MONASH University

Improving Pruning and Compression Techniques in Path Planning

Shizhe Zhao

Doctor of Philosophy (0190)

A Thesis Submitted for the Degree of Doctor of Philosophy at
Monash University in 2022
DEPARTMENT OF DATA SCIENCE & AI

Main supervisor:
Dr. Daniel D. Harabor
Associate supervisor:
Prof. Peter J. Stuckey

Copyright notice

I certify that I have made all reasonable efforts to secure copyright permissions for third-party content included in this thesis and have not knowingly added copyright content to my work without the owner's permission.

Abstract Path planning is a well-studied problem in AI. In many scenarios, such as computer games and trip planning, the environments are two dimensional Euclidean planes with obstacles. In games, the most common way to model these environments is a rasterized grid map as it is easy to generate, update and detect conflicts. In trip planning, traversable regions are restricted to road networks, these environments are modeled as graphs where each edge is a segment of a road, and the weight of each edge is time, distance or some other metrics. In many scenarios, computing high quality paths (optimal or near optimal) is an important and fundamental task and its performance is critical. In this research we focus on improving two state-of-the-art optimal methods, the purely online method *Jump Point Search* (JPS), and an offline method *Compressed Path Databases* (CPDs).

For *JPS*, we study three pathological behaviours that are not well studied by existing literature: suboptimal node generation, suboptimal node expansion, and redundant scanning. We explain how these happen, how they affect the performance and how to efficiently resolve them by a new method called *Constrained JPS*. The experiments show that, in pathological scenarios, the speed up factor can be up to 4.87 and it significantly reduces suboptimal node expansion, while the overhead is only 25% when there is no chance to prune.

For *CPD*s, we introduce a novel encoding method to reduce storage by orders of magnitude, and a bounded suboptimal method to trade-off the storage and preprocessing time with optimality. Compared to vanilla *CPD* which might be too large to fit in memory for large maps, these two methods extend its application scenarios. In addition, we generalize the key idea of *CPD* and propose a distributed oracle when the first move data cannot be well compressed. This method can be applied on road network navigation, comparing with other competitors, it supports anytime queries which returns a prefix without finding the entire path; and under highly dynamic environments, it has smaller amortized query time as it doesn't need repair.

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: _____

Print Name: Shizhe Zhao

Date: February 2023

Publications during enrolment

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 J. Benton, Nir Lipovetzky, Eva Onaindia, David E. Smith, and Siddharth Srivastava, editors, *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*, pages 106–113. AAAI Press, 2019. URL https://ojs.aaai.org/index.php/ICAPS/article/view/3465

Shizhe Zhao, Mattia Chiari, Adi Botea, Alfonso Emilio Gerevini, Daniel Harabor, Alessandro Saetti, and Peter J. Stuckey. Bounded suboptimal path planning with compressed path databases. In J. Christopher Beck, Olivier Buffet, Jörg Hoffmann, Erez Karpas, and Shirin Sohrabi, editors, *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, pages 333–342. AAAI Press, 2020. URL https://ojs.aaai.org/index.php/ICAPS/article/view/6678

Arthur Mahéo, Shizhe Zhao, Hassan Afzaal, Daniel Harabor, Peter J. Stuckey, and Mark Wallace. Customised shortest paths using a distributed reverse oracle. In Hang Ma and Ivan Serina, editors, *Proceedings of the Fourteenth International Symposium on Combinatorial Search, SOCS 2021, Virtual Conference [Jinan, China], July 26-30, 2021*, pages 79–87. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/SOCS/article/view/18554

Acknowledgements

First and foremost, I would like to thank my parents for their unconditional support.

I am extremely grateful to my supervisors, Dr. Daniel Harabor and Prof. Peter Stuckey for their invaluable advice, continuous support, and patience during my PhD study. Their immense knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life.

I would also like to thank Prof. Guido Tack and Prof. David Taniar for their support during my honors degree. Prof. Guido Tack was the supervisor of my first research unit in the Masters, and I feel extremely lucky to have you bring me to the optimisation group. Prof. David Taniar was my main supervisor for the honors thesis, who helped me with my first step of research.

I would like to offer my special thanks to Goldi and Adam for their kind help that have made me successfully submit this thesis at the last minute.

I would like to thank all the members in the optimisation group. It is their kind help and support that have made my study and life in the Monash a wonderful time.

Contents

Copyright notice 1

Abstract 2

Declaration 3

Publications during enrolment 4

Acknowledgements 5

List of Figures 9

List of Tables 12

1 Introduction 1
Games . . . 1
Trip planning . . . 2
1.1 Challenges . . . 3
1.2 Contribution . . . 3
1.3 Structure of the Thesis . . . 4

2 Literature Review 6
2.1 Problem statement . . . 7
Gridmap. . . . 7
Weighted graph. . . . 7
2.2 Search Framework . . . 8
Input graph . . . 8
Expansion policy . . . 8
Heuristic function . . . 9
2.3 Related works . . . 9
2.3.1 Heuristic . . . 10
Discussion . . . 11
2.3.2 Goal bounding . . . 11
Lossless Compression . . . 12
Lossy Compression . . . 12
Discussion . . . 13
2.3.3 Abstraction . . . 13
Lossless abstraction . . . 13
Lossy abstraction . . . 14

Discussion . . . 14
2.3.4 Symmetry breaking . . . 14
Reachability relation . . . 15
Canonical ordering . . . 15
Random noises . . . 15
2.4 Background . . . 15
2.4.1 Jump Point Search . . . 15
Concepts . . . 16
Block-based scanning. . . . 17
Discussion . . . 18
2.4.2 Compressed Path Databases . . . 18
Heuristic Ordering . . . 19
Bidirectional Wildcards . . . 20
Path Extraction . . . 21
CPD Search . . . 21
Discussion . . . 21

3 Reducing Redundant Work in Jump Point Search 22
3.1 Pathological Behaviours in JPS . . . 23
3.2 Constrained JPS . . . 24
Observation . . . 24
Computing L . . . 25
Updating and Deleting the Constraint . . . 25
Pruning and Early Termination . . . 26
3.3 Branch-less Implementation . . . 26
3.4 Eliminating Suboptimal Node Expansion . . . 27
3.4.1 How does it happen? . . . 28
3.4.2 How bad it could be? . . . 28
3.4.3 Can we eliminate all of them? . . . 29
3.5 Experiments . . . 30
3.5.1 Exp-1: Synthetic Maps . . . 32
3.5.2 Exp-2: Ablation Study . . . 33
3.5.3 Exp-3: Domain Maps . . . 34
Simulate dynamic environments. . . . 34
How to read these plots. . . . 35
3.6 Conclusion and Future Work . . . 36

4 Improving Forward CPD 37
4.1 Heuristic Redundant Symbols . . . 38
4.1.1 Distance Functions . . . 39
4.1.2 Improving the Tie-breaking . . . 41
4.2 Proximity Wildcards . . . 42
4.3 Experiment . . . 44
4.3.1 Preprocessing Results . . . 46
4.3.2 Path Extraction Results . . . 49
4.4 Conclusion . . . 49

5 Bounded Suboptimal Path Planning with CPD **50**
5.1 Suboptimal CPDs . . . 51
Building Centroids . . . 52
5.1.1 Reverse Compressed Path Databases . . . 53
"Illegal" Moves . . . 55
Preprocessing and Path Extraction . . . 56
Heuristic Path Extraction . . . 57
5.2 Experiments on Suboptimal CPDs . . . 57
5.2.1 Experiment 1: Preprocessing Time . . . 58
5.2.2 Experiment 2: CPD Size . . . 59
5.2.3 Experiment 3: Path Extraction . . . 60
5.2.4 Experiment 4: Suboptimality and Path Quality . . . 62
5.3 Discussion . . . 63
5.3.1 Other similar methods . . . 63
5.3.2 NP-hardness of optimal centroid placement . . . 63
5.4 Conclusion . . . 64

6 Reverse Oracle for weight-changing road networks **65**
6.1 Problem Statement . . . 67
6.2 Oracle Search . . . 68
6.2.1 Choosing an Oracle . . . 71
6.3 Distributed Oracle Search . . . 72
6.4 Simulating Melbourne . . . 73
6.5 Experimental Setup . . . 74
6.6 Results . . . 75
6.6.1 Experiment #1: Any Route At All . . . 75
6.6.2 Experiment #2: Bounded Suboptimal Search . . . 76
6.6.3 Experiment #3: Budgeted Search . . . 78
6.6.4 Experiment #4: COST . . . 79
6.7 Discussion . . . 80
6.8 Revisiting CCH . . . 81
6.9 Conclusion . . . 82

7 Conclusion and Future works **83**
7.1 Future work . . . 84
Constrained JPS on Higher Dimensional Problems . . . 84
General Pruning Rule Generation . . . 84
First-move Oracle for Time-dependent Routing . . . 84

Bibliography **86**

List of Figures

1.1 Two path planning applications: (a) shows the gameplay of *Starcraft*, (b) shows a trip plan provided by *Google Map*. 2

2.1 Three common heuristics that estimate distance from s to t, where the red dashed line indicates *Octile distance* on 8-connected gridmaps, the blue solid line indicates *Euclidean distance* on 2D Euclidean plane, the green dotted line indicates *Manhattan distance* on 4-connected gridmaps. The grey region may contain untraversable area. 10

2.2 An example of using Goal bounding to prune successors. The current search node s is coming from W, and it has successors in three directions: S, SE and E. The database stores bounding boxes of optimal reachable nodes for each direction, the colour and style indicates the corresponding direction (solid blue for S, dashed red for SE, dotted green for E). Successors in SE and E can be safely pruned as their bounding boxes do not contain the target t. 11

2.3 An example of Abstraction on toy graph G. G is undirected (for simplicity), the original edges of G are solid lines. To build the abstracted graph G', we remove the vertex A, and add edges (dashed lines) to G' to connect all neighbours of A, these edges represent optimal paths in the G. Then, finding the shortest path between C, B and D on the G' requires less search effort. 13

2.4 An example of symmetries on gridmaps. They are three equivalent paths from s to t: $[s, v_1, v_2, t]$, $[s, v_1, v_4, t]$, $[s, v_3, v_4, t]$, ... 14

2.5 In all figures, x is the current node, the arrow represents the incoming direction, black cells represent non-traversable cells, white cells are diagonal-first neighbours and gray cells are pruned. 17

2.6 Node a is the current search node, the black arrow represents the coming direction, blue arrows represent scanning, $a_{i \in 1 \ldots 5}$ are start points of each cardinal scan in diagonal. x_1, x_2 are jump points from the scanning direction, grey cells are corner points, the cell above a_1 is a dead-end. 17

2.7 (a) Optimal moves from the source cell s (the yellow cell) to each traversable cell t. Observe that more than one optimal move can exist towards some targets (e.g., E, SE for the bottom-right cell). (b) DFS ordering. 20

3.1 a and b are search nodes. Blue arrows shows nodes that are scanned during the diagonal recursion of a, and b, x_1, x_2 are identified successors (jump points). JPS would have the same produce on b, and green nodes are scanned during the diagonal recursion of b. 24

3.2 (a) shows constraint (a, v, d, L), where a is a jump point coming from north, or a continued diagonal move from northwest; v is a previously visited node that found from the cardinal scanning from a, and L is the maximum number of steps where the constraint is applicable, yellow cells are better reached from v, thus cardinal scanning from a_1, a_2, a_3 is stopped by the constraint at the yellow blockage; (b) shows how to evaluate g_p, where p is the stop location of a scanning before the blockage, thus the traversability of all nodes after p is unknown, noted by '?'. 26
3.3 s (yellow) is the source node, a (cyan) and b (blue) are successors of s. a_2, a_3, a_4 are suboptimal nodes propagated from a. 29
3.4 Synthetic maps, where $s = 512$, $b = 75\%$ and $r \in \{0.1, 1, 10, 20\}$. Blue and red clusters are starts and targets. 31
3.5 Speed-up on cumulative time ($\frac{\sum Time_{jps}}{\sum Time_{cjps}}$) and queries($\frac{Time_{jps}}{Time_{cjps}}$). The red line is 1, and values above it indicate that CJPS is faster. 35

4.1 (a) shows the optimal first-moves to each cell from s . (b) shows the heuristic move to each cell from s. Bold if it appears in $T_s(t)$ for the cell marked s. 39
4.2 The heuristic first move $F_o(s, t)$ to each cell of the gridmap from the cell marked s. All symbols are in bold, since each of them belongs to the corresponding set $T_s(t)$ for the marked symbol s. 40
4.3 (a) The heuristic first move $F_o(s, t)$ to each cell of the gridmap from the cell marked s and (b) the optimal first moves to each cell from s, with the heuristic moves in bold. 41
4.4 The heuristic first move $F_o^d(s, t)$ to each cell of the gridmap from the cell marked s using directional tie breaking. 42
4.5 A small gridmap to illustrate proximity wildcards. 43
4.6 Heuristic moves, for each source cell. Source cells are shown in yellow. In each case, heuristic moves that coincide with optimal moves are shown in bold. 43
4.7 Distribution of the compression factor in the maps from DAO. The compression factor is the CPD size produced by *SRC* enhanced with the setting at hand (h, w, hw) divided by the CPD size produced by *SRC*. Maps are ordered by the *SRC CPD size*. 46
4.8 Distribution of the preprocessing-time factor over all maps. The preprocessing-time factor is the preprocessing time cost of the proposed algorithm at hand (h, w, hw) divided by the preprocessing time cost of *SRC*. Maps are ordered by *SRC size*. 48

5.1 Centroid marking for a grid map with $\delta = 3$. **A**, **B**, **C** and **D** are centroids in the first stage, regions explored from them are filled with green dots (A), blue vertical lines (B), red bricks (C) and yellow starts (D). Centroids **E**–**J** are created in the second stage, and their corresponding regions are shown with lowercase letters. 52
5.2 Optimal first moves from each grid cell to the cell marked t. Moves in bold agree with the default heuristic. 55
5.3 Optimal first moves from each grid cell to the cell marked t. Moves in bold agree with the default heuristic. 55

5.4 Optimal first moves toward the grid cell t with added "illegal" move symbols for cells adjacent to obstacles. 55
5.5 Worst case behaviour for centroid CPDs when $\delta = 6$: the symbol stored for FirstMove(s,$c(t)$) is E rather than the equivalent W. The path found from s to t is length 14 rather than 2. 57
5.6 The number of maps with smaller CPD when δ increase. 59

6.1 We compare Oracle Search and CCH in a simulated setup of Melbourne (Australia). The metric periodically changes and after every change CCH *customises* (i. e., repairs) its auxiliary data. No matter the size of the changeset (1 edge, 100 edges or even all), CCH performance quickly degrades as the number of customisations grows: from one per queryset (651K total) to one per query. After 1715 customisations, CCH becomes slower than Oracle Search and eventually slower than Dijkstra search. This experiment shows the main advantage of CCH: being able to amortise the cost of customisation over many subsequent queries. 66
6.2 We compare Oracle Search and CCH in a simulated setup of Melbourne (Australia) where some percentage of all queries (651K total) requires a customised metric w_i, while all remaining queries are solved by a default metric w_0, such as network distance or freeflow travel time. CCH performance is identical to Figure 6.1 but Oracle Search is substantially faster. This experiment shows the main advantages of Oracle Search: fast query performance, which comes from exploiting precomputed w_0 paths, and no additional repair work, which CCH is forced to undertake when the metric changes. 66
6.3 Schematic view of our cluster 72
6.4 Throughput per worker when computing k-length prefixes. We do not perform search, we only extract the k first move using the CPD. The numbers below, in black, represent the mean throughput recorded across partitions. 76
6.5 We measure the quality of the unbounded suboptimal path found by blindly following the oracle vs. optimal. We plot the mean suboptimality for each percentile, and, for every tick mark, the maximum in black. . . . 76
6.6 Speedup of our system vs. single-threaded search for different workloads. We measure on the driver, from sending out the queries to receiving the response from all workers. 77
6.7 Throughput per worker when running a bounded suboptimal search with different quality guarantees – 1.0 means an optimal search. The numbers above, in black, represent the mean throughput recorded across partitions. 77
6.8 Percentage difference between the optimal path and suboptimal paths found for different admissibility percentage (% sub-optimal). 78
6.9 Throughput per worker when running a time-bounded search for different time budgets. The numbers above, in black, represent the mean throughput recorded across partitions. 78
6.10 Percentage difference between the optimal path and suboptimal paths found for different time limits (µs). 79
6.11 Throughput measured on the driver. The blue marks and numbers represent throughput on an *ideal system* – one with no communication overheads. 80

List of Tables

3.1 Statistic of each domain. We count node expansion from 100 longest queries of each map. *mean, median* are for queries, which indicates the difficulty, *sub%* shows the proportion of suboptimal node expansion, and *subp%* shows the proportion of propagated suboptimal node expansion among *sub*. 29

3.2 Improvement factors of various metrics ($\frac{Metric(JPS)}{Metric(CJPS)}$) on three settings, >1 means improvements, where: (1)*opt* measures heap operations (#expansion + #insertion); (2)*subopt* measures *opt* on suboptimal nodes; (3)*TPE* (time cost per expansion) measures average cost per node expansion in μs ($\frac{time}{\#expd}$), we also reveal the raw TPE of JPS and CJPS in parenthesis; (4)*runtime* measures average runtime per query ($\frac{time}{\#queries}$). . 32

3.3 Results on synthetic maps. We look at suboptimal expansions and run time. 34

3.4 Cumulative time per domain on static and dynamic environments. 34

4.1 Proximity distances for the nodes in the gridmap in Figure 4.5. 43

4.2 First move table with no proximity wildcards, for the example shown in Figure 4.5. In each cell, we show all the optimal moves, and the ⓗ symbol, if the heuristic move coincides with an optimal move. 44

4.3 First move table with proximity wildcards. 44

4.4 Size of CPDs of all maps from DAO (MB). 45

4.5 CPD size statistics for the large maps used in experiments. Top: the number of cells per map, and the CPD size in MB. Bottom: the number of RLE runs per map in thousands. 47

4.6 Speed up factor when extracting a full path measured as the time cost of *SRC* divided by the time cost of the proposed algorithms. 48

4.7 Number of operations in path extraction. Here l represents the number of binary searches, and c represents the number of times the target is inside the source's proximity square. 48

5.1 Number of (C)entroids, building (T)ime in minutes, and (M)emory requirements in MB for (F)orward and improved (R)everse CPDs for different radii of centroids on five representative maps. Note that for $\delta = 0$ the number of centroids is the number of cells in the map. 58

5.2 CPD size reduction between centroid CPDs and $\texttt{fwd}_0$, $\texttt{ratio} = \frac{|\texttt{fwd}_0|}{|\texttt{rev}_\delta|}$ or $\frac{|\texttt{fwd}_0|}{|\texttt{fwd}_\delta|}$. 60

5.3 Speedup, path extraction time divided by path extraction time using $\texttt{fwd}_0$. A ratio greater than 1.0 means faster. 61

5.4 62

6.1 Oracle Search performance with different path oracles on the benchmark from Figure 6.1. *Speedup* is the mean increase in query processing time. Column *Static* indicates free-flow travel times (metric w_0) and *Dynamic* indicates the congestion costs (w). In the static case the path oracle is sufficient to solve each problem optimally; in the dynamic case we perform search. 71

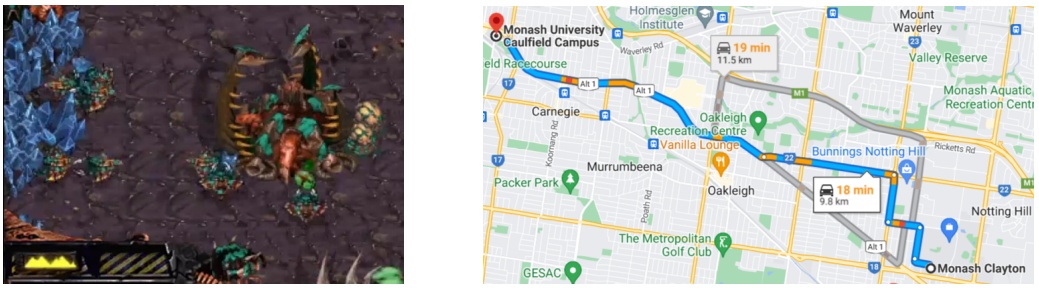
Chapter 1

Introduction

Path planning is a well-studied problem in AI. In many scenarios, such as computer games and trip planning, the environment is a two dimensional Euclidean plane with obstacles. In games, grid maps are commonly used to model these environments. The reason is that, the grid map is easy to generate and update, e.g., adding or removing objects on grid maps is simply modifying a few bits in memory. And forcing agents to move on the grid makes the conflict detection easy. In trip planning, traversable regions are restricted to road networks, and conflict avoidance is handled by other mechanisms, e.g., traffic rules. Thus, these environments are modelled as graphs with edges representing road segments, edge weights representing metrics like time or distance. Path plannings in these scenarios are challenging. Figure 1.1 shows the application of each scenario. Two concrete examples are given below.

Games Starcraft is a classic RTS (real-time strategy) game where players collect resources to build an army and engage in battles against other players or computers. Figure 1.1a[1] shows a screenshot of busy workers. In the game, workers gather minerals (blue) and gases (green) from resource points, bring them back to the base and repeat. A mineral is an obstacle but depletes after a certain number of collections, making its location traversable. This setting generates a large number of path planning tasks for workers to avoid collisions with other units in a dynamic environment. There are other types of units, e.g., soldiers, that also need to perform path planning repeatedly, and the total number of them can be up to hundreds. However, the computation resources for the path planning are very limited [4], the game engine also has other high priority tasks, e.g., rendering, decision-making and networking, that depends on the result of path planning.

[1] Download from https://starcraft.com



(a) (b)

FIGURE 1.1: Two path planning applications: (a) shows the gameplay of *Starcraft*, (b) shows a trip plan provided by *Google Map*.

In addition, Starcraft has a special mode called *Aeon of Strife*. It is a MOBA (Multiplayer Online Battle Arena) game. Players control hero units and fight against a stronger computer. The environment in *Aeon* is static, disallowing players from constructing facilities or gathering minerals. However, the movement of all units must be synchronised across the network, which significantly restricts the time available for path planning.

Trip planning Trip planning plays an important role in daily life as it not only provides users with an optimal or near optimal trip plan under the present traffic conditions, but also presents alternative options. Figure 1.1b shows a trip plan from Monash Clayton campus to Caulfield campus, provided by *Google Maps*.[2] The highlighted route is the recommended plan and the colours indicate traffic conditions, where blue, yellow, red indicates normal, light jam, heavy jam respectively. The gray routes are alternative options.

From the user's point of view, a trip planner should be both customizable and responsive. This means that the user should able to set their preferred metric (e.g., distance or time) and specify any preference, e.g., avoiding tolls, and receive immediate first instructions and real-time updates for rerouting with minimal delay, i.e., have low *first move lag*.

From the perspective of the service provider, generating a plan involves several path plannings. When the planning is done by a centralized server in the cloud, it must be able to handle hundreds of thousands of user queries simultaneously, making the performance of the path planning crucial. On the other hand, when the planning is done offline, e.g., on the mobile device, the auxiliary data for query processing must fit in memory and should be as small as possible.

In the above applications, we need to compute a high quality plan for a positive user experience by finding an optimal or near-optimal path, e.g., *Starcraft* players want a

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

game agent to move along the shortest path without manual control, while *Google Maps* users prefer a trip with minimum time cost.

1.1 Challenges

From the examples above, we can see that computing high quality paths (whether optimal or near optimal) is an critical and fundamental task in practical planning systems and its performance is important. We summarize technical challenges below:

- **Challenge #1 Responsive:** since a path planning task can be generated while the agent is in motion, the first action of the plan must be returned before the agent takes the next action or the plan becomes invalid. Therefore, the first-move lag should be minimized.
- **Challenge #2 Changing environments:** the traversability of cells in game maps and the weight of edges in road networks change from time to time. The path planner must be able to adapt to these changing environments.
- **Challenge #3 Pruning:** proving optimality of the path often requires exploring a large proportion of map. To achieve high-performance planning, it is crucial to implement effective pruning to reduce redundant work.
- **Challenge #4 Heuristic:** To guide the search process, we use heuristics to estimate the lower bound of a potential path. A more accurate estimation requires more additional data, which may not always fit in memory. As a result, the heuristic function must be able to balance the precision and memory cost effectively.

1.2 Contribution

Path planning has achieved significant improvements in recent years, and it remains an active research area, the results of state-of-the-art methods can be found in the Grid-based Path Planning Competition (GPPC) [5]. Roughly speaking, these methods can be classified as combinations of online/offline and optimal/suboptimal; in this research we will focus on improving two optimal methods, the purely online method *Jump Point Search* (JPS), and an offline method *Compressed Path Databases* (CPD).

JPS was proposed over a decade ago [6], and there are many variants [7–12] appearing in the literature. However, there are still two open issues with JPS. The first is an issue has not been documented in the literature – JPS may add suboptimal search nodes

to the open list, and the suboptimality might be propagated which can lead to more suboptimal nodes in the search space. The second issue is not properly resolved by existing literature [11, 12] – JPS has an unlimited scanning process that may explore entire columns/rows when the map is open.

In this research, we first propose an online method with stronger pruning to solve the above issues and achieve better performance, which answers the **challenges #1** and **#3**. Then we show the inner connection between these issues which reveals a potential chance of applying JPS on harder problems.

Compressed Path Databases (CPD) is an ultra fast method for path planning that is originally designed to answer **challenge #1**. The major issue is the size of precomputed index may exceed the memory limit, which reduces the ability to apply it on certain maps, and existing literature is only able to reduce the size of CPD by a few factors [13–15]. In addition, it can also be used as a heuristic on dynamic gridmaps [16],

In this research, we propose: (i) a novel encoding method to improve the compression, which answers the **challenge #4**; (ii) a way to store the auxiliary data, it allows the trade-off between memory cost and optimality and also improves the path extraction, which answers the **challenges #1** and **#4**. (iii) we apply the idea of CPD on road networks with dynamic traffic and propose the *oracle search* framework. The framework supports any-time queries which reduces the first move lag, and it outperforms traditional methods in this domain that requires repairing to handle the dynamic environment, which answers the **challenges #1, #2** and **#4**.

1.3 Structure of the Thesis

The rest of this thesis is organized as follows: in the following section, we will give a formal definition of 2D pathfinding on gridmaps and weighted graph; in Chapter 2, we first give a taxonomy study of related approaches, then we give details of JPS and CPD; in Chapter 3, we show some pathological behaviors of JPS and propose a way to resolve these issues efficiently; in Chapter 4, we introduce a method that significantly improves the compression of CPD; in Chapter 5, we demonstrate a way to trade-off between suboptimality and memory cost in a CPD, and show a new way to store information, called a *reverse CPD*, to get more benefit from the proposed method; in Chapter 6, we generalize the *reverse CPD* and apply it on a different domain than the 2D grid map; in Chapter 7, we discuss the conclusion and future works.

Publications

Publications arising from this thesis are listed as follows.

