ensure that the clusters detected are independent of each other and with the agents used in the WDG heuristic.

The algorithm then begins to compute our cluster heuristic and bypass by building a conflict state graph SG (line 4). This graph is a simple undirected graph that maintains an edge between every pair of conflicting agents in N. We use this graph to efficiently track the conflicts in the current plan $N.\mathcal{P}$. The algorithm then calls getMaxConflictAgent which iteratively accesses the agents in SG that have not been excluded and returns the agent a_m (line 5) that has the maximum number of conflicts with other non-excluded agents a_i (i.e., $a_i \notin EA$). We choose the agent a_m with the maximum number of conflicts because such an agent is more likely to find a smaller conflict cluster, thus potentially leading to a better heuristic value. The function getMaxConflictAgent does not consider an agent that was returned earlier and returns null when all agents are either excluded or were returned earlier (in which case the while loop terminates). The algorithm then calls the function findClusterOrBypass (to be detailed later) which returns either a detected conflict cluster involving a_m or a bypass for agent a_m (line 6). Based on the returned result R, the algorithm proceeds as follows.

- If R is a conflict cluster (line 7-10), the algorithm appends the conflict cluster C to $N.clusters$. All agents in C are marked as excluded agents, and the cluster heuristic h_c is increased by one, because resolving the conflict in a cluster must increase SIC by at least one (see Definition 6.1 and the following example).
- If R is a bypass (line 11-13), the algorithm takes the bypass path P_m and updates N by changing the path of a_m to P_m. The conflicts of the old path are also removed and replaced with new conflicts of P_m. The conflict state graph SG is also updated accordingly.

When the while loop terminates, the algorithm returns the heuristic value (i.e., $h_p + h_c$). Note that our algorithm could work without applying WDG heuristic, h_p. However, WDG is a relatively cheap yet effective heuristic and helps improve the performance overall.

6.3.1.1 Inherit Clusters from Parent Node

Since CBS only constrains a portion of agents (mostly one agent only) when generating child CT nodes, the incompatibility among other agents, excluding constrained agents, is not changed. Hence, we inherit the information (similar to the WDG heuristic) to avoid recomputation. To inherit a conflict cluster C from parent CT node $Pr(N)$ (at line 2), we need to ensure two conditions: (i) the path cost of every agent $a_i \in C$ of the current CT node N and its parent CT node are exactly the same (i.e., $\Sigma|N.\mathcal{P}(a_i)| = \Sigma|Pr(N).\mathcal{P}(a_i)|$) and (ii) every agent $a_i \in C$ is a non-excluded agent (i.e., $a_i \notin EA$). To ensure (i) and (ii), we iteratively scan through $Pr(N).clusters$ and filter out the clusters if $\Sigma|N.\mathcal{P}(a_i)| \neq \Sigma|Pr(N).\mathcal{P}(a_i)|$ or $a_i \in EA$. For each inherited cluster, we mark these agents as excluded and increase the cluster heuristic h_c by one (line 2).

Theorem 6.2. *Given a CT node N, the heuristic $h = h_p + h_c$ computed by Algorithm 5 is admissible.*

Proof. The pairwise heuristic h_p is computed by considering a subset of agents $A_p \subseteq A$, and h_p is an admissible heuristic of CT node N as shown by Li et al. [37]. Algorithm 5 excludes these agents and computes the cluster heuristic h_c by detecting the conflict clusters from agents $A_c = A \setminus A_p$, thus h_p and h_c are disjoint. By Definition 6.1, each conflict cluster must increase the cost of CT node N by at least one. Thus, h_c is also admissible as each conflict cluster $C \in A_c$ detected is independent of other clusters. Therefore, $h = h_p + h_c$ is an admissible heuristic. □

6.3.2 Finding Conflict Cluster or Bypass

To find a conflict cluster or bypass for an agent a_m, one can incrementally join the MDD of a_m with other non-excluded agents a_c (i.e., $a_c \notin EA$) and remove MDD nodes if there is a pair of agents in conflict. We find a conflict cluster if the joint MDD contains no feasible paths for each agent to reach its destination. Alternatively, we may explore a bypass of a_m from these feasible paths in the joint MDD. However, this naive approach has two drawbacks: (i) joining the MDDs exponentially increases the size of the joint MDD; and (ii) exhaustively checking all non-excluded agents may be time-consuming. In this work, we consider a more sophisticated algorithm to identify the cluster and bypass using mutex propagation. Recall that for a pair of MDD nodes that are mutex, the following property holds:

Property 6.3. *Iff two nodes from different MDDs at the same level are mutex, there exists no pair of conflict-free paths that traverse through the two nodes and reach their destination locations at their individual minimum cost [35].*

The key idea of our techniques is to use mutex propagation to identify incompatible nodes between a pair of MDDs according to the definition below:

Definition 6.4 (Incompatible Node)**.** Given a pair of MDDs MDD_i and MDD_j for agents a_i and a_j, a MDD node n_i at level t from MDD_i is incompatible with MDD_j iff n_i is *mutex* with all MDD nodes at level t from MDD_j.

According to Property 6.3, if an MDD node n_i from MDD_i is incompatible with MDD_j, all possible cost-minimised paths of a_i using n_i have conflicts with all cost-minimised paths of a_j. Our algorithm maintains a path P_m for a_m and uses it as guidance to detect a conflict cluster or bypass, by incrementally removing nodes of MDD_m that are incompatible with the MDDs of the agents whose paths conflict with P_m. Next, we explain the details of our algorithm.

Algorithm 5 calls the function findClusterOrBypass to find a conflict cluster or bypass for input agent a_m. The pseudo-code of this algorithm is shown in Algorithm 6. To begin, the

Algorithm 6: Find Cluster or Bypass

```
   Input: $a_m$: a selected agent, $EA$: excluded agents, $N$: a current CT node.
   Output: a conflict cluster $C$ or a bypass $P_m$ for agent $a_m$
   Initialisation: $C \leftarrow \{a_m\}$; $PA \leftarrow \emptyset$
1  $P_m \leftarrow$ getPath $(a_m, N)$;
2  $MDD_m \leftarrow$ getMDD$(a_m, N)$;
3  $CA \leftarrow$ getConflictAgents$(a_m, P_m, N, EA)$;
4  for each $a_c \in CA \setminus PA$ do
5      $PA \leftarrow PA \cup \{a_c\}$;
6      $MDD_c \leftarrow$ getMDD$(a_c, N)$;
7      $M \leftarrow$ mutexPropagation$(MDD_m, MDD_c)$;
8      if $M \neq \emptyset$ then
9          $C \leftarrow C \cup \{a_c\}$;
10         deleteNodes$(MDD_m, M)$;
11         if $MDD_m = \emptyset$ then
12             return conflict cluster $C$ ;
13         if $P_m \notin MDD_m$ then
14             $P_m \leftarrow$ getMinConflictsPath$(MDD_m)$;
15             goto line 3;
16 return $|CA|$ reduced ? Bypass$(P_m)$ : null;
```

algorithm initialises the conflict cluster C to contain the agent a_m and initialises the processed agents PA to be empty. It retrieves the current path P_m of a_m (line 1). The MDD of a_m, denoted as MDD_m, is then built which satisfies all constraints in N (line 2). The algorithm then calls getConflictAgents, which considers all non-excluded agents in N and returns the set of non-excluded agents CA that conflict with a_m (line 3).

In each iteration, the algorithm iteratively accesses the agents in CA that have not been processed before. For each such agent $a_c \in CA \setminus PA$, we build the MDD of a_c, denoted as MDD_c (line 6). The algorithm performs mutex propagation between the MDD_m and MDD_c and returns the incompatible nodes M of MDD_m which are mutex with every MDD node of MDD_c in the same level (line 7). If M is not empty, we append the agent a_c into the conflict cluster C (line 9) and recursively delete every incompatible node $n \in M$ (and the connected edges) from MDD_m (line 10). After deleting the incompatible nodes of MDD_m, it is possible that MDD_m becomes empty or the current path P_m is not valid in MDD_m as some of the nodes have been deleted. We handle each case as follows.

- If the MDD_m is empty, this implies that C is a conflict cluster which is returned (line 11-12).

- If the $P_m \notin MDD_m$, the algorithm finds an alternative path from MDD_m that has the minimal number of conflicts with the other agents (lines 13 and 14). Since path P_m is updated, there may be new agents that are in conflict with this new path. So, the algorithm goes to line 3 and re-computes CA. Since CA is changed, the algorithm continues to iteratively process the agents in $CA \setminus PA$ (line 4 onwards).

When the algorithm has processed all agents in $CA \setminus PA$, it terminates (line 16) by returning the bypass P_m if this bypass has fewer conflicts than the original path $N.\mathcal{P}(a_m)$. Otherwise, it returns null, indicating that no cluster or bypass is detected. The bypass P_m returned by Algorithm 6 is an alternative path retrieved from MDD_m, which avoids traversing through incompatible nodes. Since the MDD_m satisfies every constraint on a_m and has the same cost as $N.\mathcal{P}(a_m)$, P_m is a valid bypass.

To find the minimum-conflict path (line 14) and update the conflict agents (line 3), we must repeatedly detect conflicts between a_m and other agents, which can be time-consuming. Therefore, we use a labelling method that labels the conflict agents on each node and edge of MDD_m. Every time the algorithm extracts the path, we run a breadth-first search from source to destination of MDD_m and compute the minimum number of conflicts and its predecessor on each node visited. The minimum-conflict path and its conflict agents can be easily retrieved from a backward extraction following the predecessor node. Note that we only label MDD_m in Algorithm 6 once (when the algorithm reaches line 14 for the first time).

Theorem 6.5. *The cluster C returned by Algorithm 6 is a conflict cluster, according to Definition 6.1.*

Proof. Mutex propagation of MDD_c and MDD_m removes from MDD_m only the nodes which are incompatible with all paths in MDD_c. So unless the agent a_c increases its path length, the paths removed for a_m from MDD_m must conflict with a_c. If MDD_m becomes empty, then clearly all paths of the current path length of a_m must conflict with some other agents in the cluster. Hence, at least one agent in the cluster must increase its path length by one to avoid conflicts. □

Example 6.2. *Consider the example from Figure 6.1. Assume the paths in $N.\mathcal{P}$ for a_1, a_2 and a_3 are the solid blue, green and orange lines in Figure 6.1 (i), respectively. Algorithm 6 starts with $a_m = a_2$ and initialises $C = \{a_2\}$. The algorithm finds the set of conflicting agents $CA = \{a_3\}$ because a_2 and a_3 conflict. It then processes a_3 and performs mutex propagation between MDD_2 and MDD_3. The incompatible nodes (e.g., coloured orange in Figure 6.1 (ii)) of MDD_2 are removed and a_3 is appended to C. Since the path of a_2 no longer exists in MDD_2, the algorithm then updates its path in $N.\mathcal{P}$ to be the minimal conflicts path (e.g., the dashed green line). This new path collides with a_1. The algorithm returns to line 3 and finds conflicting agents $CA = \{a_1\}$. Agent a_1 is processed and appended to C. MDD_2 becomes empty after removing incompatible nodes (e.g., coloured blue in Figure 6.1 (ii)) from MDD_2. The algorithm returns the conflict cluster $C = \{a_1, a_2, a_3\}$.*

6.3.3 Optimisation

In this section, we introduce optimisation techniques that improve the cluster heuristic h_c and speed up our algorithm.

6.3.3.1 Solving the Cluster

Recall that Algorithm 5 increases the cluster heuristic h_c by one (line 10) whenever it detects a conflict cluster C. However, to get a better heuristic value, we can solve the cluster as a sub-instance to improve the lower bound of C. Therefore, we take the paths and constraints of each agent $a_i \in C$ from the current CT node N, and run sub-CBS search to solve C. To restrict the computation cost of this optimisation, when solving a cluster, we also set a limit $|N|$ on the number of CT nodes expanded by the sub-CBS. By default, we use the same setting (i.e., $|N| = 10$) as used in WDG heuristic [37]. Let Δ_C be the increase of the minimal f-value in the OPEN list after running sub-CBS for this cluster. We increment the heuristic by Δ_C (i.e., Algorithm 5 line 10 : h_c += Δ_C). It is easy to see that the correctness of Theorem 6.2 is preserved.

6.3.3.2 Memoisation

The algorithms have two operations that can be repetitively performed in the same or different branches of a CT tree: (i) computing the heuristic for the same conflict clusters using sub-CBS described above; and (ii) performing the mutex propagation between the same pair of MDDs (Algorithm 6 - line 7). We say that two conflict clusters (resp. MDDs) are the same if the two clusters (resp. MDDs) have the same agents with exactly the same constraints for each agent. In order to speed up the search, we apply memoisation by maintaining a centralised database in CBS. To avoid (i), we simply maintain a hash table to cache the increased cost Δ_C of a conflict cluster C by hashing all constraints $\in N.constraints$ of agents in C as a key. However, avoiding (ii) needs some modifications detailed below.

