UNIVERSITY OF CALIFORNIA
Los Angeles

Hybrid Heuristic Algorithms for Single-Agent Planning and Search With Limited Memory

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Zhaoxing Bu

2023

© Copyright by

Zhaoxing Bu

2023

ABSTRACT OF THE DISSERTATION

Hybrid Heuristic Algorithms for Single-Agent Planning and Search With Limited Memory

by

Zhaoxing Bu
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2023
Professor Richard E. Korf, Chair

Heuristic search and planning model real-world problems as graphs, where each node represents a unique state or configuration of the problem, and each edge represents an operator that changes one state to another state. The task is to find a shortest or lowest-cost path between the initial state and a goal state in such a graph, which corresponds to a sequence of operators to solve the given problem instance with minimum cost.

A* is the standard algorithm for a single-agent heuristic search or planning problem. However, A* stores all nodes generated during the search and can run out of the available memory on common heuristic search and planning domains in a few minutes. Therefore, A* usually cannot solve hard problems. On the other hand, the memory requirement of iterative-deepening-A* (IDA*) is only linear in the search depth. However, on a domain where many distinct paths exist between the same pair of nodes, IDA* may generate too many duplicate nodes and hence run for a prohibitively long time.

In this dissertation, we provide three new algorithms, A*+IDA*, A*+BFHS, and IDUCHS, for solving problems that cannot be solved by A* due to memory limitations, nor IDA* due to the existence of many distinct paths between the same pair of nodes. We show that A*+IDA* is the state-of-the-art algorithm on the classic benchmark 24-Puzzle, while A*+BFHS and

IDUCHS can generate significantly fewer nodes than A*, hence solving more problems than A* given the same memory limitations.

The dissertation of Zhaoxing Bu is approved.

Adnan Darwiche

Quanquan Gu

Cho-Jui Hsieh

Richard E. Korf, Committee Chair

University of California, Los Angeles

2023

I do not know what I may appear to the world,

but to myself I seem to have been only like

a boy playing on the seashore,

and diverting myself in now and then

finding a smoother pebble or a prettier shell than ordinary,

whilst the great ocean of truth lay all undiscovered before me.

Isaac Newton

Table of Contents

List of Figures . . . . **x**

List of Tables . . . . **xii**

Acknowledgments . . . . **xiii**

Vita . . . . **xvi**

1 Introduction **1**

1.1 Heuristic Search Problems 1

1.2 Dissertation Overview 3

2 Foundations of Heuristic Search **4**

2.1 Basic Concepts 4

2.2 Admissible Heuristics and Pattern Databases 6

2.3 A* 8

2.4 IDA* 10

2.5 Disk-Based Search 11

2.6 Heuristic Search in Planning 12

3 A* and IDA* Variations **13**

4 Frontier Search **16**

4.1 Breadth-First Frontier Search 16

4.2 Divide-and-Conquer Frontier A* 17

4.3 Breadth-First Heuristic Search 17

4.4 Breadth-First Iterative-Deepening A* 19
4.5 Drawbacks of BFHS and BFIDA* 20
4.6 Forward Perimeter Search 21
4.7 Solution Reconstruction 21
4.8 Frontier Search on Directed Graphs 23

5 A*+IDA*: A Simple Hybrid Heuristic Search Algorithm 25
5.1 Motivation 26
5.2 A*+IDA* 26
5.3 Comparisons to Previous Work 29
5.3.1 A*+IDA* vs. MREC 29
5.3.2 A*+IDA* vs. A* With Lookahead 30
5.3.3 A*+IDA* vs. External IDA* 31
5.4 Experimental Results and Analysis 31
5.4.1 24-Puzzle 31
5.4.2 Rubik's Cube 36
5.5 Combining Dual Search With A*+IDA* 37
5.6 Disk-based A*+IDA* 39
5.7 Parallel A*+IDA* on the 27 and 29-Puzzle 41
5.8 An Approximation Algorithm Based on A*+IDA* 42

6 A*+BFHS: A Hybrid Heuristic Search Algorithm 44
6.1 Motivation 44
6.2 A*+BFHS 45
6.3 Comparisons to Previous Work 47

6.3.1 A*+BFHS vs. BFIDA* 47

6.3.2 A*+BFHS vs. FPS 48

6.4 Solution Reconstruction 49

6.5 Experimental Results and Analysis 51

6.5.1 A*+BFHS vs. A* 57

6.5.2 A*+BFHS vs. BFIDA* 58

6.5.3 Calling BFHS on Nodes at Multiple Depths 61

6.5.4 Heuristic Functions and Running Time 62

7 Iterative-Deepening Uniform-Cost Heuristic Search 64

7.1 Motivation 64

7.2 Uniform-Cost Heuristic Search 65

7.2.1 Uniform-Cost Frontier Search 65

7.2.2 Uniform-Cost Heuristic Search 67

7.3 Iterative-Deepening UCHS (IDUCHS) 72

7.3.1 Calculating the Cost Bounds of Iterations 72

7.3.2 Solution Reconstruction 74

7.3.3 Algorithm Evolution 75

7.4 Experimental Results and Analysis 76

7.4.1 A* vs. IDUCHS 79

7.4.2 Cost Bounds in IDUCHS 82

7.5 A*+UCHS 83

8 Conclusions and Future Work 85

8.1 Summary of Contributions 85

8.1.1 A*+IDA* 85
8.1.2 A*+BFHS 86
8.1.3 IDUCHS 87
8.1.4 When to Use Each Algorithm? 88
8.1.5 Node Ordering Is Important 88
8.1.6 A Scheme for Combing Heuristic Search Algorithms 89
8.2 Ideas for Future Work 89
8.2.1 When Will Node Ordering Work? 90
8.2.2 Disk-based Search 91
8.2.3 Delayed Closed Node Deletion 91
8.2.4 Computing Cost Bounds Between Iterations 91
8.2.5 A*+BFHS 92
8.2.6 IDUCHS 92
8.2.7 Searching on Directed Graphs 93
8.3 Conclusions 93

Bibliography 95

List of Figures

2.1 Part of the 8-Puzzle search space. Each node represents a state and each edge represents an operator. 5

2.2 The 6-6-6-6 disjoint PDBs with reflection in [KF02]. 8

2.3 A* on a binary graph with unit-edge costs. Numbers in nodes are f-values. Nodes not generated by A* are dotted. The two light gray nodes and the dark gray node have the same f-value but different h-values. 10

4.1 An example of BFHS with a cost bound of 6. f-values are at the left of nodes. Closed nodes are gray. Deleted nodes are dotted. d_g is the delete g-value. 18

4.2 The evolution of breadth-first iterative-deepening A* (BFIDA*). 19

4.3 BFHS with $U = 4$ on the same graph from Figure 2.3. Numbers in the nodes are f-values. Nodes generated by both A* and BFHS are gray. Nodes only generated in BFHS are white. Nodes not generated by either algorithm are dotted. Nodes generated but not stored in BFHS are dashed. 20

4.4 Breadth-first frontier search on a two-dimensional grid. Stored Closed nodes are dark gray. Open nodes are light gray. Delete Closed nodes are dashed. Dotted nodes are the nodes exist in the search space but not being generated yet. . . . 22

4.5 Frontier search on a directed graph. Dashed nodes are deleted. Stored Closed nodes are gray. x, y, and z are operators. 23

5.1 An example of A*+IDA*. The f-value is at the left of each node. An arrow means the node's f-value is updated during the search. The number next to each edge is the edge cost. Closed nodes of A* are gray. Solid white nodes are the frontier nodes. Dashed nodes are the nodes generated in the IDA* phase. 28

5.2 A*+IDA*'s speedup over IDA*. 32

5.3 A*+IDA*'s last iteration vs. IDA*'s. 35

5.4 Two 7-7-7-6 disjoint PDBs for the 27-Puzzle. 41

5.5 Disjoint PDBs for the 29-Puzzle. 41

6.1 An example of A*+BFHS's search frontier. Numbers are f-values. Closed nodes are gray. 46

6.2 A* vs. A*+BFHS in time and memory. 57

6.3 BFIDA* vs. A*+BFHS in time and memory. 57

6.4 The number of nodes generated in BFIDA*'s previous iterations vs. A*+BFHS's. 59

6.5 The number of nodes generated in BFIDA*'s last iteration vs. A*+BFHS's. . . 60

7.1 An example of UCFS. Numbers next to edges are edge costs. Closed nodes are gray. Deleted nodes are dotted. d_g is the delete g-value. 66

7.2 An example of UCHS with a cost bound of 8.5. Numbers next to edges are edge costs and numbers next to nodes are f-values. Closed nodes are gray. Deleted nodes are dotted. d_g is the delete g-value. 69

7.3 The evolution of iterative-deepening uniform-cost heuristic search (IDUCHS). . 75

7.4 A* vs. IDUCHS in memory and time. 80

7.5 Comparisons of IDUCHS's last iteration. 82

7.6 IDUCHS vs. A*+UCHS in memory and time. 84

List of Tables

5.1 Cumulative data for A*+IDA* on the 24-Puzzle. 33

5.2 Nodes generated in the last iteration in A*+IDA* and Cached-IDA*+node ordering. 36

5.3 A*+IDA* vs. IDA* on Rubik's Cube. 37

5.4 A*+IDA* combined with dual search. 38

5.5 Three hard 24-Puzzle test cases in Table 5.6. 40

5.6 Disk-based A*+IDA* vs. 48 GB RAM-based A*+IDA* on three hard 24-Puzzle test cases. 40

6.1 Instances sorted by A* running times. An underline means more than 8 GB of memory was needed. Smallest memory and shortest times are in boldface. . . . 52

6.2 Instances sorted by A* running times. An underline means more than 8 GB of memory was needed. Smallest memory and shortest times are in boldface. . . . 53

6.3 Instances sorted by A* running times. An underline means more than 8 GB of memory was needed. Smallest memory and shortest times are in boldface. . . . 54

6.4 Instances where A* terminated without solving the problem (marked by >) so are sorted by BFIDA* running times. An underline means more than 8 GB of memory was needed. Smallest memory and shortest times are in boldface. . . . 55

7.1 Relationships between algorithms. 75

7.2 Peak stored nodes, total generated nodes, and running time of A* and IDUCHS on domains with non-unit edge costs. Instances sorted by A* running time. Numbers in parentheses are memory usage in GB. An underline means more than 8 GB of memory was needed. Undirected graphs are marked with *. . . . 78

Acknowledgments

First and foremost, I would like to express my deepest gratitude and admiration to my advisor Richard Korf, for his years of solid support and for never giving up on me. Rich is my role model as a teacher, researcher, and mentor, and I consider myself incredibly fortunate to have had the opportunity to learn from him. Instead of simply lecturing new knowledge to students, Rich taught me how to encourage active participation of students by leading discussions and asking thought-provoking questions. In addition to teaching, Rich has done a wonderful job at research. Rich has built many fundamental works for heuristic search and his passion for research really impressed me. Rich taught me to focus on hard problems, aiming at significant improvements, and solve problems that were not solved before instead of simply pursuing minor tweaks on easy problems. Rich also emphasized the importance of writing clear and concise papers and he meticulously edited my paper and dissertation drafts to ensure that they were of the highest quality. Rich's commitment to ethical research practices has also been invaluable to me. During my Ph.D. journey at UCLA, Rich's door is always open to not only me but also any other students who need help or guidance, even though Rich has no previous contact with them. Throughout the past years, I have had some personal difficulties and challenging times and Rich was always there, having unlimited patience and care toward me and ready to help me in both academic, psychological, and emotional aspects. Rich has established the highest standard a professor can be, and I will always be grateful and remember Rich's endeavors and help toward me.

Second, I am deeply grateful to Robert Holte for introducing me to the fascinating world of heuristic search and inspiring my passion for conducting research. Without Robert, I would never have applied to UCLA in the first place. I would also like to express my sincere gratitude to my committee members, Adnan Darwiche, Quanquan Gu, and Cho-Jui Hsieh, for their invaluable feedback and guidance throughout the dissertation process. In addition, I want to say thank you to Richard Korf, Paul Eggert, D. Stott Parker, Todd Millstein, Demetri Terzopoulos, John (Junghoo) Cho, and Adnan Darwiche for their excellent courses

at UCLA.

Third, thanks to the Computer Science Department at UCLA for providing me with the opportunity to serve as a teaching assistant, and thanks to Tuan Le for being a role model for teaching assistants. After taking Paul Eggert's CS 111 and Tuan's discussions, I discovered that I had fallen in love with teaching, which led to many rewarding and touching moments in my Ph.D. journey. Although for some quarters I spent more than 30 hours per week on teaching, I never regret that. I will always cherish the memories of my time teaching CS 111 and CS 180. I also want to say thank you to those students who attended my discussions, their enthusiasm and dedication had been a strong motivation for my devotion to teaching. Hope I have the chance to teach again in the future.

Fourth, thanks to Google for offering me three summer internships, which have been the best three summers in the past decade. I am especially grateful to my mentors and colleagues, from whom I learned how to write high-quality code, how to do code reviews, how to write design docs, and how to work in a team. I would also like to express my appreciation to the chefs and baristas at Google's cafes for their wonderful food and coffee. Till today, I still miss the coffee I used to have in the mornings.

Fifth, I would like to extend my heartfelt thanks to my friends and labmates Ethan Schreiber, Joseph Barker, Shihan Qin, Zijun Xue, Xu Wu, Huiting Zhang, Yang Pei, Hongbo Zhao, Wenting Li, Rui Shao, Jie Yu, Qianwen Zhang, Qi Zhao, Ang Li, Li Zhang, Chunnan Yao, Zhouyihai Ding, Yuan Gao, Zhihao Wu, Peng Ju, Yang Guo, Kun Hu, Yunxin Wu, Baoliang Wang, Jundong Li, Gaojian Fan, Chuan He, Liang Zhou, Long Tu, and many others who have stimulated many insightful discussions and happy moments that made my Ph.D. journey more meaningful and enjoyable.

To my parents, Xifeng Li and Ge Bu, I owe an immense debt of gratitude for their endless love and support. I could not have completed my degree without their support and encouragement. I am especially grateful to my mother for instilling in me a resilient and optimistic attitude toward life, even during the most challenging times.

Last but not least, I want to express my deepest gratitude and appreciation to my partner

Michelle Lu for her sacrifices, love, support, understanding, and belief in me throughout my Ph.D. journey. I am truly blessed to have her in my life. Thanks for pointing out all my drawbacks, pushing me to improve myself, and also taking good care of my health. I am forever grateful for her patience, kindness, inspiration, and her unwavering commitment to our shared future.

P.S. ChatGPT was used to edit the original draft of this section.

P.P.S. To any future readers, thanks for reading my dissertation. Here are a few pieces of advice for current and future Ph.D. students. Firstly, meet with your advisor regularly. Secondly, follow Rich's research taste, focus on hard or unsolved problems, and aim for significant improvements. Do not waste time on easy problems or directions that have less impact. Thirdly, think out of the box. After my initial A*+IDA* work, I spent a lot of time and energy trying to improve it, but the results were not significant, while the idea of A*+BFHS remained buried beneath the surface for years. Fourthly, work hard and avoid procrastination. I remember one weekend I met Rich at Boelter Hall and Rich told me he never rests. Even though many years have passed, I still remember that moment. Fifthly, maintain an optimistic attitude toward life and research. Lastly, believe in yourself and never never never give up.

Vita

2008 – 2012	B.E. (Software Engineering), Wuhan University, Wuhan, China.
2012 – 2014	M.Sc. student (Computing Science), University of Alberta, Edmonton, Alberta.
2015 – 2019	Teaching Assistant, Computer Science Department, UCLA.
2018 – 2020	Software Engineering Intern, Google, Mountain View and Sunnyvale, California.

Publications

1. Zhaoxing Bu and Richard E. Korf. "Iterative-Deepening Uniform-Cost Heuristic Search." In *Proceedings of the Fifteenth International Symposium on Combinatorial Search, SOCS 2022*, Vienna, Austria, July 21-23, 2022, pp. 20–28.

2. Zhaoxing Bu and Richard E. Korf. "A*+BFHS: A Hybrid Heuristic Search Algorithm." In *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022*, Virtual Event, February 22 - March 1, 2022, pp. 10138–10145.

3. Zhaoxing Bu and Richard E. Korf. "A*+IDA*: A Simple Hybrid Search Algorithm." In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019*, Macao, China, August 10-16, 2019, pp. 1206–1212.

CHAPTER 1

Introduction

1.1 Heuristic Search Problems

Heuristic search is a sub-field of artificial intelligence that focuses on solving problems and is closely related to other sub-fields like planning and games. Much of the work in heuristic search is devoted to finding a shortest path between two nodes in a graph. The graphs can be either explicitly-defined or implicitly-defined. An explicitly-defined graph has all the information for all nodes and edges explicitly stored in a file or database. A search on such graphs requires loading the information for nodes and edges from the stored file. A map of Los Angeles is an example of an explicitly-defined graph, and we can use various heuristic search algorithms to find a shortest path between two locations, like UCLA and LAX. An implicitly-defined graph is defined by simple definitions of nodes and edges representing operators that transform one node to another node. As a result, an implicitly-defined graph does not need to store the information for each node or edge in a file or database. For example, the search space of Rubik's Cube can be implicitly-defined as a graph, where each node is a unique configuration of all the cubies, and each different way of twisting the cube is defined as an operator that transforms one configuration to another.

Brute-force search algorithms can solve easy problems, such as 2 x 2 x 2 Rubik's Cube, by enumerating all possible configurations of the game. However, this is not feasible on hard problems. For example, Rubik's Cube has about $4.3252 * 10^{19}$ reachable states from any start state [Kor97]. If we use only one bit to mark if a configuration of cubies has been generated before in our brute-force search algorithms, then we need 5,406.5 petabytes (PB) of RAM to store such a bitmap. Therefore, researchers came up with various algorithms to speed up

the search. Those advanced algorithms usually use the concept of heuristics to estimate the distance between two states. For example, it is intuitive to use the Euclidean distance of two locations as a heuristic for road navigation because Euclidean distance is cheap to compute and returns a lower bound on the actual distance between two places.

Various heuristic search algorithms have been proposed in the last 50 years. Those algorithms can be roughly divided into two categories. The first type stores all the nodes generated by the algorithm. However, this is not feasible in reality as common heuristic search solvers can use all 8 GB of RAM in a few minutes. After all available memory is used, those algorithms may terminate without finding a solution. The second type overcomes the memory issue by having a space requirement that is only linear in the search depth, at the price of assuming we have enough time to solve the problems. For example, consider an m by n grid graph, where the task is to find a shortest path from (0,0) to (m,n). The first type may store $O(mn)$ nodes but generate the state (m,n) only a few times, while the second type may store only $O(m + n)$ nodes but generate the state (m,n) $O\binom{m+n}{m}$ times. As a result, in practice, when memory is not an issue, the first type is usually faster because they utilize memory to keep track of what states have been visited, hence reducing the work spent on visiting the same states again.

On many hard problems, the first type would not find a solution due to memory limitations, while it may take a very long time for the second type to find a solution. We observe that there is a gap between those two types of algorithms, which is how to utilize memory in a limited memory setting, so we can solve more hard problems relatively quickly. This dissertation investigates this gap and introduces three new algorithms for solving hard heuristic search problems where traditional algorithms may either fail due to memory limitations or run too long due to unnecessary states revisits.

1.2 Dissertation Overview

Chapter 2 provides the basic concepts of heuristic search like how to convert a problem into a search space and how to generate heuristics. Fundamental algorithms like A* [HNR68] and iterative-deepening-A* (IDA*) [Kor85] as well as their pros and cons are also introduced. Some other techniques to solve heuristic search problems are also discussed.

Chapter 3 gives a brief introduction to multiple A* and IDA* variations that were developed in the past several decades.

Chapter 4 introduces frontier search, which is a family of search algorithms that removes stored nodes from memory whenever it is possible. Two new algorithms in this dissertation are inspired by those previously invented frontier search algorithms.

Chapter 5 presents the first new algorithm A*+IDA*, which is a simple combination of A* and IDA*. A*+IDA* is the state-of-the-art algorithm on the 5×5 sliding-tile puzzle (24-Puzzle), which has been a classic benchmark for heuristic search algorithms for several decades.

Chapter 6 introduces A*+BFHS, which is based on A* and another famous heuristic search algorithm breadth-first heuristic search (BFHS) [ZH04]. We implemented A*+BFHS in a planning solver and demonstrated that it can solve more problem instances than the *de facto* algorithm A* using the same 8 GB of RAM constraint in the current International Planning Competition.

A*+BFHS only works on unit-cost domains. To overcome this limitation, Chapter 7 further introduces iterative-deepening uniform-cost heuristic search (IDUCHS), which works on domains with arbitrary non-negative edge costs. IDUCHS is also proved to require less memory than A* on planning problems.

Chapter 8 presents the conclusions and ideas for future work of this research.

CHAPTER 2

Foundations of Heuristic Search

2.1 Basic Concepts

Heuristic search models a search problem as an implicitly-defined graph. A search problem has the same meaning as a search domain, for example Rubik's Cube is a classic search problem/domain. There are two terms, *nodes* and *states*, that are often used interchangeably, but we will distinguish them. The difference is that a state means a unique configuration of the problem elements, while a node is an instance of a state that is generated during search via a particular path. This research focuses on implicitly-defined graphs and assumes that all edges/operators have non-negative edge costs. The process of applying operators to a node and generating all its children is called the expansion of a node. A search problem can have many different problem instances, where each problem instance has a start state and a set of goal states, and the task is to find an optimal solution, which is a series of operators that transforms the start state to a goal state, while minimizing the total cost of operators applied.

In both explicitly and implicitly-defined search graphs, it is possible to have multiple different paths connecting the same two states. For example, there are many different routes between UCLA and LAX. Similarly, there are different ways to transform one configuration of Rubik's Cube into any particular different configuration. When a state is generated again via a different path during search, we call the newly generated node a duplicate node.

Here we present how to model a search problem as a search space by using sliding-tile puzzles. An n-Puzzle is a rectangular board with n non-overlapping tiles, numbered from 1 to n, and one unoccupied position, which is also called the blank. We can slide a tile that is

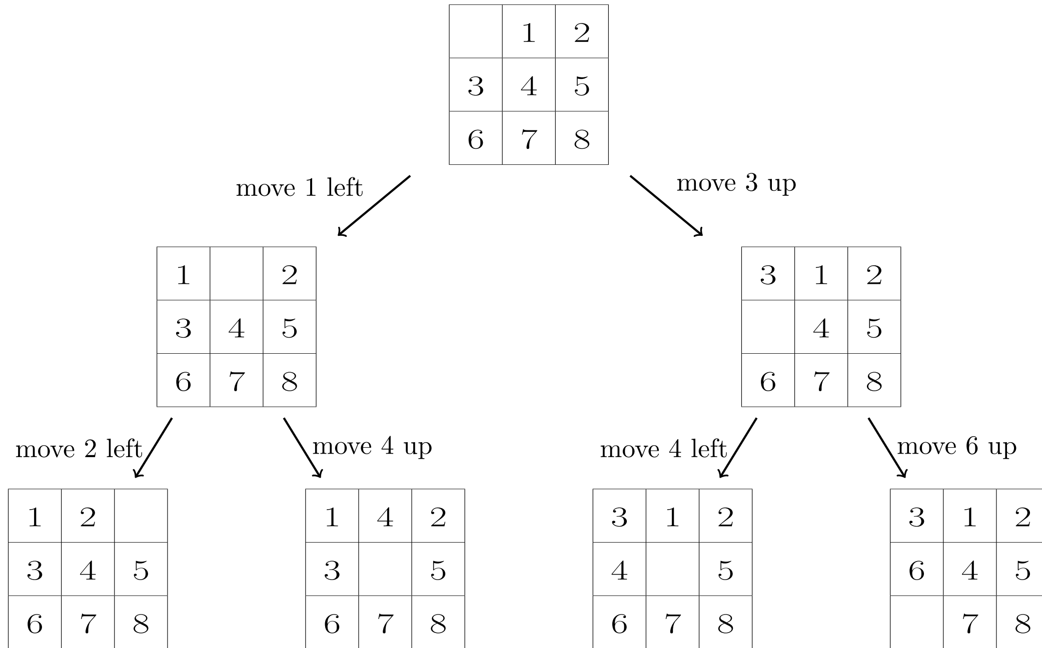

Figure 2.1: Part of the 8-Puzzle search space. Each node represents a state and each edge represents an operator.

adjacent to the blank to the position of the blank to transform one configuration of the puzzle into a different configuration. Figure 2.1 presents an example part of the 8-Puzzle search space. The search space of this puzzle consists of all different reachable configurations of the tiles, and each unique configuration corresponds to one state in the search space. The move of one tile to the blank can be considered as one operator of the problem. Each state has a limited number of applicable operators. For example, the blank is at the top-left position of the top state in Figure 2.1. There are two applicable operators to this state. The first is to move tile 1 left, and the second is to move tile 3 up. After applying one operator to a state n, we generate a new state n', which is a child state of n.