1. The material in Chapter 3 has been submitted to SoCS-2023.

2. The materials in Chapter 4 and 5 have been published in *International Conference on Automated Planning and Scheduling*

   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 J. Benton, Nir Lipovetzky, Eva Onaindia, David E. Smith, and Siddharth Srivastava, editors, *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*, pages 106–113. AAAI Press, 2019. URL https://ojs.aaai.org/index.php/ICAPS/article/view/3465

   Shizhe Zhao, Mattia Chiari, Adi Botea, Alfonso Emilio Gerevini, Daniel Harabor, Alessandro Saetti, and Peter J. Stuckey. Bounded suboptimal path planning with compressed path databases. In J. Christopher Beck, Olivier Buffet, Jörg Hoffmann, Erez Karpas, and Shirin Sohrabi, editors, *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, pages 333–342. AAAI Press, 2020. URL https://ojs.aaai.org/index.php/ICAPS/article/view/6678

3. The material in Chapter 6 has been published in *the Symposium on Combinatorial Search*

   Arthur Mahéo, Shizhe Zhao, Hassan Afzaal, Daniel Harabor, Peter J. Stuckey, and Mark Wallace. Customised shortest paths using a distributed reverse oracle. In Hang Ma and Ivan Serina, editors, *Proceedings of the Fourteenth International Symposium on Combinatorial Search, SOCS 2021, Virtual Conference [Jinan, China], July 26-30, 2021*, pages 79–87. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/SOCS/article/view/18554

Chapter 2

Literature Review

2D pathfinding is an active research topic that appears in robotics and video games. There are many undominated methods appearing in recent literature. Typically, the environment model of these methods can be classified as static and dynamic.

In practice, the environment is usually dynamic; e.g. in the warehouse, robots may drop or take a parcel, and in a game, players may build or destroy objects on the map; these map changing events are called perturbations. Under certain perturbation models, preprocessing-based methods may also work. When the perturbation is determined and rarely happens, we can just use different version of maps; when it changes more often, some methods that have light preprocessing or support fast repairing still work; while with the increasing of frequency and the size of affected region, preprocessing methods gradually become not applicable, and we have to consider purely online methods.

In some problems, the change of map is very frequent that associates with the time stamp during the processing. E.g., in multi-agent planning, we need to consider the traversability of cells that are affected by the paths of other agents at specific times, this is called a temporal obstacle problem [17, 18]; and in real-time traffic routing on road networks, the weight of graph edges are affected by real time traffic flow, this is called time-dependent routing [19–21]. For these problems, we have to add the time dimension to the model, and they become 3D path planning. In this research, we focus on solving 2D path planning problems.

In this chapter, we give the formal definition of the research problem in Section 2.1; then we show an overview of the search framework in Section 2.2; after that, we explore some representative methods of this domain in Section 2.3; finally, in Section 2.4, we take a close look on *JPS* and *CPD*, which are important methods for this thesis.

2.1 Problem statement

Gridmap. The gridmap is a two-dimensional data structure that represents the operating environment for a mobile agent, such as a robot or a game character. Gridmaps rasterize the environment into square cells, with each cell being either traversable or blocked. Each cell has up to 8 neighbours: one in each of the four cardinal (equiv. straight) directions and one in each of the four ordinal (equiv. diagonal) directions.

When moving, each agent occupies exactly one traversable cell at a time and each is allowed to step to any other adjacent traversable cell. Straight moves have a cost of 1, while diagonal moves cost $\sqrt{2}$. As in the GPPC [5], we enforce the *no-corner-cutting rule*, which says that diagonal moves are disallowed if the origin and destination cell share a common neighbour which is not traversable.

Given a grid cell n, we write $n.x$ and $n.y$ for the x (column) and y (row) coordinates of that cell. Further, let $MV = \{\text{N, NE, E, SE, S, SW, W, NW}\}$ be the set of 8 principal compass directions in which the agent can move. Given a move $m \in MV$ and cell n we define $m(n)$ as the cell n' where $n'.x = n.x + horiz(m)$ and $n'.y = n.y + vert(m)$ where

$$horiz(m) = \begin{cases} -1 & m \in \{\text{SW, W, NW}\} \\ 0 & m \in \{\text{N, S}\} \\ +1 & m \in \{\text{NE, E, SE}\} \end{cases}$$

$$vert(m) = \begin{cases} -1 & m \in \{\text{NW, N, NE}\} \\ 0 & m \in \{\text{E, W}\} \\ +1 & m \in \{\text{SE, S, SW}\} \end{cases}$$

Each gridmap induces an undirected weighted graph where traversable cells are vertices, and the moves applicable in each traversable cell are edges.

Weighted graph. Consider then a weighted graph $G = (V, E, w)$ with vertices V and edges $E \subseteq V \times V$, and a function w such that $w(u, v)$ is the cost of edge $(u, v) \in E$, it is also called the *graph metric*. In addition to the graph and the metric we are also given pairs of vertices $(s, t) \in V$ (respectively the *start* and *target*). Each pair is called an *instance* and a valid solution to an instance is any path π from s to t, that is a sequence of vertices $[n_0, n_1, n_2, \ldots, n_{k-1}, n_k]$, where $k \in \mathbb{N}^+$, $n_0 = s$, $n_k = t$, and $(n_i, n_{i+1}) \in E, 0 \leq i < k$. The *length* of the path is $len(\pi) = \sum_{i=0}^{k-1} w(n_i, n_{i+1})$. The

reverse $rev(\pi)$ of the path p is the reverse sequence of its vertices. Let $sp(s,t)$ return the shortest length path in G from s to t, and let symbol $+\!\!+$ denote sequence (and thus also path) concatenation.

2.2 Search Framework

The objective of the optimal path planning is to find the path with minimum length. Most of the state-of-the-art methods for this problem are based on the A* (equiv. Best-first search) framework. In this section, we demonstrate important components of the framework, and in Section 2.3, we discuss how existing methods work on them to improve the search.

Overall, the algorithm starts with an open list *open* that contains only the source node, then it keeps generating successors to *open* and expanding search nodes in *open*, until the target is found or *open* is empty. The open list for A* is a priority queue that maintains search nodes and their priority. The priority of each search node is called its *f-value*, it is the sum of *g-value* and *h-value*, where *g-value* stores the distance from the start and *h-value* estimates the distance to the target by the heuristic function. Algorithm 1 shows details.

The major cost of the search comes from the operations on the open list, including adding a search node (Algorithm 1 line 13), retrieving the best one (Algorithm 1 line 6) and removing a search node (Algorithm 1 line 7). Enhancements to the open list are fundamental. They are either implementation techniques [22], e.g., memory preallocation, lazy initialisation, or data structures, e.g. *Binary Heaps* [23], *Fibonacci Heaps* [24], which fall outside the scope of our research. The rest parts of A* can be decomposed to the following components.

Input graph Input graph $G = (V, E, w)$ defines the size and structure of the search space. Intuitively, the bigger the $|V|$ and $|E|$, the larger the search space and the harder the problem, but the structure of G also plays an important role. E.g., when height and width of gridmaps are fixed, an obstacle-free map has more vertices and edges than a map with obstacles, but the path planning on the former is much easier. Thus, we can improve the search by exploiting the underlying structure of the input graph.

Expansion policy The expansion policy defines the successors of each search node that needs to be added to the open list. Lines from 9 to 13 in Algorithm 1 demonstrate this procedure. The line 10 shows a simple pruning that expands a search node only if it

Algorithm 1: A$^*(s, t, G)$ computes the optimal distance from s to t for an input graph $G = (V, E, w)$.

Variables:
open: the priority queue;
f: table for the f-value of all vertices;
g: table for the g-value of all vertices;

```
1  for n ∈ V do g[n] ← ∞;
2  open ← {s};
3  g[s] ← 0;
4  f[s] ← 0;
5  while open ≠ ∅ do
6      n ← argmin{f[n'] | n' ∈ open}
7      open ← open − {n};
8      if n == t then return g[n];
9      for (n, m) ∈ E do
10         if g[n] + w(n, m) < g[m] then
11             g[m] ← g[n] + w(n, m);
12             f[m] ← g[m] + Heuristic(m, t);
13             open ← open ∪ {m};
14 return ⊥                                   ▶ No solution
```

yields a smaller g-value. We can reduce the branching-factor in the search by applying some stronger pruning rules in the expansion policy.

Heuristic function Heuristic function computes h-value which estimates the distance from the current search node to the target. Algorithm 1 line 12 is a placeholder for the heuristic function as a variety of choices exist depend on the input graph. In an optimal search, the *h-value* should be *admissible*, i.e. it is always a lower bound, so that we can confirm the first search node that reaches the target has the shortest path. A more accurate estimation gives higher priority to more promising search nodes which may reduce the node expansion, thus a stronger heuristic function provides a better guidance for the search.

2.3 Related works

In practice, we want to compute an optimal/near-optimal path to build a high quality plan. This section is a taxonomical study on relevant literature. Overall, there are four categories: *Heuristic, Goal bounding, Abstraction* and *Symmetry breaking*, each of them improves one of the components in the A* framework. In the following sections, we present the most representative methods of these categories and discuss the trade-offs between optimality, space, and time costs.

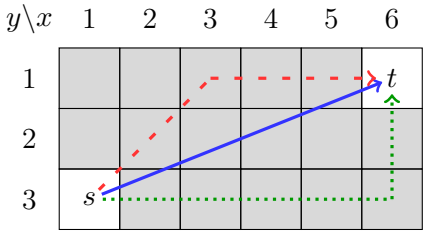


FIGURE 2.1: Three common heuristics that estimate distance from s to t, where the red dashed line indicates *Octile distance* on 8-connected gridmaps, the blue solid line indicates *Euclidean distance* on 2D Euclidean plane, the green dotted line indicates *Manhattan distance* on 4-connected gridmaps. The grey region may contain untraversable area.

2.3.1 Heuristic

Heuristics guide the search, the most common heuristics, e.g., Manhattan distance (for 4-connected grids), Octile distance (for 8-connected grids) and Euclidean distance (for Euclidean planes), use a closed-form expression to estimate the lower bound on-the-fly, Figure 2.1 illustrates how to compute them. Both of these estimate a very loose lower-bound that ignore all traversability constraints. To get a more accurate estimation, the heuristic function needs extra information based on preprocessing. Two famous methods in this family are the *Differential Heuristic* and *Fastmap*.

Differential heuristic (DH) [25] utilises the triangle inequality that selects k vertices from the graph, called landmarks L, and precomputes the optimal distance to all other vertices. Then in the search stage, the heuristic distance from a vertex n to target t is $h(n) = max\{|dist(i,n) - dist(i,t)|_{i \in L}\}$. In road networks, a similar method is called ALT [26]. DH requires $O(k \times N)$ number of entries, where each entry is the shortest distance from a landmark to a vertex.

Canonical Heuristic (CHeur) [27] is another method to compute a heuristic based on precomputed shortest distances. Similar to DH, it selects k vertices, called canonical states, and stores the shortest distance between all pairs of canonical states. Additionally, for each of the N vertices, it stores the nearest canonical state and corresponding shortest distance. It only stores $O(k^2 + 2N)$ number of entries.

Fastmap (FM) [28] computes a geometric embedding that maps the vertex in 2D (grid or Euclidean plane) to higher dimension. For a k dimensional embedding, it needs k iterations. The i-th iteration runs a constant number of Dijkstra search and computes the embedding for all vertices in the i-th dimension. In the search stage, the heuristic distance is simply the Euclidean distance in the k-dimensional system.

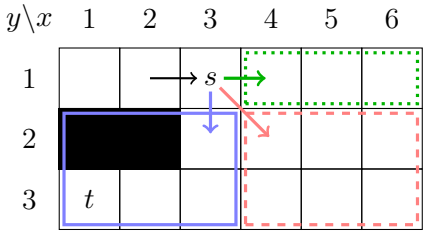


FIGURE 2.2: An example of using Goal bounding to prune successors. The current search node s is coming from W, and it has successors in three directions: S, SE and E. The database stores bounding boxes of optimal reachable nodes for each direction, the colour and style indicates the corresponding direction (solid blue for S, dashed red for SE, dotted green for E). Successors in SE and E can be safely pruned as their bounding boxes do not contain the target t.

Discussion For both *DH, CHeur* and *FM*, increasing k yields a more accurate estimation, but increases the cost, including preprocessing, repairing (in dynamic environments) and the heuristic computation at runtime, so there is a trade-off. However, the cost increases linearly in k, but the accuracy is not, e.g., doubling the k doesn't guarantee that we can reduce half of the node expansion. Thus, we tend to choose a small k (compare to $|V|$) for good performance, and for further improvements, we need to consider the expansion policy.

2.3.2 Goal bounding

This family of methods improve the A* expansion policy by reducing the branching factors in the search. They build a database for all-pair shortest paths, then exploit this information to prune unpromising successors in the query stage, Figure 2.2 shows an application of such idea in *Bounding Box* [29].

A general method to build the database is running Dijkstra on each vertex of the input graph, this procedure can be parallelised, and we can add more CPUs to reduce the total running time. So the main challenge is the memory cost. The database is $O(n^2)$ in space, when it fits in memory, we can keep extracting the optimal successor to get the shortest path and the A* becomes search-free, otherwise we need to compress.

A compression is called **lossless** if it preserves information of all-pair shortest paths, then we can still perform the search-free path extraction, the main challenge of this case is improving the compression to fit the data in memory, and when this is not achievable we need to consider lossy compression.

A compression is called **lossy** if it preserves only part of the information. In this case, we can prune some of unpromising successors, but the remaining may contains suboptimal

successors, and we still need a search to find the optimal path, the major challenge of this case is to trade off between the performance and the storage.

Lossless Compression Two representative methods in this category are *Hub labeling* and *Compressed Path Databases*. Hub labeling [30] (HL) precomputes two label sets for each vertex v: a forward label $L_f(v)$ and a backward label $L_b(v)$; labels store a collection of vertices, called *hubs*, and the corresponding shortest distance from/to vertex v. These labels satisfy a *cover property*: for any pair of vertices u and v, set $L_f(u) \cap L_b(v)$ must contain at least one vertex i that appears on the shortest path from u to v. The performance of HL depends on the total size of the labels, but minimising the total number of labels is NP-hard.

Compressed Path Databases [31] stores the optimal first move for each pair of vertices in a compressed form. In the query, it keeps extracting first-moves from the compressed data. There are many compression strategies, the most effective one is *Single Row Compression* (SRC) with *Run Length Encoding* (RLE) [32]; this compression needs a global defined ordering of all vertices, and finding the optimal global ordering is also NP-hard.

Since CPD stores first moves, in contrast to HL which stores distances, CPD can provide the current first move without extracting the entire path. This can be useful when the first-move lag is important. Thus, this research will focus on CPD, in the Section 2.4, we give more details of CPD to reveal the motivation of improving this method.

Lossy Compression *Geometric Containers* [29] is a famous method in this category. For each edge e, it computes geometric objects $S(e)$ that contain all vertices that can be optimally reached via e. In the query, it prunes successors that cannot reach the goal via the shortest path. In practice, simple bounding boxes outperforms other complex geometric objects, but they contain more vertices that are not optimally reached via edge e.

Another similar method is *Arc Flags* [33, 34], it divides the graph into k disjoint parts, and stores the "goal bounding" data in a k bit Boolean vector for each edge, where the i^{th} bit is false means that the edge doesn't appear on any shortest path to vertices in partition i.

Both Geometric Containers and Arc Flags can trade off between the performance and storage, e.g., increasing the size of $S(e)$ or k achieves stronger pruning but increases the database size.

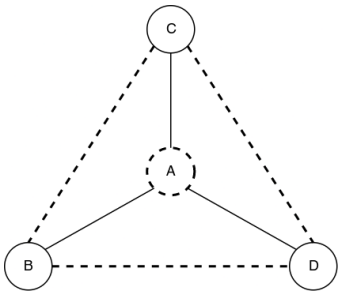


FIGURE 2.3: An example of Abstraction on toy graph G. G is undirected (for simplicity), the original edges of G are solid lines. To build the abstracted graph G', we remove the vertex A, and add edges (dashed lines) to G' to connect all neighbours of A, these edges represent optimal paths in the G. Then, finding the shortest path between C, B and D on the G' requires less search effort.

Discussion Goal bounding methods use the precomputed database to prune search space and allow flexible trade-off. The limitation is that they cannot exploit the underlying structure of the input graph, e.g., on a road network, most of optimal paths may pass some highways which are sequence of edges from the input graph, but the database still stores labels for all edges which may be redundant. To resolve this issue, we need *Abstraction* to compute the underlying structure of the input graph.

2.3.3 Abstraction

The input graph may have underlying structure, which can be abstracted into an overlay graph. For example, in a road network, there may be many small roads connecting local regions around big cities, as well as a few highways connecting cities. This family of methods exploits the abstracted overlay graph to prune the state space, and do the refinement to extract the path on the original graph. Figure 2.3 shows an example in *Contraction Hierarchy* [35].

Lossless abstraction An abstraction is called **lossless** if it preserves the optimality for all-pair paths. The most representative method in this category is *Contraction Hierarchy* (CH) [35]. CH computes an overlay graph by contracting all vertices in a predefined order which induces the hierarchy. When a vertex is contracted, the vertex and its edges are removed, and some extra edges (equiv. shortcuts) that connect pairs of its neighbours are added to preserve the shortest distance. In the query stage, it runs a modified Bi-directional Dijkstra on the overlay graph, where each search node only generate successors that appear in the higher hierarchy. The performance of CH depends on the predefined ordering. Although computing an optimal ordering is difficult, experiments show that some simple heuristics work well. *Subgoal Graph* (SG) [36] is another popular abstraction-based method for grid maps, leveraging *freespace-reachability*

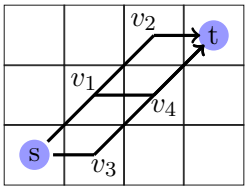


FIGURE 2.4: An example of symmetries on gridmaps. They are three equivalent paths from s to t: $[s, v_1, v_2, t]$, $[s, v_1, v_4, t]$, $[s, v_3, v_4, t]$,

(FR) in the domain, which defines a sparse visibility graph. In the preprocessing, the algorithm identifies convex corners called *subgoals*, then pairs of *subgoals* are connected based on *reachability relation*, which forms the *subgoal graph*. In the query stage, the algorithm connects start and target to the subgoal graph, then runs a best-first search on the graph and finally refine the edges on the shortest path to extract the path on the original graph.

Lossy abstraction An abstraction can also be **lossy** that aggressively abstracts the input graph to save search effort with the cost of optimality. A method that is widely used in games is *HPA** [37]. It is a near optimal (unbounded) method that is up to an order of magnitude faster than a well optimized A*. In the preprocessing stage, the method abstracts a map into multiple connected components, and computes the optimal distance for crossing the component. This allows the search to require only one step to cross the component at the abstract level. In the search, it connects the source and target to the abstracted graph, then retrieves the full path.

Discussion Designing an abstraction requires domain knowledge about the input graph. CH works on general graphs, but it is more efficient on road networks which have low highway dimension [38]. On gridmaps, due to the heavy symmetries, CH adds too many shortcuts on the abstracted graph that increases the storage and slows down the search, such behavior has been reported in [39]. SG can be generalized to other types of graphs, but need to redefine the *subgoals* and *reachability relation*.

2.3.4 Symmetry breaking

The input graph sometimes has a large number of symmetries, e.g., in Figure 2.4, reordering the cardinal moves and diagonal moves yields multiple optimal paths. When there are more cardinal and diagonal moves, the number of symmetric paths increases exponentially. Heavy symmetries slow down the search as the A* has to touch all equivalent search nodes. There are three popular methods for symmetry breaking.

Reachability relation SG builds subgoal graph by connecting convex corners only if they are *direct-h-reachable*, that is, the distance between them is the Octile distance and not via any other convex corner. Such condition allows an edge on the subgoal graph to represent symmetrical optimal paths on the input graph. During the refinement stage, the full path can be extracted from this edge by a linear search.

Canonical ordering Canonical ordering defines a preference for symmetrical paths and guide the algorithm to choose the most preferred one. In CH for gridmaps [39], it reduces some redundant edges on abstracted graph by choosing the path with minimum turns. JPS [6] utilizes *diagonal-first* ordering to reduce symmetries in the search space, i.e., it prioritizes paths with diagonal-moves appearing earlier when costs are equal. A following work [11] shows that *diagonal-first* ordering can be applied on A* and Dijkstra to achieve several factors speed up. In Section 2.4.1 we show more details of *diagonal-first* ordering and how does it works in JPS.

Random noises TRANSIT Routing [40] is a lossless abstraction technique for path planning on road networks. Despite its competitiveness in its domain, it experiences symmetry issues when applied to gridmaps. A latter work [41] resolves this issue by adding a small positive noise on all edges in the preprocessing stage. Such noise is sufficiently small that does not affect the optimality while reduces the number of nodes on the abstracted graph up to 4 times.

2.4 Background

Both CPD and JPS are the state-of-the-art in their categories. Among preprocessing-based methods, CPD has the fastest path extraction, and supports prefix queries which returns the optimal prefix without computing the entire path; meanwhile, JPS is the only purely online state-of-the-art method which is able to handle all dynamic scenarios. In this section, we demonstrate technical details of these methods for the better understanding of their pros and cons, and how to improve them.

2.4.1 Jump Point Search

The search framework of JPS is the same as A*, it uses a heuristic to prioritize search nodes in the open-list. By default the heuristic is **octile distance**. The major difference between JPS and A* is the expansion policy that prunes local suboptimal successors on-the-fly. We define the pruning rules and basic concepts of JPS below.

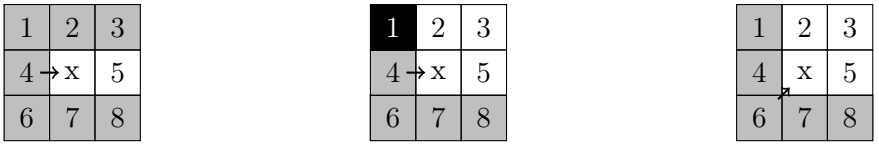
Definition 2.1. Let $NB(x)$ be adjacent vertices of x, and $p \in NB(x)$ be the predecessor of x during the search. Let $LP_p^x(t)$ be a set of optimal paths starting at p, ending at $t \in NB(x)$, and all paths are consisted with vertices from $NB(x)$. To generate a pruned successor set of x when coming from p, a vertex y is in set if $\forall l \in LP_p^x(y)$:

1. $len([p, x, y]) < len(l)$, or;
2. $rank([p, x, y]) < rank(l) \wedge len([p, x, y]) = len(l)$, where $rank$ represents the index of first diagonal move of a path.

Concepts Such a successor set is called the **diagonal-first neighbours**, denoted $neib(d, x)$, where d is the incoming direction inferred from p and x. When d is a cardinal direction, there is only one diagonal-first successor in free space. JPS keeps expanding in d until reaching a vertex x' such that $|neib(d, x')| \neq 1$ due to adjacent obstacles, this procedure is called **scanning**. When $|neib(d, x')| > 1$, x' is a **jump point**, and when $|neib(d, x')| = 0$, x' is a **dead end**. A vertex v is a **corner point** if $\exists d \in \{N, E, W, S\}$: $|neib(d, v)| > 1$. When $neib(d, x)$ has a neighbour in a diagonal direction d', JPS keeps moving in d' and scans in both cardinal directions on each step, until d' is blocked. This procedure is called **diagonal recursion**.

Since $neib(d, x)$ only requires local information, it can be computed on-the-fly in constant time. During the search, only jump points are added to the open list, *scanning* and *diagonal recursion* are procedures to identify jump points. Figure 2.5 shows visual example of *diagonal-first neighbours*:

- In (a), node 2 is diagonally reachable from 4 with shorter distance, so we prune it; similarly node 3 is reachable by path $[4, 2, 3]$ and $[4, x, 3]$ with same distance, while the latter is not *diagonal-first*, so we prune 3. Applying the same pruning on other neighbors, the only successor is 5.
- In (b), 2 is not reachable from 4 due to the no-corner-cutting rule (1 is an obstacle). Therefore, 2, 3 are successors of x with 5. In this case, x is a jump point. General speaking, we can identify a jump point in a cardinal direction if there is an adjacent row/column that has a transition from blocked to empty along the direction of movement.
- In (c), x is not a jump point as the incoming direction is not cardinal. In this case, we perform diagonal recursion. Example 2.1 demonstrates **diagonal recursion**.



(a) (b) (c)

FIGURE 2.5: In all figures, x is the current node, the arrow represents the incoming direction, black cells represent non-traversable cells, white cells are diagonal-first neighbours and gray cells are pruned.

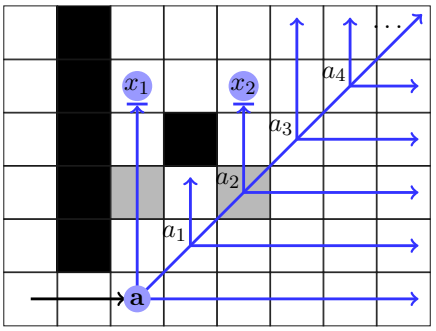


FIGURE 2.6: Node a is the current search node, the black arrow represents the coming direction, blue arrows represent scanning, $a_{i\in 1\ldots 5}$ are start points of each cardinal scan in diagonal. x_1, x_2 are jump points from the scanning direction, grey cells are corner points, the cell above a_1 is a dead-end.

Example 2.1. *In Figure 2.6, node a is a jump point with successors (diagonal first neighbour set) {N, E, NE}. JPS first scans in directions N and E, and finds a jump point x_1 to the north. Then it moves one diagonal step to a_1 and applies the same scanning, finding a dead-end at the node above a_1 during the North scan. Similarly, JPS will find a jump point x_2 on the North scan from a_2. All grey cells are corner points that wouldn't stop the scanning. JPS keeps moving diagonally and scanning cardinally until it is blocked in the diagonal direction.*

In the original JPS [6], the diagonal expansion will stop once a cardinal scan finds a jump point and push the current node on the diagonal line to the open list; in the later work [7], the diagonal move would continue until it is blocked and only adds jump point to the open list, this reduces redundant heap operation and improves JPS nearly a factor, in this thesis by JPS we refer to improved version.

Block-based scanning. *Block-based scanning* [7] represents the traversability of the gridmap by a bitmap, i.e. 0/1 means traversable/non-traversable; then instead of checking node by node, we can load a block of bits for adjacent columns/rows into memory, and quickly compute the position of the first jump point in this direction if there is one, by using three bitwise operations. This procedure is branch-less and leverages

SIMD instructions which can be extremely fast, it improves average times of JPS by nearly one order of magnitude. The only overhead is we need two bitmaps, one stored rowwise and one stored columnwise. For more details see example 2.2.

Example 2.2. *Assuming the block size is 3, in Figure 2.5c, the bit string of each row ('1,2,3','4,x,5' and '6,7,8') are* $s_1 = 100_2$, $s_2 = 000_2$, $s_3 = 000_2$. *In a left-right travel, we can quickly compute the first position that bits change from* 1 *to* 0 *by bitwise operation on* s_1, *which indicates a traversable node next to an obstacle; apply same reasoning on* s_2 *and* s_3, *we will find that* x *is a jump point.*

In practice, the block size depends on the CPU type, e.g. in a 64-bit CPU, the block size can be up to 64, which means we can check 64 cells at a time.