Algorithm 6 takes the MDD of a_m and performs mutex propagation with the MDDs of the conflicting agents $a_c \in CA$. In each iteration, the incompatible MDD nodes of MDD_m are removed which results in a smaller MDD_m. Therefore, we cannot cache the results of mutex propagation between MDD_m and MDD_c as MDD_m changes after each iteration. To overcome this issue, we propose to apply a reusable version of mutex propagation. This reusable mutex propagation does not consider the updated MDD_m, but only considers the original unmodified MDD_m and MDD_c from the CT node N. Let us denote the unmodified MDD_m as MDD'_m. The algorithm begins with MDD_m of agent a_m. Every time the algorithm performs mutex propagation between a_m and a_c, it performs the reusable mutex propagation and returns the incompatible nodes of MDD'_m to MDD'_c. We use this result to remove MDD nodes from MDD_m until MDD_m becomes empty or there is no other valid conflicting agent a_c. Although this lazy strategy weakens mutex propagation (i.e., the reusable mutex propagation may be able to detect only a subset of incompatible nodes), we can now cache the incompatible nodes between a_m and a_c based on the constraints of the two agents. In the experiments, we show that the reusable mutex propagation almost always leads to a speedup as it does not lose too much mutex information and can reuse many mutex calculations.

6.4 Experiments

In this section, we compare our algorithm against the state-of-the-art variation of CBS [34] taken from the repository[2] of the authors. This algorithm applies all leading optimisation techniques including: (i) high-level heuristics: weighted pairwise dependence graph (WDG) [37]; (ii) symmetry reasoning techniques: target reasoning, generalised rectangle and corridor reasoning [34]; and (iii) prioritising and bypassing conflicts [38, 129]. We use WDG to refer to this algorithm. Our algorithm is built on top of WDG and, in addition, uses cluster heuristic and bypass (CHBP). It is shown as WDG+CHBP in the experiments. We also compare the algorithm when only cluster heuristic is used and the bypass is ignored (i.e., WDG+CH) or when only the bypass returned by Algorithm 6 is used but the cluster heuristic is ignored (shown as WDG+BP). We do not compare our algorithm against the LR heuristic [138] because it requires us to modify the definition of MAPF by limiting the maximum cost of the paths.

6.4.1 Benchmarks

We conduct experiments on four diverse maps taken from the widely used 4-connected grid map benchmarks[3], described by Stern et al. [3]. These maps cover different real-life scenarios.

- Random map (random-32-32-20): a 32×32 grid map with 20% random blocked cells. The number of agents on the map is set to 20, 30, ..., 70.
- Empty map (empty-32-32): an empty 32×32 grid map. The number of agents in the map is set to 50, 70, ..., 150.
- Warehouse map (warehouse-10-20-10-2-1): a 161×63 grid map which simulates the warehouse environment with 10×20 stacks. Each stack has 10×2 grids. The number of agents is set to 30, 50, ..., 130.

[2] https://github.com/Jiaoyang-Li/CBSH2-RTC

[3] https://movingai.com/benchmarks/mapf

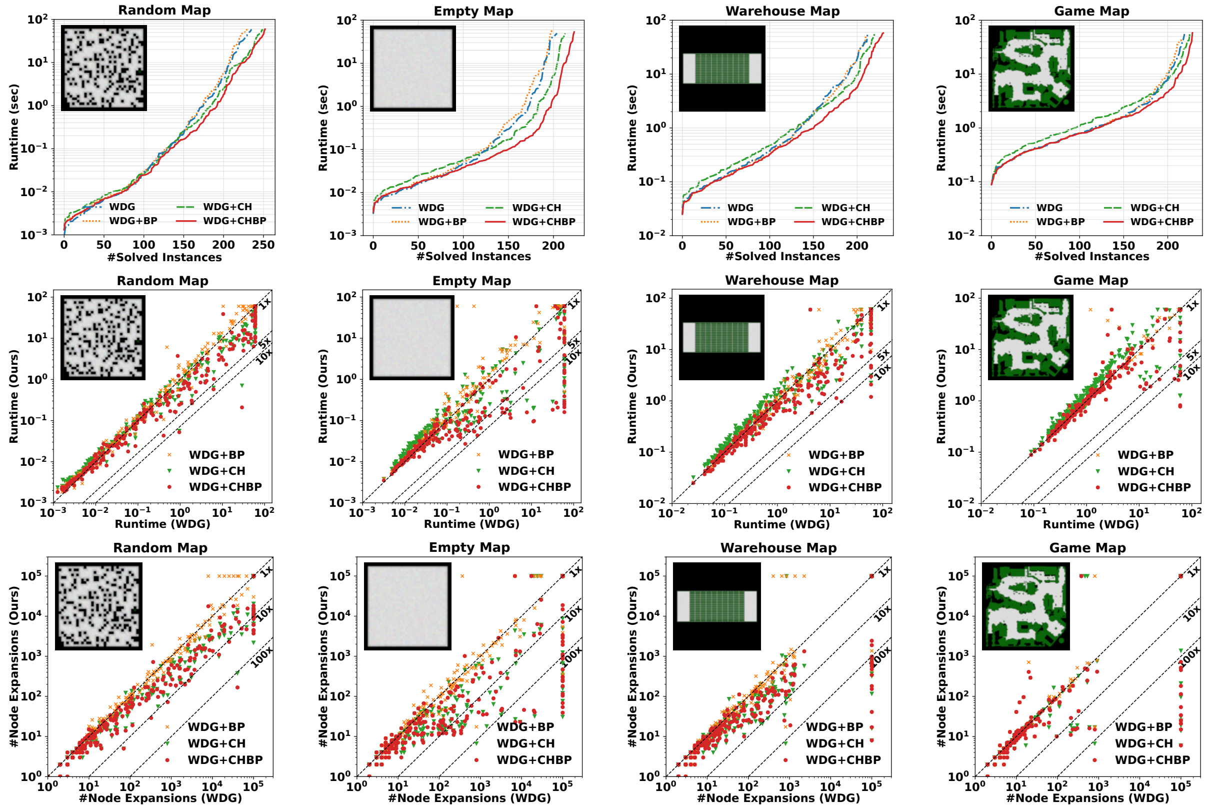


FIGURE 6.2: Cactus plots for runtime in seconds (top row), scatter plots for runtime in seconds (middle row), and scatter plots for CT node expansions (bottom row). If an approach fails to solve an instance in 60 seconds (i.e., unsolved instance), its runtime in the figure is shown as 60 seconds and its number of node expansions is shown to be 10^5 (all solved instances have runtime less than 60 and node expansions less than 10^5).

- Game map (den520d): a 256×257 grid map from a video game. The number of agents is set to 40, 60, ..., 140.

The benchmark contains, for each map setting, two sets of instances each containing 25 instances: the first set generates agents with randomly selected start and destination locations; the second set generates agents with an even mix of short and long distances between their start and destination locations. We run every instance for 1 minute and report the overall performance. The instances that cannot be solved in 1 minute by an algorithm are considered unsolved. All algorithms (including the competitor algorithms) are implemented in C++ and compiled with -O3 flag. We conduct all experiments on a Nectar research cloud with 128GB of RAM running Ubuntu 18.04.4 LTS (Bionic Beaver). For reproducibility, our implementation is available online.[4]

[4] https://github.com/bshen95/CBSH2-RTC-CHBP

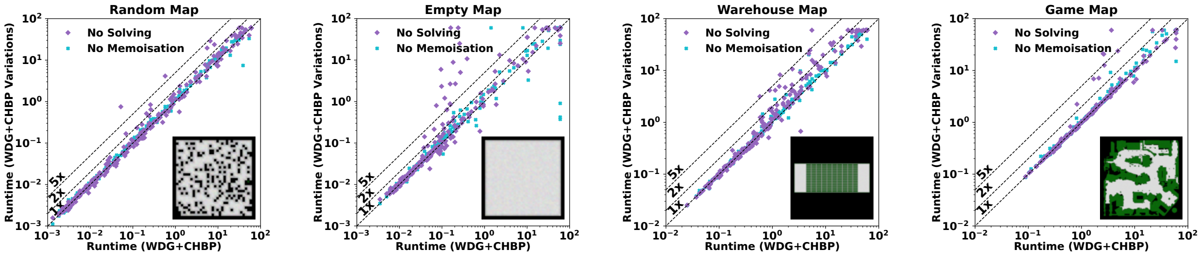


FIGURE 6.3: Effect of optimisation techniques on runtime (sec) of our final algorithm (WDG+CHBP). No Solving is when the optimisation to solve the cluster is not applied and No Memoisation is when the memoisation is not applied.

6.4.2 Runtime and CT Node Expansions

The Top row in Figure 6.2 shows the cactus plots for runtime (sec) of different algorithms. Using cluster heuristic and bypass (WDG+CHBP) significantly improves the performance on hard instances (note the log scale on the y-axis). While the cluster heuristic (WDG+CH) leads to improvements over WDG, bypassing alone (WDG+BP) does not help solve hard instances. This is because CHBP mainly benefits from increasing heuristic value whereas bypassing itself neither considers heuristic value nor excludes the agents of detected clusters which results in degraded performance. The scatter plots in the middle row (Figure 6.2) show detailed runtime comparisons versus the baseline WDG. The three diagonal lines show the performance improvement compared to WDG (1x, 5x or 10x), i.e., a point under the diagonal line 5x indicates that our algorithm is more than 5 times faster than WDG on that instance. The scatter plots show that our methods improve upon the baseline for most of the instances and rarely show significantly worse runtime. Importantly, our methods are able to solve many instances that are unsolved by the baseline (the instances shown at 60 seconds on the x-axis). There are some instances which we fail to solve but the baseline can solve (illustrated by the points at 60 seconds on the y-axis). Note that the number of such instances is much smaller than the instances that the baseline cannot solve but our algorithm can solve. Overall WDG, WDG+BP, WDG+CH and WDG+CHBP solve 875, 859, 896 and 937 instances, respectively.

The scatter plots on the bottom row in Figure 6.2 show detailed comparisons of # CT node expansions of our algorithms with WDG. Again, our methods almost always lead to fewer CT node expansions.

6.4.3 Ablation Study on Optimisation Techniques

Figure 6.3 shows the scatter plots for the runtime of our final algorithm (WDG+CHBP) versus two modified versions of the algorithm without applying optimisation techniques. The diagonal lines show how slow the two versions are compared to our final algorithm WDG+CHBP, i.e., a point above the diagonal 2x shows an instance where the algorithm is more than 2 times slower

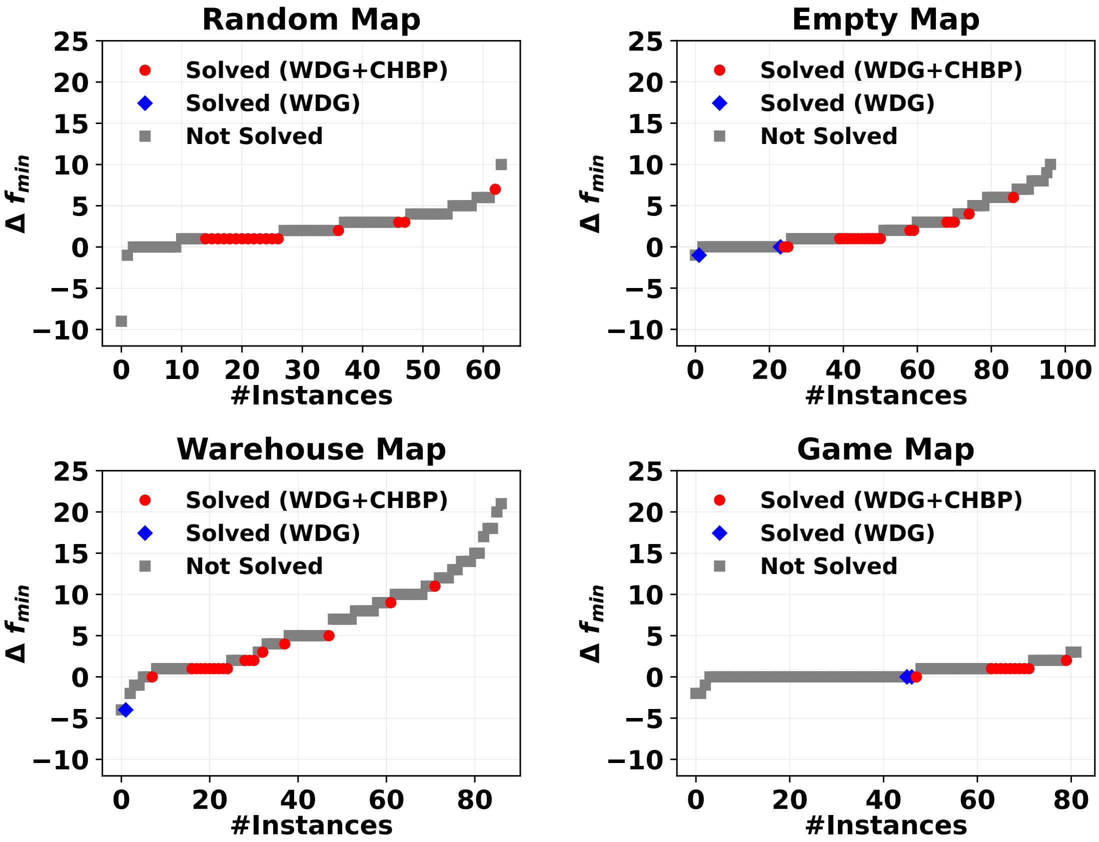


FIGURE 6.4: Δf_{min}=f_{min}(WDG+CHBP) − f_{min}(WDG). Instances solved by both WDG and WDG+CHBP are removed as they have $\Delta f_{min} = 0$. We show instances solved by only WDG (blue), solved by only WDG+CHBP (red) and unsolved by both (grey).

than WDG+CHBP. Clearly, both versions show worse performance than our final algorithm, which demonstrates the effectiveness of our proposed optimisations. In addition, No Solving is significantly worse than WDG+CHBP on many instances. This shows that solving the cluster is the most important enhancement as it significantly increases the heuristic value for some of the clusters detected. Although not as significant, memoisation also plays an important role by avoiding repeatedly solve the same clusters and perform mutex propagation between the same pair of MDDs.