Breadth-first search (BFS) and depth-first search (DFS) are two fundamental search algorithms. BFS first generates all nodes at depth one, then expands each node at depth one to generate the nodes at depth two, etc. Each time a new node is generated, it is compared to the nodes that have already being generated, and it is discarded if it has been generated before. We call this process duplicate detection. BFS terminates when either a

goal state is generated (not expanded), or all states have been generated and the goal state is still not found, which means there is no solution to the given problem instance. The main drawback of BFS is its memory requirement. Suppose a node can generate b children, and the search depth is d, then BFS needs to store $O(b^d)$ nodes in memory.

On the other hand, DFS only uses linear memory. DFS first generates the first child of the start state, then generates the first grandchild, then the first great-grandchild, etc. DFS continues doing this until it reaches a leaf node, which has no child nodes, then it backtracks to that leaf node's parent and generates the parent's second child and then expands it, etc. At any time during search, DFS only stores one node at each depth, so its memory requirement is $O(d)$, where d is the search depth, and it cannot use duplicate detection (other than the nodes on the current path).

On graphs where multiple paths from the start state to a goal state exist, DFS does not guarantee that the first solution it discovers is an optimal one. Another problem of DFS is that it may run forever if some paths do not end in a leaf node, which is the case for most implicitly-defined graphs. Therefore, researchers often use depth-first iterative-deepening (DFID) [Kor85] instead. Assume all operators have the same cost, then DFID performs a series of depth-limited DFS searches. It starts with a DFS to depth one, then a DFS to depth two, then to depth three, etc. Due to the lack of duplicate detection, both DFS and DFID have the drawback of generating and expanding duplicate nodes on graphs.

Implicitly-defined graphs often have a very large search space. Therefore, BFS, DFS, and DFID are usually not feasible on large search problems and researchers developed the concept of heuristics to help reduce the number of nodes generated during search.

2.2 Admissible Heuristics and Pattern Databases

Given a start state s and a goal state g, a heuristic function returns a heuristic value $h(s)$, which is an estimate of the optimal solution cost from s to g. A heuristic function is said to be admissible if for each state s, $h(s) \leq h^*(s)$ where $h^*(s)$ is the optimal solution cost from

s to g. There are two ways to generate a heuristic function: domain-dependent and domain-independent. An example of domain-dependent techniques to generate heuristics is the Manhattan distance in the sliding-tile puzzles. A tile's Manhattan distance is the optimal number of slides needed to move this tile from its current position to its goal position, assuming no other tiles exist on the board. The Manhattan distance heuristic function is hence the sum of all tiles' Manhattan distance values.

One way to get a heuristic function using domain-independent techniques is through the concept of abstraction, which is a simplification of the original problem. An abstraction creates a new search space that is much smaller than the original search space, and maps every state in the original search space to a state in the new smaller search space. When the number of states in the search space created through abstraction is small enough, we can just run BFS in the abstracted search space to compute the exact cost between any abstracted state and the abstracted goal state. These costs can then serve as the heuristic values for the original problem. We store such values in an array, with each abstracted state having a unique index in this array. During search, for each state n, we first compute its abstracted state and the index of this abstracted state in the heuristic value array. Then we use this index to look up the heuristic value $h(n)$ for n from the heuristic value array. This kind of heuristic function is called a pattern database (PDB) [CS98]. For example, one 8-Puzzle problem instance has $9!/2 = 181{,}440$ reachable states in total. An abstraction for this puzzle is that we treat tiles 4 through 8 as indistinguishable. The resulting search space only has 4 distinguishable tiles and $9 \times 8 \times 7 \times 6 = 3{,}024$ states. We can run BFS in this abstracted search space and store the resulting heuristic values in an array of 3,024 bytes.

PDBs enabled the first optimal solutions to random Rubik's Cube problem instances [Kor97], where 3 PDBs were built and the maximum value returned by all 3 PDBs was used as the heursitc value for each state. However, Korf and Felner [KF02] noticed that on certain domains like the sliding-tile puzzles, instead of building multiple PDBs and taking the maximum value, it is in fact possible to build multiple *disjoint* PDBs and take the sum of the heuristic values returned from all PDBs, if the cost of an operator is only taken into

consideration when building one of those PDBs. This technique is called disjoint PDBs or additive PDBs and enabled the first optimal solutions to random 24-Puzzle problems [KF02].

We present the 6-6-6-6 disjoint PDBs from [KF02] in Figure 2.2, where the 24 tiles are partitioned into 4 groups. Here Figure 2.2a shows the default 6-6-6-6 partition and Figure 2.2b shows its reflection along the diagonal. For each PDB, Korf and Felner assumed that the blank can be at any position that is not occupied by the 6 tiles in consideration. During the search, for each state, they looked up the heuristic value from each of the 4 PDBs and added them up. They did this for both the 6-6-6-6 partition in Figure 2.2a and its reflection in Figure 2.2b, and then took the maximum value among the two different partitions as the final heuristic value for a state. Furthermore, since there are only two different shapes of the PDBs, we only need to store two PDBs in RAM and we can then use tile-and-position mapping to get the heuristics from all remaining PDBs during the search. In fact, the Manhattan distance heuristic function can also be viewed as disjoint PDBs where each PDB is built upon exactly one single tile.

	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

(a) One way to partition the tiles.

	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

(b) Reflection along the diagonal.

Figure 2.2: The 6-6-6-6 disjoint PDBs with reflection in [KF02].

2.3 A*

A* [HNR68] is a classic and popular heuristic search algorithm that is guaranteed to return an optimal solution (if one exists) given an admissible heuristic function. In A*, we use two disjoint lists, Open and Closed, to denote the set of nodes that are waiting to be expanded

and have already been expanded, respectively. We use $g(n)$ to represent the cost from the start state s to state n. There are two main differences between BFS and A*. First, BFS orders nodes in Open by their g-value, while A* orders nodes in Open by their f-value, which is the sum of a node's g and h-value ($f(n) = g(n) + h(n)$). Second, on unit-cost domains, BFS terminates immediately when the goal state is generated while A* terminates only after the goal state is chosen for expansion. This is because the first solution to the goal state discovered by A* may not be optimal.

A* has two main advantages. The first is duplicate detection. A* uses its Open and Closed lists to prune all duplicate nodes. The second advantage is node ordering. A* always picks an Open node whose f-value is minimum among all Open nodes to expand next, which guarantees an optimal solution returned by A* when using an admissible heuristic function. However, tie-breaking among nodes of equal f-value significantly affects the set of expanded nodes whose f-value equals the optimal solution cost C^*. It is a common practice to choose an Open node whose h-value is minimum among all Open nodes with the same f-value, as this strategy usually leads to fewer nodes expanded. A survey of the history of tie-breaking strategies in A* can be found in [AF16].

We present an example of the effect of A*'s node ordering on a binary graph with unit-edge costs in Figure 2.3, where S is the start node, G is the goal node, the optimal solution cost is 4, and numbers in nodes are their f-values. The dotted nodes represent states existing in the search space that are not generated during the search. The two light gray nodes and the dark gray node have the same f-value but different h-values. As we can see, when A* breaks ties by expanding the nodes with the lowest h-value first, A* expands the the dark gray node and generates the goal node before expanding any light gray nodes. On the other hand, if A* uses a different tie-breaking policy, for example highest h-value first or first-in-first-out (FIFO), then A* will expand some light gray nodes before expanding the dark gray node, hence instead end up expanding and generating more nodes on the problem instance in Figure 2.3.

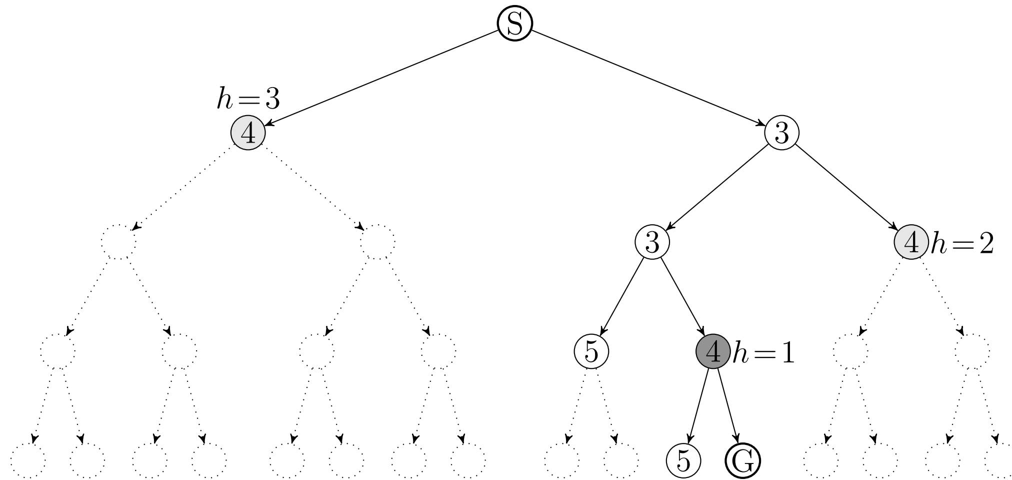

Figure 2.3: A* on a binary graph with unit-edge costs. Numbers in nodes are f-values. Nodes not generated by A* are dotted. The two light gray nodes and the dark gray node have the same f-value but different h-values.

2.4 IDA*

Although A* significantly reduces the number of nodes generated by BFS, it still has an exponential memory requirement and it is not feasible to solve hard problems since A* can use up all 8 GB of RAM in a few minutes on common search and planning problems like Rubik's Cube. Iterative-deepening-A* (IDA*) [Kor85] on the other hand, only has linear memory requirements like DFS. Unlike DFID, whose search bound is merely based on the g-value, IDA*'s search bound for each iteration is based on an f-value. The first search bound is the start state's heuristic value $h(start)$, then IDA* runs one iteration of depth-first search below the start state and only expands the nodes whose f-value is less than or equal to $h(start)$. If the goal state is not found, IDA* increases the search bound to the minimum f-value among those nodes that have been generated but not expanded. Then another depth-first search begins from the start state, with the new search bound as the cutoff. IDA* continues increasing the cost bound until it finds a solution path whose cost equals the bound, at which time it can declare it has found an optimal solution, if using an admissible heuristic function.

IDA* was the first algorithm to successfully solve randomly generated problem instances of the 15-Puzzle [Kor85], the 24-Puzzle [KT96, KF02], and Rubik's Cube [Kor97], since its memory usage is linear in the search depth. However, IDA* still has the drawback of generating duplicate nodes due to the lack of duplicate detection. On domains where many distinct paths exist between the same pair of nodes, most nodes generated by IDA* may be duplicate nodes. For example, in a grid graph, there are $\binom{m+n}{m}$ distinct shortest paths from node (0,0) to node (m,n). As a result, even with a cost bound of $m+n$, IDA* may expand state (m,n) up to $\binom{m+n}{m} = \frac{(m+n)!}{m!n!}$ times, while A* will only expand (m,n) once.

Another drawback of IDA* is the lack of node ordering. Within each iteration, IDA* orders nodes according to the order of operators being applied. In contrast, A* orders nodes according to their f-values and breaks ties by expanding the nodes with the lowest h-value first. On the example problem we presented in Figure 2.3, when the cost bound is 4, if IDA* generates the left child of each node before the right child, then IDA* would expand the left light gray node before expanding the dark gray node. Thus, IDA* would expand more nodes whose f-value equals 4 than A*. As a result, IDA* usually generates most nodes in its last iteration, where the cost bound is the optimal solution cost C^* [Kor85, Kor97, KF02].

2.5 Disk-Based Search

One way to mitigate the memory requirement of algorithms like A* is to use disks to store the generated nodes. Disks are usually much cheaper than RAM and we can increase the number of stored nodes by several orders of magnitude when using disks. For example, in BFS, when we are expanding the nodes at depth k, we can store all the generated nodes at depth $k+1$ on the disk. After all nodes at depth k are expanded, we load the nodes at depth $k+1$ from the disk, perform duplicate detection to remove the duplicate nodes, and then start to expand the nodes at depth $k+1$. This technique is called delayed duplicate detection (DDD) [Kor04] and can be applied to other algorithms like A* as well.

2.6 Heuristic Search in Planning

Heuristic search and planning are two closely related sub-fields in artificial intelligence. Many heuristic search algorithms are used in planning. For example, A* is used by many state-of-the-art optimal track planners [FTL17, KSS18, FLB18, MME18].

There are three major differences between heuristic search and planning: 1) how to represent the problem; 2) how to generate the heuristics; and 3) what algorithm to use for solving the problem. For heuristic search, problem domains are hard coded in algorithm implementations and we find the best algorithm for each different domain. Furthermore, we use domain knowledge to generate the heuristics. As a result, given a new domain, we have to implement the solver for this new domain entirely from scratch and a lot of time would be spent on finding the best algorithm and heuristic function.

In planning, domains are described using general purpose declarative languages like STRIPS [FN71] and PDDL [AHK98]. A solver, called planner, is implemented to analyze a domain's description, a start state, and the set of goal states. Then this planner will start to generate heuristics and search for the solution. In a planner, the same search algorithm and code for generating heuristics are used for all different domains. Given a new domain, there is no need to write any new code, and the only job is to describe this new domain in STRIPS or PDDL.

Another angle to view the above differences is that in heuristic search, researchers always know the problem they are trying to solve, while in planning researchers do not know the problem in advance.

In sum, those two different approaches have their own pros and cons. Heuristic search utilizes the best available algorithm and domain specific knowledge to generate the heuristics and search for solutions, and therefore is usually faster than planning on the same problems. On the other hand, given a new domain, planning requires very little work thus can adapt to new domains very quickly and hence is easy to expand to many different domains in a short time.

CHAPTER 3

A* and IDA* Variations

In the past several decades, many researchers have proposed different algorithms trying to overcome IDA*'s lack of duplicate detection and/or node ordering issues. This chapter gives a brief introduction to those algorithms in chronological order.

MREC [SB89] is IDA*, except that it stores in memory the first nodes generated up to a user-defined bound. Similar to IDA*, MREC also starts each iteration from the start state. For each node that is generated both before and after the memory bound is reached, duplicate detection is performed against the stored nodes. After the search below a stored node is done, the stored h-value (hence f-value) for this node is updated if the goal state is not found below this node under the previous f-value bound. This algorithm was rediscovered, and referred to as IDA* with a transposition table (IDA*+TT) in [RM94]. Transposition tables [SA83] were first used in two-player games to cache search states and reduce duplicate nodes. [AKF10] also implemented IDA*+TT, but did not apply it to the sliding-tile puzzles.

The original MREC paper [SB89] showed no speedup on the 15-Puzzle compared to IDA*. [RM94] reported a node generation reduction of 54%, and a speedup of 37% compared to IDA* on the 15-Puzzle. As we will see below, however, such speedups can be very sensitive to the efficiency of the implementation of IDA* they are compared to. In this research, we use Korf's implementation of IDA* on the sliding-tile puzzles, which has been widely disseminated, and is generally considered very efficient.

MA* [CGA89] and SMA* [Rus92, KK94] represent one family of A* variations where the main idea is to run A* until memory is almost full, then replace the largest f-value nodes in Open with their parents to free up memory for future expansions. However, these

algorithms do not work well on hard problems where the available memory is small compared to the amount needed, so the same states will be generated multiple times. In addition, the implementation of MA* is complex as the hash table needs to be modified to accommodate deletion of nodes from the hash table.

Dynamic Balanced IDA* (DBIDA*) [ES95] first runs A* up to a limit, then runs IDA* on the Open nodes of A*. It is similar to our A*+IDA* that will be introduced in Chapter 5. However, DBIDA* dynamically adjusts the frontier nodes by adding new frontier nodes with a large subtree below them and removing existing frontier nodes that have a small subtree below them. DBIDA* is very complex as it utilizes multiple sets and priority queues to adjust the frontier. The authors reported a speedup of less than a factor of two on the 15-Puzzle, but this was for finding all optimal solutions in the last iteration, and hence is not directly comparable to finding a single optimal solution, since node ordering has no effect in the former case.

Bidirectional A*-IDA* (BAI) [KKL95] first runs A* from one direction, then runs IDA* from the other direction. For each node generated in IDA*, it is checked against the stored nodes in A*. [AK04] further presented several other bidirectional search algorithms based on BAI and BS* [Kwa89] and showed that their algorithm is faster than IDA* by a factor of 7 on the 15-Puzzle. [Man95] presented BIDA* and showed that it is faster than IDA* by a factor of almost 8 on the 15-Puzzle. BIDA* first builds a perimeter of nodes all at the same distance d from the goal state, then runs IDA* from the start state. For each node n generated, BIDA* computes the heuristic to the perimeter nodes p, and expands those for which $g(n) + h(n, p) + d$ does not exceed the current search bound. All these previously mentioned bidirectional algorithms used the Manhattan distance heuristic and were only tested on the 15-Puzzle. However, these algorithms cannot beat IDA* on the 24-Puzzle, for several reasons: 1) These algorithms use the Manhattan distance heuristics so the heuristic between any two states can be cheaply calculated. However, heuristic calculations are more expensive with PDBs and the standard heuristic function to use on the 24-Puzzle is the 6-6-6-6 disjoint PDBs [KF02]. 2) IDA* with duplicate detection is two to three times slower

than IDA* without duplicate detection on our machine when using PDBs. 3) The search space of the 24-Puzzle is $7.4 * 10^{11}$ times larger than that of the 15-Puzzle, making any A* or BS* variations unable to solve random problem instances. Furthermore, the chance that a node generated in BAI's IDA* phase was already generated in its A* phase is very small. 4) As shown in [BK15], bidirectional algorithms with the 6-6-6-6 PDB on the 24-Puzzle would generate more nodes than unidirectional search algorithms.

All the above works were done before PDB heuristics enabled optimal solutions to the 24-Puzzle [KF02]. While [KF02] were aware of these previous works, they chose standard IDA* as their search algorithm.

Dual IDA* (DIDA*) [FZS05, ZFH06, FMS10] was believed to be the state-of-the-art solver for the 24-Puzzle and reduced the nodes generated by IDA* by a factor of 9.3. However, there are three considerations here: 1) Dual search is very complicated and hard to implement. 2) For each state generated in DIDA*, there is a corresponding dual state generated, so the total number of states generated should be doubled. 3) Korf's 24-Puzzle solver [KF02] has two PDB lookups for each new node, while the version of IDA* used for comparison in their papers does eight PDB lookups for each node. Compared to Korf's more efficient IDA* implementation, DIDA* is only 8.5% faster than IDA*.

A* with lookahead (AL*) [SKF10, BSF14] is another A* variation which performs a single iteration of IDA* with bound $f(n) + k$ on each generated node n to increase its heuristic value and hence reduce the memory requirement. Users need to provide the value of k before AL* can run and there is no single best k-value that works for all domains. In practice, given a new problem instance to solve, the standard way is to first run AL* with $k = 0$. If the problem instance is solved, then no more runs are needed. If not, we increase k gradually until AL* can finally solve the problem instance within the given memory limit.

External IDA* [ES11, Ede16] first runs disk-based A* then runs IDA* on the frontier nodes. However, they only reported data for two instances of the 24-Puzzle, did not compare it to IDA*, and did not report any running times.

CHAPTER 4

Frontier Search

Algorithms like BFS and A* never remove any stored nodes from memory. Frontier search [KZT05] is a family of algorithms that only stores the search frontier in memory and removes expanded nodes whenever possible. Unlike IDA*, frontier search algorithms detect duplicate nodes and are guaranteed not to expand a state more than once on undirected graphs.

4.1 Breadth-First Frontier Search

Breadth-first frontier search (BFFS) [Kor04] is the breadth-first version of frontier search, and applies to unit-cost domains. BFFS deletes all closed nodes at depth d after expanding all nodes at depth $d+1$. For each stored node n, we define its delete g-value, $d_g(n)$, as the minimum g-value such that once all nodes with g-value of $d_g(n)$ or less have been expanded, then node n can be safely deleted from memory. For BFFS, $d_g(n)$ is simply one level greater than the depth of n.

$$d_g(n) = g(n) + 1 \tag{4.1}$$

For example, if a node n is at depth 6, or $g(n) = 6$, then n is deleted after BFFS expands all nodes at depth 7.

Equation 4.1 prevents expanding the same state more than once on undirected graphs with unit-edge costs. However, BFFS still has to store nodes at three consecutive depths. In the above example, all nodes stored at depth 6 are deleted after all nodes at depth 8 are generated. On certain domains like the sliding-tile puzzles, where the inverse of every operator is known in advance, there is an optimization called forbidden operators or used

operators [KZ00] that can reduce the number of depths of the stored nodes into two. Suppose every node in a domain has at most n operators, then we can use n bits for each node to mark whether all its operators can be applied or not. For example, assume the parent node p generates the child node c via operator o, whose inverse is operator o'. Then we can mark the operator o' of c as forbidden. After the expansion of p, we can immediately remove p from memory. Then during the expansion of c, o' will not be applied, which means p will not be regenerated and the time for duplicate detection for p is also saved. As a result, this technique reduces both the space and time requirement of BFFS and can be applied to other frontier search algorithms as well. The drawback of this technique is that it in general only works well on undirected domains and the domain has to be analyzed in advance to find the inverse of every operator, therefore it is in general not used in planning.

4.2 Divide-and-Conquer Frontier A*

Divide-and-conquer frontier A* (DCFA*) [KZ00] is a frontier search algorithm based on A*. The difference between A* and DCFA* is that DCFA* stores the entire Open list and only a few layers of the Closed list, which can be done by deleting a parent node after expanding all its child nodes. Compared to A*, DCFA* saves a large amount of memory when the Closed list is significantly larger than the Open list. For example, in 2D-grid spaces, the problem spaces grow polynomially in the search depth d, hence we have $|\text{Open}| = O(d)$ and $|\text{Closed}| = O(d^2)$. On the other hand, little memory is saved on problems where the Open list is much larger than the Closed list, which is often the case for exponential problems.

4.3 Breadth-First Heuristic Search

Breadth-first heuristic search (BFHS) [ZH04] is another frontier search algorithm that works on unit-cost domains, and uses less memory than A*. It is BFFS with heuristics to prune nodes. BFHS requires a user-specified cost bound U, and prunes every generated node whose f-value is greater than U.

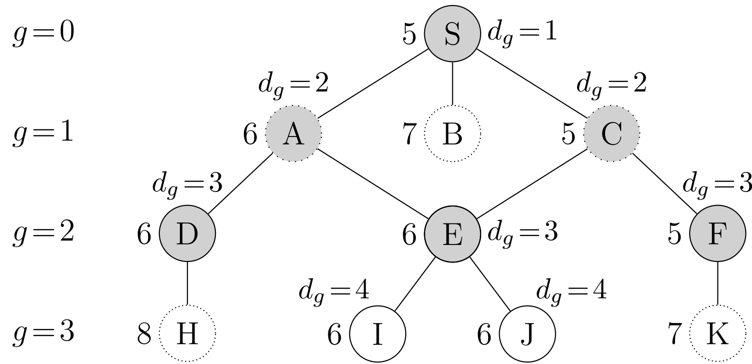

Figure 4.1: An example of BFHS with a cost bound of 6. f-values are at the left of nodes. Closed nodes are gray. Deleted nodes are dotted. d_g is the delete g-value.