Discussion JPS significantly reduces the search space and the cost of heap operations; however, in an open map, JPS scanning has less chance to find jump points and tends to explore the entire row/column. There are some literature that tries to resolve this issue. Bounded JPS (BJPS) [11] introduces a predefined jump limit, a scan is forced to stop when hitting the jump limit and is pushed to the open list; but this method has to choose a proper jump limit for different maps, otherwise it increases the heap-operations and ends up with worse performance. Boundary look-up JPS (BLJPS) [12] uses a binary search to find jump points instead of scanning, it has better complexity guarantee, but due to the large overhead from the binary search, it is outperformed by block-based scanning in practice. In Chapter 3, we will propose a new method to resolve this issue.

2.4.2 Compressed Path Databases

CPD is a lossless goal bounding technique that yields a search-free query processing. In this section, we explore more details of CPD, then show a roadmap of improving and extending it.

Definition 2.2. A *First-move matrix* T is a square two-dimensional array that stores optimal first moves from every start to every target. T_s is the s-th row of the matrix T which stores all optimal first moves from vertex s to any reachable target. For a given target t, $T_s(t)$ represents the set of optimal first moves from s to t.

The size of T quickly becomes prohibitive, being quadratic in the number of vertices. A CPD is obtained by compressing T. CPDs are constructed *offline* in a preprocessing phase that requires repeated iterations of Dijkstra search: one per graph vertex. With only slight modifications, this algorithm can be used to compute T_s. In Chapter 5, we

will introduce another way to store first-moves – each row corresponding to the target vertex t, which records *all* optimal first moves, from any source vertex s to the target t. To distinguish these two types of CPDs, we refer to the former as *forward* and the latter as *reverse*. In this section, all CPDs are *forward*.

Once computed, T_s is compressed and stored, the compression is based on run-length encoding (RLE) [13]. See Examples 2.3 for an illustration. As they are independent, distinct Dijkstra iterations can be run in parallel, with a speed-up linear in the number of processors.

Example 2.3. *A compressed T_s can be represented by a list of first-moves, where each of them belongs to the corresponding first-move set. For example, in Figure 2.7a, the first-move sets for the 1st row is W; W; W; (W, E); E; E; E. So the corresponding list can be either [W, W, W, W, E, E, E] or [W, W, W, E, E, E, E]. RLE compresses a list of symbols by representing more compactly sub-list, called runs, consisting of repetitions of the same symbol. E.g., the list [W, W, W, W, E, E, E] can have two runs, namely WWWW, and EEE. We replace each such run by a pair of values: one value indicates the starting index (where the run begins) and the other value stores the associated symbol. With RLE the example sub-list can be represented more efficiently as [1W, 5E].*

Heuristic Ordering Assume for simplicity that we order all grid cells as one single string, traversing each row from the left to the right and going row by row from the top to the bottom. We call this the *left-right-top-bottom* ordering, and it is the default option in this thesis. RLE guarantees that the compression is optimal under a fixed ordering [13]. The size of the CPD can further be reduced by replacing the default scheme with a different cell ordering. However, finding the best ordering to minimize total number of runs in the first-move matrix is NP-hard [14], thus we need a heuristic ordering. DFS (Depth First Search) ordering suggested in [13] is the state-of-the-art, that is, regarding the gridmap as an undirected graph, run DFS on an arbitrary vertex and record the order of visited vertices.

Notice that there will be multiple DFS orderings, which depend on the order of visiting successors of each vertex; in practice we usually visit edges in clockwise order. In Example 2.4, we create the DFS manually for convenience.

Example 2.4. *Obstacle cells and the source are assigned wildcard symbols "∗"; i.e., "don't care" symbols, because we never need to look up a move from* s *to any of these. Consider the gridmap shown in Figure 2.7a. When applying the* left-right-top-bottom *ordering, the entire string is compressed into 11 runs: 1W 5E 8W 12E 15W 20E 22W 26E 29SW 32S 33SE.*

W	W	W	W,E	E	E	E
W	W	W	W,E	E	E	E
W	W				E	E
W	W	W	*s*	E	E	E
W,SW	W,SW	SW	S	SE	E,SE	E,SE

(a)

9	10	11	12	16	17	18
8	7	14	13	15	20	19
5	6				21	22
4	3	2	1	25	24	23
32	31	30	29	26	27	28

(b)

FIGURE 2.7: (a) Optimal moves from the source cell s (the yellow cell) to each traversable cell t. Observe that more than one optimal move can exist towards some targets (e.g., E, SE for the bottom-right cell). (b) DFS ordering.

When applying the DFS *ordering as in figure 2.7b, the entire string is compressed into 5 runs: 1W, 15E, 26SE, 29S, 30SW.*

Bidirectional Wildcards We can improve the compression in a CPD for undirected graphs by taking into account the duplicate information [15]. In order to find the shortest path from s to t we only need to know either (a) the first move from s towards t, or (b) the first move from t towards s. We do not need both.

Assume a realised relation $R \subseteq V \times V$ which defines which pairs of (s,t) are realised correctly in the CPD. In practice the realised relation R is based on a vertex ordering, that is, $(s,t) \in R \Leftrightarrow s$ comes before t in the ordering.

The advantage of the realised relation is that if $(s,t) \notin R$ then we will never access the t entry in T_s. Hence, we can replace the t in T_s with wildcard entries "$*$". This allows to create more compressed run length encodings.

Algorithm 2: Bidirectional path extraction at runtime for an (s,t) pair

```
1  prefix ← []
2  suffix ← []
3  while s ≠ t do
4      if (s,t) ∈ R then
5          (s,n) ← FirstMove(s,t)
6          prefix ← prefix++[(s,n)]
7          s ← n
8      else
9          (t,n) ← FirstMove(t,s)
10         suffix ← [(n,t)]++suffix
11         t ← n
12 return prefix + suffix
```

Path Extraction With a CPD in hand, we may begin the *online* phase of the algorithm. Here the objective is to compute an optimal path $cpd(s,t)$ or prefixes for any given start-target pair (s,t) (equiv. *instance*). We denote as $\mathsf{FirstMove}(s,t)$ the function which returns an optimal first move from s to t. The implementation of this function requires a simple binary search through a compressed string of symbols representing the first-move row T_s [13]. The function can be used to extract entire optimal paths or optimal prefixes of any given size. The Algorithm 2 shows the path extraction procedure with bidirectional wildcards, the algorithm returns a sequence of optimal first-moves, and we can build the corresponding shortest path.

CPD Search Since CPD path extraction is much faster than search-based algorithms, it can be used as a heuristic in A* search to derive a new state-of-the-art for pathfinding in dynamic cost gridmaps [16]. The resulting algorithm, CPD Search, differs from A* in two important ways: i) the search tracks and stores best known "incumbent solutions" – i.e. feasible paths computed by the heuristic whose cost bounds the optimal solution from above (recall that A* only bounds from below); ii) early termination, which means the search stops when a quality threshold is achieved or a time limit is exceeded. Upon termination CPD Search returns the best known incumbent. For computing optimal paths, CPD Search can stop as soon as the upper and lower bounds meet for the currently expanded vertex. Often this occurs well before the expansion of the target vertex. When computing suboptimal paths, CPD Search finds a first solution fast (usually microseconds, sometimes less) as often the "CPD path" from s is feasible. Each expansion thereafter offers an opportunity to improve the incumbent and raise the lower-bound. Being anytime the algorithm can be terminated after available time for planning has elapsed. Usually there is a solution in hand that can be returned. Compared to other similar methods, such as Anytime Weighted A* [42], CPD Search computes first solutions faster, avoids re-expansions later and proves optimality sooner. In Chapter 6, we will generalize CPD search, and propose an oracle-based method on cost-changing road networks.

Discussion The major issue of CPD is that the first-move matrix may not compress well on some maps, and when the map is large, there are more rows that need to be stored, which causes the size of CPD to be prohibitively large. Two key components that affect the size are ordering and encoding. Since computing an optimal ordering is NP-hard, we focus on improving the CPD encoding. In Chapter 4 we will define a new encoding method to improve the compression, in Chapter 5, we will show how to trade off between the size and optimality, and in Chapter 6 we will show how to extend this idea and tackle a practical problem on road networks.

Chapter 3

Reducing Redundant Work in Jump Point Search

Grid-based pathfinding is a classic problem in AI and is widely used in games and robotics, as well as an active research area. Despite the many fast preprocessing-based approaches for solving this problem, online approaches are still preferable in dynamic environments where obstacles may appear on the map randomly between different queries. The reason is that preprocessing-based approaches have to rebuild or repair with the map changes, and when this happens frequently and/or large regions of the map are affected, the additional preprocessing cost can be huge. As a result, the amortized performance becomes worse than purely online approaches.

JPS (Jump Point Search) [6] is a state-of-the-art online approach to pathfinding. Although JPS is widely used in practice, and there have been many proposed improvements [9, 10, 43], the core algorithm has not been improved since 2014 [7]. Our first contribution is to identify pathological behaviours in the core JPS algorithm which are not well-studied in the literature. Our experiments show that these behaviours can significantly slow down the performance of JPS in dynamic environments.

Pathological behaviour #1: JPS identifies successors by scanning, which may result in a lot of redundant work. For example, it may potentially explore the entire map repeatedly, even if no successor exists. Bounded JPS [11] solves this issue by adding a constant jump limit during the scanning, so that a search node will be suspended and pushed back to the open list when hitting the jump limit. But suspending scans often results in extra heap operations, thus practitioners have to carefully choose a suitable limit for each different map to achieve an improvement. Boundary Lookup JPS [12] solves the issue by using binary search instead of scanning. This has better complexity but is much slower in practice than optimised JPS implementations using block-scanning [7].

Pathological behaviour #2: JPS may generate and expand suboptimal search nodes. This behaviour has not been reported previously, but is a reason for concern — if many suboptimal search nodes are created and expanded, there is a potential unbounded overhead for JPS vs. A* in pathological scenarios, since A* expands each node at most once. So it is important to understand how this happens, how much performance is affected and how to avoid it.

In this chapter, we start with an example of redundant work in JPS in Section 3.1, which gives us motivation and observation. Then, in Section 3.2, we propose a new and purely online approach, called *Constrained JPS* (CJPS), to efficiently resolve these issues. We use geometric constraints to compute dynamic jump limits on-the-fly, which helps to reduce redundant scanning, and also allows us to avoid the generation and expansion of suboptimal successors. In Section 3.3, we integrate these constraints with block-based scanning to produce a new state-of-the-art implementations of JPS. Finally, in Section 3.4 we study the root cause and impact of suboptimal node expansions.

In Section 3.5, we compare the new approach (CJPS) with vanilla JPS on both synthetic data and public benchmarks [44, 45]. Experiments show that in benchmarks drawn from real applications, when the environment is dynamic, CJPS is up to 7x faster than JPS on cumulative time, and in pathological setups, such improvement can be up to 14x. When there is little or no redundant work, CJPS has a small overhead of $\tilde{2}5\%$. Overall, *Constrained JPS* is a substantial advancement for optimal and online grid-based pathfinding in dynamic environments.

3.1 Pathological Behaviours in JPS

Vanilla JPS has the following issues that haven't been explored in-depth by previous literature: i) redundant scanning; ii) generating suboptimal search nodes; iii) expanding suboptimal search nodes. Example 3.1 shows such behaviours, in Section 3.4 we will provide more insights on suboptimal node expansion.

Example 3.1. *In Figure 3.1, a and b performs symmetric diagonal recursions. There are many nodes that are scanned by both of them, resulting in redundancy. Depending on the g-value of each search node, some successors may be suboptimal. For example, when* $g_a = g_b$, x_1, x_2 *are bettered reached from b, but JPS still generates them as successors of a. Furthermore, suboptimal nodes* x_1, x_2 *will be expanded later.*

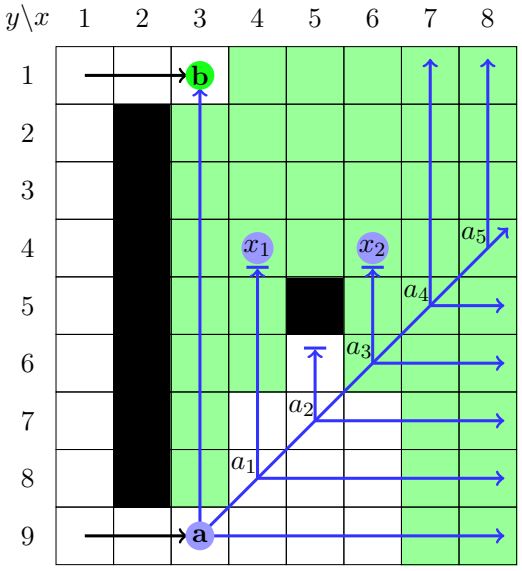


FIGURE 3.1: a and b are search nodes. Blue arrows shows nodes that are scanned during the diagonal recursion of a, and b, x_1, x_2 are identified successors (jump points). JPS would have the same produce on b, and green nodes are scanned during the diagonal recursion of b.

3.2 Constrained JPS

Observation In Example 3.1, a and b know each other's g-value during their diagonal recursion, which can be utilized to reduce redundant work.

Based on the observation above, we have developed a framework called *Constrained JPS*. At a high level, when we expand node a and the cardinal scanning finds a jump point v with an existing g value, we can create a constraint for the same cardinal direction during the current diagonal recursion. This constraint restricts cardinal scanning in the later diagonal recursion by a dynamic jump limit, and it can be deleted or updated during the recursion. In the following parts, we firstly give a formal definition of the constraint, then we demonstrate how to compute it, when to delete or update it and how to use it for pruning.

Definition 3.1. A *constraint* is defined by a tuple $\langle a, v, d, L \rangle$, where:

- a is a node which starts or continues a diagonal recursion;
- d is the direction of a cardinal scanning in diagonal recursion;
- v is a jump point found by a scanning in d from a, and has a previously stored g-value;

- L is an integer that indicates the maximum number of recursive diagonal moves from a. A constraint is **applicable** if the number of diagonal moves from a is not greater than L.

In addition, in the rest parts of this section, we use i for the i-th number of diagonal move from a and a_i for the corresponding node.

L guarantees that when a constraint is applicable, the cardinal scanning in direction d on a_i shouldn't take more than $|av| - i$ steps. Such restriction is a perpendicular blockage of the d, any path crossing the blockage is better reached from v, Figure 3.2a illustrates the idea. We now show how to compute L.

Computing L Let g_a be the g-value of node a and g_v be the g-value of node v when expanding a, to compute L, we need to consider the following cases:

(i) If $g_a + |av| \leq g_v$, all scanning that cross the blockage from further diagonal moves are better from a, and no constraint is applicable, so we set $L = 0$.

(ii) If $g_a + \sqrt{2}|av| \geq g_v + |av|$, the diagonal recursion should be terminated at $i = |av|$, since all further nodes are better reached from v, so we let $L = |av|$;

(iii) Otherwise L is the minimum integer satisfies:

$$\begin{cases} L \geq 0 \wedge L \leq |av| \wedge \\ \overbrace{g_a + \sqrt{2}L + (|av| - L)}^{\text{distance from a}} < \underbrace{g_v + L}_{\text{distance from v}} \end{cases}$$

Updating and Deleting the Constraint When the constraint is applicable, at the i-th diagonal step (i.e., at a_i), there are two cases in the cardinal scanning: i) the scan stops at the blockage; ii) the scan is stopped by a jump point or a dead end at node p, before reaching the blockage. In the first case, we can still use the constraint as long as it is applicable. In the second case, we must discard the current constraint and create a new constraint on p. To do this, we need to estimate a tight upper bound for the g-value of p:

$$\overline{g}_p = \min\{g_p, octile(v, p') + 1\} \tag{3.1}$$

where g_p is the g value currently stored with p (or $+\infty$ if it has no stored value), and p' is the cell in the previously scanned row/column adjacent to p, and $octile(v_1, v_2)$ is the octile distance between nodes v_1 and v_2. This bound is safe since the previous scans of rows/columns in from v to p' must be empty up to the blockage, since we haven't yet

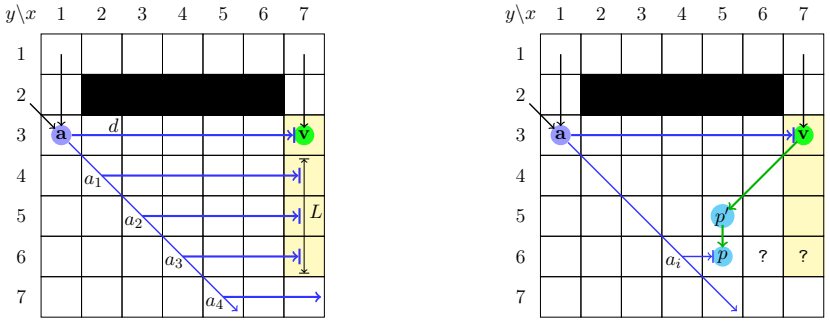


FIGURE 3.2: (a) shows constraint (a, v, d, L), where a is a jump point coming from north, or a continued diagonal move from northwest; v is a previously visited node that found from the cardinal scanning from a, and L is the maximum number of steps where the constraint is applicable, yellow cells are better reached from v, thus cardinal scanning from a_1, a_2, a_3 is stopped by the constraint at the yellow blockage; (b) shows how to evaluate g_p, where p is the stop location of a scanning before the blockage, thus the traversability of all nodes after p is unknown, noted by '?'.

deleted the constraint from v. Therefore, we update the constraint to $\langle a_i, p, d, L' \rangle$, where L' is computed based on $g_{a_i}, \overline{g}_p$ and $|a_i p|$, Figure 3.2b gives an example. When $L' = 0$, the new constraint is not applicable, then we can simply delete it.

Pruning and Early Termination Equation 3.1 can estimate whether node p is better reached from v than a. When p is a jump point, we can prune it if:

$$\overline{g}_p < g_{a_i} + |a_i p|$$

and when p is at the blockage, we can terminate the diagonal recursion early if:

$$\overline{g}_p + |a_i p| < g_{a_i}$$

Bring all pieces together, we propose *Constrained Jump Point Search* (CJPS). Algorithm 3 illustrates horizontal constraint of CJPS, vertical constraint can be done in a similar way.

3.3 Branch-less Implementation

Block-based scanning can improve JPS by nearly an order of magnitude [7]. Therefore, we need CJPS to be compatible with it. CJPS computes a jump limit on-the-fly based on the information obtained from scanning. However, blocked-based scanning is branch-less and leverages SIMD instructions. Applying a jump limit is not trivial, as adding any

Algorithm 3: Horizontal constraint can be applied in diagonal recursion, and vertical constraint can be implemented similarly. In practice, both can be applied orthogonally in the same diagonal recursion.

Input: $n = (x, y)$: the location of current search node;
$d = (dx, dy)$: the direction of the diagonal move;
$d_h = (dx, 0)$: the direction of horizontal scanning;
$c_h = (a, v, d_h, L)$: the constraint for horizontal scanning;
db: the bit map for block based scanning;

```
1  calcL(g_a, g_v, |av|) : computing L based on definition 3.1;
2  calcG(v, p) : computing upper bound of g_p based on equation 3.1;
3  scan(n, d, JL, db) : scanning at n in d with the jump limit JL (see in Algorithm 4);
4  i ← 0;
5  while empty(x+dx, y) and empty(x, y+dy) do
6      n ← n + d;
7      if i ≤ L then
8          i ← i + 1;
9      if i ≤ L and n is better reached from v then
10         break;
11     if i ≤ L then
12         JL ← |av| − i;
13     else
14         JL ← ∞;
15     p ← scan(n, d_h, JL, db);
16     if |np| < JL then
17         g_p ← calcG(v, p);
18         c_h ← n, p, d_h, calcL(g_n, g_p, |np|);
19         i ← 0;
20     if p is jump point and better reached from n then
21         successors.add(p);
```

if-then-else statement would significantly slow down the scanning process. To force the scanning to stop at certain jump limit in a branch-less way, we instead set an artificial obstacle on the map before the scanning and unset it afterwards, as shown in Algorithm 4.

3.4 Eliminating Suboptimal Node Expansion

We have shown that *CJPS* can prune redundant scanning and suboptimal nodes, but it doesn't guarantee to eliminate all of them. Other online algorithms, e.g. A* and Dijkstra, are less efficient in practice but guarantee no suboptimal node expansion. Thus we need to answer the following questions: 1) how does this happen? 2) how bad could it be? 3) can we eliminate all suboptimal node expansion?

Algorithm 4: branch-less bounded cardinal scanning

Input: C: the current location (as a bit address);
D: the cardinal direction (as a bit delta, e.g. W $= -1$, E $= +1$, S $= +mw$, N $= -mw$ where mw is the width of the map);
L: jump limit;
db: bit map for block based scanning (1 means obstacle)

1 `scanning`(C, D, db) : block-based scanning in [7]
`// store the original value in tmp`
2 $tmp \leftarrow db(C + L \times D)$;
3 $db(C + L \times D) \leftarrow 1$;
4 scanning(C, D, db);
5 $db(C + L \times D) \leftarrow tmp$;

3.4.1 How does it happen?

Let a be a non-target search node that is expanded with suboptimal g-value, then node a must be a corner point from the optimal parent rather than a jump point (as seen in Definition 2.1). This is because JPS is optimal and complete, if a were a jump point it must have been reached by the optimal parent before expanding with suboptimal g-value. But since JPS only stops at the jump point or the target, for corner point a, a search node with optimal g-value may not be generated, Figure 3.3 shows an example.

We can conclude that expanding a suboptimal node v requires two conditions: (i) v is reachable from at least two parents (i.e., jump points or the start node); (ii) v does not appear in the search space of its optimal parent due to the JPS pruning rule in definition 2.1. It is more likely to happen when the environment has many corner points and misleading heuristic, e.g., game maps.

3.4.2 How bad it could be?

The successors of a suboptimal search node may also be expanded, and the suboptimality is propagated, e.g. a_2, a_3, a_4 in Figure 3.3; when this keeps happening, it is possible that suboptimal node expansion dominates the entire search.

It is hard to perform a theoretical analysis on the number of suboptimal node expansions in general because such behavior depends on many factors, such as the topology of the map, the order of expansion, and the target location. To understand how frequently this happens in practice, we counted the number of suboptimal node expansions over a subset of queries in the benchmark, as well as the number of propagations, i.e., the number of expanded nodes whose parent is suboptimal. Table 3.1 shows the result. We see that, there are no suboptimal nodes in *maze512* domain. This is due to the underlying

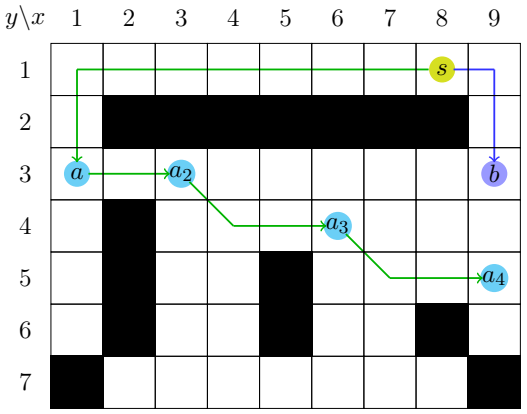


FIGURE 3.3: s (yellow) is the source node, a (cyan) and b (blue) are successors of s. a_2, a_3, a_4 are suboptimal nodes propagated from a.

domain	#maps	mean	median	sub%	subp%
dao	154	215	62	30	64
bgmaps	75	98	41	27	65
starcraft	75	1038	777	28	73
street	30	596	312	33	84
iron	35	5065	3962	43	88
maze512	6	8863	2516	00	NaN
random10	1	7574	7527	33	18
rooms	4	1766	653	25	26

TABLE 3.1: Statistic of each domain. We count node expansion from 100 longest queries of each map. *mean, median* are for queries, which indicates the difficulty, *sub%* shows the proportion of suboptimal node expansion, and *subp%* shows the proportion of propagated suboptimal node expansion among *sub*.

tree structure of these *maze512* maps, which means that search nodes are reachable from only one jump point. In other domains, more than 25% of node expansions are suboptimal, and many of these are propagated from their parent. This implies that pruning a suboptimal node earlier could avoid expanding more suboptimal nodes in the future.

3.4.3 Can we eliminate all of them?

Based on discussion above, it is clear that we can eliminate all suboptimal node expansion by storing a g-value on every corner point, rather than just on every jump point. However, this approach comes with significant overhead since, for each cardinal scan, it needs to stop at each encountered corner point. Although this does not change the total number of scanned nodes, it can significantly affect the performance of block-based scanning. Therefore, there is a trade-off between introducing the overhead and reducing redundant work.

Algorithm 5: Backwards scanning

Input: a: the starting location of the scanning;
n: the location of the found jump point;
d: the direction of the scanning;
db: the bit map for block based scanning;

```
1 boundedScan(C, D, L, db)  : Algorithm 4
2 cnt ← 0;
3 c ← n;
4 while c ≠ a and cnt ≤ 2 do
      // scan from c in the reverse direction and not cross a
5     p ← boundedScan(c, −d, |ac|, db);
6     g_p ← min{g_p, g_a + |ap|};
7     c ← p;
8     cnt ← cnt + 1;
```

In this section, we introduce two ways to store the g-value on corner points with minimal overhead, and we conduct an ablation study in experiments to demonstrate the effectiveness of these variants in pathological scenarios.

The most straightforward way is to store a g-value on each corner point found on the diagonal from the expanded node. Since the diagonal recursion stops at every cell on the diagonal, there is little overhead mainly arising from extra memory accesses during the search. However, this approach only allows pruning on cells that appear on the diagonal from an expanded node, and most cells do not meet this condition.

Another way is *backwards scanning*. When a cardinal scan in direction d finds a jump point n, we also set the g-value on all corner points along d by scanning from n in the reverse direction. The scanning in d can pass at most two corner points before reaching n - one for the row above and row below (or the column left and right). Since not all scanning finds a jump point (most are dead-end), this approach should have smaller overhead than stopping at all corner points, but is more expensive than the previous approach. See details in algorithm 5.

3.5 Experiments

We consider two distinct benchmarks: synthetic maps, which shows performance in extreme cases, and domain maps, which shows performance on a range of well-established test sets.

Synthetic maps: based on an empty map with random obstacles and diagonal blockage in the middle, which are controlled by three variables:

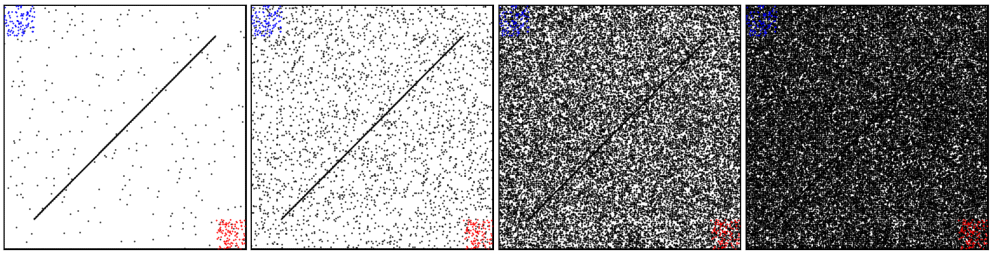


FIGURE 3.4: Synthetic maps, where $s = 512$, $b = 75\%$ and $r \in \{0.1, 1, 10, 20\}$. Blue and red clusters are starts and targets.