6.4.4 Effect of Heuristic Value and Insights

In Figure 6.4, we show Δf_{min}=f_{min}(WDG+CHBP) − f_{min}(WDG) where $f_{min}(X)$ is the minimum f-value ($f = N.cost + h$) in the OPEN list when the algorithm X terminates. Δf_{min} shows the difference in heuristic values of the two algorithms indicating how much the cluster heuristic is able to improve the search progress compared to WDG. Note that while it cannot make the heuristic worse at a CT node, it does change the CBS search tree which may lead to a smaller f_{min} for WDG+CHBP for some instances compared to WDG. Figure 6.4 shows that Δf_{min} is

Map	CBS Search		Compute WDG		Compute CHBP	
	Total(s)	PH(ms)	Total(s)	PH(ms)	Total(s)	PH(ms)
Random	1094.39	0.54	1158.83	0.57	1452.25	0.71
Empty	1061.40	0.50	1342.00	0.63	2513.68	1.23
Warehouse	2826.13	18.53	932.69	6.12	1334.28	8.75
Game	2972.42	32.93	627.17	6.95	1084.17	12.01

Table 6.2: Performance breakdown of WDG+CHBP. We show the total runtime (Total) and the average runtime per heuristic calculated (PH) for each component in WDG+CHBP.

mostly positive and there is typically a significant increase in f_{min} for WDG+CHBP compared to WDG, especially on the empty and warehouse maps. Also, note that WDG+CHBP solves many instances that WDG cannot solve. On the other hand, there are very few instances that only WDG can solve.

Table 6.2 shows the average runtime per heuristic calculated of the various components: CBS search, that is everything else than heuristics and bypass calculation; Compute WDG, the time to compute the WDG heuristic; and Compute CHBP, the time to compute our heuristic and bypasses. Clearly, the more complex heuristics are more expensive on average than the WDG heuristic, but never more than 2 times more expensive. They take less time than the remaining components on the larger maps. Overall of course the computation cost of this heuristic almost always pays off in terms of reduced high-level search.

6.5 Discussion

In this work, we propose new techniques to compute heuristics by reasoning about incompatibility beyond two agents. It dynamically finds conflict clusters and bypasses at the same time. We substantially improve CBS by solving more instances in limited time and reducing the high-level node expansion and runtime to solve problems. For instances with a timeout failure, we push the lower-bound (f_{min}) to a significantly higher value. We show that reasoning for conflict clusters is essential to solve larger MAPF problems. Future works include capturing more complex clusters, integrating conflict cluster heuristics into the integer program of the WDG heuristic, and designing strong methods to efficiently resolve all conflicts in clusters.

Chapter 7

Tracking Progress in Multi-Agent Pathfinding

7.1 Overview

In recent years, there has been a growing industrial interest in MAPF, leading to a significant increase in publications across various venues. Despite the community's effort in creating standardised MAPF benchmarks [3], tracking progress on these benchmarks remains challenging due to several issues. To address these challenges and make it easier for new researchers to enter the field, we have developed a new set of methodological and visualisation tools. These tools enable comparison of a broad range of MAPF methods and establish the pareto-frontier that currently defines the state-of-the-art. In broad strokes:

- We break down the benchmarks suites into three different levels: (i) Domain and Map-level; (ii) Scenario-level; and (iii) Instance-level. At the lowest instance-level, we identify the necessary data to collect for covering different types of MAPF algorithms in the existing literature.
- For each level of benchmarks, we provide a list of important aspects that we keep tracking on, as well as automatic visualisation tools to assist researchers in generating insight and systemic analysis from results data.
- For all the results we collect, we make them publicly available and allow other researchers to download the results at each level. Additionally, we provide submission interfaces and automated tools to enable other researchers to submit and compare their algorithms.

We have implemented our system as a website and conducted a comprehensive set of experiments, including many state-of-the-art MAPF algorithms. In the end, we systematically analyse the collected results and demonstrate how our website can be used to explore the new pareto-frontier, as well as the insights for the submitted MAPF algorithms.

7.2 Background and Literature Review

In the classic MAPF problem, a team of cooperative agents moves across a 4-connected grid map. Agents are allowed to move from a current grid cell (or vertex) to another adjacent grid cell, or else wait at their current location. Each move or wait action has unit cost. Time is also discretised into unit-sized steps. We are asked to compute paths to move every agent, from its source position to its destination location. The paths need to be collision free, which means agents do not collide with static obstacles on the map or with each other. Performance for classical MAPF is measured in a variety of ways. The most common metrics are the number of problems solved, the success rate (percentage of problems solved for a fixed number of agents within the fixed runtime), the plan cost (sum of the individual costs), and the runtime.

In this chapter, we follow the definitions and terminologies defined in Section 2.3. A comprehensive survey of MAPF algorithms is presented in Section 2.3.1 - 2.3.2, covering optimal, bounded suboptimal, and unbounded suboptimal MAPF algorithms. Next, we discuss the current benchmarks, as well as the limitations of existing works and large-scale evaluations.

7.2.1 Existing Benchmarks and Limitations

Early MAPF research (and even some recent papers) typically conduct experiments over individually generated problem sets, e.g., [89, 97, 114]. Historically these problem sets have not been published as separate artefacts, which creates difficulties trying to reproduce experimental data, even if reference implementations of the original works are made available. In response to these challenges the MAPF community developed a standard benchmark suite [3], to test the performance of (classical) MAPF solvers across 33 maps from 7 different types of grid domains: (i) Game grids, originating from real video games; (ii) City maps, originating from layouts of real cities; (iii) Maze maps, synthetically generated and featuring different corridor sizes; (iv) Room maps, synthetic grids with open areas connected by narrow entrances; (v) Empty maps, open grids without obstacles; (vi) Random maps, synthetic grids with random obstacles; and (vii) Warehouse maps, synthetic maps imitating automated warehouse environments.

Each map has 25 random and 25 even scenarios, each of which is associated with several instances. A random scenario includes randomly generated source and destination locations for up to 1000 agents (instances). On even scenarios, source and destination locations have an even distribution on distance, and the total number of location pairs varies across different maps,

	Domains	Maps	Instances	Compared algorithms
CBSH2-RCT	7	8	1,200	1
CBSH2-RCT-CHBP	4	4	1,200	1
ID-CBS	7	32	3,380	1
Lazy-CBS	3	4	1400	3
BCP-MAPF	6	16	4,430	2
EECBS	5	6	1,200	3
FEECBS+	7	8	1,800	1
PIBT2	5	10	21,075	6
LNS2	7	33	825	3
LaCAM	5	12	5,000	6
Total Available	7	33	1.5 M	—

Table 7.1: The number of domains, maps, instances and compared algorithms in recent MAPF research.

ranging up to 7000 agents (instances). There are more than 1.5 million location pairs in total, which means the same amount of problem instances. Table 7.1 gives a complete summary.

The computational burden required to run the whole benchmark is large, which means researchers only compare proposed approaches against a small number of contemporaries and only on a limited subset of problem instances. Table 7.1 shows a summary of reported results for a recent set of leading optimal and suboptimal solvers: CBSH2-RCT [34], CBSH2-RCT-CHBP [50], ID-CBS [92], Lazy-CBS [41], BCP-MAPF [39], EECBS [104], FEECBS+ [106], PIBT2 [42], LNS2 [43, 125], and LaCAM [40]. Although most domains are considered, researchers typically select only 1 to 2 maps per domain for evaluation. The number of instances selected for experiments meanwhile is small relative to the total available instances. In particular, most research increases the number of selected source and destination locations by 5 or 10 until the selected algorithm returns a timeout failure. Therefore, the coverage of reported results across the benchmark (sub-)set is incomplete.

Another related issue is that experimental results now appear in many different venues across the literature. Thus, it is increasingly difficult to keep track of all recent advancements. Moreover, although headline results are published (e.g., success rate and total problems solved), detailed supplementary material, to indicate exactly which problems were solved and how well, are usually not publicly accessible. Short of large-scale re-evaluations of many published works, it has become difficult to know exactly how much progress is being made in the area, and it is difficult to know how to select an appropriate subset of competitors for experimental comparisons. As shown in Table 7.1, most research handles these issues by comparing against only 1 or 2 closest rivals, yet there are far more researches claiming state-of-the-art results each year.

7.2.2 Existing Work on Large-scale Evaluation

Several recent initiatives have attempted to track MAPF performance. The community website, `mapf.info`[1], features a dedicated page that encourages researchers to report their results. But there is no consistent way to submit, manage, or explore these results. Thus, since its inception more than two years ago, only one algorithm has been reported here, with only summary results and no proof of claim.

Another recent attempt is due to Kaduri et al. [150], who conducted a large-scale experiment to compare 5 leading optimal MAPF algorithms. Authors evaluate all maps and scenarios from the MAPF benchmark suite. Meanwhile in Ren et al. [151], authors evaluate six recent optimal algorithms, in another large-scale experiment. In these works, authors analyse the percentage of instances solved by each algorithm and observe general trends. A main drawback of these research works is the limited coverage of algorithms, including notable exclusions such as BCP-7 [96] and CBSH-RCT [130] (two very successful and performant solvers). In addition, authors only give summary results. For example, detailed progress for each domain and instance is unavailable. Plans, plan costs and best-known bounds are also unavailable, which makes it difficult to judge progress in the area, difficult to validate results and impossible to include new points of comparison (without re-producing the experiment). Recently in Ewing et al. [152] authors evaluate seven different algorithms to determine features that make MAPF problems hard. Although wide-ranging, experiments are on non-standard problem sets. Detailed data is unavailable.

Discussion: Substantial interest and large recent advances have grown the size of the MAPF community. These are positive developments, but they come at a price: progress is harder to track, the main challenges are less clear, and barriers for entry are increasing. To continue growing the MAPF community needs effective tools: to track progress, investigate results, simplify comparisons and to help researchers stay up to date with recent developments.

7.3 Methodology

In this section, we introduce our methodology to track the progress of different methods on MAPF benchmarks. In general, there are three types of algorithms studied by the research community:

(i) **Optimal Algorithms** focus on finding exact optimal solutions. Such algorithms start from a lower-bound of the optimal solution, and progressively push the lower-bound until they find a feasible solution that is provably optimal.

[1] `http://mapf.info/index.php/Data/Data`

(ii) **Bounded Suboptimal Algorithms** find the suboptimal solution within theoretical guarantees. These algorithms explore the lower-bound and feasible solutions simultaneously, and return the solution that is within certain suboptimality w.r.t. current best lower-bound.

(iii) **Unbounded Suboptimal Algorithms** focus on finding feasible solutions. These algorithms find the feasible solution fast, and may keep improving it given sufficient time.

Our goal is to design a system that tracks different types of algorithms and their progress together. The critically important feature for us is the ability to handle all types of algorithms. Therefore, we focus on two important results reported by different MAPF algorithms: (a) best (i.e., largest) lower-bound value: we track this value to cover the algorithms in (i) and (ii); and (b) best (i.e., smallest SIC) solution: we record this result to cover the algorithms in (ii) and (iii). In addition to bounds and costs, runtime is another frequently used metric that can further distinguish between competing solvers. We do not attempt to track this aspect owing to the large variability in configuration setups from one paper to the next. In the remainder of this section, we explain our strategies for generating insight and systemic analysis from results data, as well as a list of important things that we are tracking on different levels of the benchmark.

7.3.1 Instance-level Tracking

At the instance level, our system records the best lower-bound and solution cost as explained above. For each reported lower-bound or valid plan we also keep track of additional metadata, such as the algorithm that produced the result, names of authors, publication references and links to implementations. We then use the data to provide additional insights:

Tracking the concrete plan: each instance contains a different number of agents, however, it is not clear how these agents are distributed w.r.t. the obstacles of map, and how their solution paths interact on the map. Our system records a concrete plan for each best known solution cost and provides a visualiser to better understand those solutions.

Tracking the gap: For each instance, we may have different algorithms which contribute lower-bounds and solutions (upper bounds) separately. Together, we need to analyse how close these algorithms are in terms of finding and proving optimal solutions. Therefore, we automatically track and visualise the *suboptimality ratio* of each instance defined as $(\mathcal{S} - \mathcal{L})$ / $\mathcal{L}$ where $\mathcal{L}$ and $\mathcal{S}$ are the best known lower-bound and solution of the instance, respectively.

7.3.2 Scenario-level Tracking

All instances in a scenario are categorised into three types: (i) *closed instance*: the instance has the same best lower-bound and solution cost (indicating that the solution cannot be further improved); (ii) *solved instance*: the instance has a feasible solution reported, but the current best lower-bound is less than the solution cost (i.e., improvement may be possible); and (iii) *unknown instance*: the instance has no solution reported. For each scenario, our system tracks the percentage of *closed* and *solved* instances to indicate the progress of all contributed algorithms. For scenarios of the same map, we also track the following:

Tracking progress on scenarios: For a given map, our system automatically generates plots which shows the percentage of *closed*, *solved* and *unknown* instances for every scenario. The objective here is to identify the scenarios that are hard to solve with existing MAPF algorithms, so that more attention can be paid to these.