We present an example of BFHS with $U = 6$ on an undirected graph in Figure 4.1, where all edge costs are 1, the numbers next to the nodes are their f-values, Closed nodes are gray, and deleted nodes are dotted. The start node S generates nodes A, B, and C. $f(\text{B}) = 7 > U = 6$ so B is immediately deleted, while A and C are stored at depth 1. Next, BFHS expands the Open nodes at depth 1. A generates S, D, and E. S is a duplicate node, while D and E are stored at depth 2. C generates S, E, and F. S and E are duplicate nodes, while F is stored at depth 2. Although $d_g(\text{S}) = 1$, S is not deleted since it is the start node. Then BFHS expands the Open nodes at depth 2. D generates A and H. A is a duplicate node, while H is immediately deleted since $f(\text{H}) > U$. E generates A, C, I, and J. A and C are duplicate nodes, while I and J are stored at depth 3. F generates C and K. C is a duplicate node, while K is immediately deleted since $f(\text{K}) > U$. After all Open nodes at depth 2 are expanded, BFHS deletes the nodes at depth 1, A and C, which have d_g-values of 2, since they cannot be generated again. BFHS then expands the Open nodes at depth 3 and continues until it finds a goal node, or proves that no solution exists within the cost bound U.

4.4 Breadth-First Iterative-Deepening A*

BFHS searches for solutions within a given cost bound. If the bound is smaller than the optimal solution cost C^*, it may never generate a goal node. If the bound is too large, it may generate many nodes n for which $f(n) > C^*$. C^* is rarely known in advance. Zhou and Hansen [ZH04] hence proposed breadth-first iterative-deepening A* (BFIDA*), which runs a series of iterations of BFHS with increasing cost bounds. The first cost bound is the heuristic value of the start node and the last bound is C^*. Similar to IDA*, during each call to BFHS, BFIDA* keeps track of the minimum f-value generated but not expanded in that call to BFHS and uses that f-value as the next cost bound. Although the name of BFIDA* contains IDA*, BFIDA* does not use depth-first search and the name simply means both IDA* and BFIDA* use the same strategy for increasing the cost bound.

We present the evolution of breadth-first iterative-deepening A* in Figure 4.2, where each node is an algorithm or search technique. When breadth-first search and frontier search are combined together, we get breadth-first frontier search (BFFS). Combining BFFS with heurstics and a cost bound, we get breadth-first heuristic search (BFHS). Finally, after applying iterative-deepening to BFHS, we get breadth-first iterative-deepening A* (BFIDA*).

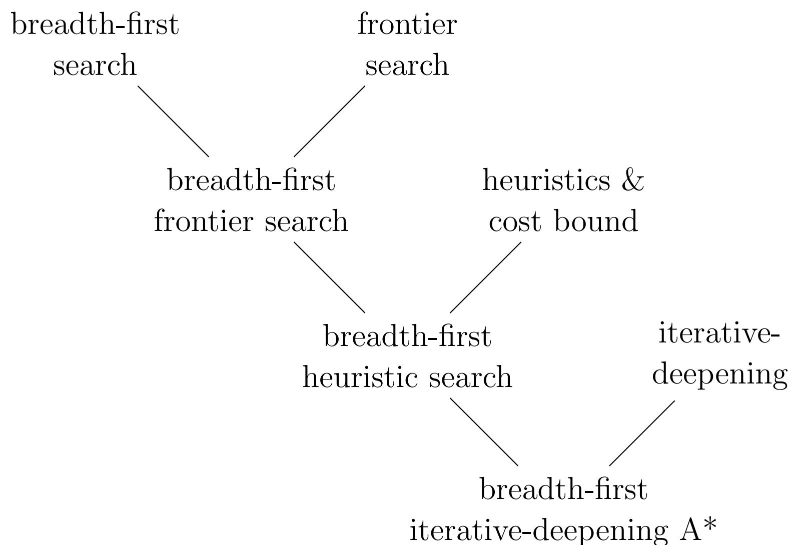

Figure 4.2: The evolution of breadth-first iterative-deepening A* (BFIDA*).

4.5 Drawbacks of BFHS and BFIDA*

Compared to A*, BFHS and BFIDA* save significant memory but generate more nodes. The main drawback of BFHS and BFIDA* is that their node ordering is almost the worst among different node ordering schemes. BFHS and BFIDA*'s breadth-first ordering means they have to expand all nodes stored at a single depth before expanding any nodes in the next depth. As a result, they have to expand almost all nodes whose f-value equals C^*, excepting only some nodes at the same depth as the goal node, while A* may only expand a small fraction of such nodes due to its best-first node ordering.

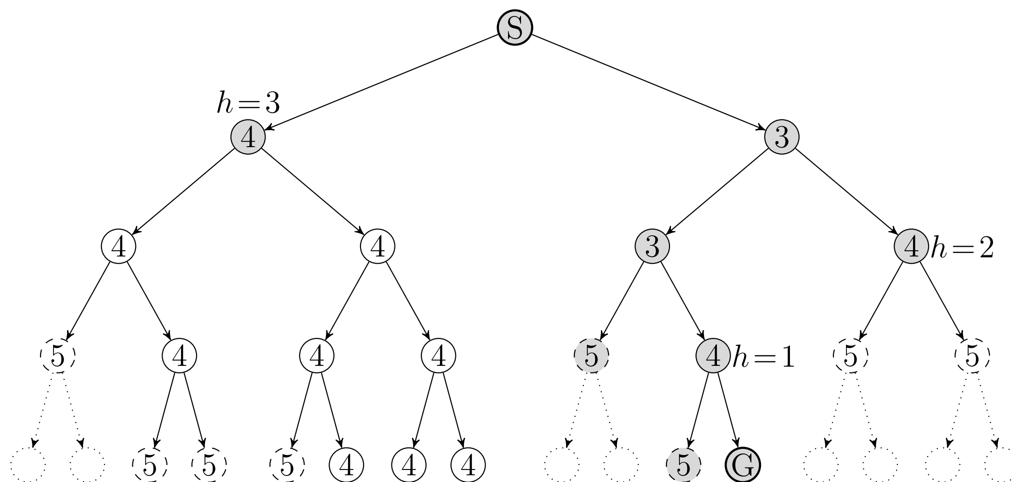

Figure 4.3: BFHS with $U = 4$ on the same graph from Figure 2.3. Numbers in the nodes are f-values. Nodes generated by both A* and BFHS are gray. Nodes only generated in BFHS are white. Nodes not generated by either algorithm are dotted. Nodes generated but not stored in BFHS are dashed.

To compare the effects of node ordering between A* and BFHS, we run BFHS with $U = C^* = 4$ on the same graph from Figure 2.3, and we present the result in Figure 4.3. Here S is the start node, G is the goal node, the optimal solution cost is 4, and numbers in the nodes are their f-values. Nodes generated by both A* and BFHS are gray. Nodes only generated in BFHS are white. The dotted nodes represent states existing in the search space but which are not generated by either algorithm during the search. The dashed nodes, which have an f-value that is greater than the cost bound 4, are generated but not stored in

BFHS. As we can see, due to the nature of breadth-first node ordering, BFHS has to expand all stored nodes at depths 1 through 3, hence generating many more nodes than A* on the same problem instance.

4.6 Forward Perimeter Search

Forward Perimeter Search (FPS) [SDR13] first builds a perimeter from the start state by using BFS, then runs BFIDA* from each node in the perimeter to find a solution. A perimeter is a set of nodes that have the same or similar distance from the start state. During search, this perimeter can also be dynamically adjusted. After the perimeter is built, FPS switches to its BFIDA* phase and uses the lowest f-value among all perimeter nodes as the first cost bound. In each iteration of the BFIDA* phase, FPS first sorts the perimeter nodes using a max-tree-first or longest-path-first policy, then calls BFHS on each perimeter node and updates the f-value of each perimeter node to the smallest f-value of the node generated but not expanded in that single call to BFHS. Here the max-tree-first policy means that in each iteration, FPS first calls BFHS on the perimeter node that generated the most nodes in the previous iteration, while the longest-path-first policy means that FPS first calls BFHS on the perimeter node that led to the longest path (largest g-value generated in a single call to BFHS) in the previous iteration.

4.7 Solution Reconstruction

All frontier search algorithms delete stored nodes whenever possible. As a result, when a goal node is expanded, we only know the optimal solution cost, but not the optimal solution path because most nodes on this path were already removed from memory. Therefore, frontier searches need to reconstruct the optimal solution path in the end.

All frontier search algorithms use the same scheme for solution reconstruction. During the search, they keep a middle layer of nodes in memory and never remove the nodes in the middle layer from memory. Every node generated below this middle layer will have a

pointer to its ancestor in the middle layer. When frontier searches expand the goal node, they know which node in the middle layer led to this goal node. Then the same algorithm can be called recursively to reconstruct the optimal path from the start node to the middle node, and from the middle node to the goal node [ZH04, KZT05].

We present an example of such a middle layer in Figure 4.4, where we apply BFFS on a two-dimensional grid. The stored Closed nodes are dark gray and the Open nodes are light gray. The dashed nodes are the deleted Closed nodes and the dotted nodes are the nodes that exist in the search space but have not been generated yet. S is the start node and G is the goal node. After G is found, BFFS knows that node M is the middle node led to G. Then we can call BFFS again to find the optimal solution path from S to M and M to G. When the subproblem is easy enough, we can just use another algorithm like A* to compute the optimal path of the subproblem.

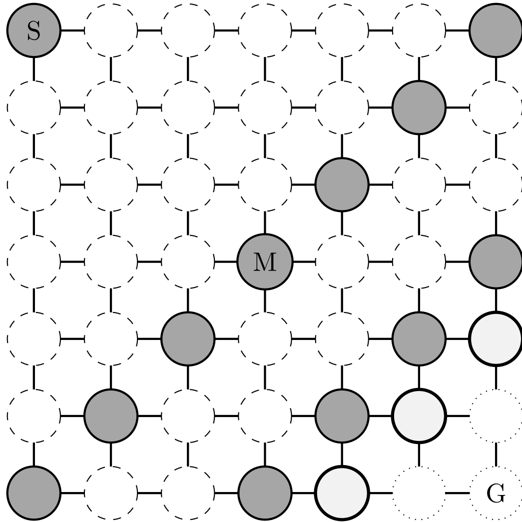

Figure 4.4: Breadth-first frontier search on a two-dimensional grid. Stored Closed nodes are dark gray. Open nodes are light gray. Delete Closed nodes are dashed. Dotted nodes are the nodes exist in the search space but not being generated yet.

In practice, there are multiple ways to determine what nodes are considered to be in this middle layer. For example, in DCFA*, we can store a node n into this middle layer if $g(n) = h(n)$ [KZ00]. In BFHS and BFIDA*, we can store the nodes whose g-value equals

$U/2$ or $U/4$, where U is the current cost bound [ZH04].

4.8 Frontier Search on Directed Graphs

All frontier search algorithms introduced in this chapter are guaranteed to detect all duplicate nodes on undirected graphs. However, in general, frontier search algorithms cannot detect all duplicates on directed graphs [ZH04, KZT05].

Consider the example of frontier search on a directed graph in Figure 4.5, where dashed nodes are deleted Closed nodes and currently stored Closed nodes are gray. x, y, and z are operators. Here the frontier search algorithm can be any algorithm described in this chapter. Frontier search first expands A and generates B. Then frontier search expands B, generates C, and removes A from memory because all of A's child nodes have been expanded. Frontier search would next expand C and generate A again via operator z. At this time, because A is not stored in memory, this new copy of A will be considered as a new state, hence is stored and put on Open. As a result, frontier search on this graph may keep generating and expanding the same states and never terminate, if there exist zero-edge costs or no search upper bound.

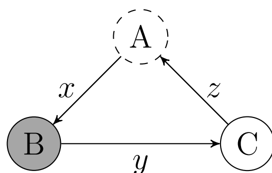

Figure 4.5: Frontier search on a directed graph. Dashed nodes are deleted. Stored Closed nodes are gray. x, y, and z are operators.

The above example is a general case where frontier search algorithms cannot detect all duplicate nodes on directed graphs. As shown in [KZT05], there is an exception where frontier search algorithms can detect all duplicates on directed graphs. Suppose for each node, we know all of its predecessor nodes, then during the search, when we expand a node n, we can create a "dummy" node for each predecessor node p of n, if p is not already in

memory. This "dummy" node will not be expanded, but will be used for duplicate detection.

Consider the example in Figure 4.5, when A is expanded, if we know C is a predecessor node of A, we can create a "dummy" node C and mark operator z as forbidden. Then B is expanded and generates C, this new copy of node C is merged with the existing "dummy" node C, which means operator z remains as a forbidden operator. As a result, during the expansion of node C, A will not be generated again via z.

In sum, if we can know all the predecessor nodes of every node in advance and create the corresponding "dummy" nodes during the search, then frontier search algorithms can still detect all duplicate nodes on directed graphs.

CHAPTER 5

A*+IDA*: A Simple Hybrid Heuristic Search Algorithm

We present a simple combination of A* and IDA*, which we call A*+IDA*. It runs A* until memory is almost exhausted, then runs IDA* below each frontier node without duplicate checking. It is widely believed that this algorithm is called MREC, but MREC is just IDA* with a transposition table. A*+IDA* is the first algorithm to run significantly faster than IDA* on the 24-Puzzle, by a factor of almost 5. A complex algorithm called dual search was reported to significantly outperform IDA* on the 24-Puzzle, but the original version does not. We made improvements to dual search and our version combined with A*+IDA* outperforms IDA* by a factor of 6.7 on the 24-Puzzle. Our disk-based A*+IDA* shows further improvement on several hard 24-Puzzle instances. We also found optimal solutions to a subset of random 27 and 29-Puzzle problems. A*+IDA* does not outperform IDA* on Rubik's Cube, for reasons we explain.

Our work on A*+IDA* is presented in the following order. Section 5.1 discusses the motivation of this research. Section 5.2 presents the new algorithm A*+IDA*. Section 5.3 compares A*+IDA* with some previous algorithms that look similar to A*+IDA* and discusses the differences. Section 5.4 presents experimental results on the 24-Puzzle and Rubik's Cube and shows why A*+IDA* is faster than IDA* on the 24-Puzzle by a factor of almost 5, but not faster on Rubik's Cube. Section 5.5 shows the result of A*+IDA* combined with dual search on the 24-Puzzle. Section 5.6 presents results of disk-based A*+IDA*. Section 5.7 presents results of parallel A*+IDA* on the 27 and 29-Puzzle. Lastly, Section 5.8 shows how to convert A*+IDA* to an approximation algorithm to quickly find a solution.

5.1 Motivation

A* detects all duplicates and expands few nodes whose f-value equals the optimal solution cost C^*, but has an exponential memory requirement. IDA*'s space is linear in the search depth at the price of no duplicate detection, and expands a lot of nodes whose f-value equals C^*. In past years, researchers have developed various algorithms to overcome these difficulties. Some of these algorithms are in the breadth-first style and use various ways to reduce the number of nodes stored. Other algorithms are in the depth-first style but utilize more memory to help reduce the number of duplicate nodes. However, most of the previous work either cannot solve large problems, or are slower than IDA* on some common benchmarks like the sliding-tile puzzles. As a result, most of these algorithms are not very popular and researchers do not often use them.

Based on the pros and cons of A* and IDA*, it is natural to consider the question: how can we combine A* and IDA* to get the advantages of both without their disadvantages? To answer this question, we present our new algorithm A*+IDA*.

5.2 A*+IDA*

A*+IDA* runs pure A* in the first phase and stores nodes up to a user-defined limit. In the A* phase, duplicate detection is performed for each newly generated node. In Open, nodes are sorted in increasing order of f-value, with ties broken by smaller h-value. All f and h-values are integers, so we can use a three-dimensional array to order the nodes such that the f-value corresponds to the first dimension, the h-value corresponds to the second dimension, and all nodes with the same f and h-value are stored in the last dimension according to the order they are put in Open. At each step a node with the smallest f-value and the smallest h-value among that f-value is removed from Open and then expanded. All new child nodes passing duplicate detection are put in Open and the parent node is then put in Closed. For a state that is generated again, the new node replaces the old node and is put in Open if the new node's g-value is smaller than that of the old node, otherwise the new node is deleted.

If A*+IDA* finds the optimal solution in the A* phase, then A*+IDA* terminates without entering its IDA* phase. If the memory bound is reached before the A* phase finds the optimal solution, A*+IDA* switches to its IDA* phase, and we define the nodes in Open at this moment as the frontier nodes.

A*+IDA*'s IDA* phase can be viewed as a doubly nested loop. Each iteration of the outer loop, which we define as an iteration of the IDA* phase, corresponds to a different cost bound for IDA*. The first cost bound is set to the smallest f-value among all frontier nodes. In the inner loop, we make one call to IDA* on each frontier node whose f-value equals the bound, in increasing order of their h-values. In contrast to pure IDA*, here each call to IDA* has a fixed bound that is the current bound of the IDA* phase. After each call to the single-bound IDA* on a frontier node n, we increase the stored $f(n)$-value to the minimum f-value of all the nodes generated but not expanded in that call to the single-bound IDA*. This inner loop continues until a solution is found or all calls to the single-bound IDA* with the current bound fail to find a solution. There are two important things to note here. First, unlike A*+IDA*'s A* phase, we do not perform any duplicate detection in the IDA* phase. Second, similar to pure IDA*, A*+IDA*'s IDA* phase terminates only when the best solution cost found so far equals the current search bound.

We store all information for all nodes in a hash table. Duplicate detection is performed by checking this hash table. The Open list is a three-dimensional vector and each element is a node's index in the hash table. Adding or removing a node from the Open list can be done in constant time. All nodes not in Open belong to the Closed list and there is no separate data structure specifically for the Closed list.

We present an example of A*+IDA* in Figure 5.1, where the f-value is at the left of each node, an arrow means the node's f-value is updated during the search, and the number next to each edge is the edge cost. Closed nodes of A* are gray. Solid white nodes are the frontier nodes, which are the Open nodes at the end of the A* phase. Dashed nodes are the nodes generated in the IDA* phase. S is the start node and G is the goal node. In the A* phase, S is expanded and generates A, B, and C, which all have an f-value of 4. Since B

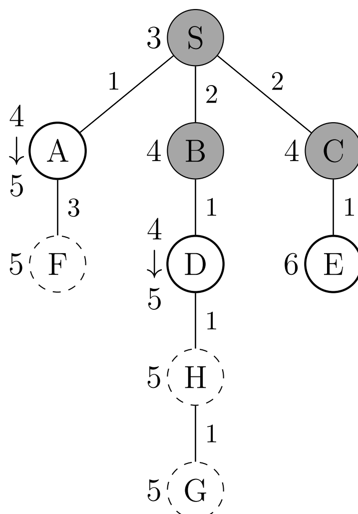

Figure 5.1: An example of A*+IDA*. The f-value is at the left of each node. An arrow means the node's f-value is updated during the search. The number next to each edge is the edge cost. Closed nodes of A* are gray. Solid white nodes are the frontier nodes. Dashed nodes are the nodes generated in the IDA* phase.

and C have an h-value of 2 while A has an h-value of 3, A*+IDA* would expand B and C before expanding A. Suppose A*+IDA* first expands C and generates E with $f(\text{E}) = 6$, then expands B and generates D with $f(\text{D}) = 4$. At this time, assume A*+IDA* has reached its storage limit, which is 6 nodes. Then A*+IDA* begins its IDA* phase with nodes A, D, and E as the frontier nodes, which are the Open nodes at the end of the A* phase. The first cost bound of the IDA* phase is 4, the smallest f-value among A, D, and E. In each iteration of the IDA* phase, IDA* is called on the frontier nodes whose f-value equals the current cost bound in increasing order of their h-values. Therefore, A*+IDA* first calls IDA* with a fixed cost bound of 4 on D and generates H, whose f-value is 5, greater than the cost bound. A*+IDA* will not expand H but updates the stored f-value of D to 5. Then A*+IDA* calls IDA* with a fixed cost bound of 4 on A and generates F, whose f-value is also 5. Thus A*+IDA* will also update A's f-value to 5. At this point, there are no more frontier nodes whose f-value is 4, so A*+IDA*'s IDA* phase will begin its second iteration with the cost bound of 5, the smallest f-value among all frontier nodes. A*+IDA* then calls IDA* with a fixed cost bound of 5 on D, which generates H. This time H is also expanded and generates

the goal node G with an f-value of 5. Since 5 is the current cost bound, A*+IDA* can claim that it has found the optimal solution to this problem with a cost of 5.

A*+IDA* is complete and admissible with admissible heuristics. A*+IDA* potentially makes calls to IDA* on all frontier nodes. When an optimal solution exists, one node on this optimal path serves as the start node for one of the calls to IDA*. Such a node is guaranteed to exist by A*'s completeness and admissibility. When the cost bound for the calls to IDA* equals C^*, the optimal solution will be found, guaranteed by IDA*'s completeness and admissibility.

5.3 Comparisons to Previous Work

5.3.1 A*+IDA* vs. MREC

A*+IDA* is a simple combination of A* and IDA*, so how did this simple algorithm escape notice for so long? The answer seems to be that most researchers believe it was discovered in 1989 as the MREC algorithm [SB89]. For example, [Kor93] described MREC as running A* until memory is full, then running iterations of IDA* below the frontier nodes. Several other researchers repeated this erroneous characterization of MREC [ES00, Fel05, SKF10, AKF10, SDR13].

In fact, as we introduced in Chapter 3, MREC is just IDA* with a transposition table (IDA*+TT) [RM94]. There are four main differences between A*+IDA* and MREC. First, A*+IDA* stores the first nodes generated by A*, while MREC stores the first nodes generated by IDA*. These sets can differ significantly among the nodes of largest f-value that are stored. Second, A*+IDA* starts each call to IDA* from the frontier nodes, while MREC always starts from the start state. Third, during the IDA* phase, A*+IDA* orders the frontier nodes of equal f-value in increasing order of their h-values. Finally, A*+IDA*'s IDA* phase does not check for duplicate nodes, while MREC always does duplicate detection for each node generated. This last difference saves significant overhead for A*+IDA* since accessing a large hash table requires an expensive access to main memory. This expensive access usu-

ally comes from three factors: hash table index calculations, comparisons with stored nodes, and page faults due to the nature of random access to a large hash table. Transposition tables can also use various replacement strategies to dynamically modify the nodes stored, but this adds additional overhead and may not really save search time [AKF10].

As we mentioned before, the first appearance of a version of A*+IDA* in the literature is in [Kor93], where he mistakenly described it as MREC. He also claimed that MREC in [SB89] did not check for duplicate nodes during the IDA* search on the 15-Puzzle, contrary to the pseudo-code in the MREC paper. He implemented a version of A*+IDA*, but with duplicate node checking during the IDA* phase, and no mention of ordering Open nodes. He reported a 41% reduction in node generations, but a 64% increase in running time compared to IDA*, presumably due to checking for duplicate nodes during the IDA* phase.

5.3.2 A*+IDA* vs. A* With Lookahead

As introduced in Chapter 3, A* with lookahead (AL*) [SKF10, BSF14] is also a combination of A* and IDA* that performs a single iteration of IDA* with bound $f(n) + k$ on each generated node n. We found that on our machine, $k = 6$ is the smallest k that enables us to solve all of Korf' 50 24-Puzzle test cases [KF02]. When $k = 6$, AL* is also faster than IDA* on the 24-Puzzle, but slower than A*+IDA*. AL* is similar to A*+IDA* but there are some significant differences. First, A*+IDA* is guaranteed to solve a problem instance if a solution exists. On the other hand, AL* may run out of memory before finding a solution if the problem instance to solve is hard enough and the user-defined k-value is too small. Second, the best k-value in terms of running time is different for each problem instance and we cannot predict whether a certain k-value would be enough for AL* to solve a problem instance. Therefore, to use AL* in practice, a lot of time would be needed to find the correct k-value. A*+IDA* does not have this issue. Third, A*+IDA* can use all the given memory for duplicate detection while AL* may only use a small portion of the available memory. For example, AL* with $k = 3$ may run out of memory before finding a goal node while AL* with $k = 4$ may only store a small number of nodes in memory before termination. In such cases,

a lot of memory could be wasted, reducing the power of duplicate detection in A*. Fourth, AL* may expand a lot of nodes whose f-value is greater than the optimal solution cost even when using an admissible heuristic function, while A*+IDA* is guaranteed to never expand such nodes.