- r: the proportion of traversable cells that are blocked, which simulates dynamic environments;
- s: the height and width of map is $s \times s$;
- b: the proportion of diagonal blockage, which controls the difficulty, i.e., less accurate heuristic;

There are 100 instances per map, where starts and targets are always traversable and clustered in top-left and bottom-right regions. Figure 3.4 shows an example.

Domain maps: We also compare *CJPS* and *JPS* on real application maps from two public benchmark sets. The first is MovingAI benchmark [44], which features a variety of domains including: games (bgmaps, starcraft, dao), grid rasterisations of real cities (streets), other three artificially generated domains (maze512, rooms, random) are excluded. The second benchmark, Iron Harvest, is a set of 35 grid maps taken from the game [45], which are intended to be more challenging than existing game benchmarks. Table 3.1 shows a summary of these domains.

Our competitor is JPS, the state-of-the-art optimal online method, which is suitable for dynamic environments. Both *Constrained JPS* (CJPS) and JPS use block-based scanning [7], a well known optimisation. For all experiments that measure execution time, we run each map 10 times in random order and choose the median, to avoid cache behaviour and reduce random noise. For all experiments that inspect suboptimal node behaviour, we run a Dijkstra to compute a true-distance table before each query.

All algorithms are implemented in C++ and compiled with *clang 13* using *-O3* under *5.10.102-1-MANJARO* with *Intel Xeon E-2276M* Processor and *32 GB* Memory. Our implementations and data are available online.[1]

impv-fr(%)	opt	subopt	TPE	runtime
0.0	1.00	nan	0.79 (5.76/ 7.34)	0.79
0.1	1.64	37.10	8.95 (3.24/ 0.36)	14.87
1.0	1.56	3.84	4.16 (1.51/ 0.36)	6.47
10.0	1.15	1.44	1.03 (0.39/ 0.37)	1.18
20.0	1.08	1.26	0.89 (0.30/ 0.34)	0.96

(a) Vary obstacle density, fix s=512, b=75%.

impv-fb (%)	opt	subopt	TPE	runtime
0.00	1.00	1.00	0.78 (12.00/ 15.19)	0.78
0.25	1.52	8.90	7.48 (5.02/ 0.66)	10.21
0.50	1.58	9.12	8.20 (3.81/ 0.46)	12.72
0.75	1.64	37.10	8.98 (3.26/ 0.37)	14.75

(b) Vary heuristic accuracy, fix r=0.1%, s=512

impv-f resolution	opt	TPE	runtime
256	1.31	2.85 (1.21/ 0.43)	3.80
512	1.29	4.62 (3.63/ 0.79)	5.97
1024	1.29	5.10 (11.32/ 2.24)	6.59
2048	1.29	5.47 (38.73/ 7.12)	7.12

(c) Vary resolution, scale up a map r=0.1%, s=256, b=75%

TABLE 3.2: Improvement factors of various metrics ($\frac{Metric(JPS)}{Metric(CJPS)}$) on three settings, >1 means improvements, where: (1)*opt* measures heap operations (#expansion + #insertion); (2)*subopt* measures *opt* on suboptimal nodes; (3)*TPE* (time cost per expansion) measures average cost per node expansion in μs ($\frac{time}{\#expd}$), we also reveal the raw TPE of JPS and CJPS in parenthesis; (4)*runtime* measures average runtime per query ($\frac{time}{\#queries}$).

3.5.1 Exp-1: Synthetic Maps

This experiment is to show how CJPS is affected by three map properties: random-obstacles density, heuristic accuracy and resolution. To do this, we run queries on three sets of synthetic maps:

- Fix s=512, b=75%, vary r $\in$ {0, 0.1, 1, 10, 20}%;
- Fix r=0.1%, s=512, vary b $\in$ {0, 25, 50, 75}%;
- Fix the synthetic map (r=0.1%, s=256, b=75%), vary resolution $\in$ {256, 512, 1024, 2048}. Note that changing resolution doesn't affect the topology of map, the number of jump points is same.

[1]https://github.com/eggeek/warthog-pathfinding

Table 3.2 shows the results.

From table 3.2a, we can see that when there is no chance to prune (r=0%), CJPS is 21% slower than JPS due to the overhead of local reasoning in diagonal recursion. With 0.1% random obstacles, CJPS's advantage shows up; it is 14.87 times faster than JPS due to the pruning. When r increases, the average expansion cost of CJPS is stable, and although there are more chance to prune (i.e., more search nodes), the improvement factor drops. The reason is that the g-value differences between optimal and suboptimal nodes are smaller, and the upper bound estimation (equation 3.1) is relatively less accurate, which weakens the pruning (i.e., *subopt* drops). Meanwhile, JPS expansion cost becomes smaller as the diagonal recursion terminates earlier. Thus, CJPS becomes less effective and eventually slower than JPS when r=20%.

From table 3.2b, we can see that CJPS is more effective when the heuristic is inaccurate because the search space becomes larger and JPS trends to generate more suboptimal search nodes.

In table 3.2c, the *opt* are the same on both maps because simply scaling up doesn't change the number of jump points in the search space. The expansion cost improvement of CJPS increases. The reason is that scanning on higher-resolution maps is slower due to the cache behaviour, so the reduced scanning in CJPS saves more time. This means that CJPS has more advantage on less powerful CPUs. CJPS also has more cache misses on larger maps as the local reasoning needs to access the g-value table. Thus, the improvement factor on expansion cost only increases by 0.37 (5.10 to 5.47) from 1024 to 2048 resolution.

In summary, CJPS makes a significant improvement in dynamic scenarios(r>0%), especially when the heuristic is inaccurate or the resolution is high, while the improvement factor drops with the increasing density of obstacles.

3.5.2 Exp-2: Ablation Study

This experiment is to show whether it is worth further eliminating redundant work by diagonal storing (*-g*) and backwards scanning (*-b*). We run both variants ({JPS, CJPS} × {*-g, -b*}) on the synthetic map set CJPS is less effective when the density increases (fix s=512, b=75%, r ∈ {0, 0.1, 1, 10, 20}%).

Table 3.3 shows the result. We can see that both *-g* and *-b* reduce the proportion of suboptimal node expansion on top of JPS and CJPS (Table 3.3a), but such improvement is not enough to pay the overhead, so they don't achieve better performance (Table 3.3b). Thus, in the next experiment, we focus on CJPS.

r(%)	jps	jps-g	jps-b	cjps	cjps-g	cjps-b
0.0	0.00	0.00	0.00	0.00	0.00	0.00
0.1	0.52	0.52	0.29	0.06	0.06	0.06
1.0	0.71	0.69	0.60	0.29	0.27	0.17
10.0	0.59	0.57	0.50	0.47	0.43	0.39
20.0	0.44	0.40	0.35	0.37	0.35	0.31

(a) Proportion of suboptimal expansion: $\frac{\sum SuboptExpd}{\sum Expd}$

r(%)	jps-g	jps-b	cjps	cjps-g	cjps-b
0.0	0.97	0.95	0.75	0.77	0.77
0.1	0.99	1.02	14.61	13.55	11.91
1.0	1.00	0.75	6.32	6.13	6.14
10.0	0.93	0.85	1.20	1.18	1.19
20.0	0.93	0.86	0.98	0.98	0.96

(b) Speedup factor: $\frac{avg(Time_{jps})}{avg(Time_{cjps})}$

TABLE 3.3: Results on synthetic maps. We look at suboptimal expansions and run time.

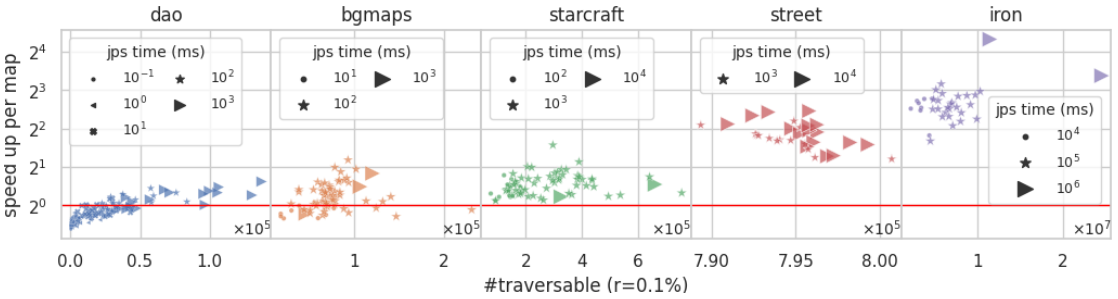
3.5.3 Exp-3: Domain Maps

r (%)	time(s) domain	cjps	jps	speed up
0.0	dao	13.93	13.58	0.97
	bgmaps	4.95	3.49	0.71
	starcraft	37.86	33.12	0.87
	street	17.14	13.44	0.78
	iron	167.35	175.17	1.05
0.1	dao	15.52	17.22	1.11
	bgmaps	6.49	7.51	1.16
	starcraft	47.20	68.88	1.46
	street	28.09	98.57	3.51
	iron	341.78	2508.75	7.34

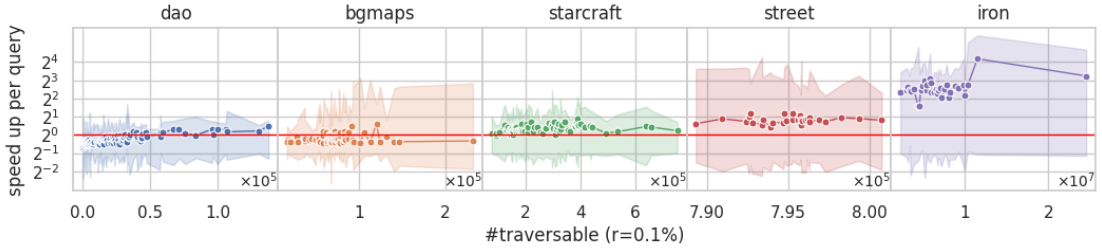
TABLE 3.4: Cumulative time per domain on static and dynamic environments.

This experiment is to show CJPS performance on public benchmarks from real applications when environments are dynamic. There are three resolutions for city maps (*street*). We pick the highest one (1024) as it is more challenging.

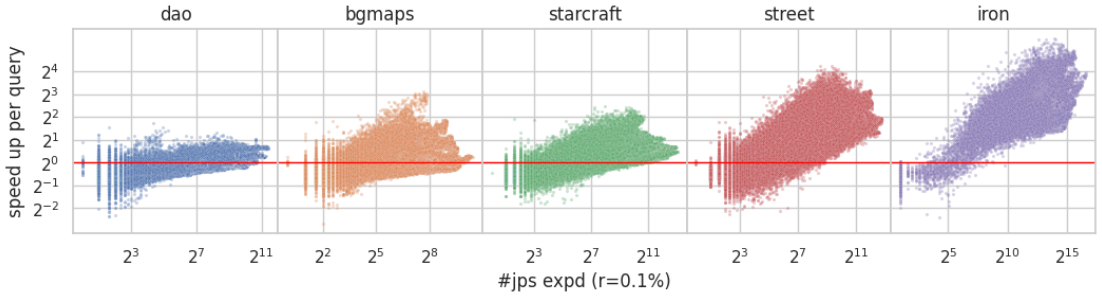
Simulate dynamic environments. In real applications, the number of dynamic changes depends on the size of the map and is not very dense. For example, in Starcraft, each player controls at most 200 agents and the number of facilities is usually much smaller than this, while the maps are usually 512×512. Thus, we assume the density of random obstacles is 0.1%. To simulate dynamic environments, since removing obstacles



(a) Speed-up per map in increasing number of traversable cells.



(b) Compact distribution of speed-up on queries per map. Points represent the median, and the bands represent the minimum and maximum values.



(c) Speed-up per query in increasing difficulty (*jps expansion*)

FIGURE 3.5: Speed-up on cumulative time ($\frac{\sum Time_{jps}}{\sum Time_{cjps}}$) and queries($\frac{Time_{jps}}{Time_{cjps}}$). The red line is 1, and values above it indicate that CJPS is faster.

to make an agent passes through a wall requires analyzing obstacle components which is out of our research scope, we add random obstacles in the same way as synthetic maps, and assuming the result represents the map after a dynamic change.

Table 3.4 shows the total time to finish all queries per domain, we can see that CJPS has no advantage when the environment is static, but wins in dynamic environments, Figure 3.5 shows detailed results in such an environment.

How to read these plots. Figure 3.5a shows speed-up on the cumulative time per map, where markers indicate the order of time to finish all queries. It describes an overview of improvement but misses the distribution of improvement per query. Figure 3.5b shows a compact distribution of speed-up on queries per map, including min,

median and max, but it doesn't directly show what kind of queries are improved. Figure 3.5c shows the query speed-up in increasing difficulty, i.e., number of JPS node expansion.

We can see that in *dao* and *bgmaps*, there are no significant improvements. Even though the speed-up of cumulative time can be up to 2 on specific maps, most instances are slower than JPS. The reason is that most maps from these domains are either small or easy. *Dao* has many small dungeon-like maps, and all maps from *bgmaps* are scaled to 512 from a smaller size which have large open spaces and accurate heuristics. In the rest of the domains, according to Figure 3.5b, CJPS has better performance on most maps. In *starcraft*, most queries get improved, i.e., the median point is above the red line. In *street*, the median speed-up factors are about 2. In iron, CJPS is more than 4x faster on most maps, and can be up to 16x on one large map. According to Figure 3.5c, an encouraging feature is that, when it is improving CJPS becomes more effective, the harder the query is.

3.6 Conclusion and Future Work

In this chapter, we study the pathological behaviours of *JPS*: redundant scanning, suboptimal node generation and suboptimal node expansion. We also introduce a new approach, *CJPS*, which effectively resolves these issues and convincingly improves JPS performance in a dynamic environment. Although JPS still wins when the environment is static, offline-based methods should be applied in this case.

While CJPS is designed for online search, it is orthogonal to and be combined with preprocessing-based improvements that work with JPS, e.g. [9, 10, 43]. This is a future work. Another interesting direction is grid-based pathfinding with higher dimension, e.g. 3D voxel grid map [46] or 2D grid maps with temporal obstacles [17], in the latter, obstacles can move and environments are changing during the query. These problems have a larger search space and more symmetries, thus there are more opportunities for CJPS to improve over the baseline algorithm.

Chapter 4

Improving Forward CPD

A main drawback of CPDs is that the preprocessed data can sometimes be prohibitively large to the point that it exceeds allocated memory — even after considering recent efforts to reduce its size; e.g. [13, 15]. In this chapter we seek to address the issue with two new compression ideas that can substantially reduce the size of existing CPDs:

- *Redundant symbols* are a generic concept that allow us to represent a single first-move with two or more symbols. The additional flexibility allows us to more effectively compress first-move data.
- *Proximity wildcards* are "don't care" symbols which allow us to avoid storing first-move data in cases where such data can be efficiently re-computed online.

We undertake a detailed evaluation on gridmaps from Sturtevant's well known benchmark sets [5]. Results indicate substantial reductions in CPD size, from several factors to over one order of magnitude. Because CPD sizes are much smaller, we also report a speedup in the time required to extract optimal shortest paths.

The rest of this chapter is organised as follows. In Section 4.1, we introduce the redundant heuristic symbol and show how this can help compress CPDs. In Section 4.2, we show how we can use proximity wild cards to help compress CPDs. In Section 4.3, we explain the experimental setup and give experimental results. Finally, in Section 4.4 we give the conclusions and mention future work. In this chapter, for simplicity, we use the *left-right-top-bottom* ordering in all examples of the compression by default.

The content of this chapter is from our published work joint with Mattia et al. [1], Mattia is the major contributor of the proximity wild cards in Section 4.2.

4.1 Heuristic Redundant Symbols

In an obstacle-free open space, the first move to be taken from a start s to target t is the "obvious move" that heads in the direction towards the target. Based on this observation, rather than record the first optimal move in the CPD, we can just record that the move is "obvious"; then in the query stage, with this information, we can just apply a predefined heuristic policy to extract the move. To do so we introduce a new *heuristic redundant symbol* ⓗ to represent such moves.

Definition 4.1. Let s be the source and t be a target, let $h(s,t)$ be the first-move extracted from the predefined heuristic policy, called heuristic move. When the heuristic move from s to t is one of the optimal first moves, i.e. $h(s,t) \in T_s(t)$ (see in Definition 2.2), then we can encode this information in the CPD using the new symbol ⓗ which represents that we follow the predefined heuristic policy, and such policy allows us to unambiguously decode the same optimal first move.

Specifically, we add to $T_s(t)$ the new symbol ⓗ, in addition to the existing contents of $T_s(t)$. This way, the compression step will have more options to choose what symbol to keep from $T_s(t)$ in the compressed string, with a potentially better compression in the end. The heuristic move symbol is able to encode large parts of the graph which are close to the source, and many other parts as well. Let's see a simple example of applying ⓗ.

Example 4.1. *Given nodes s and t we can define a* heuristic move *from s to t, $h(s,t)$ as follows. Let*

$$A = [NW, N, NE, W, \mathit{null}, E, SW, S, SE]$$

be the array of directions indexed from 1 to 9. Let $\sigma x = \text{sign}(t.x - s.x)$, and $\sigma y = \text{sign}(t.y - s.y)$, where $\text{sign}(x)$ is -1 if $x < 0$, 0 if $x = 0$, and $+1$ if $x > 0$; then $h(s,t) = A[\sigma x + 3 \times \sigma y + 5]$. Essentially, the heuristic move is the move that leads us closest to the target, assuming the map is obstacle-free. Consider the gridmap shown in Figure 4.1. For each node t, we show the heuristic move bolded if it appears in $T_s(t)$ for s, that is the set of the optimal first moves sketched in Figure 2.7a. As we can see, almost half of the graph can be encoded using the heuristic moves. Once we add the heuristic move symbol ⓗ to each entry $T_s(t)$ where the move is bolded, we can perform a better run length encoding. The encoding using the new heuristic move symbol is 1W 5E 8W 12E 15W 20E 22ⓗ, which is smaller than without using ⓗ (see in Example 2.4).

W	W	W	W,E	E	E	E
W	W	W	W,E	E	E	E
W	W				E	E
W	W	W	*s*	E	E	E
W,SW	W,SW	SW	S	SE	E,SE	E,SE

(a)

NW	NW	NW	N	NE	NE	NE
NW	NW	NW	N	NE	NE	NE
NW	NW				NE	NE
W	**W**	**W**	*s*	**E**	**E**	**E**
SW	**SW**	**SW**	**S**	**SE**	**SE**	**SE**

(b)

FIGURE 4.1: (a) shows the optimal first-moves to each cell from s . (b) shows the heuristic move to each cell from s. Bold if it appears in $T_s(t)$ for the cell marked s.

4.1.1 Distance Functions

The policy in example 4.1 is quite basic, it takes no account of information about the surroundings of s, and indeed in many cases the direction returned may not even be possible. We can improve the use of the heuristic move symbol ⓗ by considering the relationship between s and t in more detail.

Let s be the source, t be a target, let $f_x(s,t)$ be a predefined heuristic distance function. We assume $f_x(s,t)$ is simple to calculate. For our examples we will use the octile distance function

$$\begin{aligned} f_o(s,t) &= \sqrt{2} \times c + |s.x - t.x| - c + |s.y - t.y| - c \\ &\quad \text{where } c = \min(|s.x - t.x|, |s.y - t.y|) \end{aligned}$$

or the Euclidean distance function

$$f_e(s,t) = \sqrt{(s.x - t.x)^2 + (s.y - t.y)^2}.$$

We define the move chosen by the distance function $f_x(s,t)$ from s to t as the move leaving s to n such that the estimated path distance through n, $w(s,n) + f_x(n,t)$, is minimised, that is

$$F_x(s,t) = \underset{(s,n)\in E}{\operatorname{argmin}}\{w(s,n) + f_x(n,t)\}.$$

$F_x(s,t)$ can be ambiguous when there are multiple edges produce minimum estimation, i.e., $m_1 = (s,n_1), m_2 = (s,n_2)$ such that $w(m_1) + f_x(n_1,t) = w(m_2) + f_x(n_2,t)$. Such ambiguity may introduce error. For example, if m_1 is on the actual optimal path, we can apply ⓗ, but we would have a wrong move if $F_x(s,t) = m_2$ in the decoding stage. In order for ⓗ to be used in the CPD, it must be unambiguous, hence we assume a total order on all moves leaving s, and assume the argmin in F returns the least move in this

Algorithm 6: ⓗ encoding and decoding with $F_x(s,t)$

```
1 Function Encode(s, t):
2     if F_x(s, t) ∈ T_s[t]) then
3         T_s[t] = T_s[t] ∪ ⓗ;
4 Function Decode(m, s, t):
5     if m = ⓗ then
6         return F_x(s, t);
7     else
8         return m;
```

W	**W**	**W**	**E**	**E**	**E**	**E**
W	**W**	**W**	**E**	**E**	**E**	**E**
W	**W**				**E**	**E**
W	**W**	**W**	*s*	**E**	**E**	**E**
SW	**SW**	**SW**	**S**	**SE**	**SE**	**SE**

FIGURE 4.2: The heuristic first move $F_o(s,t)$ to each cell of the gridmap from the cell marked s. All symbols are in bold, since each of them belongs to the corresponding set $T_s(t)$ for the marked symbol s.

order that leads to the minimal distance. For gridmaps our default ordering is NE, NW, SE, SW, N, S, E, W. Effectively this tries to take diagonal moves first when there is a tie.

The move chosen by the distance function is immediately better than the obstacle-free approach in example 4.1 because it can never choose invalid grid moves. Just as before, if $F_x(s,t)$ returns a first move that corresponds with the start of the shortest path from s to t, then we can add the symbol ⓗ to the set $T_s(t)$ of possible first moves for s. See details in Algorithm 6.

Example 4.2. *Consider the gridmap in Figure 2.7a. Figure 4.2 shows the heuristic move to each node in the graph underlying the gridmap. Since for each target node t, $F_o(s,t) \in T_s(t)$ for s (using octile distance f_o), we can add a heuristic move symbol ⓗ to each entry in $T_s(t)$. Now when we perform run length encoding we get the result 1ⓗ. That is, the entire compressed string has only 1 run, which is substantially shorter than options discussed in Examples 4.1.*

The revised CPD lookup function simply looks up the CPD as usual but, if it finds a symbol ⓗ, it returns the heuristic move $F_x(s,t)$, for whichever version we choose to use.

s	E	E	E	E	E
S	SE	SE	■	SE	SE
S	SE	■	SE	SE	SE
S	SE	SE	SE	SE	SE
S	SE	SE	SE	SE	SE

(a)

s	**E**	**E**	**E**	**E**	**E**
S	**SE**	E,**SE**	■	E	E
S	S,**SE**	■	S,**SE**	E	E
S	S,**SE**	S,**SE**	S,**SE**	S,**SE**	S,**SE**
S	S,**SE**	S,**SE**	S,**SE**	S,**SE**	S,**SE**

(b)

FIGURE 4.3: (a) The heuristic first move $F_o(s,t)$ to each cell of the gridmap from the cell marked s and (b) the optimal first moves to each cell from s, with the heuristic moves in bold.

4.1.2 Improving the Tie-breaking

The heuristic move must be unambiguous, hence when there are moves that look equally good we must choose one of them unambiguously. This choice can be different for each source node s. While the diagonal first ordering is a good default, we can make use of the relative positions of s and t to improve the tie breaking.

Example 4.3. *Consider the gridmap in Figure 4.3. We show the heuristic move $F_o(s,t)$ to each cell of the gridmap in Figure 4.3(a), and the optimal first moves in Figure 4.3(b). Clearly all nodes can make use of a heuristic move, except those marked E on the second and third row. The resulting encoding is 1ⓗ 12E 13ⓗ 17E 19ⓗ. This pattern is common where some part of the graph is blocked. We want to have a better tie breaker for such circumstances.*

When we have blockages in the graph it can often be the case that the heuristic best move is not the diagonal move. We improve our heuristic tie breaking by breaking the graph into 8 quadrants around s, corresponding to the 8 directions, and break ties by choosing the closest direction to the straight line direction from s to t. In order to avoid complex trigonometric operations we use simple approximations: if $|s.x - t.x| \geq 2|s.y - t.y|$ then the straight line from s to t is close to the horizontal line, so we favour E or W, similarly if $|s.y - t.y| \geq 2|s.x - t.x|$ then the straight line from s to t is close to vertical so we favour N or S. In other cases we favour the closest diagonal direction. We denote the heuristic move with directional tie breaking by $F_x^d(s,t)$ where x denotes the distance function.

Example 4.4. *Consider the gridmap in Figure 4.3 once more. Using direction-based tie breaking the heuristic moves are given in Figure 4.4. Note that now each heuristic move appears in the optimal first moves (Figure 4.3(b)), and hence we can encode all the first moves as 1ⓗ, that is much shorter than without using directional-based tie breaking.*

s	E	E	E	E	E
S	SE	E	■	E	E
S	S	■	SE	E	E
S	S	SE	SE	SE	SE
S	S	S	SE	SE	SE

FIGURE 4.4: The heuristic first move $F_o^d(s,t)$ to each cell of the gridmap from the cell marked s using directional tie breaking.

4.2 Proximity Wildcards

In the previous section, we have used heuristic moves to introduce redundant symbols in CPDs, for an improved compression. In this section we further exploit the potential of heuristic moves by focusing on proximity areas around source nodes s. Often, within an open area around a node s, all the nodes will be optimally reachable using the heuristic move as a first move. To take advantage of this we introduce a new level of compression, using *proximity wildcards*.

Definition 4.2. Given a node s and a function $F_x(s,t)$, the *proximity distance* $\texttt{pd}(s)$ is the lowest value $d \in \mathbb{N}$ such that there exists a cell n for which $|s.x - t.x| \leq d+1$ or $|s.y - t.y| \leq d+1$, and $F_x(s,t) \notin T_s(t)$.

In other words, for all cells t such that $|s.x - t.x| \leq \texttt{pd}(s)$ and $|s.y - t.y| \leq \texttt{pd}(s)$, we have that ⓗ $\in T_s(t)$ for s. Generally speaking, any function that describes a geometric shape can be utilized. For simplicity, in the rest of the chapter the proximity wildcards are determined by using $F_x = f_o$, i.e., using the octile distance function.

Definition 4.3. The *proximity square* of a node s is the square centred in s and with the edge size equal to $2 \cdot \texttt{pd}(s) + 1$.

That is, the proximity square is the largest square centred on s such that the cells of the gridmap inside the square can optimally be reached from s by the heuristic move ⓗ.[1]

We can then replace the entries in T for the nodes inside a proximity square with a wildcard symbol "$*$", and use the distance $\texttt{pd}(s)$ instead to determine the move for nodes n within the proximity square. Wildcards obtained this way are called proximity wildcards. To be able to use proximity wildcards in the CPD, we change the lookup function to always use the heuristic move when the target is within the proximity square of s.

[1] We can keep the definition simple and still use a square in those cases when s is close to the border of the gridmap, by simply allowing the proximity square to partly go outside the gridmap.

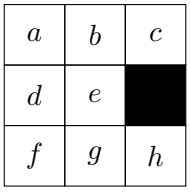


FIGURE 4.5: A small gridmap to illustrate proximity wildcards.