Tracking progress on different number of agents: Each scenario contains instances with different numbers of agents. It is important to understand the scalability of MAPF algorithms across all scenarios (i.e., at what number of agents we stop making progress). Therefore, our system includes the percentages of *closed*, *solved* and *unknown* instances for different number of agents on the same map.

7.3.3 Domain and Map-level Tracking

Finally, at the map-level of the benchmark, our system records the percentages of *closed* and *solved* instances for each map. Similar to the scenario-level, our system also generates plots to track the percentages of *closed*, *solved*, and *unknown* instances across all maps, and summarises the related maps for each domain to provide domain-level plots. This allows researchers to focus their efforts on solving those parts of the benchmark that have seen only limited progress.

7.3.4 Participation and Comparison

Another critical feature of our system is allowing other researchers to participate by submitting their algorithms/results and establish the state-of-the-art together. For all the results we collect, we make them publicly available and allow other researchers to download the results at each level. In order to make it easy for researchers to evaluate their own progress against other attempts we also provide tools to automatically compare algorithms, across every level of the system. Our principal evaluation criteria are: # of instances a given algorithm closed; # of instances that the algorithm solved; # of instances for which it achieved the best lower-bound;

# of instances for which it reported the best solution. We apply these four criteria to summarise the state-of-the-art for each type of algorithm (optimal, bounded- and unbounded-suboptimal).

7.3.5 Submission Interface

The submission interface is meant to make it easy for anyone in the community to upload results for any of the benchmarks. A submission *batch* requires a minimal set of metadata (details about the authors, the algorithm, its source code repository (if any)), and then consists of, for each tackled instance (i.e. map, scenario, number of agents) a lower bound (if the algorithm generates one), a solution cost (if the algorithm generates one), and a concrete plan that meets the solution cost. The plan format is an ASCII string that specifies the movements of agents at each timestep, e.g., the string "*udlrw*" represents a agent moving up, down, left, and right respectively, while w represents waiting at its current location. This is a compact way of storing plans, and also extensible to more complex plan formats in the future. The format for the submission is a .csv file. The system checks the validity of the plan and its solution cost as it processes each entry. Thus, we can guarantee all upper bound information is valid. We are unable to check lower bounds[2] so they trusted by default. If any later stage we find a lower bound is violated by correct plan, then we have direct evidence that the lower bounding submission was erroneous. In this case, we remove all lower bounds in the batch from the system.

7.4 Implementation and Initial Results

We implement our proposed system as a website[3]. For the back-end, we use Mongodb to manage all submissions in a database, and use Nodejs to implement APIs in order to communicate between database and web front-end. For the front-end, we use a React interface. To seed the database, we evaluate four different state-of-the-art optimal algorithms. This allows us to determine the best known lower bounds and optimal solutions. We also evaluate two leading unbounded-suboptimal algorithms, to explore the best known feasible solutions for as many instances as possible. All algorithms are implemented in C++ and compiled with -O3 flag, the details of each algorithm are shown in Table 7.2. At this stage, our system focuses on the MAPF algorithms that minimise the SIC, but could be extended to track progress on other objectives (e.g., makespan). To ease the computational burden, we run each algorithm on each instance for one minute by default. For CBSH2-RCT and CBSH2-RCT-CHBP[4], we run the

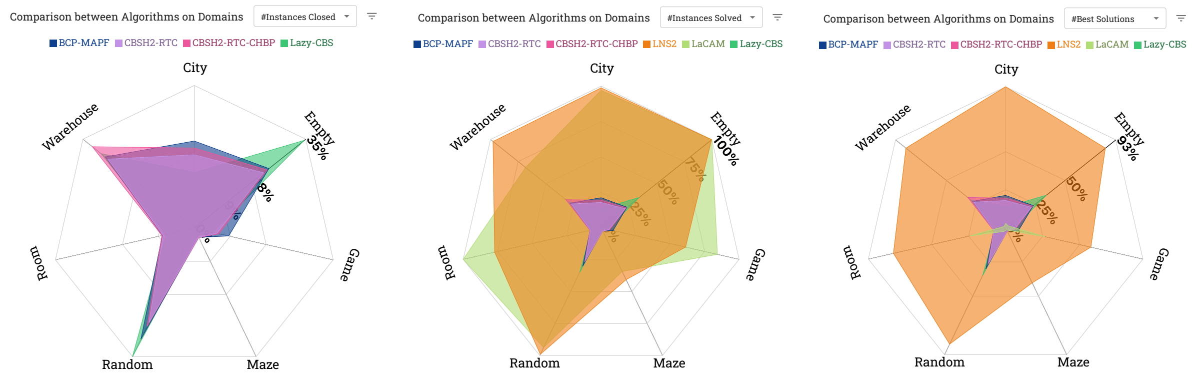
[2] Although a format for checkable proofs for lower bounds of MAPF problems would be a valuable resource for the community.

[3] Our website is accessible at: `http://tracker.pathfinding.ai`. A demo video giving an overview of the system is also available at: `http://tracker.pathfinding.ai/systemDemo`.

[4] CBSH2-RCT-CHBP represents our enhanced version of the CBS algorithm, which is detailed in Chapter 6.

Algorithm Name	Github Link	Type
CBSH2-RCT [34]	https://github.com/Jiaoyang-Li/CBSH2-RTC	Optimal
CBSH2-RCT-CHBP[50]	https://github.com/bshen95/CBSH2-RTC-CHBP	Optimal
BCP-MAPF [39]	https://github.com/ed-lam/bcp-mapf	Optimal
Lazy-CBS [41]	https://bitbucket.org/gkgange/lazycbs	Optimal
LNS2 [43]	https://github.com/Jiaoyang-Li/MAPF-LNS2	Unbound suboptimal
LaCAM [40]	https://github.com/Kei18/lacam	Unbound suboptimal

Table 7.2: We list the optimal and unbounded suboptimal algorithms that we have evaluated for our website.



(i) (ii) (iii)

Figure 7.1: Screenshots taken from our website. (i) Percentages of # of instances closed; (ii) Percentages of # of instances solved; and (iii) Percentages of # of instances achieved best solution by each algorithm for various domains.

algorithm for one minute and check whether the lower-bound value is increased when finished. If so, we increase the timeout to another minute and keep searching. Otherwise, we terminate the algorithm. For all algorithms, we run every instance of a scenario by increasing the number of agents, and terminate if two instances in a row fail.

7.4.1 Domain and Map-level Analysis

We demonstrate how our website can be used to explore the new pareto-frontier, as well as the insights for the submitted MAPF algorithms. To begin, Figure 7.1 shows the plots that summarise the submitted algorithms on each domain.

Figure 7.1 (i) shows the # of instances closed by each algorithm (shown as percentage). BCP-MAPF slightly outperforms the CBS variations, CBSH2-RTC and CBSH2-RTC-CHBP, on most of the domains, but achieves a lower number of instances closed than Lazy-CBS on the Empty and Random maps. This shows that the exponential reduction in search that Lazy-CBS can achieve pays off on these smaller maps. However, no existing solver is able to close many instances for domains such as Maze, Room and Games. Thus, more attention is needed on how to solve these domains optimally.

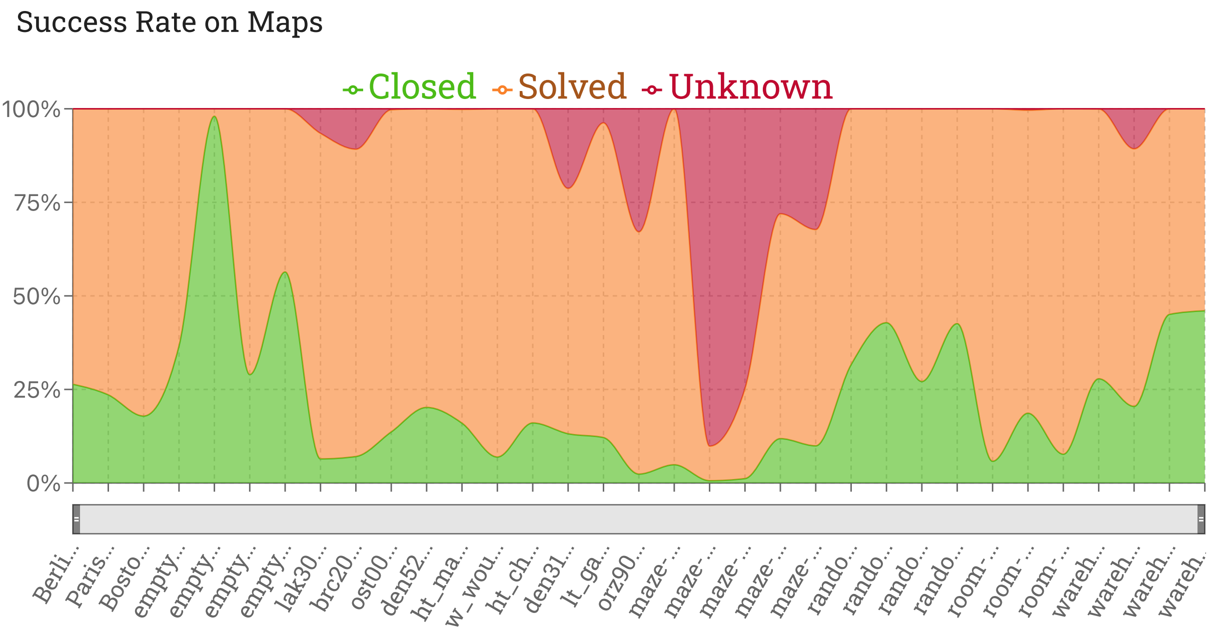
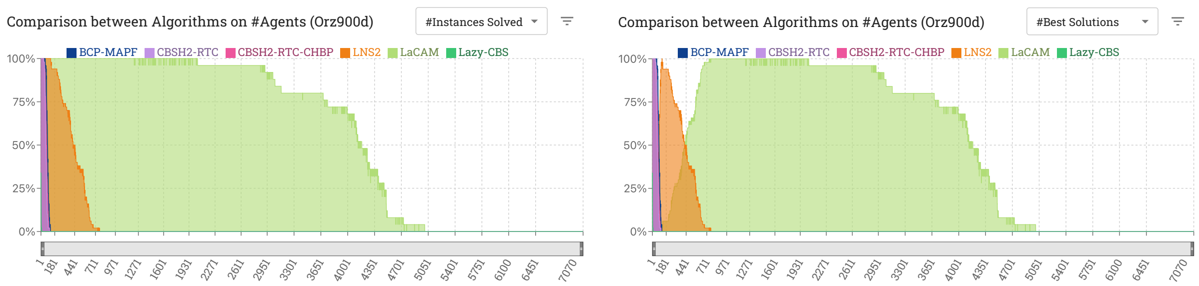


FIGURE 7.2: Screenshots taken from our website. Percentages of # closed, solved and unknown instances shown for various maps on x-axis.

Figure 7.1 (ii) shows the # of instances solved by each algorithm. Clearly, LNS2 significantly outperforms the four optimal solvers in terms of # of instances solved. LaCAM further mitigates the weaknesses of LNS2 and is able to solve more instances than LNS2 on domains such as Room and Game. But, again, no solver is able to effectively tackle maze maps.

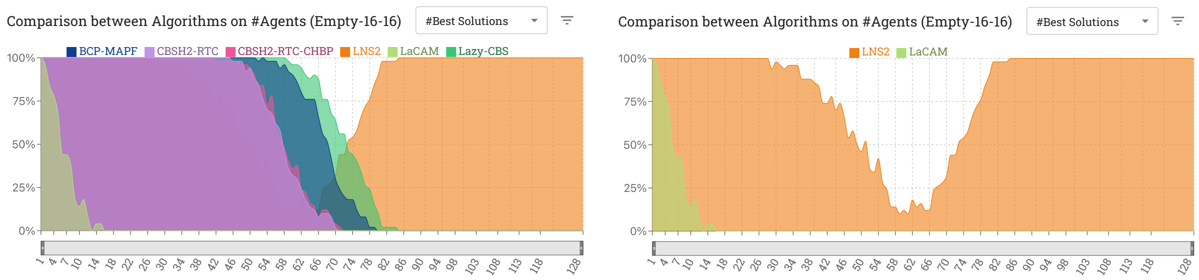
Figure 7.1 (iii) shows the # of best solutions achieved in order to compare the solution quality of the solved instances for each algorithm. Although LaCAM can solve more instances than LNS2 on Room and Game maps, surprisingly, the solution quality of the solved instances of LaCAM is dominated by LNS2 on every domain. In fact, LaCAM is dominated by all other algorithms and finds the best solution when it is the only algorithm capable of finding a solution.

To further dig into the details, Figure 7.2 demonstrates the collective progress that all algorithms have made for different maps in terms of the number of closed, solved and unknown instances. Other than Maze maps, where no solver is able to perform well, almost all instances on all maps have been completely solved. Yet we are still a long way from being able to close all instances. Additionally, we see that some of the large-scale game maps, such as `orz900d`, `den312d` and `brc202d`, are not only far from being completely solved yet, but they also have a low number of closed instances – less than 15% are provably optimal. We now move our attention to the next level down, and focus on the map `orz900d`.