5.3.3 A*+IDA* vs. External IDA*

External IDA* [ES11, Ede16] first runs disk-based A* then runs IDA* on the frontier nodes, which appears similar to our disk-based A*+IDA* that we will introduce later. However, they did not indicate whether or not duplicate detection is performed during the IDA* phase, nor whether any node ordering of the Open nodes is done.

5.4 Experimental Results and Analysis

We first present our results and analysis on the 24-Puzzle, then Rubik's Cube. All experiments were performed on a 3.33GHz Intel Xeon X5680 CPU with 96 GB of RAM.

5.4.1 24-Puzzle

We used the same 6-6-6-6 disjoint PDBs with reflection as we introduced in Figure 2.2 and the same 50 test cases as in [KF02]. During the search, we took the larger heuristic value between the values returned from the default grouping in Figure 2.2a and the reflection grouping in Figure 2.2b. We ran A*+IDA* with 8 GB, 48 GB, and 96 GB of memory, and the upper bounds of nodes stored for each test case were 96,855,259, 774,842,075, and 1,549,684,150, respectively.

We also implemented the original MREC/IDA*+TT [SB89, RM94]. However, because of the high cost of duplicate detection for each node generated during search, it is slower than IDA* by a factor of two to three, so we omit those results here.

For comparison purposes, we also include an algorithm we call Cached-IDA*, which uses

a transposition table (TT) to store the first nodes generated by IDA*. Before the TT is full, Cached-IDA* behaves the same as MREC/IDA*+TT. After TT is full, Cached-IDA* only does duplicate detection for a node if its parent is stored in TT. As a result, when Cached-IDA* generates a node c that is not in TT and TT is full, Cached-IDA* turns off duplicate detection for the search tree below c. Cached-IDA* generates more nodes than the original MREC/IDA*+TT, but is much faster as it avoids duplicate detection for most nodes. Our Cached-IDA* used the same upper bound of stored nodes as A*+IDA* with 48 GB memory.

We present the speedups of A*+IDA* over IDA* in run time on all 50 test cases in Figure 5.2. We sort the 50 test cases according to the number of nodes generated by IDA*, so the left-most test case is the easiest and the right-most test case is the hardest for IDA*. The squares, diamonds, and circles correspond to A*+IDA* with 8 GB, 48 GB, and 96 GB memory respectively. The scattered nature of this plot is due to variation in when the first optimal solution is found on the last iteration.

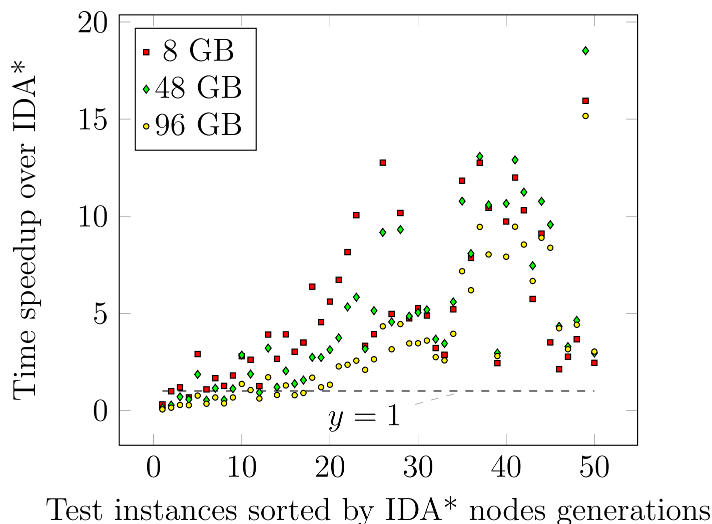

Figure 5.2: A*+IDA*'s speedup over IDA*.

We present the cumulative data of the 50 test cases in Table 5.1, where the first column is the algorithm, the second column is the total number of nodes stored, the third column is the total number of nodes generated, the fourth column is the nodes generated in the last

Algorithm	Stored nodes	Total nodes	Last iteration	Time(s)
IDA*	0	18,044,623,983,548	9,861,089,635,700	572,845
Cached-IDA*	33,149,796,158	10,416,963,057,231	5,703,252,119,005	446,293
A*+IDA*(8 GB)	4,565,459,317	4,552,871,977,112	454,939,266,782	150,387
A*+IDA*(48 GB)	30,491,473,996	3,143,297,938,874	24,670,542,049	116,963
A*+IDA*(96 GB)	56,833,924,109	2,839,875,132,068	24,133,797,786	129,871

Table 5.1: Cumulative data for A*+IDA* on the 24-Puzzle.

iteration, and the last column is total time in seconds for all 50 problems.

Over all 50 test cases, A*+IDA*'s speedup over IDA* is 3.8 (8 GB), 4.9 (48 GB), and 4.4 (96 GB) respectively. A* solved 5, 15, and 19 test cases respectively before exhausting memory. Due to A*'s expensive memory and duplicate detection operations, A* is slower than IDA* in terms of nodes generated per second by a factor of 7. As a result, A*+IDA* is slower than IDA* on the easy test cases, where most or all search time was spent on the A* phase of A*+IDA*. On the medium and hard test cases, A*+IDA* is consistently faster than IDA*. A*+IDA* (8 GB) is also faster than the 48 GB and 96 GB versions on easy test cases as its A* phase is much shorter than that of the 48 GB and 96 GB versions. Increasing the memory from 8 GB to 48 GB reduced the total number of nodes generated by 31.0%, while increasing the memory from 48 GB to 96 GB only reduced the nodes by an additional 9.7%.

In Table 5.1, the overall nodes ratio for A*+IDA* (48 GB) is 5.74, which is higher than the speedup ratio of 4.9. There are two reasons for this. First, as discussed above, A* is slower than IDA* in terms of nodes generated per second. However, this is a minor reason as the nodes generated in A* are less than 1% of the total generated nodes in A*+IDA*. The major reason is that the IDA* phase in A*+IDA* is always slower (in terms of nodes generated per second) than pure IDA* due to the overhead for each separate call to IDA*. This overhead mainly comes from fetching the start node from a large hash table, which

usually causes cache misses. This is also the reason why A*+IDA* (96 GB) is slower than A*+IDA* (48 GB). The former generated fewer nodes, but as the frontier nodes doubled, this overhead for all IDA* runs also doubled.

The speedup comes from three factors: fewer IDA* iterations, fewer duplicate nodes, and early goal termination. A*+IDA* typically performs only two to five iterations in its IDA* phase, while pure IDA* typically performs around 10 iterations. However, this is a minor reason as the last two iterations dominate the search. A*+IDA* (48 GB) reduced the nodes generated in all but the last iteration by a factor of 2.62 over IDA* and 1.51 over Cached-IDA*. By starting from the frontier nodes instead of the start state, we do not need to re-generate the nodes above the frontier nodes and the search tree generated by A*+IDA*'s IDA* phase is much smaller than that of IDA*. Our data shows that on average, each separate call to the single-bound IDA* in A*+IDA* (48 GB) only generated 190 nodes. This number is dominated by the hard test cases and for most test cases, this number is below 50. The better duplicate pruning effect of A*+IDA* over Cached-IDA* also suggests that expanding and storing nodes in a best-first order with tie-breaking by h-value is better than depth-first order, as the latter can be heavily biased to the left part of the search tree.

The last iteration of IDA* is usually the most time-consuming iteration. In Table 5.1, 54.6% of IDA* nodes were generated in the last iteration. A*+IDA* reduced the nodes generated in the last iteration by 400 times, making it less than 1% of the total nodes generated. We present the detailed last iteration comparison in Figure 5.3, where the y-axis is the number of nodes generated in IDA*'s last iteration over A*+IDA*'s. We see that the 8 GB version usually leads to more nodes generated, but there is not much difference between the 48 GB and 96 GB versions. For the 48 GB and 96 GB versions, A*+IDA* reduced the number of nodes in the last iteration by more than three orders of magnitude on most test cases. This result shows that by expanding frontier nodes of equal f-values in increasing order of h-values (decreasing order of g-values), the goal state was found very early in the final iteration. It also suggests that finding the optimal solution is much easier than verifying optimality. Expanding nodes according to the f and h-values to significantly

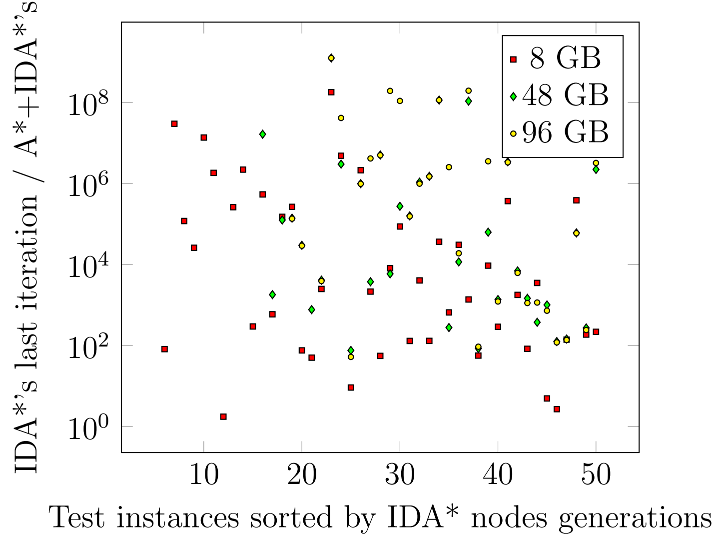

Figure 5.3: A*+IDA*'s last iteration vs. IDA*'s.

reduce the number of nodes generated in the last iteration was first done in [PK89]. For example, IDA* can order a node's child nodes based on their h-values to decide which node to expand next [PK89]. However, as noted in [PK89], the benefit of this limited node ordering may be mitigated by its extra cost. This may be the reason why node ordering was not commonly used in IDA* in the past. However, we show that effective node ordering can be easily achieved by utilizing A*+IDA*'s frontier list.

We further applied the above node ordering on Cached-IDA*'s last iteration. We scanned the transposition table, selected the nodes that do not have all their children stored in the transposition table, pruned the nodes whose f-values exceed the bound, then sorted the nodes in increasing order of stored h-values and expanded them. In this way, Cached-IDA* begins each iteration from the frontier nodes instead of the root node, and we call the resulting algorithm Cached-IDA*+node ordering. We present the results for the 31 test cases that A* (96 GB) did not solve in Table 5.2. We only list the number of nodes generated in the last iteration. IDA* generated 9,789,829,523,428 nodes in its last iteration on these 31 test cases. We see that A*+IDA* consistently generated fewer nodes than Cached-IDA*+node ordering. The most significant difference is under the 48 GB setting where Cached-IDA*+node ordering generated 13 times more nodes than A*+IDA*. This is

Memory	A*+IDA*	Cached-IDA*+node ordering
8 GB	453,763,507,684	557,116,891,535
48 GB	24,636,552,525	320,412,253,835
96 GB	24,133,797,786	40,276,211,512

Table 5.2: Nodes generated in the last iteration in A*+IDA* and Cached-IDA*+node ordering.

largely due to one test case where Cached IDA*+node ordering generated 280,591,231,353 nodes, proving that even with node ordering, Cached-IDA*+node ordering can sometimes find the goal state very late in the last iteration. Similar to Cached-IDA*'s less powerful duplicate detection compared to A*+IDA*, this late termination in the last iteration also comes from the stored nodes being biased towards the left part of the search tree.

Another way to use memory to speedup IDA* is to build larger PDBs. We built 8-8-7-1, 7-7-7-3, 7-7-6-4, and 7-7-5-5 disjoint PDBs and found that none of them is better than the standard 6-6-6-6 disjoint PDBs for the 24-Puzzle. The reason is that they all produce more small heuristic values than the 6-6-6-6 disjoint PDBs (see [HNF04] for a more detailed discussion of the effect of small heuristic values).

5.4.2 Rubik's Cube

We randomly generated 100 test cases for Rubik's Cube. We built one PDB based on 8 corner cubies and two PDBs each based on 9 edge cubies, similar to the PDBs in [Kor97], and took the maximum value among all three PDBs. These three PDBs require 38 GB of memory. We present the results in Table 5.3, where the first column is the algorithm, the second column is the number of nodes stored by A*+IDA*'s A* phase, the third column is the total number of nodes generated, the fourth column is the nodes generated in the last iteration, and the last column is time. Our A*+IDA* stored the same number of nodes (774,842,075) as A*+IDA* (48 GB) on the 24-Puzzle, resulting a 32 GB hash map for the

Algo.	A* stored	Total nodes	Last iteration	Time (s)
IDA*	0	918,896,332,771	576,086,877,811	323,083
A*+IDA*	69,909,543,136	710,571,584,688	381,865,691,696	325,950

Table 5.3: A*+IDA* vs. IDA* on Rubik's Cube.

A* phase.

A*+IDA* is 1% slower than IDA* and only reduced the nodes generated in IDA* by 23%. There are two reasons for this: 1) With the move pruning used in [Kor97], there are few duplicate nodes generated in IDA*. Depth-first search only generates 2.1%, 3.1%, 4.2%, 5.4%, 6.6%, and 8.0% more nodes as duplicates at depth 7, 8, 9, 10, 11, and 12 respectively (`http://www.cube20.org` and [Kor97]). Most of the frontier nodes stored by A*+IDA* are at depth 7 to 12. Therefore, there will not be much gain in duplicate detection as there do not exist many duplicates. 2) A*+IDA* generated more nodes than IDA* in the last iteration on 19 out of 82 test cases that A* did not solve, with the highest ratio being 39. The reason is that the frontier nodes that are on the optimal paths sometimes have a perfect heuristic value and hence a small g-value, causing them to be expanded very late in the last iteration. For IDA*, the expansion order is based on the order of operators. The lesson here is that it is hard to get good node ordering for Rubik's Cube.

As we mentioned before, there is an overhead with each call to the single-bound IDA*, so A*+IDA*'s IDA* phase generated fewer nodes per second than pure IDA*. The time spent in the A*-phase, lower speed in the IDA*-phase, rare duplicate nodes, and less powerful early termination in the last iteration make A*+IDA* not faster than IDA* on Rubik's Cube.

5.5 Combining Dual Search With A*+IDA*

We further applied dual search [ZFH08] to the IDA* phase of our A*+IDA*. Dual search (DS) is very complicated and we choose not to explain it here. This section does not require

the reader to understand DS.

[ZFH08] compared their DIDA* (IDA* with DS) code to an inefficient IDA* implementation which does 8 PDB lookups for each node. However, an efficient IDA* implementation only does two PDB lookups for each node. We compared their DIDA* with Korf's efficient IDA* code [KF02] and found that their DIDA* is only 8.5% faster than Korf's efficient IDA* code.

The original DIDA* implementation in [ZFH08] performs 8 PDB lookups for a child state and 8 PDB lookups for the child's dual state, a total of 16 PDB lookups. We optimized the DIDA* code to perform only 4.435 PDB lookups in total for each state and its corresponding dual state. We also discovered an error in the original Dual PDB lookup and Dual search implementation in [FZS05, ZFH06], resulting in 32% of the total generated nodes' regular PDB lookups and reflection PDB lookups always returning the same value. We omit the details of our optimizations here.

Algorithm	Total nodes	Time (s)	Speedup
IDA*	18,044,623,983,548	572,845	1
Original DIDA*	3,876,071,228,494	523,948	1.1
Our DIDA*	3,684,978,748,642	259,506	2.2
Original A*+IDA*	3,143,297,938,874	116,963	4.9
A*+IDA* w/ DS	858,888,100,914	85,405	6.7

Table 5.4: A*+IDA* combined with dual search.

We present the results in Table 5.4, where the first column is the algorithm, the second column is the total number of nodes generated, the third column is time, and the last column is the speedup over IDA*. We used the same 50 test cases as before. [ZFH06] used bidirectional pathmax (BPMX) [FZS05] in their DIDA*. BPMX, which is used with inconsistent heuristics, propagates greater heuristic values from a child node to its parent and siblings, thus reducing node expansions. BPMX reduced the number of nodes generated

during search by about 2%. To be consistent with IDA*, we did not use BPMX in Table 5.4.

From Table 5.4 we see that the original DIDA* is only 8.5% faster than Korf's efficient IDA* code. After our optimization and bug fix, DIDA* is faster than Korf's efficient IDA* code by a factor of 2.2. Our version of DIDA* is by far the fastest linear search algorithm on the 24-Puzzle. After we combined dual search with A*+IDA*, the speedup increases from 4.9 to 6.7. Contrary to the original dual search paper, we treat a state and its dual state as two distinct states. Therefore, the number of nodes for the original DIDA* is roughly two times that in [ZFH08]. The number is only approximate because we did not use BPMX.

5.6 Disk-based A*+IDA*

We then applied delayed duplicate-detection (DDD) [Kor04] to A*+IDA*. Traditional A* performs duplicate detection whenever a node is generated. A* with DDD first generates and stores nodes in external-memory without duplicate detection up to a certain limit of expansions or generations. Then A* with DDD loads nodes back from external-memory and prunes the duplicate nodes. Our implementation is similar to External IDA* [ES11, Ede16]. However, they did not indicate whether or not duplicate detection was performed during the IDA* phase, nor whether any node ordering of the frontier nodes was done. They reported data for only two instances of the 24-Puzzle, did not compare it to IDA*, and did not report any running times.

We first ran disk-based A* with DDD up to 10.2 billion expansions, then we removed the duplicates and ran the IDA* phase on the frontier nodes. Common disk-based A* assumes consistent heuristics [ES11]. However, the 6-6-6-6 PDBs for the 24-Puzzle is an inconsistent heuristic and pathmax cannot turn it into a consistent heuristic [Hol10]. Therefore, re-expanding the same state in A* cannot be avoided. This is not mentioned in [ES11]'s disk-based A* or External IDA* explanation. Our results show that disk-based A*+IDA* is only faster than RAM-based A*+IDA* on extremely hard problem instances. The reason is the same as why the 96 GB A*+IDA* version is slower than the 48 GB version: 1) more

State	Length
s1: 23 1 12 6 16 2 20 10 21 18 14 13 17 19 22 0 15 24 3 7 4 8 5 9 11	113
s2: 0 19 3 14 17 13 21 4 10 22 12 11 9 15 24 18 7 2 6 8 5 1 23 16 20	108
s3: 3 5 0 15 4 13 22 2 21 23 14 7 16 17 10 11 1 8 12 24 9 20 19 18 6	104

Table 5.5: Three hard 24-Puzzle test cases in Table 5.6.

Algo. & Test case	Total nodes	Time (s)	Speedup
A*+IDA*(48 GB), s1	1,343,383,770,705	44,602	1
A*+IDA*(Disk), s1	816,569,465,508	41,085	1.09
A*+IDA*(48 GB), s2	5,011,462,601,471	165,972	1
A*+IDA*(Disk), s2	1,620,137,788,282	76,547	2.17
A*+IDA*(48 GB), s3	3,039,454,921,549	100,722	1
A*+IDA*(Disk), s3	2,043,687,639,368	88,428	1.14

Table 5.6: Disk-based A*+IDA* vs. 48 GB RAM-based A*+IDA* on three hard 24-Puzzle test cases.

time is spent on the A* phase and 2) with more frontier nodes, the overhead for initiating IDA* on the frontier nodes increases.

We randomly generated 150 test cases and found two (s2 and s3) that are harder than the most difficult test case (s1) we used before. We present these three test cases and their solution lengths in Table 5.5 and the results in Table 5.6, where the first column is the algorithm and test case, the second column is the total number of nodes generated in A*+IDA*, the third column is time, and the last column is the speedup of the disk-based version over the 48 GB RAM-based version. Disk-based A*+IDA* is faster than RAM-based A*+IDA* on all three problems. On s2, the disk-based version reduced the nodes by a factor of three and achieved a speedup of 2.17. On s3, disk-based A*+IDA* reduced the nodes by one third and search time by 12%. The disk-based version performed better on s2 because

it reduced the number of nodes generated by 2,479 billion in the last iteration.

5.7 Parallel A*+IDA* on the 27 and 29-Puzzle

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

(a) First 7-7-7-6 PDBs.

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

(b) Second 7-7-7-6 PDBs.

Figure 5.4: Two 7-7-7-6 disjoint PDBs for the 27-Puzzle.

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

(a) First 7-7-7-7-1 PDBs.

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

(b) Second 7-7-7-7-1 PDBs.

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

(c) First 6-6-6-6-5 PDBs.

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

(d) Second 6-6-6-6-5 PDBs.

Figure 5.5: Disjoint PDBs for the 29-Puzzle.

We implemented a multi-threaded A*+IDA* and tested it on the 4×7 (27) and $5 \times$

6 (29) sliding-tile puzzles. We first ran a single-threaded A*, then started multiple calls to single-bound IDA* in parallel with each thread given a different frontier node. Unlike the 24-Puzzle, 27 and 29-Puzzles cannot use reflections for PDB lookups. For the 27-Puzzle, we built two 7-7-7-6 PDBs (and took their maximum) whose sparse mapping [FKM07] required 26 GB in total. For the 29-Puzzle, we built two 7-7-7-7-1 PDBs and two 6-6-6-6-5 PDBs (and took their maximum) whose sparse mapping required 64 GB in total. These PDBs are shown in Figure 5.4 and Figure 5.5 respectively.

We randomly generated 10 instances for each puzzle. To date, we have found the optimal solution for 5 and 4 instances for the 27 and the 29-Puzzle respectively. Those instances took from 4 minutes to 15 hours with 12 threads of single-bound IDA* running in parallel in the IDA* phase on our machine. To the best of our knowledge, these are the first random instances of a sliding-tile puzzle larger than the 24-Puzzle to be solved optimally. For the remaining instances, we have found solutions but not yet verified their optimality. The solutions we found range from 108 to 149 for the 27-Puzzle and 117 to 152 for the 29-Puzzle. The average (not guaranteed to be optimal) lengths for the solutions we found are 123.6 and 129.4 respectively. The average heuristic values for one million random states are 81.6, 100.3, and 107.6 for the 24, 27, and 29-Puzzles respectively.

5.8 An Approximation Algorithm Based on A*+IDA*

For the sliding-tile puzzles, A*+IDA* can be easily converted to an approximation algorithm that enables us to quickly find a solution. For example, in each iteration, we can only call the single-bound IDA* on the top 20% of the frontier nodes (sorted by h-values) that have an f-value equal to the current cost bound. Suppose a solution is not found after those 20% frontier nodes are expanded, we then just increase the f-value of the remaining frontier nodes that can be expanded in this iteration to the next cost bound and then we begin the next iteration. If a solution is found, we decrease the bound to verify the optimality or find a better solution.

We used this approximation algorithm to quickly find solutions for the 10 test cases for the 27 and 29-Puzzles. Most of the nodes generated in the IDA* phase are generated below the frontier nodes with high h-values. For example, the last 5% of the frontier nodes would generate more than two orders of magnitude more nodes than the first 5% of the frontier nodes. Therefore, even expanding the first half of the frontier nodes would generate only a small portion of the nodes generated below the second half of the frontier nodes. The optimal solutions are always generated very early in the last iteration, so the first solution found by this algorithm is usually the optimal one. However, our analysis suggests that it would take months to years to verify the optimality of the hardest 27 and 29-Puzzle test cases we generated, so we cannot claim to have optimally solved these instances.

This approximation algorithm is based on the property that A*+IDA* usually finds the solution very early in its last iteration. Given a domain and a heuristic function combination where A*+IDA* does not have such property, this approximation algorithm may not work well.

CHAPTER 6

A*+BFHS: A Hybrid Heuristic Search Algorithm

We present a new algorithm called A*+BFHS for solving problems with unit-cost operators where A* and IDA* fail due to memory limitations and/or the existence of many distinct paths between the same pair of nodes. A*+BFHS is based on A* and breadth-first heuristic search (BFHS). A*+BFHS combines advantages of both algorithms, namely A*'s node ordering, BFHS's memory savings, and both algorithms' duplicate detection. On easy problem instances, A*+BFHS behaves the same as A*. On hard problem instances, it is slower than A* but saves a large amount of memory. Compared to BFIDA*, A*+BFHS reduces the search time and/or memory requirement by several times on a variety of planning domains.