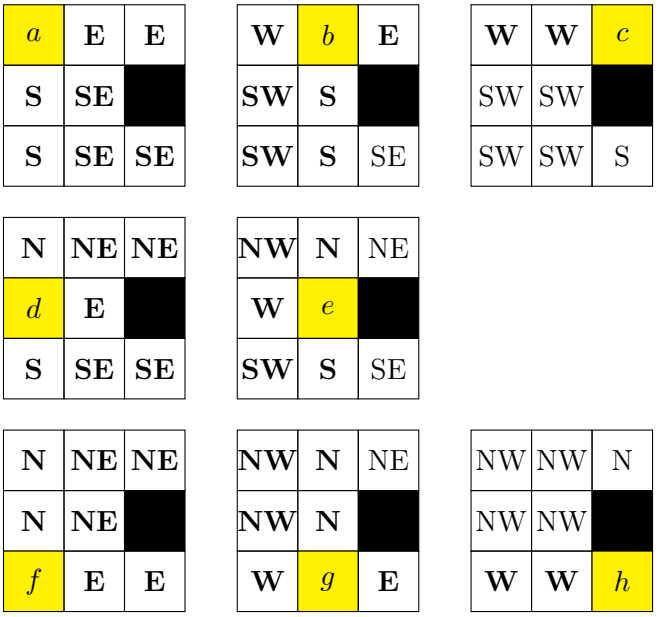


FIGURE 4.6: Heuristic moves, for each source cell. Source cells are shown in yellow. In each case, heuristic moves that coincide with optimal moves are shown in bold.

Node s	a	b	c	d	e	f	g	h
$\texttt{pd}(s)$	2	1	0	2	0	2	1	0

TABLE 4.1: Proximity distances for the nodes in the gridmap in Figure 4.5.

Example 4.5. *Consider the gridmap shown in Figure 4.5, with 8 cells labelled from a to h. Figure 4.6 illustrates, for each source cell (node), the areas on the map where heuristic moves coincide with optimal moves. As mentioned, heuristic moves are computed with* $f_o(s,t)$.

Using Definitions 4.2 and 4.3, it follows that, in Figure 4.6, the proximity square of a source node may contain the source cell, cells in bold, obstacles, and areas outside the map. The proximity square cannot contain (white) cells in normal font. This remark allows us to calculate, for each source node s*, the value* $\texttt{pd}(s)$*. We show the values in Table 4.1.*

So far we have computed the proximity squares of each node. Next we illustrate how these are used to introduce proximity wildcards in the first-move table. Table 4.2 shows

	a	b	c	d	e		f	g	h
a	$*$	ⓗ, E	ⓗ, E	ⓗ, S	ⓗ, SE	$*$	ⓗ, S	ⓗ, S, SE	ⓗ, S, SE
b	ⓗ, W	$*$	ⓗ, E	ⓗ, SW	ⓗ, S	$*$	ⓗ, S, SW	ⓗ, S	S
c	ⓗ, W	ⓗ, W	$*$	W	W	$*$	W	W	W
d	ⓗ, N	ⓗ, NE	ⓗ, NE	$*$	ⓗ, E	$*$	ⓗ, S	ⓗ, SE	ⓗ, SE
e	ⓗ, NW	ⓗ, N	N	ⓗ, W	$*$	$*$	ⓗ, SW	ⓗ, S	S
f	ⓗ, N	ⓗ, N, NE	ⓗ, N, NE	ⓗ, N	ⓗ, NE	$*$	$*$	ⓗ, E	ⓗ, E
g	ⓗ, N, NW	ⓗ, N	N	ⓗ, NW	ⓗ, N	$*$	ⓗ, W	$*$	ⓗ, E
h	W	W	W	W	W	$*$	ⓗ, W	ⓗ, W	$*$

TABLE 4.2: First move table with no proximity wildcards, for the example shown in Figure 4.5. In each cell, we show all the optimal moves, and the ⓗ symbol, if the heuristic move coincides with an optimal move.

	a	b	c	d	e		f	g	h
a	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
b	$*$	$*$	$*$	$*$	$*$	$*$	ⓗ, S, SW	ⓗ, S	S
c	ⓗ, W	ⓗ, W	$*$	W	W	$*$	W	W	W
d	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
e	ⓗ, NW	ⓗ, N	N	ⓗ, W	$*$	$*$	ⓗ, SW	ⓗ, S	S
f	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
g	ⓗ, N, NW	ⓗ, N	N	$*$	$*$	$*$	$*$	$*$	$*$
h	W	W	W	W	W	$*$	ⓗ, W	ⓗ, W	$*$

TABLE 4.3: First move table with proximity wildcards.

the first-move table of our running example before adding the proximity wildcards. This is compressed into 1ⓗ (first row); 1ⓗ 9S (second row); 1W (third row); 1ⓗ (fourth row); 1ⓗ 3N 4ⓗ 9S (fifth row); 1ⓗ (row 6); 1N 4ⓗ (row 7); 1W (last row). This sums up to 13 RLE runs.

Then, for each row, we replace all entries within the proximity square of the source node at hand with wildcards, since we do not need to use the CPD to look up their first move. The resulting first-move table is shown in Table 4.3. The resulting run length encodings are: 1; 1S; 1W; 1*; 1ⓗ 3N 4ⓗ 9S; 1*; 1N; and 1W. This gives a total of 11 RLE runs. This concludes our example.*

The revised CPD lookup function using proximity squares is shown in Algorithm 7. The first advantage is that for lookups close to the source s we may not need to perform logarithmic run length decoding. The second advantage is that since many more wildcards are added to the first move array, the run length encoding can be much smaller.

4.3 Experiment

We run experiments on two sets of maps: *Dragon Age: Origins (DAO)* and a set of 9 large maps from three different games. For the first benchmark, we used 155 maps from the game *Dragon Age: Origins* with the number of nodes ranging from about 2,000

Algorithm 7: Get next move algorithm for CPDs using proximity wildcards and heuristic moves.

```
1 d ← pd(s)
2 if |s.x − t.x| ≤ d ∧ |s.y − t.y| ≤ d then
3 |   return F(s, t)
4 else
5 |   m ← CPD(s,t)
6 |   return Decode(m, s, t);
```

to 100,000. For the second benchmark we used 3 maps from *Dragon Age: Origins*, 3 from *Starcraft* and 3 form *Baldur's Gate II* with a number of nodes ranging from about 100,000 to 300,000.

As a baseline we compare against *Single Row Compression* (*SRC*) with DFS ordering [13], the fastest optimal solver at the last edition of GPPC [5]. We further test three variants of *SRC, each enhanced with additional compression. These algorithms are denoted as follows:*

- Algorithm *h*, which implements the heuristic symbol enhancements described in Section 4.1;
- Algorithm *w*, which implements proximity wildcards, described in Section 4.2, combined with bidirectional wildcards [15];
- Algorithm *hw*, which is the combination of the previous two methods, *h* and *w*.

	mean	std	min	25%	50%	75%	max
SRC	6.52	11.2	0.011	0.41	2.37	7.25	68.6
h	1.85	3.82	**0.003**	0.09	0.70	1.87	29.6
w	1.93	3.54	0.005	0.12	0.74	2.04	24.3
hw	**1.48**	**2.82**	0.004	**0.08**	**0.52**	**1.57**	**21.7**

TABLE 4.4: Size of CPDs of all maps from DAO (MB).

In the comparison, we examine the improved size, the preprocessing time and the path extraction time.

As mentioned, in our experiments, inside setting *w* we have implemented our proximity wildcards on top of a different type of wildcards from the literature, namely bidirectional wildcards [15]. The relation is that both types of wildcards replace actual move symbols with don't care symbols, improving the compression. The difference is that they are computed significantly differently, exploiting different properties of the problem. See Section 2.4.2 for a brief description of bidirectional wildcards. Given the relation between

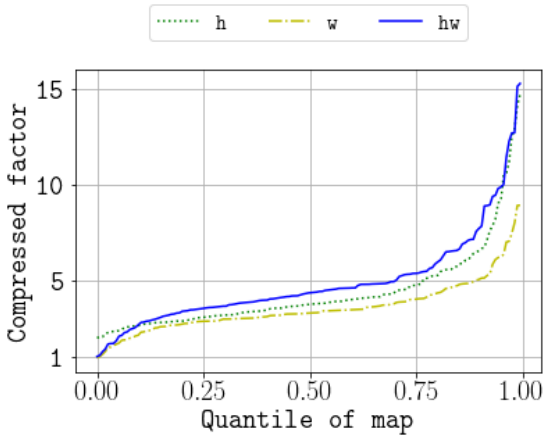


FIGURE 4.7: Distribution of the compression factor in the maps from DAO. The compression factor is the CPD size produced by *SRC* enhanced with the setting at hand (h, w, hw) divided by the CPD size produced by *SRC*. Maps are ordered by the *SRC CPD size*.

both types of wildcards, we wanted to evaluate how they work together and what are the gains of proximity wildcards on top of bidirectional wildcards.

All algorithms are implemented in C++ and compiled with *clang-902.0.39.1* using *-O3* flag, under *x86_ 64-apple-darwin17.5.0* platform with *2.5 GHz Intel Core i7 Processor and 16 GB 1600 MHz DDR3 Memory*. All benchmarks[2] and our implementations[3] are available online.

4.3.1 Preprocessing Results

From Table 4.4, we can notice that for the maps from DAO the sizes of CPDs are quite small. Figure 4.7 shows the distribution of the compression factor of the proposed methods. From the plot we can see that both proposed methods achieve better compression in 99% of maps, and hw tends to dominate other methods in most of maps. We also notice w and hw produce larger size ($\approx 7\%$) in 1% of maps. The reason is that these maps are small, while w needs to compute and store extra information (the proximity distances), and such an extra cost can sometimes be larger than the benefit in such cases.

On the DAO maps, a closer look inside the w setting reveals the following: CPDs computed with SRC [13] enhanced with bidirectional wildcards [15] have a minimum size of 0.006 MB, an average size of 2.292 MB and a maximum size of 26.526 MB. I.e., the usage of bidirectional wildcards leads to a reduction of the size of the CPDs by 41% to 86%, with an average size reduction of 61%. Neverthless, adding proximity wildcards on top of this (i.e., running the w setting) is even stronger: w further improves the size of

[2]https://movingai.com/benchmarks/grids.html
[3]https://github.com/eggeek/CPD-Hsymbol-Wildcard

Maps	#cells	The size of the CPD in MB *SRC*	*h*	*w*	*hw*
AR0044SR	231,469	507.1	14.3	34.8	**9.1**
AR0605SR	140,922	179.4	25.4	23.0	**14.2**
AR0700SR	131,852	69.0	37.8	23.3	**20.3**
Aftershock	166,076	88.0	8.9	14.7	**7.6**
DarkContinent	285,669	213.8	70.5	50.6	**40.4**
TheatreofWar	220,816	170.1	53.9	42.2	**36.6**
hrt000d	106,608	60.8	11.0	11.4	**6.8**
ost000a	130,478	44.1	15.9	13.4	**9.7**
ost000t	105,707	38.0	13.4	11.2	**8.0**

Maps	The number of the RLE runs $\times 10^3$ *SRC*	*h*	*w*	*hw*
AR0044SR	126,086	2,887	7,549	**1,128**
AR0605SR	44,418	5,917	5,042	**2,854**
AR0700SR	16,856	9,050	5,166	**4,422**
Aftershock	21,512	1,730	2,846	**1,057**
DarkContinent	52,594	16,767	11,226	**8,678**
TheatreofWar	41,852	12,803	9,437	**8,045**
hrt000d	14,878	2,418	2,325	**1,175**
ost000a	10,644	3,574	2,695	**1,763**
ost000t	9,191	3,036	2,267	**1,460**

TABLE 4.5: CPD size statistics for the large maps used in experiments. Top: the number of cells per map, and the CPD size in MB. Bottom: the number of RLE runs per map in thousands.

the CPDs, compared to SRC with bidirectional wildcards, by 6.33% to 43.68%, to an average improvement of 24.86%.

We also examine the compression performance on larger maps, with results shown in Table 4.5. Notice that even in larger maps, the final CPD size produced by our methods are still small. For example, map *AR0044SR* has 231469 nodes; an uncompressed first move matrix needs 231469^2 Bytes $\approx$ 53.5GB; for this map, *SRC* requires 507 MB, while *hw* requires only 9.1MB, which is more than 50 times smaller. In fact, *h* alone leads to a reduction by a factor of 25.46 on this map.

Figure 4.8 shows the rate between the preprocessing time of *SRC using methods h, w, hw and the preprocessing time of SRC* for all maps (including the DAO set and the large maps). From the figure we can see that, on preprocessing, *h*, *w*, and *hw* require more time than *SRC*. The worst-case preprocessing slow-down does not exceed a factor of 4. In 80% of the cases, the preprocessing slow-down does not exceed a factor of 1.5.

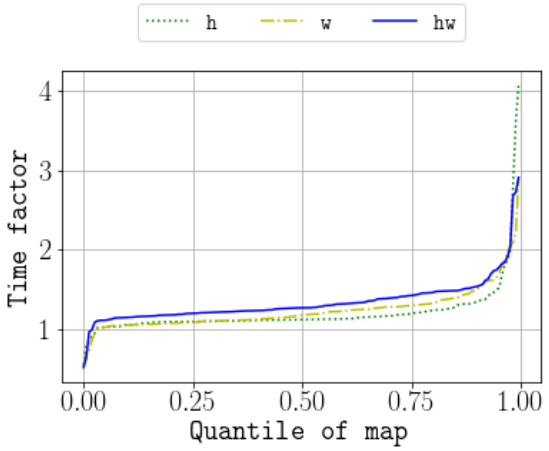


FIGURE 4.8: Distribution of the preprocessing-time factor over all maps. The preprocessing-time factor is the preprocessing time cost of the proposed algorithm at hand (h, w, hw) divided by the preprocessing time cost of *SRC*. Maps are ordered by *SRC size*.

	Full path extraction					
	std dev.	min	25%	50%	75%	max
h	0.43	0.23	1.03	1.16	1.33	8.44
w	0.53	0.28	1.10	1.28	1.52	9.68
hw	0.64	0.22	1.03	1.24	1.50	11.05

TABLE 4.6: Speed up factor when extracting a full path measured as the time cost of *SRC* divided by the time cost of the proposed algorithms.

#op	mean	std	min	25%	50%	75%	max
l_{src}	382.6	383.0	0.0	109	268	525	2563
c_{src}	0.00	0.00	0.0	0.0	0.0	0.0	0.0
l_h	382.6	383.0	0.0	109	268	525	2563
c_h	0.00	0.00	0.0	0.0	0.0	0.0	0.0
l_w	365.6	383.9	0.0	91	251	508	2563
c_w	17.04	22.49	0.0	5.0	11.0	22.0	428.0
l_{hw}	368.5	385.2	0.0	93	254	510	2563
c_{hw}	14.14	20.75	0.0	2.0	9.0	18.0	428.0

TABLE 4.7: Number of operations in path extraction. Here l represents the number of binary searches, and c represents the number of times the target is inside the source's proximity square.

4.3.2 Path Extraction Results

Our proposed methods achieve impressive compression performance without paying any cost in path extraction. Table 4.6 shows the speed up factor for the full path extraction. From the table, we can see that the proposed algorithms tend to be faster and data shown in Table 4.7 helps understand why: i) h runs the same number of binary searches as *SRC*, but the former has smaller CPD size, so that the binary search is faster; also, smaller CPDs could improve the rate of cache hits; ii) using w if a target is inside the source's proximity square can avoid looking for the optimal first move in the CPD, via a binary search, and compute it directly in constant time instead; iii) hw can benefit from both.

4.4 Conclusion

In this chapter, we have shown how to substantially reduce the size required to store compressed path databases (CPDs) for a grid map, by using a heuristic move symbol, and introducing wildcard symbols which we store for nodes in the proximity to the source. Using these methods together we can reduce the size of a CPD significantly. The reduction in CPD size also leads to an improvement in lookup time, even though the lookup function is slightly more complex. With these improvements we define a new state of the art for CPDs.

While this chapter has focused on grid maps, most of what we have talked about can be extended to more general graphs. Given a distance function we can apply the distance function approach to any graph. We can extend directional tie-breaking to any graph embedded in the Euclidean plane. Proximity wildcards can also be extended to this case by discovering the largest radius within which the move to all target nodes follows the distance function heuristic.

Generally speaking, instead of storing all pairs of the first move, the CPD can store the first k moves, thereby reducing the number of lookups by k in the path extraction. However, the number of symbols increases quadratically, which may negatively impact the compression rate of RLE. For example, when $k = 1$, at least 8 symbols are required to encode all possible moves, and when $k = 2$, 24 symbols are needed. As a result, in practice, only the first move is stored.

Interesting directions for the future work include extending the area of the proximity wildcards (e.g., with rectangles instead of squares), evaluating the effectiveness of the proposed techniques for the multi-row compression of the CPD [13], and investigating the usage of CPDs and heuristic moves on road maps.

Chapter 5

Bounded Suboptimal Path Planning with CPD

The principal advantage of CPDs is speed: optimal paths can be found quickly and without any state-space search. Furthermore, optimal prefixes can be found faster still. This feature is important to reduce the *first-move lag*, where an agent needs to wait until it knows in which direction to start moving. By contrast, consider that search-based methods can only know an optimal prefix once they know the entire path. The main drawback of CPDs is build cost: each database requires an all-pairs precompute with space and time being worst-case quadratic in the size of the input graph; i.e., we need to run one Dijkstra search per node in the graph and then compress and store the result. Researchers in this area prioritize fast online performance and for this reason works tend to focus on reducing the size of the database so that it better fits in working memory [1, 13, 15], or they focus on improving the lookup performance, so that moves can be extracted faster still [10]. These efforts have improved CPD performance, for offline storage and online performance, by over one order of magnitude beyond the 2014 GPPC baseline. Despite these gains the preprocessing time required to construct each database has not substantially reduced and this aspect can be prohibitive.

In this chapter we consider a new bounded-suboptimal take on CPDs which dramatically cuts the time required for precomputation. The approach can also reduce storage costs and yield faster online performance – all in exchange for a small amount of additional cost per extracted path. Our idea involves selecting from the graph, induced from the input map, a subset of nodes C, which we call *centroids*, such that every node is at most a path distance δ from some centroid $c \in C$. During precomputation we store first move data from every node in the graph to every centroid. The entire procedure requires at most $|C|$ offline instantiations of Dijkstra search. More, the resulting database is sufficient to

determine a bounded suboptimal path, from any node s to any other node t. We give a theoretical description of the method and show that, for a given value of δ, the cost of each extracted path is at most 2δ larger than optimal, with δ being an input parameter.

In experiments, we test this idea with different δ and with different compression schemes. Results on standard grid benchmarks indicate order-of-magnitude reductions in preprocessing times, and up to several factors improvement in database size and lookup speed. We also show that, in most cases, the suboptimality cost per path is substantially smaller than the guaranteed upperbound, and the relative suboptimality is usually small.

5.1 Suboptimal CPDs

For some application settings, it is possible that the space and time required to build a CPD are considered too large. To address such situations, we propose to compute and compress first-move data for only a subset of grid nodes C, called *centroids*.

We associate the nearest centroid $c(t) \in C$ for every grid node t. Then, a path from any s to any t, which we call a *centroid path* (*cp* for short), can be built as:

$$cp(s,t) = cpd(s,c(t)) {+\!\!+} rev(cpd(t,c(t)).$$

We can do better by joining the two paths at the first common point p, which may not be $c(t)$:

$$\begin{aligned} cp(s,t) \quad = \quad & [n \,|\, n \in cpd(s,c(t)), n \notin cpd(t,c(t))] \\ & {+\!\!+} [p] \\ & {+\!\!+} [n \,|\, n \in rev(cpd(t,c(t)), n \notin cpd(s,c(t))], \end{aligned}$$

where p is the first node in $cpd(s,c(t))$ and also in $cpd(t,c(t))$.

Given we have chosen centroids so that no target t is more than δ from its centroid $c(t)$, we can show that the centroid path is never longer than 2δ than the optimal path.

Theorem 5.1. *If* $len(sp(t,c(t)) \leq \delta$ *for all* t *in* V*, then* $len(cp(s,t)) \leq len(sp(s,t)) + 2\delta$.

Proof. Suppose for the purpose of a contradiction that we have $len(cp(s,t)) > len(sp(s,t)) + 2\delta$. By definition, the CPD returns a path $cpd(s,c(t))$ which is the shortest possible path, and similarly for $cpd(t,c(t))$, hence we have $len(sp(s,c(t)) + len(sp(t,c(t)) \geq len(cp(s,t))$. Consider the path $sp(s,t){+\!\!+}sp(t,c(t))$ from s to $c(t)$. By assumption, its length is at most $len(sp(s,t))+\delta$, and hence we have that $len(sp(s,t)) + \delta \geq len(sp(s,c(t)))$. Finally,

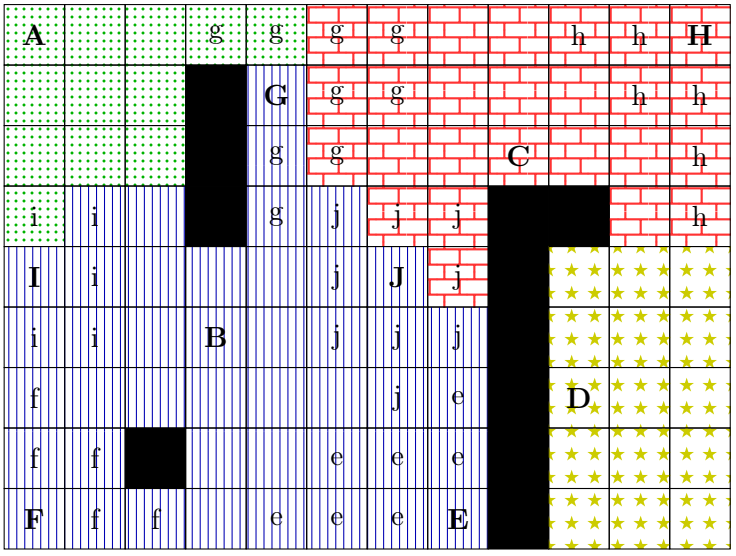


FIGURE 5.1: Centroid marking for a grid map with $\delta = 3$. **A**, **B**, **C** and **D** are centroids in the first stage, regions explored from them are filled with green dots (A), blue vertical lines (B), red bricks (C) and yellow starts (D). Centroids **E**–**J** are created in the second stage, and their corresponding regions are shown with lowercase letters.

we also have that $\delta \geq len(sp(t, c(t))$. This leads to the reasoning

$$\begin{aligned} & len(sp(s, c(t)) + len(sp(t, c(t)) \\ \geq\ & len(cp(s, t)) \\ >\ & len(sp(s, t)) + 2\delta \\ \geq\ & len(sp(s, c(t))) + \delta \\ \geq\ & len(sp(s, c(t))) + len(sp(t, c(t)) \end{aligned}$$

Contradiction. □

Building Centroids Choosing a minimum number of centroids to cover all grid nodes for a given radius δ is a NP-hard problem. In this section, we propose the following algorithm which guarantees that there may be at most $\frac{2\sqrt{2}|V|}{\delta}$ centroids in each connected component, where $|V|$ is the number of traversable cells in the component.

The algorithm (Algorithm 8) has two stages: the first stage makes the distance of the furthest node to a centroid in $[\delta, 2\delta]$; the second stage makes the distance of the furthest node to a centroid lower than or equal to δ. To achieve this, we need to prioritize all grid nodes based on two criteria:

- d_c: the shortest distance to the nearest centroid, which are ∞ at the beginning and get updated during the algorithm.

- d_o: the shortest distance to the nearest obstacle, which can be precomputed by flood-fill.

In the first stage, we keep selecting node which $d_c > 2\delta$, with **minimum** d_o, and **minimum** d_c if there is a tie. For each selected node v, we run a Dijkstra search to explore neighbors of v within $2\delta + 1$ and update their d_c. In the second stage, we keep selecting node which $d_c > \delta$, with **maximum** d_c, and **minimum** d_o if there is a tie. For each selected node v, similarly to the first stage, we run a Dijkstra search to explore the neighbors of v within δ and update their d_c. Notice that, in the first stage, selecting the node with a minimum d_o is to choose centroids from the "border" to the "center" of the gridmap; selecting the node with a minimum d_c is to make a new centroid around the existing centroid regions. The motivation for such a strategy is to make more border cells covered by centroids in the first stage. We recognize there can be many other ways of selecting centroids. For this work, we choose this strategy as our preliminary investigation showed that it performs well.

Theorem 5.2. *Assume that the gridmap induces a graph with only one connected component, and let $|V|$ be the number of traversable cells where $|V| \gg \delta$, then the Algorithm 8 creates at most $\frac{2\sqrt{2}|V|}{\delta}$ centroids.*

Proof. After the first stage, because we always select a centroid with $d_c > 2\delta$, for each centroid c, there is a grid node v covered by c such that $d_c[v] \geq \delta$. After the second stage, because we always select centroid with $d_c > \delta$, for each centroid c, there is a grid node v covered by c such that $d_c[v] \geq \frac{\delta}{2}$, which consists with at least $\frac{\delta}{2\sqrt{2}}$ moves - assuming all moves are diagonal, so each centroid covers at least $\frac{\delta}{2\sqrt{2}}$ nodes, thus there are at most $\frac{2\sqrt{2}|V|}{\delta}$ centroids. □

5.1.1 Reverse Compressed Path Databases

A standard CPD compresses first-move matrix *rows* (i.e., an array of moves from one source to many targets). We call this *forward CPDs*. In this section we introduce CPDs that compress *columns* (i.e., arrays with moves from many sources to one target), we call these *reverse CPDs*.

Forward and reverse CPDs have different compression rate. Forward CPDs are less affected by column ordering compared to the reverse CPDs. Specifically, in a forward CPD, targets (columns) with the same first-move for a given source (a row) tend to be close to each other spatially. In contrast, in a reverse CPD, the location of sources (columns) with the same first-move for a given target (a row) depends on topology of the

Algorithm 8: Centroid creation Centroids(V,G) given a graph (V,G).

```
1  Calculate d_o[v], the shortest distance from vertex v to an obstacle by flood fill in;
2  d_c[v] ← ∞, v ∈ V;
3  c[v] ← ⊥, v ∈ V;
4  Q ← V;
5  while Q ≠ ∅ do
6      v ← argmin{(d_o[v], d_c[v])|v ∈ Q};
7      Q ← Q − {v};
8      if d_c[v] > 2δ then
9          for v' ∈ V where d = len(sp(v, v')) ≤ 2δ + 1 do
10             if d < d_c[v'] then
11                 d_c[v'] ← d;
12                 if d ≤ δ then
13                     c[v'] ← v
14 Q ← V;
15 while Q ≠ ∅ do
16     v ← argmax{(d_c[v], −d_o[v])|v ∈ Q};
17     Q ← Q − {v};
18     if d_c[v] > δ then
19         for v' ∈ V where d = len(sp(v, v')) ≤ δ do
20             if d < d_c[v'] then
21                 d_c[v'] ← d;
22                 c[v'] ← v
```

graph (e.g., along the path in the same direction). Comparing to forward CPDs, reverse CPDs have following advantages: (a) Reverse CPDs can achieve better size savings than the forward ones for certain (but not all) δ values; (b) Reverse CPDs return moves faster, due to a combination of better caching and the ability to return several moves with a single lookup. Both forward and reverse CPDs enjoy a dramatic cut in the preprocessing time, as δ grows.