(i) (ii)

FIGURE 7.3: Screenshots taken from our website. (i) Percentages of instances solved; and (ii) Percentages of instances achieving best solution by each algorithm for different number of agents on x-axis, we show the result for game map: `orz900d`.



(i) (ii)

FIGURE 7.4: Screenshots taken from our website. Percentages of instances achieving best solution (i) by each algorithm; and (ii) by LNS2 and LACAM only, for different number of agents on x-axis, we show the result for game map: `empty-16-16`.

7.4.2 Scenario-level Analysis

Figure 7.3 shows the comparison between different solvers on different number of agents for the game map, `orz900d`. On the left, Figure 7.3 (i) shows # of instances solved by each algorithm, where we see the optimal solvers (e.g., BCP-MAPF, Lazy-CBS, CBSH2-RTC and CBSH2-RTC-CHBP) can only scale up to no more than 181 agents. LNS2 is able to solve the instances on more # of agents than the optimal solvers with a maximum improvement of up to four times. LaCAM can scale to many more agents. Clearly, LaCAM outperforms the other solvers, the # of instances it can solve safely scales to around two thousand agents and slowly decreases until reaching around five thousand agents. Figure 7.3 (ii) presents the # of instances where each algorithm achieved the best solution. Unsurprisingly, as we have seen before, LaCAM almost never achieves the best solutions on these instances that are solved by LNS2 or any other solvers.

Note that Figure 7.3 (ii) does not really allow us to contrast the optimal solvers, since they all can only handle a few instances. Figure 7.4 (i) show the same plots for the much smaller map `empty-16-16`, here we can see that CBSH2-RTC finds the best solution for about half the instances, and CBSH2-RTC-CHBP slightly improves upon this, BCP improves further, and Lazy-CBS even further. All of the optimal solvers fail after 85 agents. We can see that on some

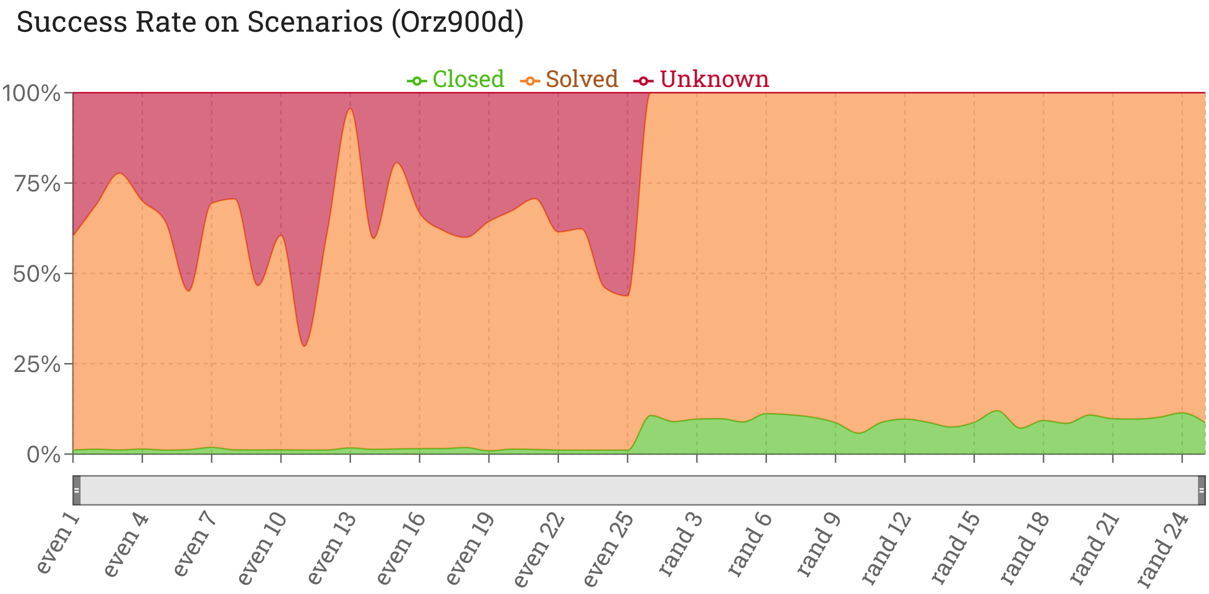


FIGURE 7.5: Screenshots taken from our website. Percentages of closed, solved and unknown instances shown for various scenarios of `orz900d` map on x-axis.

small instances where LaCAM also finds the optimal solution. LNS2 finds best solutions early-on (obscured in the plot) and then dips down, as it finds suboptimal solutions, then returns to 100% (best known solution) for instances where the optimal solvers are unable to prove optimality. The platform allows us to just plot some solvers, restricting to LaCAM and LNS2, Figure 7.4 (ii) shows how good LNS2 is at finding optimal solutions on smaller instances. This illustrates how we can use the platform to easily compare solvers on a single map. We now return to analysing `orz900d`.

Figure 7.5 shows the collective progress that all algorithms have made across the different scenarios of `orz900d`, again in terms of the number of closed, solved and unknown instances. Interestingly, although we have seen there is a large proportion of instances not solved when the # of agents is high, all randomly generated scenarios (i.e., rand 1 - 25) are 100% solved, the unsolved instances only arise in the evenly generated scenarios (i.e., even 1 - 25). This is because the randomly generated scenarios contain up to one thousand agents, whereas the evenly generated scenarios have around six thousand agents in each scenario file. In even scenarios, the source and destination locations are generated based on the maximum grid distance[5], d_{max}, between any two traversable cells on the map. The map is divided into $\lfloor d_{max}/4+1 \rfloor$ buckets of (s, g) pairs, where the i-th bucket contains 10 (s, g) pairs with a grid distance between them within the range of $i \times 4$ to $(i+1) \times 4$. Thus, the number of agents can become large as the size of the map increases. Next, we zoom in to the next-level, to further explore the insight of even-1 scenarios.

[5] The grid distance refers to the optimal distance on a grid map between the source and destination, while ignoring other agents in the map.

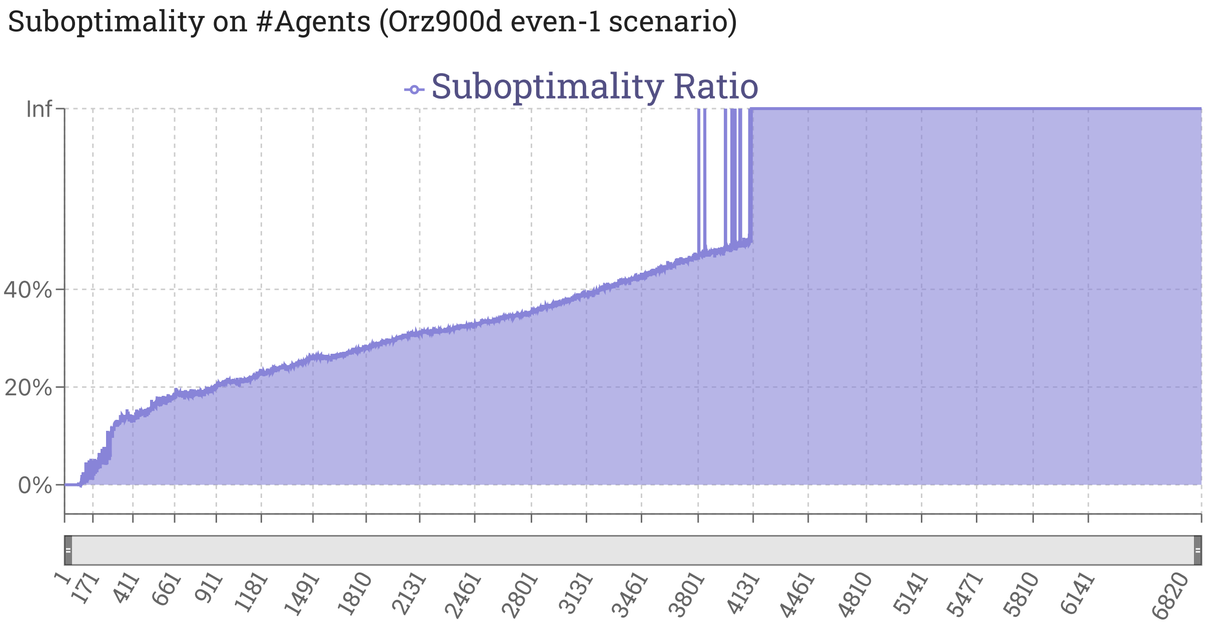


Figure 7.6: Screenshots taken from our website. The suboptimality ratio (i.e., $(\mathcal{S} - \mathcal{L})\ /\ \mathcal{L}$ where $\mathcal{L}$ and $\mathcal{S}$ are the best known lower-bound and solution of the instance) on different # agents for even-1 scenario on `orz900d` map.

7.4.3 Instance-level Analysis

Figure 7.6 shows the suboptimality ratio between best known lower-bound and solution of instances with various # agents on x-axis. Since most instances for the even-1 scenario of `orz900d` are solved only by LaCAM, the suboptimality is computed based on the trivial lower bound (i.e., the SIC of each agent follows the shortest path by ignoring other agents), thus the value indicates the upper bound of suboptimality, where the solution quality of solved instances for LaCAM is no more than around 40% worse than the optimal solution.

Now, let us focus on a particular instance in order to analyse why other algorithms can not solve it. Figure 7.7 shows a screenshot of a feasible solution for an instance of even-1 scenario on part of the `orz900d` map, where the # agents is 4119. From the figure we see that substantial congestion forms around obstacle corners. Tightly bending around corners is a necessary condition for individually optimal paths, but with many agents on the map, following those individually optimal paths incurs substantial delays. We see that agents wait and must form queues to reach their destination locations. This behaviour is characteristic of the Push-and-Swap [42, 119] strategy used in LaCAM. This strategy essentially asks each agent to follow its shortest path if possible, but tries to push or swap the locations with other agents when they are in conflict. Optimal solvers can not solve this instance, because they rely on complicated reasoning techniques to resolve the conflicts occurring in an infeasible plan. Indeed such techniques can only scale up to hundreds of agents. LNS2 meanwhile uses a two-step framework that first plans an initial solution using prioritised planning and then improves that

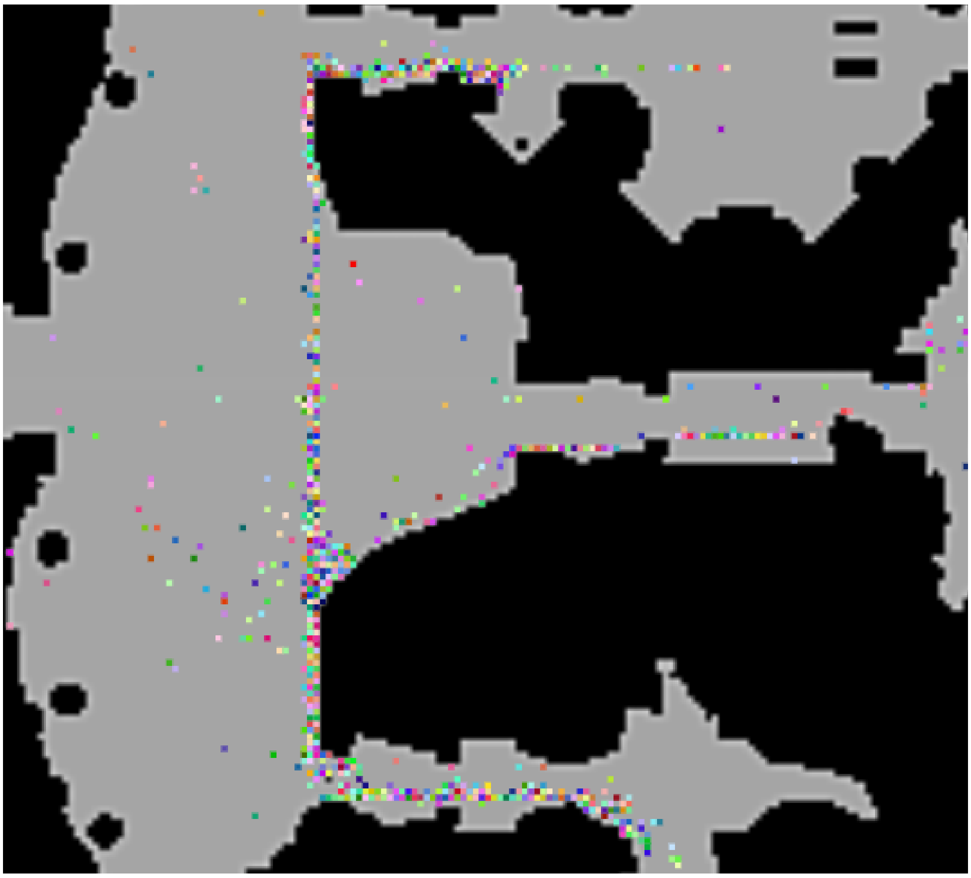


FIGURE 7.7: A feasible solution has been found for an even-1 scenario instance on the `orz900d` map, where # of agents is 4119. We show the screenshot on part of the map from the visualizer of our website. The colored points represent individual agents.

solution using a Large Neighbourhood Search (LNS). The core of this approach – prioritised planning – is known for being very fast and effective, especially on large maps where many paths exist for each agent. Why then is LNS2 unable to solve this instance?