Our work on A*+BFHS is presented in the following order. Section 6.1 discusses the motivation of this research. Section 6.2 presents the new algorithm A*+BFHS. Section 6.3 compares A*+BFHS with previous algorithms and discusses the differences. Section 6.4 describes the solution reconstruction strategy we used for A*+BFHS in our experiments. Section 6.5 presents experimental results of A*, BFIDA*, and A*+BFHS on 32 hard instances from 18 unit cost International Planning Competition (IPC) domains.

6.1 Motivation

Compared to IDA*, A*+IDA* reduces the number of duplicate nodes expanded and may reduce the number of nodes expanded in the last iteration, depending on the domain and heuristic functions. However, on a domain where many distinct paths exist between the same pair of nodes, for example a grid graph or many planning domains, A*+IDA* would still expand many duplicate nodes during its IDA* phase. This raises the question: how to

further reduce the number of duplicate nodes expanded by A*+IDA*?

On the other hand, BFHS and BFIDA* solve the duplicate nodes issue while still requiring less memory than A* by using a breath-first search order. However, as we show in Section 4.5, BFIDA* would then expand almost all nodes whose f-value equals the optimal solution cost C^*, hence end up expanding and generating many more nodes than A*. Therefore, BFIDA* is usually much slower than A* due to its almost worst-case node ordering scheme.

Enlightened by BFIDA*'s memory-efficient duplicate detection scheme, we came up with the idea: of replacing depth-first search with breadth-first search in A*+IDA* to reduce the number of duplicate nodes expanded. Based on this idea, we present our new algorithm A*+BFHS.

6.2 A*+BFHS

We propose A*+BFHS, a hybrid algorithm to solve problems with many distinct paths between the same pair of nodes. A*+BFHS first runs A* until a storage threshold is reached, then runs a series of BFHS iterations on sets of frontier nodes, which are the Open nodes at the end of the A* phase.

The BFHS phase can be viewed as a doubly nested loop. Each iteration of the outer loop, which we define as an iteration of the BFHS phase, corresponds to a different cost bound for BFHS. The first cost bound is set to the smallest f-value among all frontier nodes. In each iteration of the BFHS phase, we first partition the frontier nodes whose f-value equals the cost bound into different sets according to their depths. Then the inner loop makes one call to BFHS on each set of frontier nodes, in decreasing order of their depths. This is done by initializing the BFS queue of each call to BFHS with all the nodes in the set. This inner loop continues until a solution is found or all calls to BFHS with the current bound fail to find a solution. After each call to BFHS on a set of frontier nodes, we increase the f-value of all nodes in the set to the minimum f-value of all nodes generated but not expanded in the previous call to BFHS.

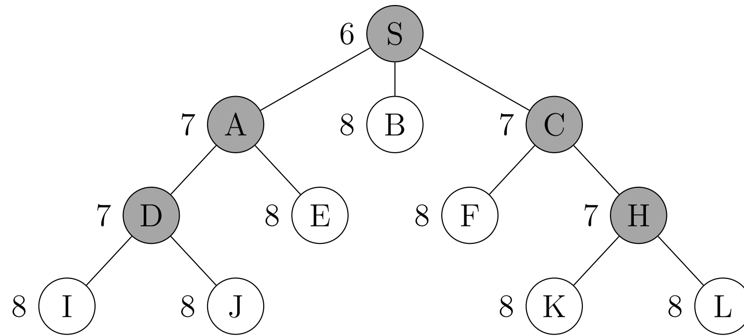

Figure 6.1: An example of A*+BFHS's search frontier. Numbers are f-values. Closed nodes are gray.

Figure 6.1 presents an example of the Open and Closed nodes at the end of the A* phase. Node S is the start node. All edge costs are 1 and the number next to each node is its f-value. Closed nodes are gray. The Open nodes B, E, F, I, J, K, L are the frontier nodes for the BFHS phase. A*+BFHS first makes a call to BFHS with a cost bound of 8 on all frontier nodes at depth 3, namely nodes I, J, K, L. If no solution is found, A*+BFHS updates the f-values of all these nodes to the minimum f-value of all the nodes generated but not expanded in that call to BFHS. A*+BFHS then makes a second call to BFHS with bound 8, starting with all frontier nodes at depth 2, namely nodes E and F. If no solution is found, A*+BFHS updates the f-values of these nodes, then makes a third call to BFHS with bound 8, starting with the frontier node B at depth 1. Suppose that no solution is found with bound 8, the updated f-values for nodes E, F, I, J, K, L are 9, and the updated f-value for node B is 10. A*+BFHS then starts a new iteration of BFHS with a cost bound of 9, making two calls to BFHS on nodes at depth 3 and 2 respectively. If the solution is found in the first call to BFHS with bound 9, BFHS will not be called again on nodes E and F.

A*+BFHS is complete and admissible with admissible heuristics. A*+BFHS potentially makes calls to BFHS on all frontier nodes. When an optimal solution exists, one node on this optimal path serves as one of the start nodes for one of the calls to BFHS. Such a node is guaranteed to exist by A*'s completeness and admissibility. When the cost bound for the calls to BFHS equals C^*, the optimal solution will be found, guaranteed by BFHS's

completeness and admissibility.

A state can be regenerated in separate calls to BFHS in the same iteration. To reduce such duplicates, we can decrease the number of calls to BFHS in each iteration by making each call to BFHS on a combined set of frontier nodes at adjacent depths. For the example in Figure 6.1, we can make one call to BFHS on the frontier nodes at depths 2 and 3 together instead of two separate calls to BFHS, by putting the frontier nodes at depth 3 after the frontier nodes at depth 2 in the initial BFS queue.

In practice, we can specify a maximum number of calls to BFHS per iteration. Then in each iteration, we divide the number of depths of the frontier nodes by the number of calls to BFHS to get the number of depths for each call to BFHS. For example, if the depths of the frontier nodes range from 7 to 12 and we are limited to three calls to BFHS per iteration, each call to BFHS will start with frontier nodes at two depths. We used this strategy in our experiments.

For each node generated in the BFHS phase, we check if it was generated in the A* phase. If so, we immediately prune the node if its current g-value in the BFHS phase is greater than or equal to its stored g-value in the A* phase.

The primary purpose of the A* phase is to build a frontier set, so that A*+BFHS can terminate early in its last iteration. In the A* phase we have to reserve some memory for the BFHS phase. In our experiments, we first generated pattern databases or the merge-and-shrink heuristic, then allocated 1/10 of the remaining memory of 8 GB for the A* phase.

6.3 Comparisons to Previous Work

6.3.1 A*+BFHS vs. BFIDA*

A*+BFHS's BFHS phase also uses the iterative-deepening concept of BFIDA*, but there are two key differences. First, in each iteration, BFIDA* always makes one call to BFHS on the start node, while we call BFHS multiple times, each on a different set of frontier nodes.

Second, in each iteration, we order the frontier nodes based on their depth, and run BFHS on the deepest frontier nodes first.

These differences lead to one drawback and two advantages. The drawback is that A*+BFHS may generate more nodes than BFIDA*, as the same state can be regenerated in separate calls to BFHS in the same iteration.

The first advantage is that A*+BFHS may terminate early in its last iteration. If A*+BFHS generates a goal node in the last iteration below a relatively deep frontier node, no frontier nodes above that depth will be expanded. Therefore, A*+BFHS may generate only a small number of nodes in its last iteration. In contrast, BFIDA* has to expand almost all nodes whose f-value is less than or equal to C^* in its last iteration. As a result, A*+BFHS can be faster than BFIDA*.

The second advantage is that A*+BFHS's memory usage, which is the maximum number of nodes stored during the entire search, may be smaller than that of BFIDA* for two reasons. First, the partition of frontier nodes and separate calls to BFHS within the same iteration can reduce the maximum number of nodes stored in the BFHS phase. Second, BFIDA* stores the most nodes in its last iteration while A*+BFHS may store only a small number of nodes in the last iteration due to early termination. Thus, A*+BFHS may store the most nodes in the penultimate iteration instead.

6.3.2 A*+BFHS vs. FPS

Forward Perimeter Search (FPS) [SDR13] first builds a perimeter from the start state by using BFS, then runs BFIDA* from each node in the perimeter to find a solution. A perimeter is a set of nodes that have the same or similar distance from the start state. After the perimeter is built, FPS switches to its BFIDA* phase and uses the lowest f-value among all perimeter nodes as the first cost bound. In each iteration of the BFIDA* phase, FPS first sorts the perimeter nodes using a max-tree-first or longest-path-first policy, then calls BFHS on each perimeter node and updates the f-value of each perimeter node to the smallest f-value of the node generated but not expanded in that single call to BFHS. Here the max-

tree-first policy means that in each iteration, FPS first calls BFHS on the perimeter node that generated the most nodes in the previous iteration, while the longest-path-first policy means that FPS first calls BFHS on the perimeter node that led to the longest path (largest g-value generated in a single call to BFHS) in the previous iteration.

FPS looks similar to A*+BFHS, but there are several fundamental differences. First, FPS builds the perimeter using a breadth-first approach while A*+BFHS builds the frontier via a best-first approach. FPS can also dynamically extend the perimeter but this approach does not always speed up the search [SDR13]. Second, in each iteration of FPS's BFIDA* phase, FPS makes one call to BFHS on each perimeter node. In contrast, in A*+BFHS each call to BFHS is on a set of frontier nodes. Third, FPS sorts the perimeter nodes at the same f-value using a max-tree-first or longest-path-first policy, while A*+BFHS sorts the frontier nodes at the same f-value in decreasing order of their depth. Fourth, FPS needs two separate searches for solution reconstruction while A*+BFHS only needs one.

6.4 Solution Reconstruction

Each node generated in A*+BFHS's BFHS phase has a pointer to its frontier node from the A* phase. When a goal node is generated, the solution path from the start node to this node is stored in the A* phase and only one more search is needed to reconstruct the solution path from this node to the goal node. This subproblem is much easier than the original problem and we can use the same heuristic function as for the original problem. Therefore, we just use A* to solve this subproblem. In addition, since we know the optimal cost of this subproblem, we can immediately prune any node whose f-value exceeds this cost. During our experiments, such subproblems were all easily solved by A*. However, if A* runs out of memory on this subproblem, we can still switch to A*+BFHS or BFHS to solve this subproblem.

In BFIDA*, we have to solve two subproblems to recover the paths from the start node to the middle node and from the middle node to the goal node. [ZH04] called BFHS recursively

to solve these two subproblems. However, pattern database heuristics (PDB) [CS98] only store heuristic values to the goal state, and not between arbitrary pairs of states, which complicates finding a path to a middle node. Similar to A*+BFHS, we use A* to solve the second subproblem. For the first subproblem, we use A* to compute the path from the start node to the middle node using the same heuristic function as for the original problem, which measures the distance to the goal node, not the middle node. To save memory, we prune any node whose g-value is greater than or equal to the depth of the middle node, and any node whose f-value exceeds the optimal cost of the solution. We saved the layer at the 1/4 point of the solution length as the middle layer instead of the 3/4 point in [ZH04]. In this way, we do not need to build a new heuristic function for the middle node. In our experiments, the search time for solution reconstruction in BFIDA* was usually less than 1% of the total search time.

When using A* to solve the two subproblems, the choice of the depth of the middle layer in BFIDA* involves a tradeoff between the middle layer size and the search cost of the two subproblems. A shallower depth of the middle layer, for example 1/8 point or even 1/16 point, would in general lead to a smaller middle layer to store and an easier subproblem from the original start node to the middle node, but a harder subproblem from the middle node to the goal node. On the other hand, a deeper depth of the middle layer, for example 1/2 point, would in general lead to a larger middle layer and a harder subproblem from the original start node to the middle node, but an easier subproblem from the middle node to the goal node.

Instead of using A*, if we call BFHS resursively to solve these two subproblems, then the ideal choice of the depth of the middle layer would be at the 1/2 point. However, in this case, we either have to build PDB heuristics for the middle node, or we user other heuristic functions like the LM-cut heuristics for the middle node. This approach is in general much more complicated and won't speed up search, so we did not use it in our experiments.

6.5 Experimental Results and Analysis

We implemented BFIDA*, A*+IDA*, and A*+BFHS in the planner Fast Downward 20.06 [Hel06], using the existing code for node expansion and heuristic functions. A*+BFHS's A* phase used the existing A* code. A* stores all nodes in one hash map. We used the same hash map implementation with the following difference. In each call to BFHS in both BFIDA* and A*+BFHS, we saved three depth layers of nodes for duplicate detection and we created one hash map for each layer of nodes. We did this because storing all nodes in one hash map in BFHS involves a lot of overhead, and is more complicated. [SDR13] did not test FPS on planning domains and we do not know the optimal perimeter radius and sorting strategy for each domain, so we did not implement FPS.

We solved about 550 problem instances from 32 unit-cost domains. We present the results of A*, BFIDA*, and A*+BFHS on the 32 hardest instances. All remaining instances were easily solved by A* in seconds. We tested two A*+BFHS versions. A*+BFHS (∞) starts each call to BFHS on frontier nodes at a single depth. A*+BFHS (4) makes each call to BFHS on frontier nodes at multiple depths with at most four calls to BFHS in each iteration. All tests were run on a 3.33 GHz Intel Xeon X5680 CPU with 236 GB of RAM. We used the landmark-cut heuristic (LM-cut) [HD09] for the *satellite* domain, the merge-and-shrink heuristic (M&S) with the recommended configuration [SWH14, SWH16, Sie18] for the *tpp* and *hiking14* domains, and the iPDB heuristic with the default configuration [HBH07, SOH12] for all other domains.

We present the results in Tables 6.1, 6.2, 6.3, and 6.4. Tables 6.1, 6.2, and 6.3 contain the 26 instances solved by A*. Table 6.4 contains the remaining 6 instances where A* terminated early without finding a solution due to the limitation of the hash map size in Fast Downward 20.06. The instances in Tables 6.1, 6.2, and 6.3 are sorted by the A* running times and the instances in Table 6.4 are sorted by the BFIDA* running times.

All four tables have the same columns. The first column gives the domain name, the instance ID, the optimal solution cost C^*, and the heuristic function used. The second

Instance	Algorithm	Peak stored	Total nodes	Prev. iterations	Last iteration	Time (s)
depot	A*	70,504,763	344,658,749	344,639,234	19,515	**233**
14	BFIDA*	**17,042,841**	1,390,466,785	582,348,193	795,336,992	1,708
C^*=29	A*+BFHS (∞)	21,023,657	556,674,817	540,764,124	15,909,899	596
iPDB	A*+BFHS (4)	22,882,537	446,204,987	432,278,188	13,926,005	475
termes18	A*	80,012,545	211,514,579	211,514,568	11	**245**
05	BFIDA*	**9,370,587**	3,757,844,868	3,413,500,020	221,186,298	4,796
C^*=132	A*+BFHS (∞)	30,874,300	10,702,979,649	10,701,959,808	911,786	15,415
iPDB	A*+BFHS (4)	30,076,170	2,271,661,960	2,270,262,609	1,291,296	3,319
freecell	A*	53,080,996	243,947,771	243,244,703	703,068	**250**
06	BFIDA*	38,054,162	1,220,132,074	732,920,409	485,268,534	1,883
C^*=34	A*+BFHS (∞)	**30,481,377**	327,209,951	312,812,283	14,388,579	441
iPDB	A*+BFHS (4)	35,120,076	403,465,250	302,581,091	100,875,070	561
logistics00	A*	57,689,357	107,083,712	106,929,666	154,046	**255**
14-1	BFIDA*	**15,441,813**	3,137,204,256	106,929,666	3,020,315,591	10,381
C^*=71	A*+BFHS (∞)	19,472,255	354,438,805	354,058,774	368,595	1,160
iPDB	A*+BFHS (4)	20,169,648	227,903,318	110,674,320	117,217,562	752
driverlog	A*	144,065,288	420,609,830	420,609,777	53	**344**
12	BFIDA*	35,034,406	1,718,350,515	678,644,177	1,030,180,074	1,676
C^*=35	A*+BFHS (∞)	**24,712,720**	1,020,438,794	1,020,410,754	27,959	944
iPDB	A*+BFHS (4)	30,270,816	643,723,984	641,790,459	1,933,444	631
freecell	A*	107,183,015	531,379,136	531,378,858	278	**522**
07	BFIDA*	77,196,602	4,152,881,254	2,897,339,576	1,143,762,584	6,416
C^*=41	A*+BFHS (∞)	**54,171,433**	3,095,608,289	2,370,094,738	725,267,629	4,775
iPDB	A*+BFHS (4)	58,058,327	2,430,947,097	1,896,369,611	534,331,564	3,769
depot	A*	172,447,963	764,608,339	764,607,971	368	**550**
11	BFIDA*	**27,192,174**	3,037,154,042	1,260,718,486	1,755,157,316	3,544
C^*=46	A*+BFHS (∞)	37,977,775	6,268,318,349	3,092,746,859	3,175,552,575	7,314
iPDB	A*+BFHS (4)	46,923,423	3,319,995,622	1,262,429,685	2,057,547,022	4,078
tpp	A*	187,011,066	610,996,630	610,995,018	1,612	**562**
11	BFIDA*	93,759,836	4,290,825,940	754,905,369	3,525,135,895	7,214
C^*=51	A*+BFHS (∞)	**30,856,159**	5,504,314,294	5,504,268,064	46,111	9,550
M&S	A*+BFHS (4)	33,368,912	1,419,143,562	1,285,410,734	133,732,709	2,426
mystery 14	A*	139,924,686	652,569,481	650,036,341	2,533,140	**578**
C^*=11	BFIDA*	135,963,227	6,213,135,253	727,753,687	5,430,082,105	7,628
iPDB	A*+BFHS (∞/4)	**20,302,860**	730,971,724	676,473,465	54,497,630	839

Table 6.1: Instances sorted by A* running times. An underline means more than 8 GB of memory was needed. Smallest memory and shortest times are in boldface.

Instance	Algorithm	Peak stored	Total nodes	Prev. iterations	Last iteration	Time (s)
tidybot11	A*	69,953,936	171,363,621	170,286,720	1,076,901	**662**
17	BFIDA*	42,080,838	776,084,110	486,518,217	281,131,278	3,684
C^*=40	A*+BFHS (∞)	**33,969,968**	661,386,777	467,282,853	194,103,710	3,223
iPDB	A*+BFHS (4)	37,090,062	547,745,706	397,125,094	150,620,398	2,694
logistics00	A*	82,161,805	167,974,727	163,970,672	4,004,055	**663**
15-1	BFIDA*	**13,638,319**	2,847,571,079	163,970,672	2,660,698,165	19,062
C^*=67	A*+BFHS (∞)	18,827,830	730,154,067	722,390,335	7,763,336	4,897
iPDB	A*+BFHS (4)	18,827,830	251,960,077	198,537,096	53,422,585	1,627
pipesworld-	A*	123,553,926	284,884,903	284,880,335	4,568	**727**
notankage 19	BFIDA*	86,818,434	1,227,115,669	634,454,295	576,633,809	4,140
C^*=24	A*+BFHS (∞)	**42,192,503**	619,095,459	619,013,855	81,147	2,072
iPDB	A*+BFHS (4)	44,706,153	574,957,328	570,451,612	4,505,259	1,942
parking14	A*	351,976,816	828,472,606	828,472,562	44	**971**
16_9-01	BFIDA*	183,832,715	4,846,132,188	1,023,897,982	3,821,980,237	6,236
C^*=24	A*+BFHS (∞)	**30,675,587**	1,191,570,432	1,191,514,776	55,283	1,468
iPDB	A*+BFHS (4)	51,147,740	1,013,776,888	1,011,227,268	2,549,247	1,290
visitall11	A*	407,182,291	795,670,561	795,669,929	632	**1,045**
08-half	BFIDA*	172,474,497	3,159,596,842	1,332,828,069	1,824,866,109	4,220
C^*=43	A*+BFHS (∞)	**34,406,966**	1,639,641,152	1,639,585,228	55,798	2,233
iPDB	A*+BFHS (4)	64,671,078	1,346,690,454	1,312,333,974	34,356,354	1,902
tidybot11	A*	115,965,857	246,756,618	246,756,201	417	**1,086**
16	BFIDA*	86,095,996	1,090,011,154	652,777,121	431,816,881	5,512
C^*=40	A*+BFHS (∞)	**41,342,908**	583,309,116	570,082,820	13,225,950	2,923
iPDB	A*+BFHS (4)	57,026,598	598,365,499	519,723,294	78,641,859	3,080
snake18	A*	94,699,640	129,288,606	129,273,608	14,998	**1,131**
08	BFIDA*	44,231,998	1,852,488,086	1,517,078,892	325,204,785	14,877
C^*=58	A*+BFHS (∞)	**44,081,853**	391,010,354	390,681,641	328,706	3,445
iPDB	A*+BFHS (4)	51,166,308	356,988,514	348,015,242	8,973,265	3,192
hiking14	A*	287,192,625	3,299,939,168	3,299,937,850	1,318	**1,297**
2-2-8	BFIDA*	**42,570,885**	11,376,337,161	5,757,334,602	5,582,502,874	10,847
C^*=42	A*+BFHS (∞)	44,454,322	16,233,911,987	12,346,881,620	3,886,689,991	14,897
M&S	A*+BFHS (4)	53,148,260	9,850,751,126	6,310,295,933	3,540,114,817	9,696
pipesworld-	A*	292,998,092	907,283,307	907,283,301	6	**1,364**
tankage 14	BFIDA*	158,262,429	5,354,342,623	3,680,871,467	1,661,344,123	10,609
C^*=38	A*+BFHS (∞)	**84,077,693**	5,768,933,724	5,763,927,002	5,002,176	11,622
iPDB	A*+BFHS (4)	103,288,306	3,300,541,977	3,220,772,288	79,765,143	6,896

Table 6.2: Instances sorted by A* running times. An underline means more than 8 GB of memory was needed. Smallest memory and shortest times are in boldface.

Instance	Algorithm	Peak stored	Total nodes	Prev. iterations	Last iteration	Time (s)
blocks	A*	555,864,249	1,185,065,570	205,172,261	979,893,309	**1,540**
13-1	BFIDA*	99,782,317	1,742,819,669	463,603,038	1,224,383,750	2,142
C^*=44	A*+BFHS (∞)	**54,601,577**	2,261,321,708	425,991,501	1,827,341,160	2,817
iPDB	A*+BFHS (4)	79,572,108	1,817,197,763	401,559,990	1,407,648,726	2,317
parking14	A*	606,117,759	1,430,911,954	1,430,746,610	165,344	**1,714**
16_9-03	BFIDA*	291,822,896	8,077,642,530	1,796,305,162	6,280,923,558	10,059
C^*=24	A*+BFHS (∞)	**48,304,204**	2,519,414,336	2,328,368,930	191,043,484	3,124
iPDB	A*+BFHS (4)	63,455,874	2,151,415,198	1,992,188,756	159,224,520	2,679
tidybot11	A*	175,574,760	372,772,055	372,771,560	495	**1,730**
18	BFIDA*	114,747,861	1,718,896,347	1,093,273,564	613,928,542	8,810
C^*=44	A*+BFHS (∞)	**40,540,308**	1,045,166,148	1,028,635,660	16,529,544	5,410
iPDB	A*+BFHS (4)	65,784,369	1,204,942,101	931,501,196	273,439,961	6,365
blocks	A*	704,938,102	1,568,547,017	342,339,737	1,226,207,280	**1,990**
13-0	BFIDA*	137,821,868	2,421,546,636	775,076,076	1,628,338,675	2,977
C^*=42	A*+BFHS (∞)	**81,918,224**	3,498,922,607	774,231,514	2,710,189,950	4,483
iPDB	A*+BFHS (4)	126,629,640	2,615,897,101	698,028,054	1,903,367,904	3,378
hiking14	A*	368,433,117	6,711,042,999	6,710,971,209	71,790	**2,480**
2-3-6	BFIDA*	**124,686,777**	38,476,138,468	29,175,130,389	8,123,329,545	42,379
C^*=28	A*+BFHS (∞)	146,623,619	107,138,328,055	106,429,883,507	682,558,443	120,494
M&S	A*+BFHS (4)	148,357,537	68,496,320,172	65,779,382,852	2,691,051,215	76,603
pipesworld-	A*	442,232,520	1,028,882,844	1,028,880,896	1,948	**2,693**
notankage 20	BFIDA*	301,349,348	4,454,789,871	2,384,958,671	2,032,377,777	15,245
C^*=28	A*+BFHS (∞)	**133,708,317**	3,325,668,014	3,267,529,384	58,132,775	11,499
iPDB	A*+BFHS (4)	148,029,967	2,988,248,448	2,728,140,813	260,097,006	10,629
snake18	A*	265,033,991	367,639,596	365,927,487	1,712,109	**3,967**
17	BFIDA*	60,041,363	2,162,411,969	1,464,995,207	639,565,966	20,418
C^*=62	A*+BFHS (∞)	**56,839,243**	877,934,374	871,327,013	6,607,339	8,785
iPDB	A*+BFHS (4)	73,365,792	855,342,127	776,892,002	78,450,103	8,916
satellite	A*	107,395,076	463,747,690	463,744,251	3,439	**11,834**
08	BFIDA*	20,846,202	3,656,980,017	520,525,131	3,125,446,334	398,884
C^*=26	A*+BFHS (∞)	**18,870,254**	552,221,751	551,990,933	230,549	54,551
LM-cut	A*+BFHS (4)	19,763,323	546,211,783	479,810,475	66,401,039	56,296

Table 6.3: Instances sorted by A* running times. An underline means more than 8 GB of memory was needed. Smallest memory and shortest times are in boldface.