Reverse CPDs can be computed with only small adjustments to Dijkstra's algorithm. Specifically, we run a Dijkstra search from the target, and use the results to extract moves from each node towards the target at hand. Reverse CPDs are compatible with heuristic moves (Section 4.1) and other CPD improving techniques.

Example 5.1. *Figure 5.2 shows reverse first-move data that we compress to 1S 3SW 5SE 6S 10W 12E 13S 15SW 16S 21SE 22E 26W 29NE 32N 33NW, which is 15 runs. Our encoding assumes a left-right-top-bottom cell ordering. Adding h-moves reduces this to 1S 3SW 5SE 6S 10W 12E 13ⓗ, which is 7 runs. For comparison, in the forward direction the first-move data for source t can be encoded with just a single run: 1ⓗ.*

S,**SE**	S	SW	SW,SE	SE	S	S,**SW**
S,**SE**	**S**	W	**W**,E	E	**S**	S,**SW**
SW	**S**				**S**	**SE**
E	**E**	**E**	*t*	**W**	**W**	**W**
E,**NE**	E,**NE**	**NE**	**N**	**NW**	W,**NW**	W,**NW**

FIGURE 5.2: Optimal first moves from each grid cell to the cell marked t. Moves in bold agree with the default heuristic.

t		S	S,SW	S,SW	SW
N		S	S,SW	SW	W,SW
N		S	SW	W,SW	W,SW
N	**W**	W	W	W	W

FIGURE 5.3: Optimal first moves from each grid cell to the cell marked t. Moves in bold agree with the default heuristic.

	SE	S,SE	S
	S,SE, **E**	SE	S
		S,SE, **E**	S
			S,SW
t	**W**,NW	**W**,NW	**W**,NW

FIGURE 5.4: Optimal first moves toward the grid cell t with added "illegal" move symbols for cells adjacent to obstacles.

"Illegal" Moves One way to improve the run length encoding is allowing "illegal" moves to be part of the set of possible moves. More specific, assume the current node is x, we consider all 8 neighbors of x in a 3 by 3 square, a move from x to $t \in neighbors(x)$ is illegal if:

1. t is a blocked cell;
2. t is cut by a corner;
3. all traversable neighbors of t are also traversable neighbors of x.

In both cases "illegal" moves are useful **only if** they can be unambiguously *decoded* by some function to extract a correct optimal move. One possible *decoder* can be finding the first legal move in a clockwise direction from N. The Example 5.2 illustrates this idea in the first case, and the same reasoning can also be applied on the second case, the Example 5.3 illustrates the third case. Note that heuristic symbol is not applied in these examples for the convenience.

Example 5.2. *Figure 5.3 shows reverse first-move data where S moves appear as the only symbols of three cells in column 3. These symbols force us to create one additional run per row. To improve compression we will store at each of these locations an additional*

Algorithm 9: The function $\mathsf{FirstMove}^r(s,\, t)$ for reverse CPDs with heuristic moves and "illegal" moves.

```
1 m ← FirstMove(s,t)
2 if m = ⓗ then
3 |  m = F_x^d(s,t)
4 if ¬legal(m,s,t) then
5 |  m = closest(m, {mv|mv ∈ MV, legal(mv,s,t)})
6 return m
```

symbol, SW, which can be used to reduce the database by three runs. Although following this move produces an obstacle we can still proceed if we detect the situation at runtime. To decode the SW symbol we choose the closest move to SW which is not blocked: here, S. The definition of "illegal" move guarantees that our decoding function always returns an optimal first move.

Example 5.3. *Consider the example in Figure 5.4. We show a reverse first-move table which already includes some "illegal" moves; i.e., those that produce obstacles and which can be appropriately decoded.*

Further compression here is hampered by the two tiles with SE-only moves. Notice however that while the S move from each of these locations is valid it is not helpful. *The only reason to apply move S from these positions is to reach the cell directly below and no other. If we identify such unhelpful moves at compression time, we can add S to the set of symbols for the currently SE-only tiles. When extracted at runtime, the "illegal" move S can be decoded to SE. With this enhancement the table can compress into two runs: 1S 18W.*

Preprocessing and Path Extraction Building a reverse CPD for target t can be performed by a single Dijsktra search from the target and recording for each grid node s all best moves for reaching t. Compression proceeds as usual. In the experimental section, we will show that build times for reverse CPDs and forward CPDs using the same set of centroids C are similar.

Algorithm 9 gives a modified version of $\mathsf{FirstMove}(s,\, t)$; i.e., the CPD first move extraction function which is here modified for reverse CPDs and which decodes both "illegal" and ⓗ. This algorithm extracts a first move m as usual, and, if this move is not legal, it finds the closest move that is legal and returns that instead. Function $legal(m, s, t)$ returns *false* if moving in direction m from s: (i) leads to a blocked square, or (ii) cuts a corner, or (iii) moves to a position $m(s) \neq t$ where all successors are better reached directly from s. In all other cases $legal(m, s, t)$ returns *true*. Function $closest(m, S)$ returns the closest

move to m in set S breaking ties by moving clockwise, e.g., $closest(\text{SW}, \{\text{N}, \text{NE}, \text{E}\}) = \text{N}$ and $closest(\text{S}, \{\text{N}, \text{NE}, \text{E}\}) = \text{E}$. Note that forward CPD can also store "illegal" moves, but only get a slight improvement.

Heuristic Path Extraction Paths extracted with centroid CPDs, whether forward or reverse, can experience a pathological worst-case where the cost of all suboptimal detour can be substantially larger than the cost of the optimal path. Such cases occur when the start and target position are: (i) in close proximity; (ii) when there exist multiple paths from s to the centroid $c(t)$ and; (iii) when the CPD stores the "wrong" path from s to $c(t)$ for a specific and pathological choice of t. Figure 5.5 gives an example, the path getting from the CPD from s to t via the centroid $c(t)$ is $[s, p, c(t), t]$, while the optimal path is $[s, t]$. Notice that the relative suboptimality can be large in a relative sense, but it is also local to the centroid. Thus we can often improve the path computation by trying a direct heuristic path, from the current node to the target, when we arrive close to the centroid. Our approach is as follows.

When the current position p in the path has the same centroid as the target $c(p) = c(t)$ we check whether following the default heuristic function $F_x^d(p, t)$ repeatedly leads us to the target, and instead use this path. For the example of Figure 5.5 starting from s and following the moves derived from the CPD we hit the region of centroid $c(t)$ at position p and find the direct path straight west to the target. We extract the direct path of length 2 by omitting overlapping paths from s to p and back.

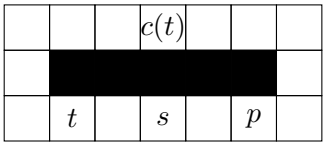


FIGURE 5.5: Worst case behaviour for centroid CPDs when $\delta = 6$: the symbol stored for FirstMove(s,$c(t)$) is E rather than the equivalent W. The path found from s to t is length 14 rather than 2.

5.2 Experiments on Suboptimal CPDs

We run experiments on maps from the GPPC 2014 benchmarks [5], including 105 game maps and 6 artificial maps - 2 out of 9 maps each in *mazes, rooms, random.* Although all forward CPDs and most of reverse-centroid CPDs work on the rest of the 21 artificial maps, for small values of δ the size of the reverse CPD can be prohibitively large for these adversarial environments. To avoid missing values we choose to exclude these maps. All

	δ	0		2		4		8		16		32		64	
map	stat	F	R	F	R	F	R	F	R	F	R	F	R	F	R
Aurora	C	493772		93346		22078		6777		2278		837		372	
	T	551.3	576.0	104.2	108.9	24.6	25.7	7.5	7.9	2.5	2.6	0.9	0.9	0.4	0.4
	M	**341.8**	10015.2	**320.0**	1892.1	**225.1**	448.4	**123.0**	139.6	72.6	**48.8**	51.6	**19.9**	40.2	**10.5**
orz103d	C	40392		7977		2134		759		287		105		44	
	T	2.0	2.0	0.4	0.4	0.1	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0	0
	M	**1.4**	176.1	**1.5**	35.2	**1.4**	9.7	**1.3**	3.7	**1.3**	1.7	1.2	**0.9**	1.2	**0.6**
maze-400-4	C	127996		24526		7899		3687		1443		607		257	
	T	17.2	24.1	5.4	7.5	2.0	2.8	0.7	1.0	0.2	0.3	0.0	0.1	0.0	0.0
	M	**4.1**	3862.0	**4.5**	741.3	**4.3**	239.8	**4.2**	112.7	**4.0**	45.0	**3.9**	19.8	**3.8**	9.2
room-400-40	C	152811		27480		6524		2111		615		190		55	
	T	43.3	45.8	7.7	8.2	1.8	1.9	0.6	0.6	0.1	0.1	0.0	0.0	0.0	0.0
	M	**74.2**	2149.9	**69.4**	388.2	**44.1**	93.6	**25.6**	31.4	15.1	**10.3**	10.3	**4.4**	7.7	**2.6**
random-400-33	C	103535		32381		12019		4515		1566		456		105	
	T	17.2	24.1	5.4	7.5	2.0	2.8	0.7	1.0	0.2	0.3	0.0	0.1	0.0	0.0
	M	**40.7**	5208.1	**34.2**	1629.8	**27.4**	605.7	**21.1**	228.3	**15.1**	80.0	**9.5**	24.1	**5.6**	6.5

TABLE 5.1: Number of (C)entroids, building (T)ime in minutes, and (M)emory requirements in MB for (F)orward and improved (R)everse CPDs for different radii of centroids on five representative maps. Note that for $\delta = 0$ the number of centroids is the number of cells in the map.

algorithms are implemented in `C++` and compiled with `-O3` flag. We use the following abbreviations for convenience:

- $\texttt{fwd}_\delta$: forward CPD with centroid size δ
- $\texttt{rev}_\delta$: reverse CPD with centroid size δ.

Note that $\delta = 0$ means the full forward or reverse CPDs. Our principal point of comparison in all experiments is $\texttt{fwd}_0$. This variant corresponds to CPDs in Section 4.1. Like the previous work we employ proximity wildcards for all forward CPDs. For reverse CPDs we exclude this feature as we found that it does not lead to substantial improvement in compression. Our test machine is `Linux 4.19.45-1-MANJARO` with `i5-8600 CPU @ 3.10GHz` CPU and `15GB` memory.

5.2.1 Experiment 1: Preprocessing Time

Massive preprocessing-time improvements are observed on all maps, compared to previous work, which has been limited to full CPDs. For example, Table 5.1 shows, for 5 representative maps, preprocessing and size statistics. The use of centroids reduces the CPD construction time by up to three orders of magnitude (i.e. the number of centroids is up to thousands of times smaller than the number of cells in a full CPD). For a fixed δ, the preprocessing time is similar for forward and reverse CPDs since both are dominated by the Dijkstra search time. Forward CPDs are built slightly faster because they avoid

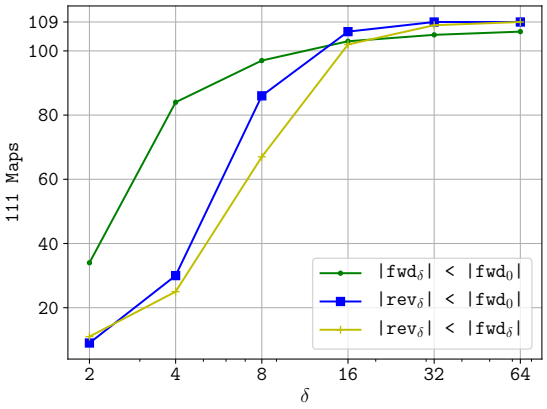


FIGURE 5.6: The number of maps with smaller CPD when δ increase.

the extra step of "illegal" move processing. The savings are achieved due to the fact that the preprocessing time is proportional to the number of Dijkstra searches, which is equal to the number of centroids.

5.2.2 Experiment 2: CPD Size

Assume that $|V|$ is the number of cells and $|C|$ is the number of centroids. A forward CPD starts from a competitive size, but its size decreases at a moderate pace (smaller than linear) as the number of centroids decreases. The reason is that we always have $|V|$ compressed strings (rows) in a forward CPD, independent of $|C|$. Each compressed string compresses a row of length $|C|$. Rows in a full CPD compress well (much better than columns), and cutting symbols from a row that compresses well anyway has a moderate impact on further size reductions. By contrast, a reverse CPD starts from a much larger size (as columns compress more poorly than rows), but it decreases more aggressively, at a linear pace with the number of centroids. This is because a reverse CPD has exactly $|C|$ compressed strings (columns). Often, a reverse CPD gets smaller than a forward CPD as δ increases.

The interplay of these factors is shown in Figure 5.6, which shows a summary of size comparison. We can see that for $\delta \geq 8$ the size of $\texttt{fwd}_\delta$ and also $\texttt{rev}_\delta$ improves on $\texttt{fwd}_0$ in a large majority of cases. There are however 5 small game maps where $\texttt{fwd}_{64}$ CPDs are larger than $\texttt{fwd}_0$, and there are 2 large artificial maps where $\texttt{rev}_{64}$ CPDs are larger than $\texttt{fwd}_0$. Notice however that $\texttt{rev}_\delta$ outperforms $\texttt{fwd}_\delta$ on more than half of all maps when $\delta = 8$ and compression improves further beyond this point. Table 5.2 shows in more detail how much size reduction centroid CPDs can achieve – $\texttt{rev}_\delta$ (resp. $\texttt{fwd}_\delta$) can be up to 32.43x (resp. 9.65x) smaller than fwd_0.

δ	type	mean	min	25%	50%	75%	max
2	`rev`	0.33	0.01	0.12	0.20	0.35	1.57
	`fwd`	0.97	0.85	0.93	0.98	1.01	1.19
4	`rev`	0.82	0.02	0.45	0.72	1.05	2.34
	`fwd`	1.13	0.85	1.02	1.12	1.24	1.68
8	`rev`	1.54	0.04	1.13	1.56	1.99	3.45
	`fwd`	1.34	0.86	1.08	1.27	1.56	2.90
16	`rev`	2.59	0.09	1.87	2.48	3.12	7.39
	`fwd`	1.60	0.86	1.14	1.36	1.86	4.90
32	`rev`	3.81	0.21	2.39	3.18	4.16	17.88
	`fwd`	1.84	0.87	1.19	1.42	2.10	7.20
64	`rev`	4.93	0.44	2.70	3.64	5.12	32.43
	`fwd`	2.08	0.87	1.24	1.50	2.21	9.65

TABLE 5.2: CPD size reduction between centroid CPDs and $\texttt{fwd}_0$, $\texttt{ratio} = \frac{|\texttt{fwd}_0|}{|\texttt{rev}_\delta|}$ or $\frac{|\texttt{fwd}_0|}{|\texttt{fwd}_\delta|}$.

Illegal moves: We can additionally report (not in Table 5.1) that without *illegal move* encoding, reverse CPDs can be 1.5x *larger* for maps such as `room-400-40` and `maze-400-4`. For game maps, illegal moves provide even more benefit – up to a factor of 3x vs. no illegal moves.

Trade-offs: We have seen that as we increase δ, the reduction in size of forward CPDs is often limited, while the reduction in size of reverse CPDs is proportional to the reduction in the number of centroids. This introduces a tradeoff between the size of the CPD and the suboptimality guarantee provided by δ: the smaller the CPD the looser the guarantee.

In the remaining experiments we focus on this tradeoff and restrict our attention to $\delta \geq 16$ where both forward and reverse CPDs improve on $\texttt{fwd}_0$ across almost all maps and where we construct corresponding databases of the smallest size.

5.2.3 Experiment 3: Path Extraction

Reverse CPDs have a runtime advantage due to memory caching which can be understood as follows. When extracting a path with reverse CPDs we require only a single compressed string: the one computed for the target t. Once this string is loaded into CPU cache, it resides there throughout the entirety of path extraction. By comparison, forward CPDs require a different compressed string at each step toward t and extracting each new move incurs the possibility of a cache miss.

	mean	min	25%	50%	75%	max
$\texttt{rev}_{16}$	1.839	0.061	1.311	1.747	2.162	235.606
$\texttt{rev}_{32}$	1.738	0.031	1.194	1.666	2.091	229.882
$\texttt{rev}_{64}$	1.580	0.008	0.998	1.490	1.953	207.992
$\texttt{fwd}_{16}$	1.209	0.013	1.038	1.125	1.312	175.824
$\texttt{fwd}_{32}$	1.233	0.033	1.041	1.144	1.355	163.937
$\texttt{fwd}_{64}$	1.230	0.012	1.013	1.139	1.389	184.361
$\texttt{FS}_{32}$	0.037	0.000	0.001	0.002	0.025	18.451
$\texttt{FS}_{64}$	0.039	0.000	0.000	0.001	0.047	17.233
$\texttt{FS}_{128}$	0.041	0.000	0.000	0.002	0.051	17.431
TC	6.951	0.009	4.570	5.961	8.170	949.790

TABLE 5.3: Speedup, path extraction time divided by path extraction time using $\texttt{fwd}_0$. A ratio greater than 1.0 means faster.

The second runtime advantage is that we can extract the entire run from a reverse CPD rather than just a single move. That means we can avoid additional extractions if the first move of next location $m(s)$ is compressed in the same run as s. This behavior is surprisingly frequent due to the row ordering. On average, $\texttt{rev}_0$ performs just 20% of extractions needed by $\texttt{fwd}_0$, i.e., each extracted run is used 5 times.

Table 5.3 examines the efficiency of forward and reverse CPDs ($\delta = 16, 32, 64$) for path extraction, Here we compare against $\texttt{fwd}_0$ as well as two other suboptimal pathfinding algorithms, each with characteristics similar to our work:

- TC: *Tree Cache* [47], a very fast **unbounded-suboptimal** algorithm based on spanning trees. We compare against this method because like our work path extraction is implemented as a series of recursive lookups. TC was the fastest suboptimal algorithm at GPPC 2014.
- $\texttt{FS}_{\delta_2}$: *FOCAL Search* [48], a **bounded-suboptimal** variant of $\texttt{A}^*$, Usually this algorithm is implemented with a relative suboptimality bound; i.e. the path returned is guaranteed to be no larger than some multiplicative factor $w \geq 1$. Here we adapt FOCAL search to compute solutions that are no more than a constant factor 2δ from optimal. This allows us to compare CPD solution quality against a competitor with equivalent guarantees.

We run each algorithm on all 105 *game maps* in our benchmark set. We have 142,534 instances in total, and we solve each one 5 times, taking the mean to eliminate random noise.

Table 5.3 gives a summary of results, relative to $\texttt{fwd}_0$. We observe that both forward and reverse CPDs benefit from improved cache locality due to smaller CPD size and that

	mean	25%	50%	75%	99%	max
$\texttt{rev}_{16}$	0.88	0.00	0.00	1.17	7.41	26.00
$\texttt{FS}_{32}$	2.52	0.00	0.00	2.34	26.08	31.98
$\texttt{rev}_{32}$	1.55	0.00	0.00	1.66	16.68	54.59
$\texttt{FS}_{64}$	5.95	0.00	0.82	8.14	46.54	63.99
$\texttt{rev}_{64}$	4.38	0.00	0.00	3.51	50.65	113.40
$\texttt{FS}_{128}$	13.22	0.00	4.97	19.74	82.04	127.99
TC	143.10	18.97	46.53	138.12	1644.88	3194.47

(a) Suboptimality comparison (absolute difference).

	mean	25%	50%	75%	99%	max
$\texttt{rev}_{16}$	0.01	0.00	0.00	0.00	0.08	1.95
$\texttt{FS}_{32}$	0.01	0.00	0.00	0.01	0.15	0.79
$\texttt{rev}_{32}$	0.01	0.00	0.00	0.00	0.15	6.04
$\texttt{FS}_{64}$	0.03	0.00	0.00	0.03	0.20	0.95
$\texttt{rev}_{64}$	0.03	0.00	0.00	0.01	0.50	8.77
$\texttt{FS}_{128}$	0.04	0.00	0.02	0.07	0.26	2.40
TC	0.83	0.06	0.18	0.53	8.62	2005.07

(b) Quality comparison (relative difference).

TABLE 5.4

both improve baseline performance. The largest gains are observed for reverse CPDs, which can be almost two times faster for path extraction. The speedup decreases as the compression increases due to overheads from checking the heuristic direct path. Notice that only a few factors separate CPDs and Tree Cache. TC can be understood as a performance lowerbound for CPD-like methods since the cost of path extraction for this algorithm is only one memory operation per step. Unsuprisingly, any form of CPD is orders of magnitude faster than state-space FOCAL search.

5.2.4 Experiment 4: Suboptimality and Path Quality

We evaluate the (absolute) suboptimality of a path as $diff = l - o$, that is, the length of the extracted path l minus the length of an optimal path o. Similarly, we evaluate the (relative) quality of a path as $\frac{diff}{o}$. Table 5.4a and Table 5.4b shows summaries of path suboptimality and path quality for different approaches. We observe that both suboptimality and quality for the unbounded-suboptimal approach TC are prohibitively large. Meanhwhile reverse CPD paths have better suboptimality and better quality than FS for most queries, but FS performs better in the worst case.

Heuristic Extraction: We can further report that, without heuristic path extraction, the worst case behavior we can have slightly faster path extraction but worse path quality: without this addition the mean increase for *diff* is approximately around $\delta/3$.

Overall: Based on results above, it is clear that $\texttt{rev}_\delta$ ($\delta \geq 16$) are better for path extraction in terms of tradeoff between suboptimality and extraction time. Each of our nearest competitors suffers from significant drawbacks, either in terms of a large and unbounded suboptimality (*Tree Cache*) or a much slower path extraction (*Focal Search*).

5.3 Discussion

5.3.1 Other similar methods

In a very high level, centroid CPDs are approaches that precomputing information for a subset of nodes to improve path planning; such an idea also appears in Differential Heuristic (DH) [25] and Canonical Heuristic (CHeur) [27]. However, they are very different in terms of the algorithm: (i) DH and CHeur improve path planning search by providing improved heuristic estimates, while CPDs are search-free path planning techniques that do not depend on heuristics; (ii) DH nodes usually do not appear on computed paths while in our case the target centroid almost always appears on computed paths; (iii) DH and CHeur store true distance information, while centroid CPDs store first-moves, which is compressible.

Recent work [16] shows CPDs can be used to derive heuristic estimates in dynamic graphs. This work compares CPD heuristics vs. DH and shows that CPDs offer substantial advantages.

5.3.2 NP-hardness of optimal centroid placement

We can model the optimal centroid placement as a Min Edge Cost Max Flow (MECM) problem, which is known to be NP-hard [49].

Let s be a pseudo-source node, t be a pseudo-sink node, $v \in V$ be traversable vertices in the gridmap, and δ be the radius such that the distance between all centroids are no less than δ.

- For each vertex v, build edge $e_0(s, v)$ to link s. Their capacities are 1 and costs are 0.

- For each vertex v, build edge that $e_1(v, i)|dist(i, v) \leq \delta, v, i \in V$. Their capacities are 1 and costs are 0.
- For each vertex v, build edge $e_2(v, t)$ to link t. Their capacities are infinite and costs are 1.

The maximum flow of the graph is $|V|$ and the minimum edge cost $\sum cost(edge)$ is the optimal number of centroids for the given δ. Note that this is different from Minimum Cost Flow problem, in which the pricing function is $\sum flow \times cost(edge)$.

5.4 Conclusion

CPDs provide a state-of-the-art solution for path planning, but they may be unattractive because of substantial overhead required to construct and store them. In this chapter, we explore the use of bounded suboptimal CPDs based on centroids, that massively reduces the precomputation time. We apply the idea to both standard forward CPDs and to the reverse CPDs that we introduce in this work. As the number of centroids decreases, reverse CPDs shrink more aggressively, eventually overtaking forward CPDs in terms of size reduction. Our approach leads to faster path extraction than forward CPDs and, in practice, to a much smaller difference in path length than the suboptimality bound guarantees. Results show an excellent speed vs suboptimality tradeoff compared to other techniques from the literature.

Chapter 6

Reverse Oracle for weight-changing road networks

In commuter road routing a centralised routing oracle aims to serve driving directions to as many simultaneous users as possible. There exists a large number of simultaneous queries, tens or even hundreds of thousands, and each one must be solved in close to real-time. The planned routes meanwhile must have guarantees: optimal or close to optimal. Further complications include real-time data updates, to reflect changing road conditions, and *customised metrics*, which means the objective function can change from one query to the next.

This is an extremely challenging problem which has been considered by all major map providers. Some of these approaches are proprietary and their implementations are unavailable for scientific experimentation. Such is the case for Bing Maps and Google Maps. Other providers (e. g., OpenStreetMap, OpenTraffic) do make available routing implementations and these are based on modern speedup algorithms including Contraction Hierarchies (CHs) [35, 50] and other similar techniques [51, 52]. Though each of these works can be orders of magnitude faster than reference algorithms, such as Dijkstra and A*, their performance advantages are achieved under the assumption that the input graph remains static. When the graph changes, for example because of a new metric or because new traffic-related information becomes available, these algorithms no longer guarantee returning a shortest or even feasible path. The reason is that such modern algorithms rely on auxiliary data, created offline during a preprocessing step. When the graph changes this data is invalidated and all guarantees are lost.

To handle dynamically changing costs some authors propose to repair auxiliary data [50, 53] or else to compute *metric independent* auxiliary data and then *customise* the costs online (Customizable Contraction Hierarchies, abbr., CCH) [54]. After these operations

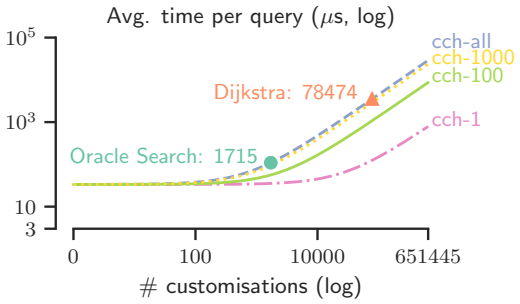


FIGURE 6.1: We compare Oracle Search and CCH in a simulated setup of Melbourne (Australia). The metric periodically changes and after every change CCH *customises* (i. e., repairs) its auxiliary data. No matter the size of the changeset (1 edge, 100 edges or even all), CCH performance quickly degrades as the number of customisations grows: from one per queryset (651K total) to one per query. After 1715 customisations, CCH becomes slower than Oracle Search and eventually slower than Dijkstra search. This experiment shows the main advantage of CCH: being able to amortise the cost of customisation over many subsequent queries.