In this case, our visualisation shows that planning in the congested area requires avoiding many temporal obstacles. We also see that agent source locations are far from their destination positions. The combination (long path, many temporal obstacles) explodes the size of the time domain. The effect is that individual path planning times increase, from milliseconds per query on other maps to several seconds per query on this map. We now see that LNS2 fails because there is not enough time to even compute an initial plan. This analysis points out a place where researchers might try to improve the state-of-the-art: by developing better and more effective low-level solvers for the LNS2 algorithm. It seems likely that such cases were not considered during the original development. Fortunately, LNS2 can work with any initial solution, just using its second phase to improve it. One notable takeaway from our analysis is the potential for combining LNS2 and LaCAM. In the combination, we can leverage LaCAM to obtain an initial solution quickly, while utilising LNS to continuously improve the solution quality in remaining computation time. This combination effectively addresses the drawbacks of both algorithms at the same time, and represents an interesting direction for future work.

7.5 Discussion

We introduce a new set of methodologies to track the progress of MAPF from various perspectives. We evaluate several currently leading optimal and suboptimal solvers on a large-scale experiment. The detailed results are publicly accessible and illustrated by a set of well-built visualisation tools on our online platform. We believe that our platform helps: identify the remaining challenges in MAPF problems; establish state-of-the-art performance; and provide a valuable dynamic resource for understanding the various challenges arising in MAPF.

Currently, the platform concentrates only on lower and upper bounds for problem instances. This is usual in the world of optimisation problems, since it allows us to understand the gap between lower and upper bounds, and hence how far away an instance is likely to be from being *closed*. It also allows us to attribute the closure of instances to particular algorithms. The platform does not consider run times, and indeed many of the results currently there could be improved by giving more runtime to the algorithms.[6] Clearly, runtimes are of interest in practice. But runtime is hard to compare fairly (over different machines), and indeed there is no way to guarantee claimed runtimes as part of a submission are indeed correct. A future direction for the platform is to allow executables to be submitted, and then run for a fixed amount of wall clock time on each instance. This would give us a more detailed view of the state-of-the-art from the run-time perspective. Another interesting direction is to extend the platform to different types of agent models. This would allow the community to track progress on a variety of different MAPF variants, which all share common benchmarks; e.g., Continuous-time MAPF [153]. Lastly, we would like to mention that the objective of our website is not merely to determine a winning algorithm, but rather to generate valuable insights that can assist researchers in their endeavors. Indeed, comprehending the performance of algorithms on a benchmark suite presents a multifaceted challenge [154], and perhaps the establishment of these benchmarking utilities should follow an entirely open process involving the entire community. Nevertheless, it is essential to establish a starting point for this journey.

[6] Although in many cases doubling or even multiplying available time by 10 will make little difference, and CBS algorithms may run out of memory before reaching the time limit [92].

Chapter 8

Final Remarks

Pathfinding queries are one of the most ubiquitous uses of computing and are fundamental for many real-world applications. In this dissertation, we develop efficient yet effective algorithms for solving pathfinding problems in various environments with different computational challenges that have been identified in Chapter 3. Specifically, our algorithms advance the state-of-the-art in several distinct parts of the literature: computer games, route planning software, and automated warehouses.

8.1 Advances in Pathfinding for Computer Games

In Chapter 3, we investigated the pathfinding problem in computer games where the environment comprises a Euclidean plane with polygon-shaped obstacles. Limited researches have been conducted on pathfinding in a Euclidean space, and most of the existing solutions have to suffer the issue of *first move lag* (i.e., **Challenge #1**). In this dissertation, we developed End-Point Search (EPS), an optimal algorithm that overcomes this issue through the use of any-time behaviour. Not only does EPS exhibit strong any-time behaviour, but it is also significantly faster than several state-of-the-art algorithms, e.g., EPS is approximately 2-3 times faster than SUB-NL (i.e., the fastest suboptimal grid planner), 2-4 times faster than ENLSVG (i.e., the fastest visibility graph planner), and up to an order of magnitude faster than Polyanya (i.e., the fastest online planner). Our second strategy of migrating *first move lag* is through the development of Centroid Path Extraction (CPE), a bounded suboptimal algorithm that completely avoids the search process by constructing an oracle to efficiently extract suboptimal paths. Our experiments demonstrate that CPE outperforms leading suboptimal algorithms SUB-NL variations, in terms of path quality. Additionally, CPE achieves faster run times than EPS by at least an order of magnitude.

Although we have developed efficient algorithms for pathfinding in two-dimensional space, with the game industry's growing interest in developing games in higher-dimensional spaces, the need for efficient navigation techniques in such spaces has become increasingly urgent and indispensable. Thus, a promising future direction could be:

Pathfinding in Higher-Dimensional Space. While pathfinding in 2D space is well-studied, research on pathfinding in 3D space is limited. Most current approaches involve modelling the 3D environment as a grid map and adapting popular grid-based pathfinding techniques to address the challenges of 3D navigation. Despite attempts to reduce the search space by adapting Jump Point Search (JPS) [155], pathfinding in 3D still runs extremely slowly due to a large number of grid cells, and may even run out of memory before a solution is found. Similar to the 2D space, the traversable area in 3D contains a large number of grid cells but a limited number of convex vertices around obstacles. Therefore, extending EPS to solve the pathfinding problem in 3D shows promise. However, designing an efficient 3D pathfinding algorithm to construct a visibility graph between convex vertices and insert connections from the source and destination points to these vertices is a challenging task. One potential solution could be to extend Anya [58] or Polyanya [14] to the 3D environment.

8.2 Advances in Pathfinding for Route Planning Software

Chapters 4 to 5 focus on the pathfinding problem in road networks for route planning software. Our investigation includes two types of networks: (i) static road networks, where each edge is assigned a non-negative constant value; and (ii) time-dependent road networks, where the cost of each edge is represented by a travel time function that varies over time.

- In Chapter 4, we focus on the static road network. Although the leading oracle-based approaches often find optimal paths fast, it still requires substantial time and memory to build and store the oracle (i.e., **Challenge #2**). In this dissertation, we overcome this issue by developing a novel oracle called CH-CPD. CH-CPD utilises the shortcuts introduced by Contraction Hierarchies (CH) to address the drawbacks of conventional Compressed Path Databases (CPD) and further improves the preprocessing time by caching the distances of "important nodes" from CH. In our experiments, we demonstrate that CH-CPD requires less preprocessing time than conventional CPD and outperforms several existing algorithms, including CH and CPD. Additionally, we also introduce the partial CH-CPD search, which guarantees finding the optimal path by constructing the CH-CPD only for a portion of the CH nodes. This approach further reduces preprocessing costs (i.e., time and space) while maintaining competitive runtime.

- In Chapter 5, we focus on the time-dependent road network. Unlike static road networks, time-dependent networks require more complex travel time functions that change over a large period of time. Thus, existing algorithms often compute solutions slowly and require substantial time and memory to construct auxiliary data structures covering the entire time domain (i.e., **Challenge #3**). To address the issue, we focus on enhancing the state-of-the-art algorithm, Time-dependent contraction Hierarchies (TCH). First, we demonstrate that TCH can be improved by incorporating time-independent heuristics such as landmarks and advanced path databases heuristic. While these heuristics require preprocessing, the supporting data structure does not scale larger with time domain increasing. Second, we show that TCH can be accelerated by dividing the time domain to construct multiple sub-TCHs that only cover a smaller time period of time domain. Experiments show that our heuristic methods improve TCH by a factor of 2-3. Additionally, our time-splitting algorithms are approximately 3 times faster than TCH. By combining both heuristic and time-splitting algorithms, we achieve the best performance, resulting in a speedup by a factor of 5.

Throughout our studies of pathfinding problem in road network, we have seen the superiority of oracle-based approaches for finding the optimal path. However, the dynamic nature of road networks requires frequent updates to edge costs to reflect real-life road conditions, such as road closures, event schedules, etc. As a result, the pre-computed oracles may become unreliable and unable to accurately respond to pathfinding queries. Therefore, a future direction could be:

Dynamically Updating the Path Oracle. Most of the existing works of path oracle focus on improving the compression [79] and storing the oracle distributively [25]. However, there is no clear way of updating the path oracle without completely reconstructing it. One potential approach for handling updates is to assign labels to each edge, similar to the technique described for Arc flags [20], which involves identifying a set of nodes that could be affected by any change in the cost of the edge. During the update phase, we can reconstruct only the rows or columns of the path oracle that contain these nodes. However, creating and maintaining such an edge index poses an interesting problem. Another possible method for updating is through graph decomposition [156], where we divide the network into a set of sub-graphs and construct a path oracle for each sub-graph. While updating a single path oracle becomes less expensive, using multiple sub-oracles to find the optimal path presents another challenging problem.

8.3 Advances in Pathfinding for Automated Warehouses

Chapter 6 to 7 focus on the pathfinding problem in automated warehouses, which requires the coordination of numerous agents simultaneously. The key coordination challenge, known as

Multi-Agent Path Finding (MAPF), considers the environment as a discretised grid map and aims to find collision-free paths for each agent that minimise the sum of their individual path costs. Our contributions to this area are twofold:

- In Chapter 6, we focus on improving Conflict-Based Search (CBS), a leading search-based algorithm that finds the optimal solution for MAPF problems. Although there has been massive advances for improving the CBS, existing works only reason about incompatibility between at most two agents at a time (i.e., **Challenge #4**). In this dissertation, we overcome this issue by developing a new algorithmic technique called Cluster Heuristic and ByPass (CHBP). CHBP leverages mutex propagation to determine the incompatibility of clusters of more than two agents. These conflict clusters assist CBS in obtaining stronger bounds and devising innovative bypasses, resulting in a significant reduction in the size of the CT tree. Through a range of experimental results, we demonstrate that CHBP is able to solve more instances than CBSH2-RTC (i.e., the leading CBS variation) within the same runtime. For the co-solvable instances, CHBP runs faster than CBSH2-RTC with speedups up to one order of magnitude. On the other hand, CHBP pushes the lower-bound (i.e., f_{min}) to a significantly higher value for the instances that cannot be solved within the time limit.
- In Chapter 7, we aim to improve the standardised MAPF benchmarks [3]. As the industrial interests continue to grow, the number of publications on MAPF have exploded, making it difficult to track the progress (i.e., **Challenge #5**). In this dissertation, we present an online platform that provides researchers with a suite of methodological and visualisation tools to systematically analyse benchmarks and compare between a wide range of MAPF methods. By undertaking a large set of experiments with several leading optimal and suboptimal solvers, we demonstrate how our system can be used to identify the current Pareto frontier, highlight the strengths of existing research, and uncover remaining challenges within the field. Moving forward, we believe our proposed platform will alleviate the barriers to entry for new research on MAPF.

In the classic Multi-Agent Path Finding (MAPF) problem, we have improved the CBS algorithm by addressing one of its open problems - reasoning about the incompatibility of more than two agents. Our innovative cluster reasoning technique CHBP represents a new approach that opens up exciting possibilities for further improvement of the CBS family of algorithms. Specifically, we detail the three future directions that are promising:

- **Improving Clusters Detection:** CHBP detects conflict clusters by incrementally performing mutex propagation from one agent to others. However, this approach has two main limitations. Firstly, it may result in the detection of superset clusters that include unnecessary agents. Secondly, it may fail to detect some of the more complex clusters. These drawbacks arise because the cluster detection process only considers the incompatible nodes identified by the mutex propagation. Therefore, one possible approach for enhancing cluster detection is to employ a more powerful reasoning technique or a more sophisticated use of mutex propagation.

- **Integrating WDG Heuristic:** Currently, CHBP identifies conflict clusters that are independent of each other and with the WDG heuristic. To generate a stronger heuristic, a possible strategy is to perform greedy cluster detection on each agent and then formulate a linear integer problem to compute the overall increasing cost of clusters and agent pairs. However, a key challenge is to ensure that these more sophisticated heuristics can be computed efficiently, as the performance may be degraded even if a better heuristic is computed but at a much higher computation cost.

- **Resolving Cluster Conflicts:** In their study, Li et al. [34] have shown that resolving symmetric conflicts between pairs of agents can substantially decrease high-level node expansion, resulting in improved performance for CBS. Similarly, the CBS algorithm also needs to generate a considerable number of CT nodes to resolve conflicts within a conflict cluster. Consequently, developing efficient algorithms to generate constraints for resolving conflict clusters is the next critical challenge that needs to be addressed to enable further improvement of the CBS family of algorithms.

As discussed in Chapter 7, another promising future direction for the classic Multi-Agent Path Finding (MAPF) problem is to combine the Large Neighbourhood Search (LNS2) [43] with the Lazy Constraint Addition Search (LaCAM) [40]:

- **Combining LNS2 with LaCAM:** The initial results from our online platform demonstrate that LaCAM can find feasible solutions on most instances, though those solutions almost always have a lower solution quality than other solvers. In contrast, LNS2 typically finds solutions with reasonable quality, but often fails on these challenging instances with a large number of agents. This is because LNS2 relies on a prioritised planning algorithm to find an initial solution, which often fails to return a solution when large groups of agents are congested in a particular area of the map. Fortunately, LNS2 is a two-step framework that can work with any initial solution. Therefore, a notable takeaway from our analysis is the potential to combine LNS2 and LaCAM. In this combination, we can leverage LaCAM to obtain an initial solution quickly, while utilising LNS to continuously improve the solution quality in remaining computation time. We believe this combination can lead to a new state-of-the-art unbounded suboptimal solver for MAPF, effectively addressing the drawbacks of both algorithms at the same time.