Instance	Algorithm	Peak stored	Total nodes	Prev. iterations	Last iteration	Time (s)
blocks	A* (unfinished)	>814,951,324	>1,562,632,802	256,247,910	>1,306,384,892	>2,284
15-0	BFIDA*	113,471,990	2,408,362,561	579,842,889	1,827,125,272	**3,058**
C^*=40	A*+BFHS (∞)	**68,070,197**	3,861,465,924	550,007,126	3,291,490,500	4,889
iPDB	A*+BFHS (4)	106,482,059	2,656,641,036	492,390,560	2,144,282,178	3,514
storage	A* (unfinished)	>799,907,374	>1,741,590,894	>1,741,590,894		>2,358
17	BFIDA*	397,798,456	13,297,651,168	4,430,334,119	8,825,291,425	19,086
C^*=26	A*+BFHS (∞)	**118,138,352**	13,403,671,261	13,364,290,422	39,380,047	18,914
iPDB	A*+BFHS (4)	133,800,503	7,895,157,984	6,819,827,727	1,075,329,465	**11,354**
driverlog	A* (unfinished)	>786,467,847	>2,028,764,217	>2,028,764,217		>1,853
15	BFIDA*	453,643,579	24,705,660,389	6,388,627,692	18,280,039,412	24,297
C^*=32	A*+BFHS (∞)	**88,449,751**	16,928,608,100	16,913,831,869	14,773,242	15,311
iPDB	A*+BFHS (4)	123,602,679	9,160,294,407	8,974,814,158	185,477,260	**8,447**
rovers	A* (unfinished)	>801,124,989	>4,427,878,559	>4,427,878,559		>2,776
09	BFIDA*	235,386,020	20,666,689,222	7,239,737,785	13,401,874,237	25,336
C^*=31	A*+BFHS (∞)	**96,100,365**	34,236,064,765	34,235,937,332	123,597	42,290
iPDB	A*+BFHS (4)	99,498,513	12,845,107,625	12,752,327,728	92,776,061	**16,770**
rovers	A* (unfinished)	>766,016,316	>3,690,650,688	>3,690,650,688		>2,378
11	BFIDA*	274,612,697	18,975,576,425	6,574,504,656	12,391,406,745	26,022
C^*=30	A*+BFHS (∞)	**112,783,085**	32,143,105,562	32,139,546,138	3,549,575	43,538
iPDB	A*+BFHS (4)	113,594,902	12,342,784,453	11,789,007,437	553,767,167	**16,661**
parking14	A* (unfinished)	>770,874,998	>1,681,926,228	>1,681,926,228		>2,306
16_9-04	BFIDA*	1,045,614,854	27,924,183,007	6,292,017,194	21,628,727,845	37,701
C^*=26	A*+BFHS (∞)	**156,758,802**	9,778,837,190	9,777,264,498	1,570,687	12,304
iPDB	A*+BFHS (4)	181,535,647	7,588,132,706	7,586,728,152	1,402,549	**9,813**

Table 6.4: Instances where A* terminated without solving the problem (marked by >) so are sorted by BFIDA* running times. An underline means more than 8 GB of memory was needed. Smallest memory and shortest times are in boldface.

column lists the algorithm names. We ran each algorithm until it found the optimal cost and returned the optimal path, or terminated without finding a solution due to hash map size limitations. The third column gives the maximum number of nodes stored by each algorithm. For A*, this is the number of nodes stored at the end of the search. For BFIDA*, this is the largest sum of the number of nodes stored in all three layers of the search, plus the nodes stored in the 1/4 layer for solution reconstruction. For A*+BFHS, this is the largest number of nodes stored in the BFHS phase plus the number of nodes stored in the A* phase.

An underline means the specific algorithm needed more than 8 GB of memory to solve the problem. The fourth column is the total number of nodes generated, including the nodes generated during solution reconstruction. The fifth column is the number of nodes generated in all but the last iteration. For A*, this is the number of nodes generated before expanding an Open node whose f-value is C^*. For A*+BFHS, this number includes the nodes generated in its A* phase. The sixth column is the number of nodes generated in the last iteration. For A*, this is the number of nodes generated while expanding the Open nodes whose f-value equals C^*. The last column is the running time in seconds, including the time for solution reconstruction but excluding the time spent on precomputing the heuristic function, which is the same for all algorithms. For each instance, the smallest maximum number of stored nodes and shortest running time are indicated in boldface. For the A* data in Table 6.4, we report the numbers of nodes and running times just before A* terminated, with a $>$ symbol to indicate such numbers.

For A*+IDA*, we set the stored nodes threshold for the A* phase to A*+BFHS's peak stored nodes. A*+IDA* was faster than A*+BFHS by a factor of 2 on *tidybot11* 18, but was slower than A*+BFHS by around 50% on *mystery* 14, a factor of 2 on *visitall11* 08-half, 4 on *parking14* 16_9-04, and 8 on *snake18* 17. Furthermore, A*+IDA* was orders-of-magnitude slower than A*+BFHS on domains such as *blocks*, *depot*, *driverlog*, *hiking14*, *logistics00*, *pipesworld-tankage*, *rovers*, *satellite*, *storage*, *termes18*, and *tpp*. Thus we omit the results of A*+IDA*.

We further compare the time and memory between A* and A*+BFHS in Figure 6.2, and between BFIDA* and A*+BFHS in Figure 6.3, where the x-axis is A* or BFIDA*'s peak stored nodes over A*+BFHS's and the y-axis is A* or BFIDA*'s running time over A*+BFHS's. Figure 6.2 contains the 26 instances solved by A* and Figure 6.3 contains all 32 instances. The red circles and green triangles correspond to A*+BFHS (4) and A*+BFHS (∞) respectively. The data points above the $y = 1$ line or to the right of the $x = 1$ line represent instances where A*+BFHS outperformed the comparison algorithm in terms of time or memory.

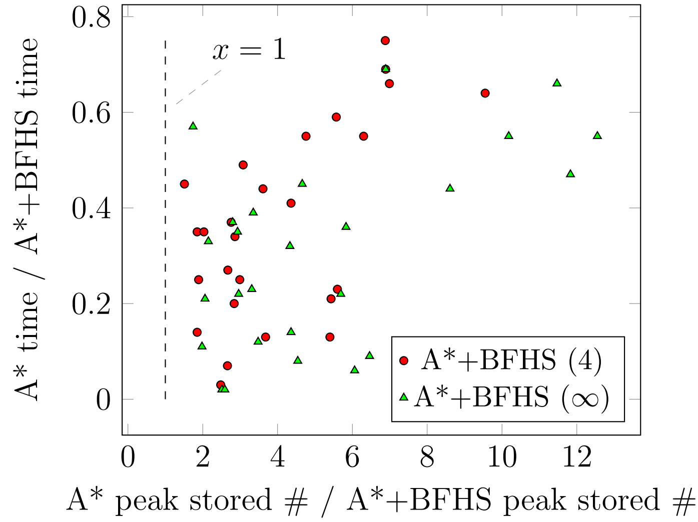

Figure 6.2: A* vs. A*+BFHS in time and memory.

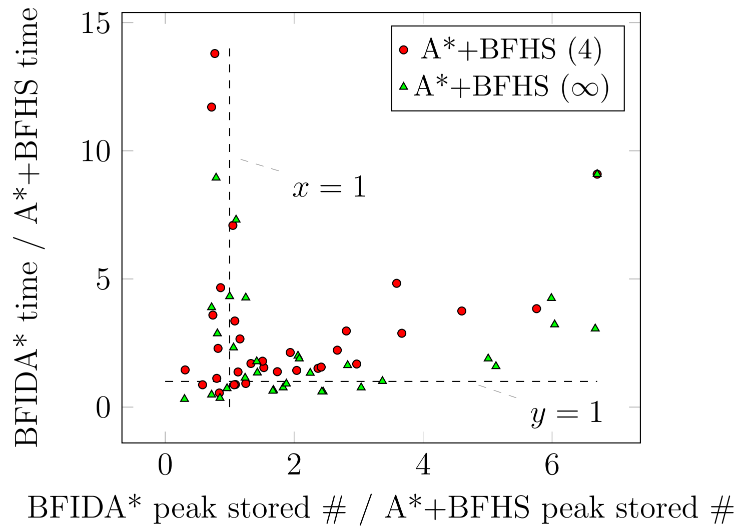

Figure 6.3: BFIDA* vs. A*+BFHS in time and memory.

6.5.1 A*+BFHS vs. A*

A* was the fastest on all problem instances that it solved, but also used the most memory. Among the 32 hardest problem instances we present, A* required more than 8 GB of memory on 22 instances and could not find a solution on 6 of those after exhausting the hash map used by Fast Downward 20.06. On some of these instances, A* used 30 GB to 40 GB of memory before it terminated. This means that A* cannot solve these 22 instances under

the current IPC memory limit of 8 GB. A*+BFHS required several times, and sometimes an order of magnitude, less memory than A*. As a result, A*+BFHS only used more than 8 GB of memory on one instance. An interesting comparison is the space-time trade-off. For example, on *parking14*, A*+BFHS increased the running time by less than a factor of 2 while saving more than an order of magnitude in memory.

6.5.2 A*+BFHS vs. BFIDA*

In summary, on easy problem instances that A*+BFHS can solve in its A* phase, A*+BFHS behaves the same as A*, and is always faster than BFIDA*. We solved around 500 such problem instances, which are not included here. On the 32 hardest problem instances we present, A*+BFHS is faster than BFIDA* on 27 instances and at least twice as fast on 16 of those. Furthermore, A*+BFHS requires less memory than BFIDA* on 25 of the 32 instances and saves more than half the memory on 14 of those. In addition, these time and memory reductions exist on both the relatively easy and hard problem instances of the 32 instances presented, demonstrating that A*+BFHS is better than BFIDA* on very hard problem instances as well as just hard problem instances. In the following paragraphs, we compare A*+BFHS with BFIDA* in four aspects: duplicate detection, node ordering, memory, and running time.

Figure 6.4 compares the number of nodes generated prior to the last iteration of BFIDA* and A*+BFHS. For BFIDA*, this is the number of nodes generated in all but the last iteration. For A*+BFHS, this is the sum of the nodes generated in its A* phase and all but the last iteration in its BFHS phase. The y-axis in Figure 6.4 is the number of nodes generated in BFIDA*'s previous iterations divided by A*+BFHS's. We sort the 32 instances on the x-axis according to BFIDA*'s running time, so the left-most instance is the easiest and the right-most instance is the hardest for BFIDA*. The data points above the $y = 1$ line represent instances where A*+BFHS generated fewer nodes than BFIDA* in the previous iterations. Compared to BFIDA*, A*+BFHS (4) generated a similar number of nodes in the previous iterations on most instances. *Hiking14* 2-3-6 is the only instance where

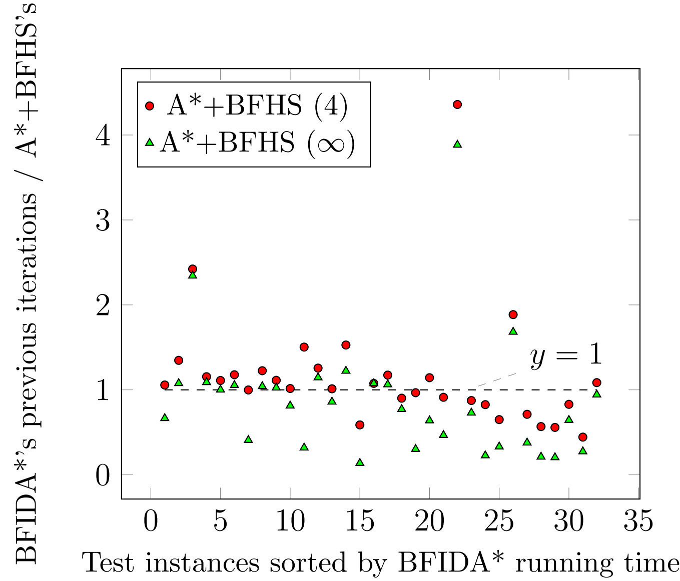

Figure 6.4: The number of nodes generated in BFIDA*'s previous iterations vs. A*+BFHS's.

A*+BFHS (4) generated at least twice as many nodes in the previous iterations as BFIDA*. However, A*+BFHS (∞) generated 2 to 7 times as many nodes in the previous iterations as BFIDA* on 11 instances. This contrast shows that, compared to BFIDA*, significantly more duplicate nodes are generated by making each call to BFHS on frontier nodes at a single depth. However, most of those duplicate nodes can be avoided by making each call to BFHS on frontier nodes at multiple depths.

A*+BFHS generates fewer duplicate nodes than BFIDA* due to fewer BFHS iterations and making each call to BFHS on a set of frontier nodes. A*+BFHS reduced the number of nodes in previous iterations by around 50% on *freecell* 06 and *snake18* 17, and a factor of 4 on *snake18* 08. To our surprise, we found that on *snake18* 08, the number of nodes generated in the penultimate iteration of BFIDA* was twice as many as the sum of the nodes generated in A*+BFHS's A* phase and the penultimate iteration of the BFHS phase. This means that many duplicate nodes were generated in BFIDA*. *Snake18* generates a directed graph, in which case frontier search cannot detect all duplicate nodes [ZH04, KZT05].

Compared to BFIDA*, A*+BFHS reduced the number of nodes in the last iteration significantly, and usually by several orders of magnitude, on 28 of the 32 instances. We

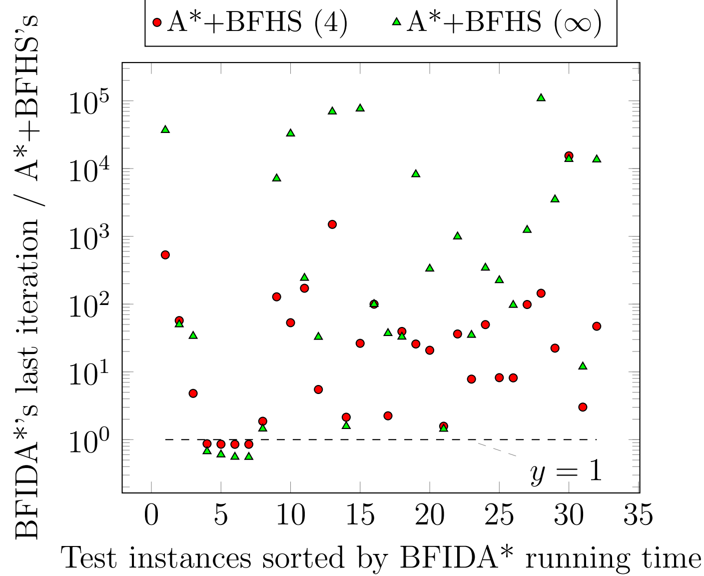

Figure 6.5: The number of nodes generated in BFIDA*'s last iteration vs. A*+BFHS's.

present this comparison in Figure 6.5, where the y-axis is the number of nodes generated in BFIDA*'s last iteration divided by A*+BFHS's. We also sort the 32 instances on the x-axis according to BFIDA*'s running time, so the left-most instance is the easiest and the right-most instance is the hardest for BFIDA*. The data points above the $y = 1$ line represent instances where A*+BFHS generated fewer nodes than BFIDA* in the last iteration. Both A*+BFHS versions usually generated several orders of magnitude fewer nodes in the last iteration than BFIDA*, while A*+BFHS (∞) generated the fewest nodes on most instances. This large reduction proves that when ordering the frontier nodes by deepest-first, A*+BFHS can terminate early in its last iteration. On the three *blocks* instances and *depot* 11, A*+BFHS did not terminate early in its last iteration because the last ancestor of the goal in the A* phase had a relatively low g-value. In fact, A* generated the most nodes while expanding the Open nodes whose $f = C^*$ on the three *blocks* instances, which shows that node ordering is also difficult for A* on those instances. In contrast, A* generated very few nodes while expanding the Open nodes whose $f = C^*$ on *depot* 11, suggesting that A*+BFHS may terminate early in its last iteration given more memory for its A* phase.

A*+BFHS's A* phase usually stored from 10 to 20 million nodes, with the exception of

the *snake18* domain where 40 to 50 million nodes were stored. Comparing the maximum number of stored nodes, A*+BFHS (∞) required less memory than BFIDA* on 25 instances and less than half the memory on 14 of those. For A*+BFHS (4), these two numbers were 23 and 11 respectively. In contrast, *termes18* 05 is the only instance where the maximum number of stored nodes of A*+BFHS was at least twice that of BFIDA*.

Comparing the two versions of A*+BFHS, A*+BFHS (4) was usually faster, sometimes significantly, due to the reduction in duplicate nodes. Compared to BFIDA*, A*+BFHS (4) was slightly slower on four instances and 80% slower on one instance. On the other 27 instances, A*+BFHS (4) was faster than BFIDA*, and at least twice as fast on 16 of those. The large speedups usually were on the instances where BFIDA* generated the most nodes in its last iteration. The best result was on the *logistics00* domain, where an order of magnitude speedup was achieved. This is because BFIDA* performed very poorly on this domain due to its breadth-first node ordering. Compared to BFIDA*, A*+BFHS (∞) was slower on 11 instances and at least twice as slow on three of those, but also at least twice as fast on 12 instances. The main reason for the slower cases is the presence of many duplicate nodes generated in certain domains.

6.5.3 Calling BFHS on Nodes at Multiple Depths

Comparing the two A*+BFHS versions, each has its pros and cons. A*+BFHS (4) always generated fewer duplicate nodes. Comparing the number of nodes generated in the previous iterations, A*+BFHS (∞) generated at least twice as many nodes on 7 instances. A*+BFHS (∞) generated significantly fewer nodes in the last iteration than A*+BFHS (4) on 22 instances. However, the number of nodes generated in the last iteration of A*+BFHS is usually only a small portion of the total nodes generated, so the large difference in the last iteration is not very important. A*+BFHS (4) stored a larger maximum number of nodes than A*+BFHS (∞) on almost all instances. However, the difference was usually small and never more than a factor of two. The difference in the running times was usually less than 50%. Compared to A*+BFHS (∞), A*+BFHS (4) was faster by a factor of 3 on *logistics00*

15-1, 2.5 on *rovers* 09 and 11, 4.6 on *termes18* 05, 3.9 on *tpp* 11, and never more than 30% slower.

In general, making each call to BFHS on frontier nodes at multiple depths increases both the memory usage and the number of nodes generated in the last iteration, but reduces the number of duplicate nodes and hence is often faster. Considering the memory-time trade-off, given a new problem, we recommend making each call to BFHS on frontier nodes at multiple depths. However, if we limit the number of calls to BFHS in each iteration to one, then A*+BFHS (1) will generate about the same number of nodes as BFIDA*, and early termination is no longer possible. Therefore, at least two calls should be used. So far, we have only limited BFHS to four calls in each iteration. Determining the optimal number of calls to BFHS is a subject for future work.

6.5.4 Heuristic Functions and Running Time

For each node generated, A* first checks for duplicates then looks up its heuristic value if needed. Thus for each state, A* only computes its heuristic value once, no matter how many times the state is generated. However, the situation is different in BFHS. Even in a single call to BFHS, a state's heuristic value may be calculated multiple times. For example, if a state's f-value is greater than the cost bound of BFHS, then this state is not stored in this call to BFHS and its heuristic value has to be computed every time it is generated. In addition, A* has only one hash map but our BFHS implementation has one hash map for each layer of nodes. Consequently, for each node generated, A* does only one hash map lookup while BFHS may have multiple lookups.

Due to the above differences, the number of nodes generated per second of BFIDA* and A*+BFHS was smaller than that of A*. For the iPDB and M&S heuristics, this difference was usually less than a factor of two. For the LM-cut heuristic, A* was faster by a factor of four in terms of nodes generated per second on the *satellite* domain. This is because computing a node's LM-cut heuristic is much more expensive than its iPDB and M&S heuristics. This contrast shows that the choice of heuristic function also plays an important

role in comparing the running time of different algorithms.

CHAPTER 7

Iterative-Deepening Uniform-Cost Heuristic Search

Breadth-first heuristic search (BFHS) is a well-known algorithm for optimally solving heuristic search and planning problems. BFHS is slower than A* but requires less memory. However, BFHS only works on unit-cost domains, meaning all operators have the same cost. We propose a new algorithm that extends BFHS to domains with different edge costs, which we call uniform-cost heuristic search (UCHS). Experimental results show that the iterative-deepening version of UCHS, IDUCHS, is slower than A* but requires less memory on a variety of planning domains.

Our work on UCHS is presented in the following order. Section 7.1 discusses the motivation of this research. Section 7.2 presents the new algorithms uniform-cost frontier search (UCFS), which does not use heuristics, and uniform-cost heuristic search (UCHS). Section 7.3 introduces the iterative-deepening version of UCHS, IDUCHS, as well as the solution reconstruction scheme we used. Section 7.4 presents experimental results of A* and IDUCHS on 20 hard instances from 11 non-unit International Planning Competition (IPC) domains. Section 7.5 further introduces A*+UCHS, which replaces the BFHS phase in A*+BFHS with UCHS, and compares it with IDUCHS and discusses why it is not better than IDUCHS.

7.1 Motivation

A*+IDA* utilizes a depth-first search in its IDA* phase, works on domains with arbitrary edge costs, but still may generate many duplicate nodes on domains where many distinct paths exist between a pair of nodes. On the other hand, BFIDA* and A*+BFHS's BFHS phase utilize a breadth-first search to detect duplicate nodes and hence works well on domains

where A*+IDA* would run for a very long time. However, both BFIDA* and A*+BFHS only work on unit-cost domains.

Dijkstra's single-source shortest path algorithm [Dij59] generalizes breadth-first search to the case of non-uniform edge costs. The version that terminates when a goal node is chosen for expansion is usually called uniform-cost search (UCS), despite the fact that it deals with non-uniform edge costs. This terminology comes from the fact that the nodes on Open tend to have uniform g-values.

This suggests the question: how can we extend breadth-first heuristic search, and hence BFIDA* and A*+BFHS, to domains with arbitrary edge costs? To answer this question, we present uniform-cost heuristic search and its iterative-deepening version IDUCHS.

7.2 Uniform-Cost Heuristic Search

Previously we introduced breadth-first frontier search (BFFS) [Kor04] in Section 4.1. Here we describe uniform-cost frontier search (UCFS), a brute-force algorithm that extends BFFS to the case of non-unit edge costs.