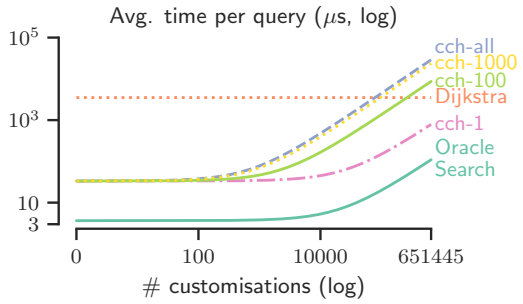


FIGURE 6.2: We compare Oracle Search and CCH in a simulated setup of Melbourne (Australia) where some percentage of all queries (651K total) requires a customised metric w_i, while all remaining queries are solved by a default metric w_0, such as network distance or freeflow travel time. CCH performance is identical to Figure 6.1 but Oracle Search is substantially faster. This experiment shows the main advantages of Oracle Search: fast query performance, which comes from exploiting precomputed w_0 paths, and no additional repair work, which CCH is forced to undertake when the metric changes.

every query is again guaranteed optimal with respect to the new costs and/or metric function. This technology is known to power Bing Maps [55] and is believed to be at the heart of Google Maps [56]. The problem is that the cost of each update/repair depends on the size of the changeset. With only a few edges changed these procedures add only a small overhead per query. When the changeset grows large (e. g., the metric changes with each new query) the overhead can dominate runtime, to the point where it becomes faster to find a shortest path using a reference algorithm (i.e., Dijkstra search). Figures 6.1 and 6.2 illustrate such behaviour. Although improving the customisation time helps [57], it does not address the issue.

In this chapter we consider new perspectives and priorities for the design of centralised routing oracles, especially in settings with dynamically changing costs. To wit, instead of optimal routing and repair, we propose that the following characteristics are more desirable:

1. **Anytime search:** centralised oracles should aim for initial solutions fast and better/optimal solutions eventually. This allows actionable plans to be computed and returned sooner, including by a given time budget, as compared to better or best plans that are returned to the user too late.

2. **Prefix paths:** centralised oracles should prioritise and return only the first k steps toward the destination. Prefixes provide concrete directions commuters can execute and they can be faster to compute than entire paths. This increases throughput

(e. g., the oracle can handle more simultaneous queries) and reduces replanning time when directions are invalidated (e. g., due to missing a turn).

3. **No repair:** centralised oracles should seldom repair auxiliary data. When edge costs change, precomputed data should instead be exploited to guide online search by providing strong upper- and lower-bounds on the optimal cost. This allows frequent graph updates and large changesets without costly online updates.

We describe the design and implementation of such a system and demonstrate its efficacy using a small cluster of six commodity machines. Our approach combines a fast real-time path planning algorithm, called Oracle Search, with a simple workload management system that distributes queries across a cluster. Our system is thus a *realtime*, *centralised* routing engine with a *distributed* oracle. We differentiate real- and anytime as, while the underlying search algorithm is indeed anytime, returning successive paths over the network would incur too much overhead.

Oracle Search generalises CPD Search [16], a recent and database-driven path planner designed for dynamic cost settings. A main feature of CPD Search is that the A* heuristic function provides incumbent paths as well as lower bound estimates. In this work we consider a variety of new database types which can substantially improve performance and also dramatically reduce the *resident set size* – i. e., the amount of data loaded into RAM to solve any given query. For evaluation, we consider a simulated traffic scenario of Melbourne, Australia. Results show that we can process hundreds of thousands of queries *per second* in driving conditions similar to Melbourne's morning-hour peak.

The content of this chapter is extract from a joint work with Afzaal Hassan [3]. The major Contribution of Afzaal is building the traffic perturbation model.

6.1 Problem Statement

We consider path planning problems in graphs with dynamically changing edge costs (see definition of weighted graph in Section 2.1). We assume such problems can be solved in two phases: online and offline. During the *offline phase* the graph is fixed with given weights w_0 and instances are unknown. We assume that we are free to preprocess the graph for as long as necessary in order to create additional auxiliary data structures. During the *online phase* we are given sets of instances together with revised weights which must be solved together as quickly as possible. In other words we aim to maximise throughput (queries per second) for the set of instances. We assume auxiliary data is available at this point and that we are free to exploit it. Our suggested approach, Oracle

Search, has access to an auxiliary database with perfect information w. r. t. w_0 but needs to perform search to find optimal paths w. r. t. new weights.

Cost updates. Edge costs can change from one query to the next. These changes model exogenous events, such as changing traffic conditions, or they can represent penalties, added on a per query basis in order to derive individualised plans – e. g., safest routes, simplest routes, scenic routes and so on. We use $w(i, j)$ to indicate the cost of edge (i, j) with respect to the current metric and $w_0(i, j)$ to indicate the cost with respect to the initial or preprocessing-time metric. We allow edge costs to increase or decrease with each update but we assume the updated value is always at least as large as its corresponding initial cost – i. e., for any edge $(i, j) \in E$ we have that $w_0(i, j) \leq w(i, j)$. We base our oracle on the *free-flow* cost, which is the fastest travel time possible on a road segment – subject to respecting the law.

Note that our problem is distinct from *Personalised Routing* [58] where the authors assume a fixed set of metrics (i. e., each edge has a vector of static costs) and where each query is *personalised* with respect to a linearised set of user-specified weights (one per metric).

6.2 Oracle Search

We describe a family of anytime search algorithms intended for path planning in settings with dynamically changing costs. Collectively known as *Oracle Search*, each member of this family has access to an eponymous *path oracle* $O(s, t)$.

Definition 6.1. A *path oracle* is a function $O(s, t)$ that returns a tuple $\langle \pi, \text{lb}, \text{ub} \rangle$ where π is an s-t path and the values $ub = w(\pi)$ and $lb = w_0(\pi)$ are upper and lower bounds on the cost of the shortest s-t path $w(sp(s, t))$.

Once constructed, a path oracle can provide strong heuristic estimates in the context of anytime A* search.

Two important differences that distinguish our approach, and which provide compelling advantages, are the following:

1. For each expanded node the heuristic returns a concrete path π from s to t. The cost $w_0(\pi)$ bounds the optimal solution from below and drives the search toward the target. The cost $w(\pi)$ meanwhile bounds the optimal solution from above; allowing more nodes to be pruned and helping the search to close the optimality gap faster (recall that A* only bounds from below).

Algorithm 10: Oracle-Search(w, s, t, ϵ): Parameters s and t indicate the start and target, w encodes actual edge-costs. O is the shortest path oracle. The fst function returns the first argument of a tuple. This algorithm guarantees solutions are ϵ-optimal.

```
1  closed ← ∅; open ← {s}
2  for n ∈ N do g[n] ← ∞;
3  g[s] ← 0; f[s] ← 0; p[s] ← []
4  u ← ∞; I ← s
5  while open ≠ ∅ do
6      n ← argmin{f[n'] | n' ∈ open}
7      if n = t then return p[n] ;
8      if εf[n] ≥ u then return p[I]++fst O(I, t) ;
9      open ← open − {n}
10     closed ← closed ∪ {n}
11     for (n, m) ∈ E, m ∉ closed do
12         if g[n] + w(n, m) < g[m] then
13             p[m] ← p[n]++ [(n, m)]
14             g[m] ← g[n] + w(n, m)
15             ⟨π, lb, ub⟩ ← O(m, t)
16             f[m] ← g[m] + lb
17             if u > g[m] + ub then
18                 u ← g[m] + ub; I ← m
19             open ← open ∪ {m}
20 return ⊥                                   ▶ No solution
```

2. Early termination, which means the search stops when a quality threshold is achieved or a time limit is exceeded. The threshold can be controlled by a parameter ϵ, and the returned solution is called *ϵ-optimal*. In many cases these conditions can be satisfied long before the target comes off the OPEN list. Upon termination Oracle Search returns the best known feasible path. We give a pseudo-code description in Algorithm 10.

There are many possible instantiations of Oracle Search, each one characterised by the choice of path oracle. For example in CPD Search [16] the path oracle takes the form of a Compressed Path Database [31, 59]. In this work we generalise CPD Search and we consider several alternative path oracles which can improve performance and reduce the size of active memory during pathfinding search.

Oracles. The usual approach to constructing an oracle is to undertake (offline) an all pairs shortest path computation w. r. t. the initial graph metric w_0. For each vertex s and t we record in a *first-move table*, $\mathbf{fm}[s, t]$, the identity of the next edge on the optimal path, from s toward t. Once the table is computed we *extract* any w_0-shortest

path using the following recursive procedure: extract the move $\mathbf{fm}[s,t]$ and follow the corresponding arc; repeating as necessary until the target is reached.

A first-move table stores $|V|^2$ entries and as $|V|$ grows the total space consumption can become prohibitive. This situation has motivated a variety of works that try to reduce the size of the table using lossless compression, with the currently most successful strategy being some variety of run-length encoding [32]. The resulting *Compressed Path Database* (CPD) trades a small amount of online performance[1] for a two or even three orders reduction in space.

Reverse oracles. A reverse oracle is a first-move table indexed on a target node t – i. e., for every node t we store an array that contains the optimal first move for every s towards t. This way of indexing has the same precomputation costs as in the forward case but comes with some compelling advantages. For example, in a forward oracle (e. g., CPD) each first move is extracted from a different row and the set of required rows is *a priori* unknown (i. e., the entire CPD must be loaded into RAM). With a reverse oracle however we query *only a single row* to extract a complete s to t path.

Forward CPDs encode one-to-all information, where many adjacent target nodes share the same first move from a common source. This compresses very efficiently. Reverse CPDs meanwhile encode all-to-one information, and here first moves can differ even for adjacent source nodes (due to topological changes in the graph). A main consequence is that a reverse CPD can be many times larger than an equivalent forward CPD. For a more detailed discussion see Chapter 5.

In this work we omit the compression step entirely and propose a new type of reverse oracle that operates directly on first-move tables. As we will see, this seemingly naïve approach actually has several strong advantages:

1. **Hardware caching:** we need only one row per query and we can store that row in a low level cache on the processor. This can eliminate expensive memory read operations which otherwise occur after every cache miss.

2. **Software caching:** if two queries share the same target and metric their w-optimal paths can overlap. By caching extracted path data in these cases we can further improve oracle performance. This is similar to the "heuristic cache" optimisation developed for forward CPDs in [16].

3. **Memory requirements:** each row is independent from all the rest, which means we can load arbitrary subsets of the first-move table into memory, and we can store those subsets across different machines. This allows us to trivially parallelise the

[1] Extraction takes log-time w. r. t. the compressed string length.

Type		Size (MB)	Speedup (vs. Dijkstra) Static (w_0)	Dynamic (w)
Dijkstra		0	1	1
CPD	Fwd	118	182.312	42.877
	Rev[2]	49938	n/a	n/a
Table	Fwd	13423	586.015	62.917
	Rev	13423	812.236	73.941
	(*Trim*	*3800*)		

TABLE 6.1: Oracle Search performance with different path oracles on the benchmark from Figure 6.1. *Speedup* is the mean increase in query processing time. Column *Static* indicates free-flow travel times (metric w_0) and *Dynamic* indicates the congestion costs (w). In the static case the path oracle is sufficient to solve each problem optimally; in the dynamic case we perform search.

queryset computation and it also mitigates the increased storage cost (since there is no compression).

6.2.1 Choosing an Oracle

Choosing a graph oracle is a case of balancing memory requirements with performance. To recap: *forward oracles* allow first-move data to be effectively compressed but this comes at the cost of online performance. Meanwhile *reverse oracles* suffer from ineffective compression but improved performance, as only one row is required to extract a complete path and computing each first move requires only constant time. Table 6.1 gives a summary, comparing tradeoffs for different instantiations of optimal Oracle Search.

The experiment shows reverse oracles can be several times faster than forward oracles. Notice too reverse tables are more storage efficient than reverse CPDs. This is because compressing reverse data with run-length encoding is inefficient. In our implementation the reverse table stores 4 bits of information per move while our reverse CPD stores 32 bits per run (4 bits for the move and 28 bits for the index). Although there is some evidence of compression (the reverse CPD is not seven times larger; *only* 2.72 times) the savings are overwhelmed by the index size. The last row, Trim, shows the actual/active space requirements for the reverse oracle – i. e., the total storage for all reverse rows accessed during search. This indicates that not all nodes appear as targets in the queryset. If we knew these nodes in advance we could have only computed and stored their corresponding rows. This reinforces one of the the main advantages of reverse oracles: that the data can be trivially partitioned and, potentially, computed only in part.

[2]We omit a full Reverse CPD comparison due to size; tests on smaller instances indicate performance on par with Fwd Table.

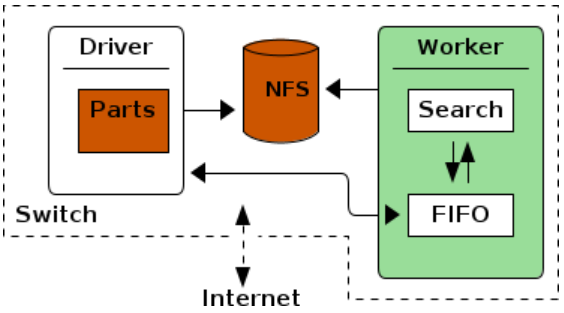


FIGURE 6.3: Schematic view of our cluster

6.3 Distributed Oracle Search

We deploy Oracle Search on a six-machine cluster managed with a simple script that distributes queries using standard GNU utilities. We use the following nomenclature:

Driver means the **head node** of the cluster. We read and distribute queries from this machine.

Worker means a machine in the cluster that is not the driver. On each **worker node**, we will load a *resident process* (the solver) with which we will communicate using a **FIFO**: a kernel-level file descriptor which allows for inter-process communication.

Partition means a division of a larger set of queries, all of which must be solved. The driver defines the partitions and subsequently allocates each one to a worker.

Our entire distributed approach works as follows.

During the *offline phase*, we compute the reverse oracle. This involves a complete Dijkstra search for every node in the graph which records the first move in the (reverse) path towards the target. The rows are distributed across the different machines. (Notice that this limits each worker to solving only queries for its associated target nodes). We also load the underlying graph on each worker. During the *online phase*, we distribute path planning queries to the workers, including edge-cost changes, and we solve them in parallel, with one instance of Oracle Search for each available core. Finally, we collect a summary of results from each distributed computation.

Currently our system is a prototype which we use for feasibility testing. That means we use batched sets of pre-generated queries instead of reading them in real-time from a stream. We also do not simulate real-time edge-cost changes but assume that cost updates from the changeset have already been applied to the graph, before receiving an associated query. We follow the same strategy in Figures 6.1 and 6.2 when comparing

with CCH – i. e., we suppose the graph labels have already been updated and we measure only the time needed to customise auxiliary data.

Systems such as ours often come with a set of trade-offs, mainly due to the overhead incurred from distributing jobs across the (local) network [60]. In the experimental section we consider ways to reduce such overheads when most of the computational load comes from external functions (in our case, external calls to Oracle Search). Our main experiment draws on simulated congestion data for the city of Melbourne, Australia. We use congestion as an example of a "changing metric." When evaluating the efficacy of the system we consider throughput (as measured in queries per second) and solution quality (as measured by path cost w. r. t. the current graph metric w).

6.4 Simulating Melbourne

We use a road graph of Melbourne, Australia provided by OpenStreetMaps [61]. The graph has 167,758 nodes and 459,793 edges. We compute free-flow travel time for every edge by taking the edge length and dividing by max speed.

Our demand is computed from 785 unique origin-destination pairs provided by the 2012–2016 Victorian Integrated Survey of Travel and Activity [62]. Every VISTA origin or destination represents a unique statistical area for census data collection purposes. We use the data as follows:

1. **Compute scenarios.** For each VISTA origin-destination pair, the survey reports an associated number of trips occurring during different periods of the day. The total number of commuter trips in the VISTA data is 651,445. We take these trips as the total set of queries to be solved. The VISTA data aggregates queries by centroids, which map to statistical areas in Melbourne. From the centroids, we generate concrete start and target locations within those areas and use these as the queryset.

2. **Model congestion.** We create a 8–9am model of Melbourne, the busiest time during a typical day. VISTA reports 77K sampled trips in that time period but the 2016 Australian Census [63] reports 1.2M *commuter trips* per day by car. Assuming approximately half of all trips occur in the morning, we scale the 8–9am VISTA data by a factor of 8 to obtain 616K peak-hour trips. We then simulate these trips on the graph of Melbourne using the software package Aimsun [64]. The result is a perturbed graph with increased edge costs which we treat as a congestion model.

To compute the congestion model we undertake a mesoscopic simulation of Melbourne. In a mesoscopic simulation every vehicle is considered a separate entity, but their behaviour is simplified – e. g., vehicles are either stopped or travelling at speed; there is no acceleration, but turn costs are taken into account. To model route choice (i. e., how vehicles react to traffic) we used a dynamic user equilibrium with the method of successive averages [65], as implemented in Aimsun. Vehicles were released in a uniform fashion for four routing cycles and reaction time was set at 1.2 s, mimicking human drivers.

The simulation was warmed up using 5 minutes of the same demand and then ran for a further 60 minutes. This produces modified travel times on about 12% of all edges – i. e., a likely optimistic but still plausible model for 8–9am congestion in Melbourne. Indeed, this relatively small perturbation actually impacts 89% of queries (580K).[3]

6.5 Experimental Setup

We implement Oracle Search[4] in C++ using G++ 7.4 with `-O3` on Ubuntu 18.04. Our test environment is a dedicated hardware cluster comprising six Intel NUCs (model `NUC817BEH`). These are commodity system-on-a-chip hardware, each featuring a four-core i7-8559U CPU running at 2.7GHz. We install 2 × 16GB RAM per machine and an NVMe solid state drive for storage. The machines have access to most of these resources but 2GB RAM is reserved for the operating system. The cluster is interconnected with an off-the-shelf gigabit router.

On each worker, we spin a thread which loads into memory the path oracle data, the graph and (for simplicity) any perturbed costs. The worker then opens a FIFO and *waits* for configuration data and for the start signal which begins the solving phase.

On the driver, we load the queryset and divide the instances into partitions – one per worker for simplicity. Each partition is copied to a shared network drive (NFS) and then the driver sends a signal to the workers along with configuration data.

The configuration data sets the parameters for the search (e. g., termination criteria) and the location of the queryset. Upon receiving the start signal each worker reads the queryset, performs the search and returns summary statistics per instance – e. g., solution cost, time elapsed, nodes expanded, etc.

Small querysets. The performance of a distributed system can be strongly impacted by communication overheads. For example, the time spent copying query data to the

[3] Measured as #queries which require some amount of search.

[4] We use *warthog* as pathfinding library (`https://bitbucket.org/dharabor/pathfinding`, tag: `socs21`) and have uploaded the supplementary code and data for the experiments at: `https://doi.org/10.5281/zenodo.4785122`

workers must be carefully weighed against the expected time savings that can be derived from parallelisation. To mitigate such issues we apply a simple rule: for querysets with < 10K instances[5] we perform the search on the driver. This means we also run a resident process on the driver. This configuration still involves copying data but only locally and the overheads are *much smaller* than over the network.

6.6 Results

To measure the performance of our distributed system we consider queries per second (i. e., throughput), which we measure in two different ways.

On the workers: we measure the time taken to read the queries, the time spent doing search, and the time to output the aggregated statistics. We will report one boxplot per configuration where each datapoint is the throughput on *a single worker*. An ideal system with no overheads would thus have a throughput five times higher on our cluster.

On the driver: we measure the time elapsed between sending out queries to workers and collecting all results; this includes the communication overheads.

In distinct experiments we then consider a variety of different settings including unbounded suboptimal search, bounded suboptimal search and time-budgeted search. In a fourth experiment we examine the scalability of our system and the impact of communication overheads on throughput.

6.6.1 Experiment #1: Any Route At All

In this experiment every query is solved by returning the shortest free-flow path. Solving such queries requires no search, only recursive move extractions from the oracle. Each path is w_0-optimal but its cost w. r. t. to w is unbounded suboptimal. We consider full path extraction and k-prefix queries where we only extract and return the first k moves of the w_0-optimal path. Results in Figure 6.4 are *per worker*.

We see that our system scales extremely well for any value of k including up to 720K full path queries per second on average – i. e., enough to provide simultaneous directions to every commuter in our demand model at every second (communication overheads notwithstanding).

[5] This value was found experimentally.

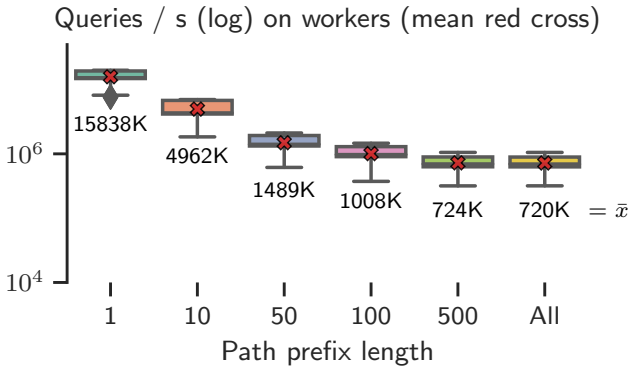


FIGURE 6.4: Throughput per worker when computing k-length prefixes. We do not perform search, we only extract the k first move using the CPD. The numbers below, in black, represent the mean throughput recorded across partitions.

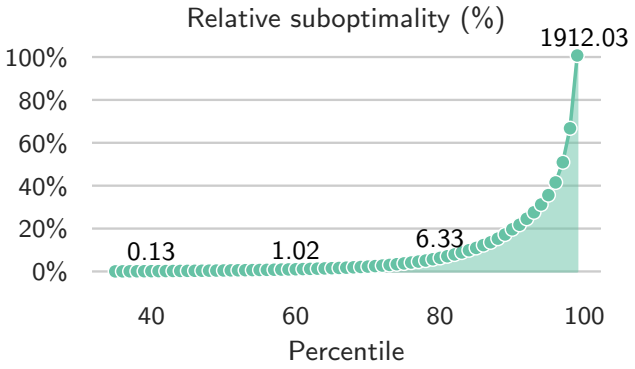


FIGURE 6.5: We measure the quality of the unbounded suboptimal path found by blindly following the oracle vs. optimal. We plot the mean suboptimality for each percentile, and, for every tick mark, the maximum in black.

In Figure 6.5 we measure the relative cost increase from returning w_0-optimal paths instead of w-optimal paths. The average difference between the two metrics is approximately 7.2%, with a standard deviation of 26%. Notice that 33% of paths are not affected by congestion while 1% of queries have double the w-optimal cost on average. In the worst case we report a 20x increase in travel time.

6.6.2 Experiment #2: Bounded Suboptimal Search

When computing driving directions we may not need to return the optimal path. Indeed, driving is inherently chaotic, with many stops and small unexpected delays. In these cases we can aim for faster throughput by running a bounded suboptimal search. In this setup we terminate Oracle Search as soon as the gap between the best feasible incumbent and the best lower-bound (as represented by the f-value of the current node) satisfies

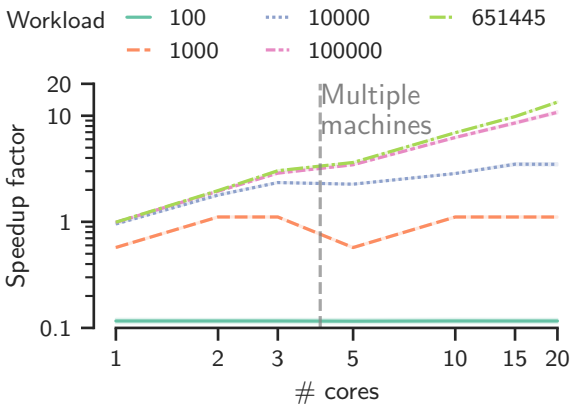


FIGURE 6.6: Speedup of our system vs. single-threaded search for different workloads. We measure on the driver, from sending out the queries to receiving the response from all workers.

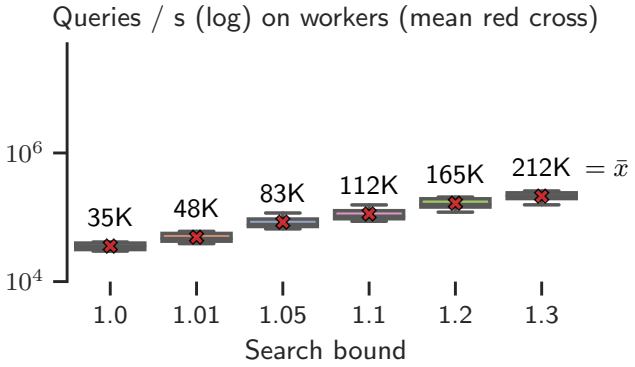


FIGURE 6.7: Throughput per worker when running a bounded suboptimal search with different quality guarantees – 1.0 means an optimal search. The numbers above, in black, represent the mean throughput recorded across partitions.

some criteria. Figure 6.7 shows *per worker* results for a range of multiplicative-factor-guarantees.

We see that when paths with larger suboptimality are acceptable the search terminates sooner and results in higher throughput: from 35K queries per second for optimal solutions to 212K queries per second for solutions up to 30% bounded suboptimal. In Figure 6.8 we report the percentage difference in solution cost between the optimal path and the path returned by suboptimal search. Results are presented as percentiles, with one plot representing one suboptimality guarantee. For example, when the cut-off is set at 10%, around 91% of paths returned are within 5% of the optimum.

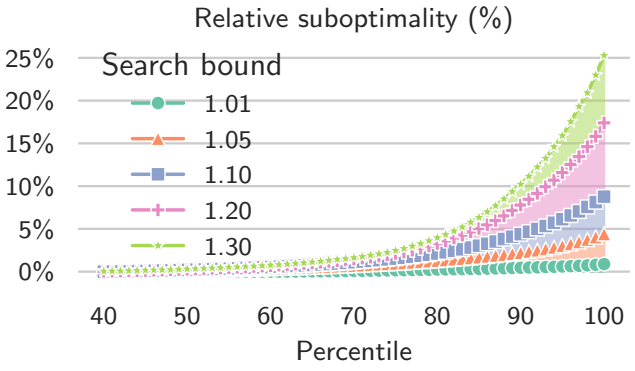


FIGURE 6.8: Percentage difference between the optimal path and suboptimal paths found for different admissibility percentage (% sub-optimal).

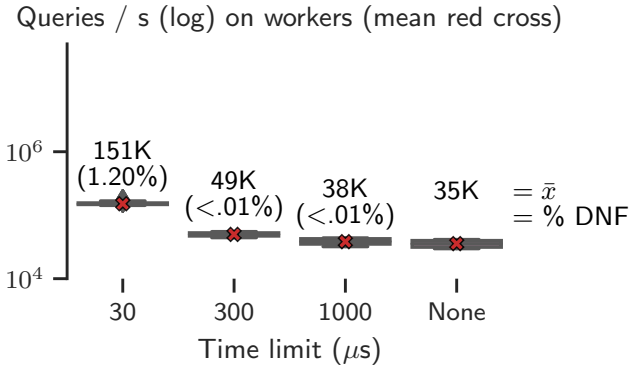


FIGURE 6.9: Throughput per worker when running a time-bounded search for different time budgets. The numbers above, in black, represent the mean throughput recorded across partitions.

6.6.3 Experiment #3: Budgeted Search

Another way to tackle search is to use a fixed budget, such as for time. In this setup we terminate Oracle Search after the maximum allowable time period has elapsed. This is useful in cases where we want to provide a response-time guarantee – e. g., "we will provide a route in under 1 s." This works because ours is a *realtime* system where we do not have to complete a search to return a solution. Figure 6.9 shows *per worker* results using different time cut-offs.