Bibliography

[1] Nathan R. Sturtevant. Moving path planning forward. In *Proceedings of the International Conference on Motion in Games*, pages 1–6. Springer, 2012.

[2] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable route planning in road networks. *Transportation Science*, 51(2):566–591, 2017.

[3] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, T. K. Satish Kumar, Roman Barták, and Eli Boyarski. Multi-agent path finding: Definitions, variants, and benchmarks. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 151–159. AAAI Press, 2019.

[4] Wolfgang Hönig, James A Preiss, TK Satish Kumar, Gaurav S Sukhatme, and Nora Ayanian. Trajectory planning for quadrotor swarms. *IEEE Transactions on Robotics*, 34 (4):856–869, 2018.

[5] Muhammad Aamir Cheema. Indoor location-based services: challenges and opportunities. *ACM SIGSPATIAL Special*, 10(2):10–17, 2018.

[6] Maoguo Gong, Guanjun Li, Zhao Wang, Lijia Ma, and Dayong Tian. An efficient shortest path approach for social networks based on community structure. *CAAI Transactions on Intelligence Technology*, 1(1):114–123, 2016.

[7] Adi Botea, Bruno Bouzy, Michael Buro, Christian Bauckhage, and Dana S. Nau. Pathfinding in games. In *Artificial and Computational Intelligence in Games*, volume 6, pages 21–31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013.

[8] Julian Berteling. Beyond 'killzone': Creating new ai systems for 'horizon zero dawn'. In *Game Developers Conference*, 2018.

[9] Daniel Damir Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1114–1119. AAAI Press, 2011.

[10] Peter Yap, Neil Burch, Robert C. Holte, and Jonathan Schaeffer. Block a*: Database-driven search with applications in any-angle path-planning. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 120–125. AAAI Press, 2011.

[11] Tansel Uras and Sven Koenig. An empirical comparison of any-angle path-planning algorithms. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, pages 206–211. AAAI Press, 2015.

[12] Nicolás Hormazábal, Antonio Díaz, Carlos Hernández, and Jorge A. Baier. Fast and almost optimal any-angle pathfinding using the 2^k neighborhoods. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, pages 139–143. AAAI Press, 2017.

[13] Alex Nash and Sven Koenig. Any-angle path planning. *AI Magazine*, 34(4):9, 2013.

[14] Michael Cui, Daniel Damir Harabor, and Alban Grastien. Compromise-free pathfinding on a navigation mesh. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 496–502. AAAI Press, 2017.

[15] Shunhao Oh and Hon Wai Leong. Edge n-level sparse visibility graphs: Fast optimal any-angle pathfinding using hierarchical taut paths. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, pages 64–72. AAAI Press, 2017.

[16] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 156–165. SIAM, 2005.

[17] 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, 1968.

[18] Ira Pohl. *Bi-directional and heuristic search in path problems.* PhD thesis, Stanford Linear Accelerator Center, USA, 1969.

[19] Robert C. Holte, Ariel Felner, Guni Sharon, and Nathan R. Sturtevant. Bidirectional search that is guaranteed to meet in the middle. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 3411–3417. AAAI Press, 2016.

[20] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In *The Shortest Path Problem, Proceedings of a DIMACS Workshop*, volume 74, pages 41–72. DIMACS/AMS, 2006.

[21] Ulrich Lauther. An experimental evaluation of point-to-point shortest path calculation on road networks with precalculated edge-flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem, Proceedings of a DIMACS Workshop*, volume 74, pages 19–39. DIMACS/AMS, 2006.

[22] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms: 7th International Workshop (WEA)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008.

[23] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *ESA*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[24] Ronald J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 100–111. SIAM, 2004.

[25] Arthur Mahéo, Shizhe Zhao, Hassan Afzaal, Daniel Harabor, Peter J. Stuckey, and Mark Wallace. Customised shortest paths using a distributed reverse oracle. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, pages 79–87. AAAI Press, 2021.

[26] Adi Botea. Ultra-fast optimal pathfinding without runtime search. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AI-IDE)*, pages 122–127. AAAI Press, 2011.

[27] Ben Strasser, Daniel Harabor, and Adi Botea. Fast first-move queries through run-length encoding. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, pages 157–165. AAAI Press, 2014.

[28] Daniel Delling and Dorothea Wagner. Landmark-based routing in dynamic graphs. In *Experimental Algorithms, 6th International Workshop (WEA)*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2007.

[29] Giacomo Nannicini, Daniel Delling, Dominik Schultes, and Leo Liberti. Bidirectional A^* search on time-dependent road networks. *Networks*, 59(2):240–251, 2012.

[30] Daniel Delling and Giacomo Nannicini. Core routing on dynamic time-dependent road networks. *INFORMS Journal on Computing*, 24(2):187–201, 2012.

[31] Daniel Delling. Time-dependent sharc-routing. *Algorithmica*, 60(1):60–94, 2011.

[32] Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-dependent contraction hierarchies. In *Proceedings of the Eleventh Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 97–105. SIAM, 2009.

[33] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent path finding. *Artificial Intelligence*, 219:40–66, 2012.

[34] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, Graeme Gange, and Sven Koenig. Pairwise symmetry reasoning for multi-agent path finding search. *Artificial Intelligence*, 301:103574, 2021.

[35] Han Zhang, Jiaoyang Li, Pavel Surynek, TK Satish Kumar, and Sven Koenig. Multi-agent path finding with mutex propagation. *Artificial Intelligence*, 311:103766, 2022.

[36] Ariel Felner, Jiaoyang Li, Eli Boyarski, Hang Ma, Liron Cohen, T. K. Satish Kumar, and Sven Koenig. Adding heuristics to conflict-based search for multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 83–87. AAAI Press, 2018.

[37] Jiaoyang Li, Ariel Felner, Eli Boyarski, Hang Ma, and Sven Koenig. Improved heuristics for multi-agent path finding with conflict-based search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 442–449. AAAI Press, 2019.

[38] Eli Boyarski, Ariel Felner, Guni Sharon, and Roni Stern. Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 47–51. AAAI Press, 2015.

[39] Edward Lam, Pierre Le Bodic, Daniel Harabor, and Peter J. Stuckey. Branch-and-cut-and-price for multi-agent path finding. *Computers & Operations Research*, 144:105809, 2022.

[40] Keisuke Okumura. Lacam: Search-based algorithm for quick multi-agent pathfinding. *CoRR*, abs/2211.13432, 2022.

[41] Graeme Gange, Daniel Harabor, and Peter J. Stuckey. Lazy CBS: implicit conflict-based search using lazy clause generation. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 155–162. AAAI Press, 2019.

[42] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. Priority inheritance with backtracking for iterative multi-agent path finding. *Artificial Intelligence*, 310:103752, 2022.

[43] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. MAPF-LNS2: Fast repairing for multi-agent path finding via large neighborhood search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 10256–10265. AAAI Press, 2022.

[44] Eric A. Hansen and Rong Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28:267–297, 2007.

[45] Bojie Shen, Muhammad Aamir Cheema, Daniel Harabor, and Peter J. Stuckey. Euclidean pathfinding with compressed path databases. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4229–4235. AAAI Press, 2020.

[46] Bojie Shen, Muhammad Aamir Cheema, Daniel Damir Harabor, and Peter J. Stuckey. Fast optimal and bounded suboptimal euclidean pathfinding. *Artificial Intelligence*, 302: 103624, 2022.

[47] Bojie Shen, Muhammad Aamir Cheema, Daniel Damir Harabor, and Peter J. Stuckey. Contracting and compressing shortest path databases. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 322–330. AAAI Press, 2021.

[48] Massimo Bono, Alfonso Emilio Gerevini, Daniel Damir Harabor, and Peter J. Stuckey. Path planning with CPD heuristics. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1199–1205. AAAI Press, 2019.

[49] Bojie Shen, Muhammad Aamir Cheema, Daniel Damir Harabor, and Peter J. Stuckey. Improving time-dependent contraction hierarchies. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 338–347. AAAI Press, 2022.

[50] Bojie Shen, Zhe Chen, Jiaoyang Li, Muhammad Aamir Cheema, Daniel Damir Harabor, and Peter J. Stuckey. Beyond pairwise reasoning in multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 338–347. AAAI Press, 2023.

[51] Bojie Shen, Zhe Chen, Muhammad Aamir Cheema, Daniel Damir Harabor, and Peter J. Stuckey. Tracking progress in multi-agent path finding. *CoRR*, abs/2305.08446, 2023.

[52] Tansel Uras, Sven Koenig, and Carlos Hernández. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 224–232. AAAI Press, 2013.

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

[54] Nathan R. Sturtevant, Ariel Felner, Max Barer, Jonathan Schaeffer, and Neil Burch. Memory-based heuristics for explicit state spaces. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 609–614. AAAI Press, 2009.

[55] Jiaoyang Li, Ariel Felner, Sven Koenig, and T. K. Satish Kumar. Using fastmap to solve graph problems in a euclidean space. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 273–278. AAAI Press, 2019.

[56] Daniel Damir Harabor and Alban Grastien. Improving jump point search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 128–135. AAAI Press, 2014.

[57] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1(1):1–30, 2004.

[58] Daniel Damir Harabor, Alban Grastien, Dindar Öz, and Vural Aksakalli. Optimal any-angle pathfinding in practice. *Journal of Artificial Intelligence Research*, 56:89–118, 2016.

[59] Tansel Uras and Sven Koenig. Speeding-up any-angle path-planning on grids. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 234–238. AAAI Press, 2015.

[60] Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Felner. Theta*: Any-angle path planning on grids. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1177–1183. AAAI Press, 2007.

[61] Dave Ferguson and Anthony Stentz. Field D*: An interpolation-based path planner and replanner. In *Proceedings of the International Symposium of Robotic Research (ISRR)*, pages 239–253. Springer, 2005.

[62] Tomás Lozano-Pérez and Michael A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–570, 1979.

[63] *AI Game Programming Wisdom 4*. Charles River Media, 2008.

[64] Marcelo Kallmann and Mubbasir Kapadia. Navigation meshes and realtime dynamic planning for virtual worlds. In *ACM SIGGRAPH 2014 Courses*, page 3. AAAI Press, 2014.

[65] L. Paul Chew. Constrained delaunay triangulations. *Algorithmica*, 4(1):97–108, 1989.

[66] Marcelo Kallmann. Path planning in triangulations. In *Proceedings of the IJCAI workshop on reasoning, representation, and learning in computer games*, pages 49–54. AAAI Press, 2005.

[67] John Hershberger and Jack Snoeyink. Computing minimum length paths of a given homotopy class. *Computational Geometry*, 4:63–97, 1994.

[68] Douglas Demyen and Michael Buro. Efficient triangulation-based pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 942–947. AAAI Press, 2006.

[69] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient point-to-point shortest path algorithms. In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 129–143. SIAM, 2006.

[70] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Highway hierarchies star. In *The Shortest Path Problem, Proceedings of a DIMACS Workshop*, volume 74, pages 141–174. DIMACS/AMS, 2006.

[71] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. *ACM Journal of Experimental Algorithmics*, 15:2–1, 2010.

[72] Jens Maue, Peter Sanders, and Domagoj Matijevic. Goal-directed shortest-path queries using precomputed cluster distances. *ACM Journal of Experimental Algorithmics*, 14: 316–327, 2009.

[73] Hannah Bast, Stefan Funke, and Domagoj Matijevic. Ultrafast shortest-path queries via transit nodes. In *The Shortest Path Problem, Proceedings of a DIMACS Workshop*, volume 74, pages 175–192. DIMACS/AMS, 2006.

[74] Reinhard Bauer and Daniel Delling. SHARC: fast and robust unidirectional routing. *ACM Journal of Experimental Algorithmics*, 14:2–4, 2009.

[75] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing speed-up techniques is hard. In *Proceedings of the International Conference on Algorithms and Complexity (CIAC)*, volume 6078 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 2010.

[76] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *Journal of Experimental Algorithmics (JEA)*, 21:1–49, 2016.

[77] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[78] Andy Diwen Zhu, Hui Ma, Xiaokui Xiao, Siqiang Luo, Youze Tang, and Shuigeng Zhou. Shortest path and distance queries on road networks: towards bridging theory and practice. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 857–868. ACM, 2013.

[79] Mattia Chiari, Shizhe Zhao, Adi Botea, Alfonso Emilio Gerevini, Daniel Harabor, Alessandro Saetti, Matteo Salvetti, and Peter J. Stuckey. Cutting the size of compressed path databases with wildcards and redundant symbols. In *Proceedings of the International*

Conference on Automated Planning and Scheduling (ICAPS), pages 106–113. AAAI Press, 2019.

[80] Ben Strasser, Adi Botea, and Daniel Harabor. Compressing optimal paths with run length encoding. *Journal of Artificial Intelligence Research*, 54:593–629, 2015.