7.2.1 Uniform-Cost Frontier Search

The difference between UCFS and UCS is that UCFS deletes a node after all its child nodes have been expanded, similar to the difference between BFFS and BFS. This prevents a parent node from being re-expanded. Suppose p is a parent node, and $\Phi = \{n_i | i = 0, 1, 2...\}$ are its children. In UCFS, the delete g-value of node p, $d_g(p)$, is the maximum of the g-values of its children.

$$d_g(p) = \max_{n_i \in \Phi} g(n_i) \tag{7.1}$$

For example, if p has two children with g-values of 5 and 6, then p can be deleted after expanding all nodes with $g = 6$.

UCFS works as follows. It first expands the start node and puts all child nodes on Open.

The Open list is kept sorted by the g-values of the nodes. After all nodes on Open whose g-value is 0 are expanded, in the case of zero-cost edges, assume the next smallest g-value on Open is 3. UCFS deletes the expanded nodes whose d_g-value is less than 3. In fact, at this time, all expanded nodes have a g-value of 0, some expanded nodes may have a d_g-value of 0, while the other expanded nodes' d_g-values are greater than or equal to 3. Next, UCFS expands all nodes on Open whose g-value is 3. It continues expanding and deleting nodes until it finds a goal node on an optimal path. On unit-cost domains, BFFS can terminate when a goal node is generated. On domains with non-unit edge costs, however a node may be generated with a g-value greater than its optimal cost. Therefore, UCFS only terminates when a goal node is chosen for expansion.

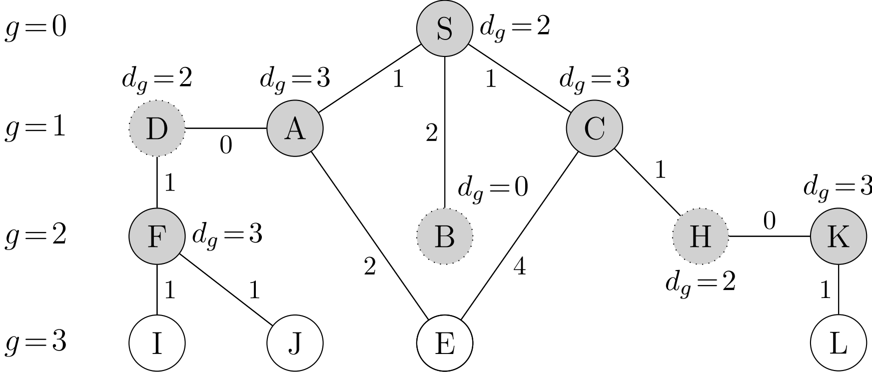

Figure 7.1: An example of UCFS. Numbers next to edges are edge costs. Closed nodes are gray. Deleted nodes are dotted. d_g is the delete g-value.

Figure 7.1 presents an example of UCFS on an undirected graph, where numbers next to edges are edge costs, Closed nodes are gray, and deleted nodes are dotted. First the start node S is expanded, generating nodes A, B, and C. $g(\text{A}) = g(\text{C}) = 1$ and $g(\text{B}) = 2$. $d_g(\text{S}) = 2$, the maximum g-value of A, B, and C. Next, A generates nodes S, D, and E. S is a duplicate node. D and E are stored in memory with $g(\text{D}) = 1$ and $g(\text{E}) = 3$, thus $d_g(\text{A}) = 3$. D generates nodes A and F. A is a duplicate node while $g(\text{F}) = 2$, hence $d_g(\text{D}) = 2$. Then C generates nodes S, E, and H. S and E are duplicate nodes. Since $g(\text{E}) = 3$ and $g(\text{H}) = 2$, $d_g(\text{C}) = 3$. At this point, all Open nodes with $g = 1$ have been expanded. Since no node has a $d_g \leq 1$, UCFS expands nodes at $g = 2$. B only generates one node which is S, thus

d_g(B) = 0, and B can be deleted immediately. F generates D, I, and J. D is a duplicate node while g(I) = g(J) = 3, thus d_g(F) = 3. Then H generates C and K. C is a duplicate node and g(K) = 2, hence d_g(H) = 2. K generates H and L. H is a duplicate node, g(L) = 3, and d_g(K) = 3. After expanding all Open nodes with $g = 2$, nodes D and H are deleted, as they have a d_g-value of 2. S cannot be deleted since it is the start node.

A key difference between BFFS and UCFS is the order of deleting nodes. In BFFS, nodes at depth d are always deleted before nodes at depth $d + 1$. In UCFS, however, a node m with a larger g-value may be deleted before a node n with a smaller g-value, if any of n's children have a higher g-value. A child node may even be deleted before its parent node. For example, in Figure 7.1, H is removed at $d_g = 2$, while H's parent node C will be removed later at $d_g = 3$.

7.2.2 Uniform-Cost Heuristic Search

We now extend UCFS by adding heuristics and a cost bound U, allowing us to prune nodes whose f-value exceeds U. We call the resulting algorithm uniform-cost heuristic search (UCHS). UCHS generalizes BFHS to handle non-unit edge costs. UCHS is to UCFS as BFHS is to BFFS, while UCFS is to UCS as BFFS is to BFS.

We present pseudo-code for UCHS in Algorithm 1, where Open$_k$ and Closed$_k$ are Open and Closed nodes whose g-value is k. UCHS first expands the start node and adds to Open its child nodes n for which $f(n) \leq U$. If a node n is generated with $f(n) > U$, it is immediately deleted. The Open list is kept sorted by the g-values of the nodes. After all nodes on Open whose g-value is d are expanded, assume the next smallest g-value on Open is d'. UCHS deletes the expanded nodes for which $d_g < d'$. It then expands all nodes on Open whose g-value is d'. It continues until it finds a goal node, and verifies that no cheaper path to a goal exists.

To show how UCHS works, we run it on the same undirected graph from Figure 7.1. We present the result in Figure 7.2, where the numbers next to the nodes are their f-values, Closed nodes are gray, and deleted nodes are dotted. Assume UCHS is called with cost

Algorithm 1 Uniform-Cost Heuristic Search

```
1:  g(start) ← 0
2:  Open_0 ← {start}
3:  d ← 0
4:  while d ≤ U do
5:     for p ∈ Open_d do
6:        if p is goal then
7:           return  d
8:        d_g(p) ← 0
9:        for n ∈ children(p) do
10:          /* Check duplicates against all stored nodes. */
11:          if n ∈ Open_k or n ∈ Closed_k then
12:             /* New path to n is not cheaper than old path. */
13:             if g(n) ≥ k then
14:                d_g(p) ← max(k, d_g(p))
15:                continue
16:             else
17:                Open_k ← Open_k\n
18:          d_g(p) ← max(g(n), d_g(p))
19:          if f(n) ≤ U then
20:             Open_g(n) ← Open_g(n) ∪ n
21:    /* All nodes with a g-value of d are expanded. */
22:    Closed_d ← Open_d
23:    Open_d ← ∅
24:    /* g-value of the next node to expand. */
25:    d′ ← min{k|Open_k ≠ ∅}
26:    for n ∈ Closed do
27:       if d_g(n) < d′ then
28:          /* Delete Closed node n from memory. */
29:          Closed ← Closed \n
30:    d ← d′
31: return ∞
```

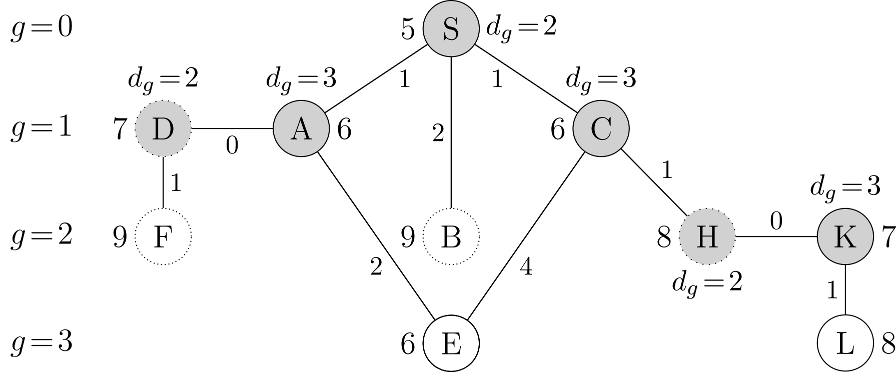

Figure 7.2: An example of UCHS with a cost bound of 8.5. Numbers next to edges are edge costs and numbers next to nodes are f-values. Closed nodes are gray. Deleted nodes are dotted. d_g is the delete g-value.

bound $U = 8.5$. S generates nodes A, B, and C. $f(\mathrm{A}) = f(\mathrm{C}) < U$, so they are stored. $g(\mathrm{B}) = 2$ and $f(\mathrm{B}) = 9 > U$, so B is deleted, but we still consider B when calculating $d_g(\mathrm{S})$, which is 2. Next, A generates S, D, and E. S is a duplicate node, D and E are stored with $g(\mathrm{D}) = 1$, $g(\mathrm{E}) = 3$, and $d_g(\mathrm{A}) = 3$. D generates A and F. A is a duplicate node, F is deleted since $f(\mathrm{F}) > U$, and $d_g(\mathrm{D}) = 2$. C generates S, E, and H. S and E are duplicate nodes, while H is stored, and $d_g(\mathrm{C}) = 3$. At this point, all Open nodes with $g = 1$ have been expanded. UCHS continues expanding nodes as there are no nodes to be deleted. H generates C and K. C is a duplicate node, but K is stored with $g(\mathrm{K}) = 2$, and $d_g(\mathrm{H}) = 2$. K then generates H and L. H is a duplicate node, $g(\mathrm{L}) = 3$, and $d_g(\mathrm{K}) = 3$. After expanding all Open nodes with $g = 2$, nodes D and H are deleted, as they have a d_g-value of 2. S cannot be deleted since it is the start node.

Similar to UCFS, in UCHS a node stored at a larger g-value may be deleted before a node stored at a smaller g-value, or a child node may be deleted before its parent.

The definition of d_g in Equation 7.1 is simple and intuitive, but on undirected graphs, it results in UCHS storing nodes for longer than it needs to. For example, D is only removed after expanding all nodes with $g = 2$ since $g(\mathrm{F}) = 2$, even though F is not stored. This suggests that when calculating $d_g(p)$, we just ignore any of p's child nodes n for which $f(n) > U$. This choice looks tempting but it can lead to a node being re-expanded. If we

used this strategy, then d_g(D) $= 1$. In our example, D is expanded before C. Suppose the problem in Figure 7.2 changes such that C can also generate F with an edge cost of 0.5, thus $g(\text{F}) = 1.5$, $f(\text{F}) = 1.5 + 7 = 8.5 = U$, and F will be stored. UCHS would have deleted D prior to expanding F, since $d_g(\text{D}) = 1$. F would then generate D again with $g(\text{D}) = 2.5$ and $f(\text{D}) = 2.5 + 6 = 8.5 = U$. As a result, D will be treated as a new node and will be stored and expanded again.

This shows that we have to consider all child nodes while calculating the d_g-values, but we can actually delete D earlier. For D to be generated by F in UCHS, F must be stored and expanded. Given $U = 8.5$, the largest $g(\text{F})$ that would allow F to be stored is $U - h(\text{F}) = 8.5 - 7 = 1.5$. Therefore, if we remove D after expanding all nodes up to $g = 1.5$, then we are guaranteed that D will not be generated again by F, as F cannot still be in memory when $g(\text{F}) > 1.5$.

Another case to consider is when to delete K. $d_g(\text{K}) = 3$ due to $g(\text{L}) = 3$. Since $c(\text{K,L})$, the cost of the edge between K and L, is 1, when L generates K, we will have $g(\text{K}) = g(\text{L}) + 1 = 3 + 1 = 4$ and $f(\text{K}) = g(\text{K}) + h(\text{K}) = 4 + 5 = 9 > U = 8.5$, which means this new copy of K will be deleted. This suggests that instead of deleting K after expanding all nodes with $g = 3$, we can delete K before expanding any node with $g = 3$. To calculate $d_g(\text{K})$, in addition to the value of $g(\text{L})$, we also consider $U - c(\text{K,L}) - h(\text{K}) = 8.5 - 1 - 5 = 2.5$. This means that if L is stored with $g(\text{L}) \leq 2.5$, then during the expansion of L, K will be generated with $f(\text{K}) = g(\text{L}) + c(\text{K,L}) + h(\text{K}) \leq 2.5 + 1 + 5 = 8.5 = U$. In other words, if L is expanded with $g(\text{L}) > 2.5$, then the new copy of K generated by L will have $f(\text{K}) > U$, and will be deleted. Therefore, UCHS can use $d_g(\text{K}) = 2.5$, and K will be deleted before UCHS expands any nodes with $g = 3$.

Taking the above examples into consideration, we propose a new definition of d_g. Suppose U is the cost bound, p is a parent node, $\Phi = \{n_i | i = 0, 1, 2...\}$ are the children of p, and c_i is the cost of the edge between p and n_i. Then $d_g(p)$ can be defined as follows:

$$d_g(p) = \max_{n_i \in \Phi} \big\{ \min\{g(n_i), U - h(n_i), U - c_i - h(p)\} \big\} \tag{7.2}$$

Theorem 1. *UCHS using Equation 7.2 is guaranteed not to expand a state more than once*

on undirected graphs.

Proof. We consider $f(n_i)$ of a newly generated node n_i.

Case 1: $f(n_i) \leq U$. n_i is stored and $g(n_i) \leq U - h(n_i)$, therefore we only need to compare $g(n_i)$ with $U - c_i - h(p)$.

(a): $g(n_i) \leq U - c_i - h(p)$. When UCHS expands n_i, p is still in memory, so the new copy of p will be deleted.

(b): $g(n_i) > U - c_i - h(p)$. UCHS may delete p before expanding n_i. However, p will only be deleted after expanding all nodes at $g = U - c_i - h(p)$. Thus, even if n_i is expanded at $g(n_i) > U - c_i - h(p)$, we still have $f(p) = g(n_i) + c_i + h(p) > U - c_i - h(p) + c_i + h(p) = U$. Therefore, this new copy of p will be immediately deleted. It is possible that after the expansion of p, n_i is generated again with a smaller g-value, but the above analysis still holds.

Case 2: $f(n_i) > U$. Since $g(n_i) > U - h(n_i)$, we only need to consider $U - h(n_i)$ and $U - c_i - h(p)$.

(a): $U - h(n_i) \leq U - c_i - h(p)$. p will not be deleted before expanding all nodes up to $g = U - h(n_i)$, the largest value of $g(n_i)$ such that n_i is stored and expanded. Therefore, if n_i is ever expanded, for example via a different and cheaper path, it must be expanded before p is removed.

(b): $U - h(n_i) > U - c_i - h(p)$. The analysis is similar to Case 1 (b) and is omitted here. □

On directed graphs, the above analysis and the guarantee that no state will be expanded more than once no longer holds, and p may be expanded multiple times. The reason is that a node we have not seen yet can regenerate node p. However, this is the price we have to pay as frontier search algorithms in general, as we show in Section 4.8, cannot prevent expanded nodes from being re-generated and re-expanded on directed graphs [ZH04, KZT05]. A special case where frontier search algorithms can detect all duplicates on directed graphs is also discussed in Section 4.8.

The graph search version of Iterative Budgeted Exponential Search (IBES) [HLL19] utilizes UCS to minimize the worst-case number of node expansions on domains with various edge costs and inconsistent heuristics. However, IBES does not remove Closed nodes from memory so it is not directly comparable to our new algorithm.

7.3 Iterative-Deepening UCHS (IDUCHS)

7.3.1 Calculating the Cost Bounds of Iterations

UCHS requires a user-specified cost bound, but the optimal solution cost C^* is rarely known in advance. Therefore, we add iterative-deepening to UCHS, which we call IDUCHS. IDUCHS generalizes BFIDA* to domains with non-unit edge costs. On unit-cost domains, BFIDA* usually increases the cost bound by one with each successive iteration. This strategy does not work well on domains with non-unit edge costs, however, as there may be too many iterations with very few new nodes expanded in each iteration.

On domains with non-unit edge costs, IDA* [Kor85] has the same problem, as it uses the smallest f-value among the nodes generated but not expanded on the previous iteration as the cost bound of the next iteration. IDA*_CR [SCG91] addresses this issue by using a larger cost bound increment between iterations. During each iteration, it counts the number of nodes generated at each f-value greater than the current cost bound. After each iteration, it decides how many new nodes to expand in the next iteration, say m. It then determines an f-value that will result in at least m new nodes being expanded in the next iteration, and uses that as the cost bound for the next iteration.

The same idea can also be applied to UCHS on domains with non-unit edge costs. We call the resulting algorithm UCHS_CR, but it did not work well, for several reasons. First, UCHS_CR uses duplicate detection while IDA*_CR does not. As a result, UCHS_CR is not guaranteed to expand m new nodes in the next iteration. Second, on planning domains, doubling the number of expanded nodes does not guarantee that the number of generated nodes will also be doubled. Third, sometimes the number of nodes generated but not expanded is

lower than the desired m-value, making it difficult to choose the next cost bound.

[KRE01] showed that compared to brute-force search, the effect of heuristic functions in IDA* is to reduce the search depth instead of reducing the branching factor. They accurately predicted the performance of IDA* on sliding-tile puzzles and Rubik's Cube. Inspired by their work, we propose a new way to calculate the cost bounds for IDUCHS on domains with non-unit edge costs. Let f_k be the cost bound of iteration k and N_k be the number of nodes generated in iteration k. We define a pseudo branching factor b by the equation $b^{f_k - f_{k-1}} = N_k/N_{k-1}$. Note that b is not the brute-force branching factor of the problem domain, but tends to remain constant between iterations. Therefore, b is sometimes called the effective branching factor. Suppose we want $r = N_{k+1}/N_k$, where r is a user-specified ratio of the generated nodes between iterations. Then we have $b^{f_{k+1} - f_k} = r$. Taking the logarithm of both sides, we get $(f_k - f_{k-1}) \ln b = \ln (N_k/N_{k-1})$ and $(f_{k+1} - f_k) \ln b = \ln r$. Combining both equations, we get

$$\frac{\ln (N_k/N_{k-1})}{(f_k - f_{k-1})} = \ln b = \frac{\ln r}{f_{k+1} - f_k}$$
$$(\ln N_k - \ln N_{k-1})(f_{k+1} - f_k) = (f_k - f_{k-1}) \ln r$$

and finally

$$f_{k+1} = \frac{(f_k - f_{k-1}) \ln r}{\ln N_k - \ln N_{k-1}} + f_k \tag{7.3}$$

Thus, given the cost bounds and the numbers of generated nodes of the previous two iterations, and the desired ratio of generated nodes between iterations, we can calculate an estimated cost bound for the next iteration. This equation does not precisely model the generated nodes in IDUCHS, but is an efficient way to calculate successive cost bounds.

The above equation cannot be used to calculate the second iteration's cost bound. In the first iteration, we calculate the average g-value ($avg(g)$) and depth ($avg(d)$) of all generated nodes, and increase the cost bound of the first iteration by $\frac{avg(g)}{avg(d)}$ to get the second cost bound.

IDUCHS works on all domains with non-negative edge costs, including unit-cost domains,

and hence subsumes BFIDA*. IDUCHS is admissible and complete on both directed and undirected graphs when using admissible heuristics. Given a cost bound $U \geq C^*$, where C^* is the optimal solution cost, there always exists a node n on Open such that n is on an optimal path, and hence reached via its minimal g-value. Furthermore, on undirected graphs, a single iteration of IDUCHS is guaranteed to never expand a state more than once when using the d_g-value defined in Equation 7.2.

7.3.2 Solution Reconstruction

Similar to BFIDA*, when IDUCHS stops, it only has the solution cost, and additional searches are needed to reconstruct the actual solution path. Therefore, we also use two A* searches to reconstruct the solution path for IDUCHS, just like our solution reconstruction strategy for BFIDA* in Section 6.4.

To compute the solution path from the middle node to the original goal node, we use A* with the original heuristic function. Computing the solution path from the start node to the middle node is more complex, as we do not have a heuristic estimate to the middle node. For this task in BFIDA*, we use A* with the heuristic to the original goal node, but we modify A* so that any node whose g-value exceeds that of the middle node, or whose f-value exceeds the original problem's C^*-value, is pruned. For IDUCHS, we similarly prune nodes n for which $f(n) > C^*$ or $g(n) > g(middle)$. To accommodate zero-cost edges, a node n for which $g(n) = g(middle)$ is not pruned.

In BFIDA*, we save the nodes at depth $U/4$ as the middle layer when U is the current iteration cost bound. This is because the layer at depth $U/2$ is usually larger than that of $U/4$. For IDUCHS, we save as the middle layer the Open nodes that exist at the point when all nodes up to $g = U/4$ have been expanded. Therefore the middle layer contains nodes with various g-values. A detailed discussion of the choice of the middle layer depth can be found in Section 6.4.

7.3.3 Algorithm Evolution

Base algorithm	Adding frontier search	Adding heuristics & cost bound	Adding iterative-deepening
breadth-first search	breadth-first frontier search	breadth-first heuristic search	breadth-first iterative-deepening A*
uniform-cost search	uniform-cost frontier search	uniform-cost heuristic search	iterative-deepening uniform-cost heuristic search

Table 7.1: Relationships between algorithms.

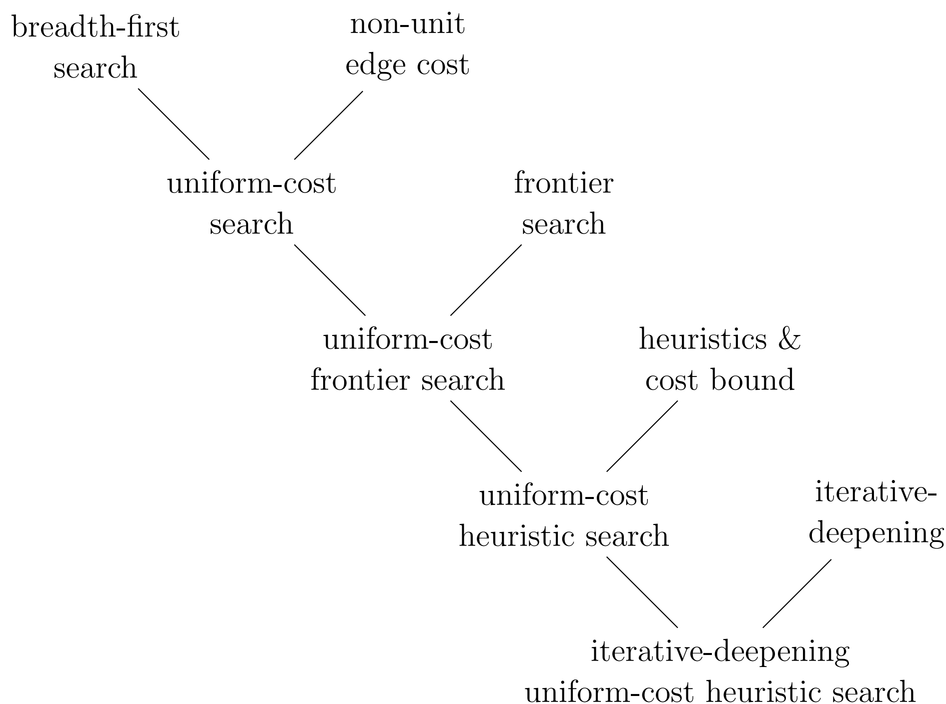

Figure 7.3: The evolution of iterative-deepening uniform-cost heuristic search (IDUCHS).

We present the relationships between various algorithms mentioned in this chapter in Table 7.1. Breath-first search (BFS) and uniform-cost search (UCS) are the two base algorithms. Combined with frontier search, BFS becomes breadth-first frontier search (BFFS) and UCS becomes uniform-cost frontier search (UCFS). After adding heuristics and a cost bound, BFFS becomes breadth-first heuristic search (BFHS) and UCFS becomes uniform-cost heuristic search (UCHS). Finally, after applying iterative-deepening, BFHS becomes

breadth-first iterative-deepening A* (BFIDA*) and UCHS becomes iterative-deepening uniform-cost heuristic search (IDUCHS).