We see that the lower the cut-off the higher the throughput. Moreover, as shown in Figure 6.10, allowing more time produces solutions of higher quality. Indeed, when running the search for 1000 µs, we achieve performance almost equal to a full search. Notice how the change in throughput is not a direct function of the time limit. For example, moving from 30 µs to 300 µs does not yield a tenfold decrease in throughput. This is because not all queries exhaust their budget: for some queries with little or no congestion the optimal response response can be returned in less than 10 µs. Only a very

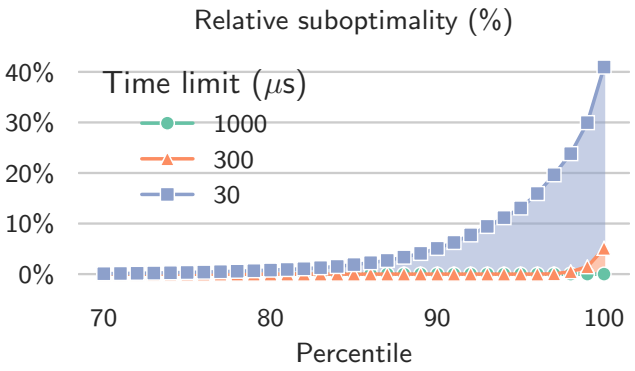


FIGURE 6.10: Percentage difference between the optimal path and suboptimal paths found for different time limits (µs).

small number of queries receive a timeout. Perhaps the best example is for 300 µs, where less than 0.00004% of queries have no solution but throughput (vs. optimal search) increases almost 50%.

6.6.4 Experiment #4: COST

The COST of a distributed system is defined in [60] as the Configuration that Outperforms a Single Thread. This metric measures the impact on performance from unavoidable overheads which systems such as ours introduce to distribute work and collect results. The COST of a system therefore is measured as the number of cores (or machines) required to outperform a single-threaded solver that tackles the same workload. In our case we measure speedup as compared to a single-threaded implementation. Figure 6.6 shows the COST of the system for different workloads and number of cores. For workloads smaller than 10K queries, the system does not distribute queries to workers but does all routing on the driver.

When the workload is very small – e. g., 100 queries – we do not manage to scale up, the communication costs[6] being larger than simply running the queries. With 1K queries, we can see that the performance does not increased with more than 2 cores. The curve looks identical on both sides because there is no distribution yet. Once we exceed 10K queries, our system is able to perform better than the single threaded configuration. After that, we are at least as fast (speedup = 0.98) as the single threaded implementation with a single core. The COST of our system is thus 2, the number of cores needed to outperform the single threaded implementation.

[6]The working thread still needs to read the queryset.

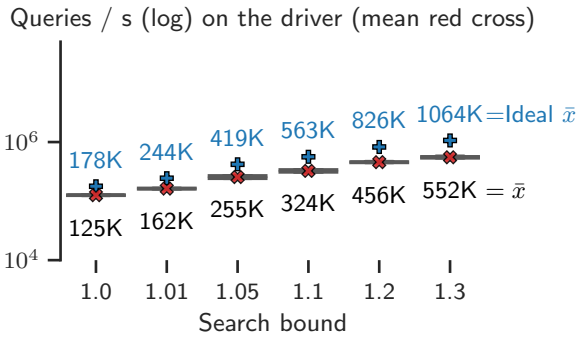


FIGURE 6.11: Throughput measured on the driver. The blue marks and numbers represent throughput on an *ideal system* – one with no communication overheads.

6.7 Discussion

Our experimental results show that we scale well in a variety of scenarios. However most results are on the workers and with full queryset (651K). This is an *ideal setup* where we do not have to worry about communication overheads. The gap between the ideal throughout and the achieved throughput on the driver is shown in Figure 6.11.

Notice that as the suboptimality tolerance increases the gap between our system and the ideal widens. This is because the ideal setup only considers the time spent doing search in parallel whereas in practice one incurs a number of unavoidable overheads, from network latency to reading queries. When the workload is small (or easy) the overheads can dominate total time, resulting in workers becoming *starved*. Figure 6.6 shows that when there is sufficient work to distribute our system is up to 14 times faster than an equivalent single thread implementation. This means a speedup of 2.8 for each (4-core) machine in the system.

Workload balancing. In the experiments, we use a naïve allocation for the workers: we allocate an equal number of rows of the oracle to each. This leads to workers sometimes getting starved because the queries are not distributed evenly across the nodes. Because we know the queryset *a priori*, we can ensure similar workloads across workers. Doing this results in a 10% throughput gain, on the workers and on the driver. This is due to a smaller makespan – i. e., the system waits less for the slowest worker to finish.

Leveraging caches. We consider two optimisations that can improve cache behaviour. First, we *sort the queries by target*. After solving the first query, all subsequent queries with the same target will share the heuristic cache. When the w_0 paths overlap, the next query can be solved faster. We also *allocate targets to specific threads*. In this case queries can not only share a heuristic (software) cache but also the CPU can have first-move data already loaded in (hardware) cache.

When sorting the queryset beforehand and running the same experiments, we measured a throughput increase on the workers of 50% (52K query / s) when sorting and 60% (59K query / s) when adding thread allocation. This translated to an increase of 25% (153K query / s) on the driver when sorting and 62% (201K query / s) when adding thread allocation. Again, the significant increase when measuring on the driver is due to a reduced makespan on the workers.

6.8 Revisiting CCH

In Figures 6.1 and 6.2 we compared Customizable Contraction Hierarchies [54] with Oracle Search and Dijkstra search to highlight the potential drawbacks of customising auxiliary data online. We now give a more detailed discussion of these experiments.

Our CCH implementation is from *RoutingKit*,[7] a freely available C++ library with overlapping authors to CCH. We used the same graph and queryset as in our main experiment with free-flow travel time being the initial metric (w_0).

We experimented with a variety of changesets, from one modification per update (CCH-1) to every modified edge in the congested graph (CCH-all). In each case we pick as the changeset the top n perturbed edges from the congestion model where edges are ordered by the frequency with which they appear on a shortest path in the queryset. We notice that the cost of customising CCH data, even for a comparatively small (100 edges) changeset, is almost as expensive as modifying *all graph edges*, even when the repair is performed in parallel.[8] Figure 6.1 shows that for a modest number of customisations ($<$ 1715 per queryset) CCH can be faster than Oracle Search, on average. As more customised queries are added, it becomes advantageous to switch. Eventually even Dijkstra search is faster than CCH, on average.

The largest speedups for Oracle Search (approximately 6x) are in the zero updates case – i. e., for queries where w_0 is the graph metric. In this setting all problems can be optimally solved using only the path oracle. Figure 6.2 shows an experiment where some proportion of queries are solved with w_0 and all the rest have a customised metric w. Here we find that Oracle Search is always faster, on average.

Our analysis shows that CCH improves on Oracle Search in one setting only: when the cost of customisation can be amortised over a large number of subsequent queries. When the metric changes often, such as per query, CCH performance drops quickly and no-repair methods are better.

[7] https://github.com/RoutingKit/RoutingKit (commit: 613b725)

[8] The RoutingKit implementation we use supports parallel customisation only for the all-graph-edges setting.

6.9 Conclusion

In this chapter, we consider new ideas for the design of centralised routing oracles that provide customised driving directions. We propose a family of anytime algorithms, called Oracle Search, which can identify feasible routes quickly and optimal routes eventually. We combine Oracle Search with a distributed workload management system and experiment with a congestion model for the city of Melbourne, Australia. Using a small commodity cluster we show that our approach scales to hundreds of thousands of simultaneous queries per second and more. We achieve these results in a range of settings including optimal, bounded-suboptimal and time-budgeted search. We further show compelling performance advantages vs. CCH [54], a leading method from the recent literature.

As future work one may consider heuristics to determine a good/performant division of available work to worker machines. Also for dynamically adjusting search parameters during a budgeted/bounded setting.

Another promising direction involves computing several different oracles, each with a different w_0 metric. This may improve performance in cases where the customised w metrics have different cost floors (cf. the current assumption which assumes w_0 is always freeflow travel-time).

Chapter 7

Conclusion and Future works

Path planning problems appear widely in games, robotics and complex planning systems, e.g., trip planning. A path planning solver needs to have fast query processing and be flexible to handle different application scenarios. In this PhD project, we study two methods — *JPS*, an online method that works for dynamic environments, and *CPD*, a preprocessing-based method that works for static environments. The major contributions are as follows.

For *JPS*, we study three pathological behaviours that are not well studied by existing literature: suboptimal node generation, suboptimal node expansion, and redundant scanning. We explain how these happen, how they affect the performance and how to efficiently resolve them by a new method called *Constrained JPS*. The experiments show that, the overhead is only $\tilde{2}5\%$ when there is no chance to prune, while in pathological scenarios, the improvement can be up to 14x and significantly reduce suboptimal node expansion. Thus, *Constrained JPS* is a new online state-of-the-art method.

For *CPD*, we introduce the heuristic symbol to significantly improve the compression, and a bounded suboptimal method to trade-off between the memory cost and suboptimality, as well as the preprocessing time. Compared to vanilla *CPD* which might be too large to fit in memory for large maps, these two methods extend its application scenarios. In addition, we generalize the key idea of *CPD* and propose a distributed oracle when the first move data cannot be well compressed. This method can be applied to road network navigation, comparing with other competitors, it supports anytime queries which returns a prefix without finding the entire path; and under highly dynamic environments, it has smaller amortized query time as it doesn't need repair.

7.1 Future work

In this project, we identify the following future directions.

Constrained JPS on Higher Dimensional Problems Experiments in section 3.5 show that performance improvement of *CJPS* in movingAI benchmarks is not as much as in the synthetic data set. This is because there are not much chance to prune arising in the instances from the movingAI benchmarks. But when it comes to 3D voxel pathfinding, or problems with temporal obstacles which adds a time dimension, the search space is much larger, symmetries are more frequent, and there are many more corner points, hence *CJPS* may prune much more redundant work and outperform vanilla *JPS* by a larger margin.

General Pruning Rule Generation *JPS* can also be applied on other environment models, e.g. 3D gridmaps, hexagon maps, but we need to design and implement a pruning rule manually for each type. This is not an easy job — when the number of available moves becomes large, we need to consider more combinations, it becomes likely to miss a case. In practice, we may need to consider kinematic constraints for different types of agent, e.g. there is an extra cost for changing direction. The combination of map models and agent models makes the design of pruning rules even harder.

At a high level, the *JPS* pruning rule is a symmetry breaking technique based on a predefined preference of moves, called *canonical ordering*, and any equal or larger path with less preferred moves is pruned. Based on this observation, instead of implementing rules manually, for a fixed environment and agent model, we can automatically generate pruning rules by trying all combination of local moves.

First-move Oracle for Time-dependent Routing The real time road network is a highly dynamic environment where the cost of each edge changes from time to time, and similar to temporal obstacles on gridmaps, we need to consider the cost changing in query processing, this is called time-dependent routing. To model this problem, researchers use a time-dependent function on each edge to describe the real time traffic. This causes the time-dependent routing to become extremely complex as time-dependent functions are accumulated along the path, and become very expensive when the path is long, e.g. the number of segments in a time-dependent function can be hundreds of thousands. Online approaches are too slow to handle such complex time-dependent functions, while the index of a time-dependent distance table is also huge [19].

However, first-move data is not accumulated along the path and not very sensitive to the time. For example, although the minimum cost from a to b keeps changing, the first move may always be toward the highway. Based on this observation, a first-move oracle might have smaller index size than existing methods.

Bibliography

[1] 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 J. Benton, Nir Lipovetzky, Eva Onaindia, David E. Smith, and Siddharth Srivastava, editors, *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*, pages 106–113. AAAI Press, 2019. URL https://ojs.aaai.org/index.php/ICAPS/article/view/3465.

[2] Shizhe Zhao, Mattia Chiari, Adi Botea, Alfonso Emilio Gerevini, Daniel Harabor, Alessandro Saetti, and Peter J. Stuckey. Bounded suboptimal path planning with compressed path databases. In J. Christopher Beck, Olivier Buffet, Jörg Hoffmann, Erez Karpas, and Shirin Sohrabi, editors, *Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling, Nancy, France, October 26-30, 2020*, pages 333–342. AAAI Press, 2020. URL https://ojs.aaai.org/index.php/ICAPS/article/view/6678.

[3] Arthur Mahéo, Shizhe Zhao, Hassan Afzaal, Daniel Harabor, Peter J. Stuckey, and Mark Wallace. Customised shortest paths using a distributed reverse oracle. In Hang Ma and Ivan Serina, editors, *Proceedings of the Fourteenth International Symposium on Combinatorial Search, SOCS 2021, Virtual Conference [Jinan, China], July 26-30, 2021*, pages 79–87. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/SOCS/article/view/18554.

[4] Ingimar Hólm Guðmundsson, Hendrik Skubch, Fabien Gravot, and Youichiro Miyake. Predictive animation control using simulations and fitted models. In *Game AI Pro 360*, pages 211–222. CRC Press, 2019.

[5] 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 Eighth Annual*

Symposium on Combinatorial Search (SOCS-15), pages 241–251, 2015. URL http://www.aaai.org/ocs/index.php/SOCS/SOCS15/paper/view/11290.

[6] Daniel Harabor and Alban Grastien. Online graph pruning for pathfinding on grid maps. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25, pages 1114–1119, 2011.

[7] Daniel D. Harabor and Alban Grastien. Improving Jump Point Search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 128–135, 2014. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS14/paper/view/7914.

[8] Yue Hu, Daniel Harabor, Long Qin, and Quanjun Yin. Regarding goal bounding and jump point search. *Journal of Artificial Intelligence Research*, 70:631–681, 2021. doi: 10.1613/jair.1.12255. URL https://doi.org/10.1613/jair.1.12255.

[9] Daniel Damir Harabor, Tansel Uras, Peter J. Stuckey, and Sven Koenig. Regarding jump point search and subgoal graphs. In Sarit Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1241–1248. ijcai.org, 2019. doi: 10.24963/ijcai.2019/173. URL https://doi.org/10.24963/ijcai.2019/173.

[10] Matteo Salvetti, Adi Botea, Alfonso Emilio Gerevini, Daniel Harabor, and Alessandro Saetti. Two-oracle optimal path planning on grid maps. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS)*, pages 227–231, 2018. URL https://aaai.org/ocs/index.php/ICAPS/ICAPS18/paper/view/17758.

[11] Nathan R. Sturtevant and Steve Rabin. Canonical orderings on grids. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 683–689. IJCAI/AAAI Press, 2016. URL http://www.ijcai.org/Abstract/16/103.

[12] Jason M. Traish, James R. Tulip, and Wayne Moore. Optimization using boundary lookup jump point search. *IEEE Trans. Comput. Intell. AI Games*, 8(3):268–277, 2016. doi: 10.1109/TCIAIG.2015.2421493. URL https://doi.org/10.1109/TCIAIG.2015.2421493.

[13] Ben Strasser, Daniel Harabor, and Adi Botea. Fast First-Move Queries through Run Length Encoding. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search (SOCS-14)*, pages 157–165, 2014.

[14] Adi Botea, Ben Strasser, and Daniel Harabor. Complexity Results for Compressing Optimal Paths. In *Proceedings of the 29th National Conference on AI, AAAI-15*, volume 29, pages 1100–1106, 2015.

[15] Matteo Salvetti, Adi Botea, Alessandro Saetti, and Alfonso Emilio Gerevini. Compressed path databases with ordered wildcard substitutions. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, (ICAPS-17)*, pages 250–258, 2017.

[16] Massimo Bono, Alfonso E. Gerevini, Daniel D. Harabor, and Peter J. Stuckey. Path planning with CPD heuristics. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 1199–1205. International Joint Conferences on Artificial Intelligence Organization, 7 2019. doi: 10.24963/ijcai.2019/167. URL https://doi.org/10.24963/ijcai.2019/167.

[17] Shuli Hu, Daniel Damir Harabor, Graeme Gange, Peter J. Stuckey, and Nathan R. Sturtevant. Jump point search with temporal obstacles. In Susanne Biundo, Minh Do, Robert Goldman, Michael Katz, Qiang Yang, and Hankz Hankui Zhuo, editors, *Proceedings of the Thirty-First International Conference on Automated Planning and Scheduling, ICAPS 2021, Guangzhou, China (virtual), August 2-13, 2021*, pages 184–191. AAAI Press, 2021. URL https://ojs.aaai.org/index.php/ICAPS/article/view/15961.

[18] Mike Phillips and Maxim Likhachev. SIPP: safe interval path planning for dynamic environments. In *IEEE International Conference on Robotics and Automation, ICRA 2011, Shanghai, China, 9-13 May 2011*, pages 5628–5635. IEEE, 2011. doi: 10.1109/ICRA.2011.5980306. URL https://doi.org/10.1109/ICRA.2011.5980306.

[19] Lei Li, Sibo Wang, and Xiaofang Zhou. Time-dependent hop labeling on road network. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 902–913. IEEE, 2019. doi: 10.1109/ICDE.2019.00085. URL https://doi.org/10.1109/ICDE.2019.00085.

[20] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum time-dependent travel times with contraction hierarchies. *ACM J. Exp. Algorithmics*, 18, 2013. doi: 10.1145/2444016.2444020. URL https://doi.org/10.1145/2444016.2444020.

[21] Bojie Shen, Muhammad Aamir Cheema, Daniel Damir Harabor, and Peter J. Stuckey. Improving time-dependent contraction hierarchies. In Akshat Kumar, Sylvie Thiébaux, Pradeep Varakantham, and William Yeoh, editors, *Proceedings of*

the Thirty-Second International Conference on Automated Planning and Scheduling, ICAPS 2022, Singapore (virtual), June 13-24, 2022, pages 338–347. AAAI Press, 2022. URL https://ojs.aaai.org/index.php/ICAPS/article/view/19818.

[22] Ethan Andrew Burns, Matthew Hatem, Michael J. Leighton, and Wheeler Ruml. Implementing fast heuristic search code. In Daniel Borrajo, Ariel Felner, Richard E. Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan R. Sturtevant, editors, *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*. AAAI Press, 2012. URL http://www.aaai.org/ocs/index.php/SOCS/SOCS12/paper/view/5404.

[23] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms second edition. 1994.

[24] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3): 596–615, 1987.

[25] Nathan R. Sturtevant, Ariel Felner, Max Barer, Jonathan Schaeffer, and Neil Burch. Memory-based heuristics for explicit state spaces. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 609–614, 2009. URL http://ijcai.org/Proceedings/09/Papers/107.pdf.

[26] 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 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, pages 156–165. SIAM, 2005. URL http://dl.acm.org/citation.cfm?id=1070432.1070455.

[27] Ariel Felner, Nathan R Sturtevant, and Jonathan Schaeffer. Abstraction-based heuristics with true distance computations. In *Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009, Lake Arrowhead, California, USA, 8-10 August 2009*. AAAI, 2009. URL http://www.aaai.org/ocs/index.php/SARA/SARA09/paper/view/825.

[28] Liron Cohen, Tansel Uras, Shiva Jahangiri, Aliyah Arunasalam, Sven Koenig, and T. K. Satish Kumar. The fastmap algorithm for shortest path computations. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1427–1433. ijcai.org, 2018. doi: 10.24963/ijcai.2018/198. URL https://doi.org/10.24963/ijcai.2018/198.

[29] Dorothea Wagner, Thomas Willhalm, and Christos D. Zaroliagis. Geometric containers for efficient shortest-path computation. *ACM Journal of Experimental Algorithmics*, 10, 2005. doi: 10.1145/1064546.1103378. URL https://doi.org/10.1145/1064546.1103378.

[30] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato Fonseca F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In Panos M. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms - 10th International Symposium, SEA 2011, Kolimpari, Chania, Crete, Greece, May 5-7, 2011. Proceedings*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2011. doi: 10.1007/978-3-642-20662-7_20. URL https://doi.org/10.1007/978-3-642-20662-7_20.

[31] Adi Botea. Ultra-fast optimal pathfinding without runtime search. In Vadim Bulitko and Mark O. Riedl, editors, *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2011, October 10-14, 2011, Stanford, California, USA*. The AAAI Press, 2011. URL http://www.aaai.org/ocs/index.php/AIIDE/AIIDE11/paper/view/4050.

[32] Ben Strasser, Adi Botea, and Daniel Harabor. Compressing Optimal Paths with Run Length Encoding. *Journal of Artificial Intelligence Research*, 54:593–629, 2015.

[33] Ulrich Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. *Geoinformation und Mobilität-von der Forschung zur praktischen Anwendung*, 22:219–230, 2004.

[34] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 13-14, 2006*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 41–72. DIMACS/AMS, 2006. doi: 10.1090/dimacs/074/03. URL https://doi.org/10.1090/dimacs/074/03.

[35] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In Catherine C. McGeoch, editor, *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2008. doi: 10.1007/978-3-540-68552-4_24. URL https://doi.org/10.1007/978-3-540-68552-4_24.

[36] Tansel Uras, Sven Koenig, and Carlos Hernández. Subgoal graphs for optimal pathfinding in eight-neighbor grids. In Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini, editors, *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. AAAI, 2013. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS13/paper/view/6058.

[37] Adi Botea, Martin Müller, and Jonathan Schaeffer. Near optimal hierarchical pathfinding. *J. Game Dev.*, 1(1):1–30, 2004.

[38] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato Fonseca F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In Moses Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, Austin, Texas, USA, January 17-19, 2010*, pages 782–793. SIAM, 2010. doi: 10.1137/1.9781611973075.64. URL https://doi.org/10.1137/1.9781611973075.64.

[39] Sabine Storandt. Contraction hierarchies on grid graphs. In Ingo J. Timm and Matthias Thimm, editors, *KI 2013: Advances in Artificial Intelligence - 36th Annual German Conference on AI, Koblenz, Germany, September 16-20, 2013. Proceedings*, volume 8077 of *Lecture Notes in Computer Science*, pages 236–247. Springer, 2013. doi: 10.1007/978-3-642-40942-4_21. URL https://doi.org/10.1007/978-3-642-40942-4_21.

[40] Hannah Bast, Stefan Funke, and Domagoj Matijevic. Ultrafast shortest-path queries via transit nodes. 74:175–192, 2006. doi: 10.1090/dimacs/074/07. URL https://doi.org/10.1090/dimacs/074/07.

[41] Leonid Antsfeld, Daniel Damir Harabor, Philip Kilby, and Toby Walsh. TRANSIT routing on video game maps. In Mark O. Riedl and Gita Sukthankar, editors, *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE-12, Stanford, California, USA, October 8-12, 2012*. The AAAI Press, 2012. URL http://www.aaai.org/ocs/index.php/AIIDE/AIIDE12/paper/view/5459.

[42] Eric A. Hansen and Rong Zhou. Anytime Heuristic Search. *Journal of Artificial Intelligence Research*, 28:267–297, 2007. doi: 10.1613/jair.2096.

[43] Steve Rabin and Nathan R. Sturtevant. Combining bounding boxes and JPS to prune grid pathfinding. In Dale Schuurmans and Michael P. Wellman, editors, *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pages 746–752. AAAI Press, 2016. URL http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12336.

[44] N. Sturtevant. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148, 2012. URL http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf.

[45] Daniel Harabor, Ryan Hechenberger, and Thomas Jahn. Benchmarks for pathfinding search: Iron harvest. In Lukás Chrpa and Alessandro Saetti, editors, *Proceedings of the Fifteenth International Symposium on Combinatorial Search, SOCS 2022, Vienna, Austria, July 21-23, 2022*, pages 218–222. AAAI Press, 2022. URL https://ojs.aaai.org/index.php/SOCS/article/view/21770.

[46] Thomas K. Nobes, Daniel Harabor, Michael Wybrow, and Stuart D. C. Walsh. The JPS pathfinding system in 3d. In Lukás Chrpa and Alessandro Saetti, editors, *Proceedings of the Fifteenth International Symposium on Combinatorial Search, SOCS 2022, Vienna, Austria, July 21-23, 2022*, pages 145–152. AAAI Press, 2022. URL https://ojs.aaai.org/index.php/SOCS/article/view/21762.

[47] Ken Anderson. Tree cache. In Daniel Borrajo, Ariel Felner, Richard E. Korf, Maxim Likhachev, Carlos Linares López, Wheeler Ruml, and Nathan R. Sturtevant, editors, *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*. AAAI Press, 2012. URL http://www.aaai.org/ocs/index.php/SOCS/SOCS12/paper/view/5410.

[48] Judea Pearl and Jin H. Kim. Studies in semi-admissible heuristics. *IEEE Trans. Pattern Anal. Mach. Intell.*, 4(4):392–399, 1982. doi: 10.1109/TPAMI.1982.4767270. URL https://doi.org/10.1109/TPAMI.1982.4767270.

[49] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN 0-7167-1044-7.

[50] 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.

[51] Ittai Abraham, Daniel Delling, AndrewV. Goldberg, and RenatoF. Werneck. Hierarchical hub labelings for shortest paths. In Leah Epstein and Paolo Ferragina, editors, *European Symposium on Algorithms (ESA)*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-33089-6.

[52] Julian Arz, Dennis Luxen, and Peter Sanders. Transit node routing reconsidered. In Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela, editors, *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, volume 7933 of *Lecture Notes in Computer Science*,

pages 55–66. Springer, 2013. doi: 10.1007/978-3-642-38527-8_7. URL https://doi.org/10.1007/978-3-642-38527-8_7.

[53] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In *International Workshop on Experimental and Efficient Algorithms*, pages 66–79. Springer, 2007.

[54] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, volume 8504 of *Lecture Notes in Computer Science*, pages 271–282. Springer, 2014. doi: 10.1007/978-3-319-07959-2_23. URL https://doi.org/10.1007/978-3-319-07959-2_23.

[55] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning in Road Networks. *Transportation Science*, 51(2):566–591, 2017. doi: 10.1287/trsc.2014.0579.

[56] Robert Geisberger. Route planning. US Patent 9,175,972, November 3 2015. Accessed: 2021-02-15.

[57] Daniel Delling, Moritz Kobitzsch, and Renato F Werneck. Customizing driving directions with gpus. In *European Conference on Parallel Processing*, pages 728–739. Springer, 2014.

[58] Stefan Funke and Sabine Storandt. Personalized route planning in road networks. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–10, 2015.

[59] Adi Botea and Daniel Harabor. Path planning with compressed all-pairs shortest paths data. In Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi, and Simone Fratini, editors, *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. AAAI, 2013. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS13/paper/view/6002.

[60] Frank McSherry, Michael Isard, and Derek Gordon Murray. Scalability! but at what cost? In George Candea, editor, *15th Workshop on Hot Topics in Operating Systems, HotOS XV, Kartause Ittingen, Switzerland, May 18-20, 2015*. USENIX Association, 2015. URL https://www.usenix.org/conference/hotos15/workshop-program/presentation/mcsherry.

[61] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org, 2017.

[62] Department of Transport, Victoria. Victorian Integrated Survey of Travel and Activity . https://transport.vic.gov.au/about/data-and-research/vista, 2016.

[63] Australian Bureau of Statistics. 2016 census quickstats. https://quickstats.censusdata.abs.gov.au/census_services/getproduct/census/2016/quickstat/2GMEL?opendocument, 2016. Accessed: 2021-02-15.

[64] Aimsun. Aimsun next 8.4 user's manual. qthelp://aimsun.com.aimsun.8.4/doc/UsersManual/Intro.html, 2019. Accessed on: 2020-02-15.

[65] Michael Florian, Michael Mahut, and Nicolas Tremblay. Application of a simulation-based dynamic traffic assignment model. *European Journal of Operational Research*, 189(3):1381–1392, 2008.