[81] Ittai Abraham, Daniel Delling, Andrew V Goldberg, and Renato F Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *International Symposium on Experimental Algorithms*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2011.

[82] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[83] Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 147–154. SIAM, 2014.

[84] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. An experimental study on hub labeling based shortest path algorithms. *Proceedings of the VLDB Endowment*, 11 (4):445–457, 2017.

[85] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *ACM Journal of Experimental Algorithmics*, 18:1–1, 2013.

[86] Ben Strasser, Dorothea Wagner, and Tim Zeitz. Space-efficient, fast and exact routing in time-dependent road networks. *Algorithms*, 14(3):90, 2021.

[87] Jingjin Yu and Steven M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1443–1449. AAAI Press, 2013.

[88] Trevor Scott Standley. Finding optimal solutions to cooperative pathfinding problems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 173–178. AAAI Press, 2010.

[89] Meir Goldenberg, Ariel Felner, Roni Stern, Guni Sharon, Nathan R. Sturtevant, Robert C. Holte, and Jonathan Schaeffer. Enhanced partial expansion A*. *Journal of Artificial Intelligence Research*, 50:141–187, 2014.

[90] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *Artificial Intelligence*, 219:1–24, 2015.

[91] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, 2013.

[92] Eli Boyarski, Ariel Felner, Daniel Harabor, Peter J. Stuckey, Liron Cohen, Jiaoyang Li, and Sven Koenig. Iterative-deepening conflict-based search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4084–4090. AAAI Press, 2021.

[93] Richard E. Korf. Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1034–1036. AAAI Press, 1985.

[94] 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 International Symposium on Combinatorial Search (SoCS)*, pages 97–104. AAAI Press, 2012.

[95] Jingjin Yu and Steven M. LaValle. Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics*, 32(5):1163–1177, 2016.

[96] Edward Lam and Pierre Le Bodic. New valid inequalities in branch-and-cut-and-price for multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 184–192. AAAI Press, 2020.

[97] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, volume 285, pages 810–818. IOS Press, 2016.

[98] Pavel Surynek. Unifying search-based and compilation-based approaches to multi-agent path finding through satisfiability modulo theories. In Sarit Kraus, editor, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1177–1183. AAAI Press, 2019.

[99] 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 International Symposium on Combinatorial Search (SoCS)*, pages 961–962. AAAI Press, 2014.

[100] Ira Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1(3): 193–204, 1970.

[101] Cornelia Ferner, Glenn Wagner, and Howie Choset. ODrM* optimal multirobot path planning in low dimensional search spaces. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3854–3859. IEEE, 2013.

[102] Faten Aljalaud and Nathan R. Sturtevant. Finding bounded suboptimal multi-agent path planning solutions using increasing cost tree search (extended abstract). In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 203–204. AAAI Press, 2013.

[103] Judea Pearl and Jin H. Kim. Studies in semi-admissible heuristics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(4):392–399, 1982.

[104] Jiaoyang Li, Wheeler Ruml, and Sven Koenig. EECBS: A bounded-suboptimal search for multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 12353–12362. AAAI Press, 2021.

[105] Jordan Tyler Thayer and Wheeler Ruml. Bounded suboptimal search: A direct approach using inadmissible estimates. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 674–679. AAAI Press, 2011.

[106] Shao-Hung Chan, Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. Flex distribution for bounded-suboptimal multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 9313–9322. AAAI Press, 2022.

[107] Pavel Surynek. Bounded sub-optimal multi-robot path planning using satisfiability modulo theory (SMT) approach. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 11631–11637. IEEE, 2020.

[108] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Sub-optimal sat-based approach to multi-agent path-finding problem. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 90–105. AAAI Press, 2018.

[109] Michael A. Erdmann and Tomás Lozano-Pérez. On multiple moving objects. *Algorithmica*, 2:477–521, 1987.

[110] Jur P. van den Berg and Mark H. Overmars. Prioritized motion planning for multiple robots. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 430–435. IEEE, 2005.

[111] Jiangxing Wang, Jiaoyang Li, Hang Ma, Sven Koenig, and T. K. Satish Kumar. A new constraint satisfaction perspective on multi-agent path finding: Preliminary results. In *Proceedings of the International Joint Conference on Autonomous Agents and Mult-Agent Systems (AAMAS)*, pages 2253–2255. International Foundation for Autonomous Agents and Multiagent Systems, 2019.

[112] Shuyang Zhang, Jiaoyang Li, Taoan Huang, Sven Koenig, and Bistra Dilkina. Learning a priority ordering for prioritized planning in multi-agent path finding. In *Proceedings*

of the International Symposium on Combinatorial Search (SoCS), pages 208–216. AAAI Press, 2022.

[113] Hang Ma, Daniel Harabor, Peter J. Stuckey, Jiaoyang Li, and Sven Koenig. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 7643–7650. AAAI Press, 2019.

[114] David Silver. Cooperative pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 117–122. AAAI Press, 2005.

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

[116] Pavel Surynek. A novel approach to path planning for multiple robots in bi-connected graphs. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 3613–3619. IEEE, 2009.

[117] Dingyuan Shi, Nan Zhou, Yongxin Tong, Zimu Zhou, Yi Xu, and Ke Xu. Collision-aware route planning in warehouses made efficient: A strip-based framework. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. IEEE, 2023.

[118] 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.

[119] Ryan Luna and Kostas E. Bekris. Push and swap: Fast cooperative path-finding with completeness guarantees. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 294–300. AAAI Press, 2011.

[120] Qandeel Sajid, Ryan Luna, and Kostas E. Bekris. Multi-agent pathfinding with simultaneous execution of single-agent primitives. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 88–96. AAAI Press, 2012.

[121] Boris de Wilde, Adriaan ter Mors, and Cees Witteveen. Push and rotate: cooperative multi-agent path planning. In *Proceedings of the International Joint Conference on Autonomous Agents and Mult-Agent Systems (AAMAS)*, pages 87–94. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[122] Adam Wiktor, Dexter Scobee, Sean Messenger, and Christopher Clark. Decentralized and complete multi-robot motion planning in confined spaces. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1168–1175. IEEE, 2014.

[123] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. Priority inheritance with backtracking for iterative multi-agent path finding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 535–542. AAAI Press, 2019.

[124] Keisuke Okumura, Yasumasa Tamura, and Xavier Défago. winpibt: Expanded prioritized algorithm for iterative multi-agent path finding. *CoRR*, abs/1905.10149, 2019.

[125] Jiaoyang Li, Zhe Chen, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. Anytime multi-agent path finding via large neighborhood search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 4127–4135. AAAI Press, 2021.

[126] Taoan Huang, Jiaoyang Li, Sven Koenig, and Bistra Dilkina. Anytime multi-agent path finding via machine learning-guided large neighborhood search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 9368–9376. AAAI Press, 2022.

[127] Jiaoyang Li, Zhe Chen, Yi Zheng, Shao-Hung Chan, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. Scalable rail planning and replanning: Winning the 2020 flatland challenge. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 477–485. AAAI Press, 2021.

[128] Florian Laurent, Manuel Schneider, Christian Scheller, Jeremy Watson, Jiaoyang Li, Zhe Chen, Yi Zheng, Shao-Hung Chan, Konstantin Makhnev, Oleg Svidchenko, Vladimir Egorov, Dmitry Ivanov, Aleksei Shpilman, Evgenija Spirovska, Oliver Tanevski, Aleksandar Nikov, Ramon Grunder, David Galevski, Jakov Mitrovski, Guillaume Sartoretti, Zhiyao Luo, Mehul Damani, Nilabha Bhattacharya, Shivam Agarwal, Adrian Egli, Erik Nygren, and Sharada Mohanty. Flatland competition 2020: Mapf and marl for efficient train coordination on a grid world. In *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*, volume 133 of *Proceedings of Machine Learning Research*, pages 275–301, 2021.

[129] Eli Boyarski, Ariel Felner, Roni Stern, Guni Sharon, David Tolpin, Oded Betzalel, and Solomon Eyal Shimony. ICBS: improved conflict-based search algorithm for multi-agent pathfinding. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 740–746. AAAI Press, 2015.

[130] Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. New techniques for pairwise symmetry breaking in multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 193–201. AAAI Press, 2020.

[131] Jiaoyang Li, Daniel Harabor, Peter J. Stuckey, Hang Ma, and Sven Koenig. Symmetry-breaking constraints for grid-based multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 6087–6095. AAAI Press, 2019.

[132] Pavel Surynek, Jiaoyang Li, Han Zhang, T. K. Satish Kumar, and Sven Koenig. Mutex propagation for SAT-based multi-agent path finding. In *Proceedings of the International Conference on Principles and Practice of Multi-Agent Systems (PRIMA)*, volume 12568 of *Lecture Notes in Computer Science*, pages 248–258. Springer, 2020.

[133] Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[134] XuanLong Nguyen and Subbarao Kambhampati. Extracting effective and admissible state space heuristics from the planning graph. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 798–805. AAAI Press, 2000.

[135] Henry A. Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1194–1201. AAAI Press, 1996.

[136] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1): 99–118, 1977.

[137] Eli Boyarski, Ariel Felner, Pierre Le Bodic, Daniel Harabor, Peter J. Stuckey, and Sven Koenig. f-aware conflict prioritization & improved heuristics for conflict-based search. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 12241–12248. AAAI Press, 2021.

[138] Jayanth Krishna Mogali, Willem-Jan van Hoeve, and Stephen F. Smith. Template matching and decision diagrams for multi-agent path finding. In *Proceedings of the International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR)*, volume 12296 of *Lecture Notes in Computer Science*, pages 347–363. Springer, 2020.

[139] Yun-Hui Liu and Suguru Arimoto. Path Planning Using a Tangent Graph for Mobile Robots Among Polygonal and Curved Obstacles. *International Journal of Robotics Research*, 11:376–382, August 1992.

[140] Thomas Young. Optimizing Points-of-Visibility Pathfinding. In *Game Programming Gems 2*, pages 324–329. 2001.

[141] Ryan Hechenberger, Peter J. Stuckey, Daniel Harabor, Pierre Le Bodic, and Muhammad Aamir Cheema. Online computation of euclidean shortest paths in two dimensions.

In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 134–142. AAAI Press, 2020.

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

[143] Robert J. Fowler, Mike Paterson, and Steven L. Tanimoto. Optimal packing and covering in the plane are np-complete. *Information Processing Letters*, 12(3):133–137, 1981.

[144] Nathan R. Sturtevant, Jason M. Traish, James R. Tulip, Tansel Uras, Sven Koenig, Ben Strasser, Adi Botea, Daniel Harabor, and Steve Rabin. The grid-based path planning competition: 2014 entries and results. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, pages 241–251. AAAI Press, 2015.

[145] Robert C. Holte, M. B. Perez, Robert M. Zimmer, and Alan J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 530–535. AAAI Press, 1996.

[146] Daniel Damir Harabor and Peter J. Stuckey. Forward search in contraction hierarchies. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*, pages 55–62. AAAI Press, 2018.

[147] Lingkun Wu, Xiaokui Xiao, Dingxiong Deng, Gao Cong, Andy Diwen Zhu, and Shuigeng Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *Proceedings of the VLDB Endowment*, 5(5):406–417, 2012.

[148] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Dynamic time-dependent route planning in road networks with user preferences. In *Proceeding of the International Symposium on Experimental Algorithms (SEA)*, volume 9685 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2016.

[149] Spyros C. Kontogiannis, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, and Christos D. Zaroliagis. Improved oracles for time-dependent road networks. In *Proceedings of the Symposium on Algorithmic Approaches for Transportation Modeling, Optimization and Systems (ATMOS)*, pages 4:1–4:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[150] Omri Kaduri, Eli Boyarski, and Roni Stern. Experimental evaluation of classical multi agent path finding algorithms. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 126–130. AAAI Press, 2021.

[151] Jingyao Ren, Vikraman Sathiyanarayanan, Eric Ewing, Baskin Senbaslar, and Nora Ayanian. MAPFAST: A deep algorithm selector for multi agent path finding using shortest path embeddings. In *Proceedings of the International Joint Conference on Autonomous*

Agents and Multi-Agent Systems (AAMAS), pages 1055–1063. International Foundation for Autonomous Agents and Multiagent Systems, 2021.

[152] Eric Ewing, Jingyao Ren, Dhvani Kansara, Vikraman Sathiyanarayanan, and Nora Ayanian. Betweenness centrality in multi-agent path finding. In *Proceedings of the International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 400–408. International Foundation for Autonomous Agents and Multiagent Systems, 2022.

[153] Anton Andreychuk, Konstantin S. Yakovlev, Pavel Surynek, Dor Atzmon, and Roni Stern. Multi-agent pathfinding with continuous time. *Artificial Intelligence*, 305:103662, 2022.

[154] Wheeler Ruml. The logic of benchmarking: A case against state-of-the-art performance. In *Proceedings of the Symposium on Combinatorial Search (SoCS)*. AAAI Press, 2010.

[155] Thomas K. Nobes, Daniel Harabor, Michael Wybrow, and Stuart D. C. Walsh. The JPS pathfinding system in 3D. In *Proceedings of the International Symposium on Combinatorial Search (SoCS)*, pages 145–152. AAAI Press, 2022.

[156] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11(1-2):1–21, 1993.