Similar to the evolution of BFIDA* in Figure 4.2, we further present the evolution of iterative-deepening uniform-cost heuristic search in Figure 7.3, where each node is an algorithm or search technique. After combining with non-unit edge cost, frontier search, heuristics, a cost bound, and iterative-deepening, breadth-first search finally evolves into iterative-deepening uniform-cost heuristic search.

7.4 Experimental Results and Analysis

We implemented IDUCHS in the planner Fast Downward 20.06 [Hel06]. We used the original code for node expansions and heuristic functions. The original hash map in Fast Downward does not support removing a stored node, so we modified the hash map to allow this. Whenever a node was removed from the hash map, we stored the next new node into the resulting vacant slot. Therefore, we used a single hash map to store all nodes. The nodes in the middle layer used for solution reconstruction were not deleted.

We have solved about 300 problem instances from 19 International Planning Competition (IPC) domains with non-unit edge costs. These domains fall into three categories. First, binary domains, where the edge costs are either 0 or 1. Second, positive domains, where the edge costs are positive integers. Third, integer domains, where the edge costs are positive integers or 0. We are most interested in solving hard problems, so we present the results of the 20 hardest problem instances we solved from 11 domains. The remaining 280 problem instances were easily solved by A*, usually within a few seconds. All experiments were run on a machine with a 3.33 GHz Xeon X5680 CPU and 236 GB of RAM.

We did not test IDUCHS on unit-cost domains, where IDUCHS will perform similarly to BFIDA*. A*+BFHS is the state-of-the-art algorithm for unit-cost domains when A* fails due to memory limitations [BK22a].

sokoban08 and *spider18* are binary domains. *data-network18*, *elevators08*, and *ged14*

are integer domains. The rest are positive domains. *elevators08*, *transport11*, and *transport14* generate undirected graphs, while the other 8 domains generate directed graphs. We tested three different heuristic functions and picked the best one for each domain: the iPDB heuristic with the default configuration in Fast Downward [HBH07, SOH12], the merge-and-shrink heuristic (M&S) with the recommended configuration in Fast Downward [SWH14, SWH16, Sie18], and the landmark-cut heuristic (LM-cut) [HD09]. We used LM-cut for *floortile11*, M&S for *agricola18*, *barman11*, and *woodworking08*, and iPDB for the rest.

We present the results of A* and IDUCHS in Table 7.2. We used Equation 7.2 to calculate the d_g-values with $r = 2$ in Equation 7.3 to calculate the cost bounds of IDUCHS. The first column is the problem instance. The second column is the optimal solution cost C^* and the third column is the cost bound of IDUCHS's last iteration. The fourth and fifth columns are the peak numbers of stored nodes in A* and IDUCHS. The numbers in parentheses are the peak memory usage in GB reported by Fast Downward. The sixth column is the total nodes generated by A*. The seventh column is the number of nodes generated in a single iteration of UCHS with the cost bound equal to C^*, excluding the nodes generated in solution reconstruction. The eighth column is the total number of nodes generated by IDUCHS, including the nodes generated in solution reconstruction. The last two columns show the running time of A* and IDUCHS, including solution reconstruction in IDUCHS but excluding the time spent on building the heuristic function. Problem instances from the three undirected graph domains are marked with *. An underline means that more than 8 GB of memory was needed.

We found that using $U/4$ for the g-cost of the middle layer was not very practical, as the A* search from the middle node to the original goal node could be relatively expensive. Instead, we dynamically set the depth of the middle layer. In each iteration of IDUCHS, we checked if the size of the middle layer was less than 1% of the peak stored nodes for that iteration. If so, we then increased the g-value threshold for the middle layer by $U/10$ in the next iteration, with $U/2$ as the upper bound. The number of nodes generated during solution reconstruction was usually around 1% of the total nodes generated in IDUCHS, and

Instance	C^*	Last Bound	Peak Stored Nodes & RAM in GB A*	IDUCHS	Total Nodes A*	UCHS (C^*)	IDUCHS	Time (s) A*	IDUCHS
elevators08 16*	87	88	53,943,052 (3.1)	3,379,185 (0.2)	151,476,407	177,867,354	410,304,955	131	337
transport11 09*	313	319	56,090,444 (3.1)	5,656,701 (0.4)	177,076,746	187,555,930	551,348,016	166	496
transport11 15*	1,079	1,086	81,297,837 (3.5)	8,577,209 (0.7)	237,858,963	244,074,950	587,047,357	238	593
data-network18 06	107	110	54,856,222 (3.2)	22,677,495 (1.4)	451,733,882	1,491,433,606	3,486,933,470	242	3,646
transport14 08*	1,173	1,338	62,784,307 (2.9)	13,200,288 (0.9)	396,833,140	397,501,043	1,325,938,181	248	1,375
sokoban08 28	43	46	42,280,857 (1.8)	19,696,951 (1.4)	104,786,508	287,762,405	938,253,805	275	2,566
spider18 18	27	27	32,956,806 (3.5)	14,725,684 (1.8)	35,247,917	197,343,317	245,024,146	313	1,859
ged14 8-9	6	5	138,961,033 (6.4)	51,194,955 (3.6)	155,044,171	333,194,875	176,316,713	332	231
transport11 08*	322	322	128,705,840 (6.0)	8,051,402 (0.4)	387,834,091	412,873,224	884,718,548	346	801
spider18 13	39	39	22,425,296 (3.0)	11,237,994 (1.7)	24,212,999	90,945,206	136,020,346	386	1,998
woodworking08 27	350	357	147,870,490 (7.4)	11,904,397 (0.7)	339,097,806	861,630,216	2,069,219,635	459	3,832
barman11 02-005	121	132	120,520,775 (6.1)	68,296,332 (5.5)	521,644,317	2,608,774,309	7,918,031,472	544	15,723
transport14 09*	966	996	127,472,989 (5.9)	37,110,556 (2.6)	741,922,587	747,591,228	2,308,571,006	549	2,327
agricola18 02	878	935	118,891,472 (6.7)	47,721,893 (2.9)	494,412,749	512,747,517	1,288,107,521	617	3,087
agricola18 03	788	793	152,528,391 (7.5)	112,549,403 (7.5)	706,797,822	823,748,774	2,287,807,416	790	5,617
agricola18 01	786	811	165,044,134 (7.9)	80,711,867 (5.5)	748,674,870	862,715,079	1,870,507,373	801	4,170
elevators08 29*	101	102	510,125,709 (24)	39,385,742 (2.6)	1,585,443,774	1,978,650,724	4,915,391,696	1,404	4,477
agricola18 05	979	980	262,635,603 (13)	79,632,741 (5.7)	1,184,932,854	1,398,771,125	3,238,751,246	1,533	8,788
elevators08 19*	112	115	765,075,247 (31)	114,952,004 (6.7)	2,159,252,861	2,571,563,855	8,409,534,183	2,140	7,957
floortile11 05-009	76	76	55,090,801 (2.9)	5,458,652 (0.3)	117,955,924	186,575,430	370,354,510	5,332	47,932

Table 7.2: Peak stored nodes, total generated nodes, and running time of A* and IDUCHS on domains with non-unit edge costs. Instances sorted by A* running time. Numbers in parentheses are memory usage in GB. An underline means more than 8 GB of memory was needed. Undirected graphs are marked with *.

never more than 10%.

7.4.1 A* vs. IDUCHS

IDUCHS required less memory than A* on 19 out of these 20 problem instances, sometimes significantly less, while A* was faster than IDUCHS. IDUCHS reduced the peak number of stored nodes by up to an order of magnitude over A* on the *elevators08*, *floortile11*, *transport11*, and *woodworking08* domains, by a factor of about 4 on *transport14*, and by a factor of around 2 on the other 6 domains. On the other hand, IDUCHS always generated more nodes than A*, and sometimes significantly more. As a result, IDUCHS was slower than A* on 19 out of these 20 problem instances.

To our surprise, on *ged14* 8-9, IDUCHS generated more nodes than A* but was faster, for three reasons. First, A* stored more nodes, resulting in more cache misses and more time spent on rehashing, which occurs when the hash map size is increased. Second, A* performs duplicate detection for each generated node, while IDUCHS prunes a node whose f-value is above the cost bound without duplicate checking. Third, we optimized our IDUCHS code to perform goal checking on every generated node. As a result, on *ged14* 8-9, IDUCHS found the goal node at its optimal g-value 6 in the iteration where the cost bound was 5. After IDUCHS verified that no optimal solution with a cost of 5 exist, IDUCHS did not need to run the iteration with cost bound $U = C^* = 6$ at all.

We present the memory and time comparisons in Figure 7.4, where the x-axis is A*'s peak stored nodes over IDUCHS's, and the y-axis is IDUCHS's running time over A*'s. Each circle represents one problem instance in Table 7.2. The points below the $y = x$ line represent problems for which the memory reduction ratio of IDUCHS over A* is higher than the increased running time ratio, while the points above the line represent the problems with the opposite property.

Comparing the memory usage in GB, the memory reduction ratio is usually less than the peak stored nodes reduction ratio for two reasons. First, the memory usage in GB includes memory used by the planner itself and the heuristic functions. On some problem instances,

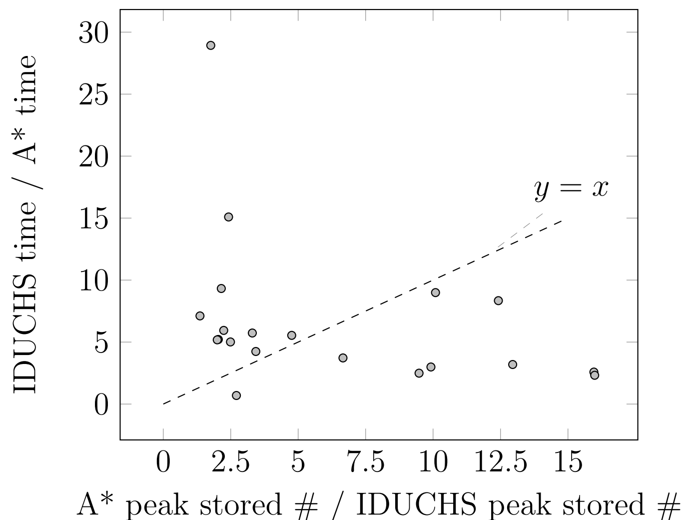

Figure 7.4: A* vs. IDUCHS in memory and time.

the heuristic function took up to 1 GB of memory. Second, our implementation was not optimized for memory, compared to the original A* implementation in Fast Downward. In our code, we used several C++ STL `vector`s, whose memory usage grows multiplicatively by a factor of 2. Optimizing memory usage of our code is for future work. Furthermore, a state's heuristic value is computed only once in A* no matter how many times the state is generated. In IDUCHS, the heuristic value of a state can be calculated more than once if that state is generated but then deleted. Therefore, the running time ratio of IDUCHS over A* is different from the total nodes ratio.

We compared IDUCHS using Equations 7.1 and 7.2 for computing the d_g-values. Using Equation 7.1 stored around 60% more nodes on *elevators08*, 30% more nodes on *floortile11* 05-009, 40% to 90% more nodes on *transport11*, and 50% more nodes on *woodworking08* 27. In contrast, using Equation 7.1 generated 60% fewer nodes than Equation 7.2 on *data-network18* 06 while storing 8% more nodes.

When the cost bound equals or exceeds the optimal solution cost C^*, the order in which nodes are generated can significantly impact the time of this final iteration, depending on how soon an optimal solution is found. This is due to node ordering. BFIDA* in general has very poor node ordering, since its breadth-first search order usually generates almost all

nodes in the final iteration. As a result, on unit-cost domains, the number of nodes generated by a single call to BFHS with a cost bound C^* is usually significantly larger than that of A*. However, the situation is different on domains with non-unit edge costs. In Table 7.2, we see that UCHS (C^*) generated a similar number of nodes to A* on many problem instances, including all instances from the three undirected graph domains. In fact, the only domains where UCHS (C^*) generated significantly more nodes than A* due to node ordering are *ged14*, *spider18*, and *woodworking08*. This is because nodes generated on domains with non-unit edge costs usually have many more different f-values than that of unit-cost domains. For example, there are hundreds of different f-values on the two *transport* domains, which is very rare in unit-cost domains. As a result, on domains with non-unit edge costs, if the cost bound of IDUCHS is increased by one, usually only a few new nodes will be expanded. Thus, the almost-worst node ordering of IDUCHS is not a serious drawback on these problems.

On undirected graphs, frontier search can avoid leaking back into the interior of the search space, and re-expanding nodes that have already been expanded. This cannot be done in general with directed graphs, since there is nothing to prevent the generation of a node in the interior of the search [ZH04, KZT05]. As a result, on the directed-graph domains *barman11*, *data-network18*, and *sokoban08*, UCHS (C^*) generated many more nodes than A*. To reduce the number of node re-expansions on directed graphs, we modified IDUCHS to increase the d_g-value of each node by $U/5$, where U was the cost bound. For example, if a node n originally had $d_g(n) = 50$ according to Equation 7.2 and $U = 100$, then we increased $d_g(n)$ to $50 + 100/5 = 70$. Compared to IDUCHS with Equation 7.2, IDUCHS with this new d_g formula stored around 15% more nodes on *barman11* 02-005 and *data-network18* 06, but reduced the total generated nodes by 78% and 61% respectively. On *sokoban08* 28, IDUCHS with this new d_g formula also pruned most duplicate nodes, but did not save memory compared to A* due to the last iteration being larger than A*.

7.4.2 Cost Bounds in IDUCHS

Table 7.2 shows the cost bound of the last iteration of IDUCHS. These bounds were calculated by Equation 7.3. The last cost bound was usually greater than C^*, with an exception on *ged14* 8-9, where the last cost bound was $C^* - 1$, for reasons we have explained in the previous subsection.

To analyze our cost-bound equation, we present the last iteration analysis in Figure 7.5, where the x-axis is the number of nodes generated in IDUCHS's last iteration divided by that of UCHS (C^*), the y-axis is the number of nodes generated in IDUCHS's last iteration divided by that of all iterations, and each triangle corresponds to one problem instance in Table 7.2. With $r = 2$ for Equation 7.3, we expected to see all data points having $y \approx 50\%$ and $x \leq 2$.

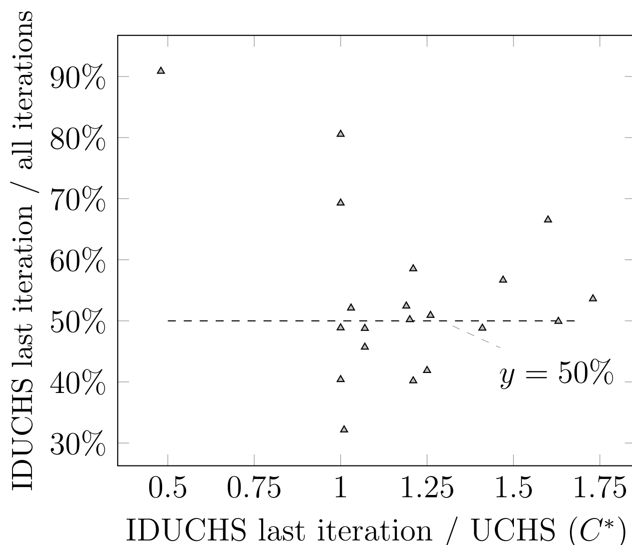

Figure 7.5: Comparisons of IDUCHS's last iteration.

From Figure 7.5, we see that the number of nodes generated in the last iteration of IDUCHS was within a factor of two of the nodes generated in UCHS (C^*) on all 20 problem instances. The number of nodes generated in the last iteration of IDUCHS was around 50% of the nodes generated in all IDUCHS iterations on most problem instances. This percentage is higher for some problem instances. For example, *ged14* 8-9 had $y = 91\%$ and *spider18* 13

and 18 had $y = 70\%$ and $y = 81\%$ respectively. This is because those instances have a small number of unique f-values, and the cost bound difference between iterations was just one, as in unit-cost domains. As a result, a generated nodes ratio of two between iterations was not possible, as one is the minimum cost bound increment. On the other hand, *transport14 08* had a percentage of 32%. This is because compared to the penultimate iteration, not many new nodes were generated in the last iteration, despite the large cost bound. In short, Figure 7.5 and the above analysis show that Equation 7.3 was very accurate in calculating the cost bounds of IDUCHS on domains where each iteration is not much larger than its previous iteration when using a cost bound increment of one.

7.5 A*+UCHS

Inspired by A*+BFHS, we implemented A*+UCHS, a hybrid algorithm combining A* and UCHS. A*+UCHS first runs A* up to a user-specified memory threshold, then runs a series of UCHS iterations on the Open nodes of A*. A*+UCHS uses Equation 7.3 to calculate the cost bounds in the UCHS phase. Similar to our experiments in Section 6.5, we first generated the heuristics, then allocated 1/10 of the remaining 8 GB of memory for the A* phase.

Figure 7.6 shows the comparison between IDUCHS and A*+UCHS. The x-axis is the peak stored nodes of IDUCHS divided by that of A*+UCHS, the y-axis is IDUCHS's time divided by A*+UCHS's, and each square represents one problem instance in Table 7.2. In Figure 7.6, the data points to the right of the $x = 1$ line represent problem instances where A*+UCHS used less memory than IDUCHS, while the data points above the $y = 1$ line represent problem instances where A*+UCHS was faster than IDUCHS.

Little speedup was achieved when switching from IDUCHS to A*+UCHS on domains with non-unit edge costs. A*+UCHS was faster than IDUCHS by a factor of at least two on only three of the 20 problem instances in Table 7.2. The speedups of A*+BFHS over BFIDA* mainly come from A*+BFHS terminating early in the last iteration, as A*+BFHS can im-

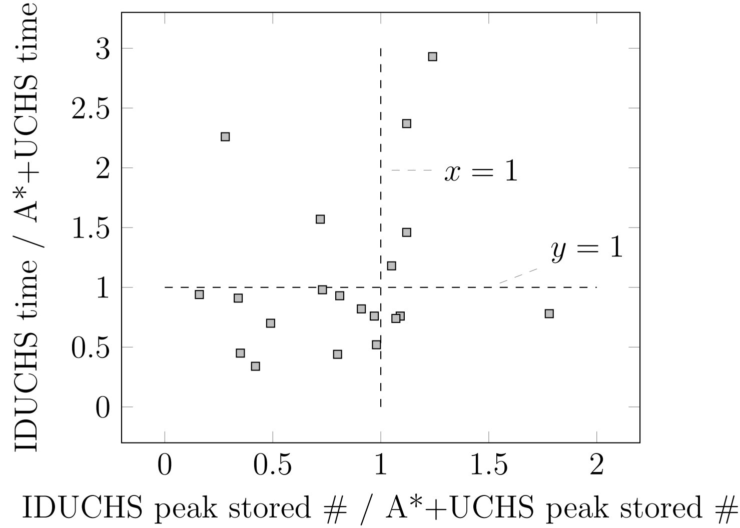

Figure 7.6: IDUCHS vs. A*+UCHS in memory and time.

mediately stop when a goal node is generated. However, on domains with non-unit edge costs, cost bounds usually increase by more than one in A*+UCHS's UCHS phase. Thus, when A*+UCHS finds a goal node, it usually cannot stop but has to continue searching to verify that the solution is optimal. Therefore, early termination is usually not possible in A*+UCHS, with the exception of problem instances where the cost bounds between iterations increase by one. It is worth noting that on the easy non-unit cost problem instances which can be solved in A*+UCHS's A* phase, A*+UCHS is faster than IDUCHS but IDUCHS also quickly solves the easy problem instances. On unit-cost domains, however, A*+UCHS behaves similarly to A*+BFHS, and hence is faster than IDUCHS.

CHAPTER 8

Conclusions and Future Work

8.1 Summary of Contributions

A* [HNR68] is usually the first algorithm to try for a new heuristic search domain and the *de facto* algorithm for classic optimal planning. However, A* can run out of 8 GB of RAM on common heuristic search and planning domains in a few minutes. On the other hand, the memory requirement of iterative-deepening-A* (IDA*) [Kor85] is only linear in the search depth, and therefore does not run out of memory on common heuristic search and planning domains. However, on a domain where many distinct paths exist between the same pair of nodes, IDA* may generate too many duplicate nodes, and hence run for a very long time.

In this research, we provide three new algorithms, A*+IDA*, A*+BFHS, and IDUCHS, for solving problems that cannot be solved by A* due to memory limitations, nor IDA* due to the existence of many distinct paths between the same pair of nodes.

8.1.1 A*+IDA*

A*+IDA* is a simple combination of A* and IDA* to make use of additional memory. It is widely believed that this algorithm was discovered in 1989 and called MREC [SB89], but in fact MREC is just IDA* with a transposition table. While there are several combinations of A* and IDA* in the literature, to the best of our knowledge, none of them forego duplicate detection during IDA* and order the frontier nodes of equal f-value by increasing h-values at the same time.

A*+IDA* is faster than the most efficient implementation of IDA* on the 24-Puzzle by a

factor of almost 5, and the first algorithm to significantly outperform IDA* on this problem. We corrected the original dual search implementation and optimized it to be faster than IDA* by a factor of 2.2, which is the fastest linear search algorithm on the 24-Puzzle. We further combined A*+IDA* with our improved dual search and found that it is faster than IDA* by a factor of 6.7. We also show that using a disk-based A* search in A*+IDA* speeds up search on several very hard 24-Puzzle instances. We also found optimal solutions to a subset of random 27 and 29-Puzzle problems and we show how to use A*+IDA* to quickly find a solution on sliding-tile puzzles. A*+IDA* is not faster than IDA* on Rubik's Cube, due to many fewer duplicate nodes and less effective node ordering in the last iteration, but not significantly slower either.

In general, if A* solves a problem, there is no overhead in using A*+IDA*. If not, then compared to pure IDA*, the overhead of A*+IDA* is likely to be more than compensated for by the duplicate pruning in its A* phase, and the improved node ordering of the last iteration in its IDA* phase.

This work was published in the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI), 2019, under the title "A*+IDA*: A Simple Hybrid Search Algorithm" [BK19].

8.1.2 A*+BFHS

A*+BFHS is a hybrid heuristic search algorithm based on A* and breath-first heuristic search (BFHS) [ZH04] that can be used for solving problems with unit-cost operators that cannot be solved by A* or IDA*. A*+BFHS first runs A* until a user-specified storage threshold is reached, then runs multiple iterations of BFHS on the frontier nodes, which are the Open nodes at the end of the A* phase. Each iteration has a unique cost bound and contains multiple calls to BFHS. Each call to BFHS within the same iteration starts with the same cost bound but a different set of frontier nodes. Within an iteration, frontier nodes are sorted deepest-first so that A*+BFHS can terminate early in its last iteration.

We tested A*+BFHS on 32 unit-cost planning domains. On about 500 easy problem

instances solved, A*+BFHS behaves the same as A*, and is always faster than breath-first iterative-deepening A* (BFIDA*) [ZH04]. On the 32 hard instances presented, A*+BFHS is slower than A* but uses significantly less memory. A*+BFHS is faster than BFIDA* on 27 of those 32 instances and at least twice as fast on 16 of those. Furthermore, A*+BFHS requires less memory than BFIDA* on 25 of those 32 instances and saves more than half the memory on 14 of those. Another contribution of this research is a comprehensive testing of BFIDA* on many planning domains, which is lacking in the literature.

This work was published in the Thirty-Sixth AAAI Conference on Artificial Intelligence (AAAI), 2022, under the title "A*+BFHS: A Hybrid Heuristic Search Algorithm" [BK22a].

8.1.3 IDUCHS

We also propose uniform-cost heuristic search (UCHS) and its iterative-deepening version IDUCHS, for optimally solving heuristic search and planning problems. Unlike breadth-first heuristic search [ZH04], which only works on unit-cost domains, UCHS works on domains with arbitrary non-negative edge costs. UCHS is a uniform-cost frontier search that uses a heuristic evaluation function to prune nodes. UCHS is guaranteed not to expand a state more than once on undirected graphs. On directed graphs, UCHS can re-expand nodes, but we show how to adjust UCHS to avoid many such re-expansions. We also propose a new way to calculate the cost bounds of IDUCHS iterations.

We have solved around 300 problem instances from 19 planning domains with non-unit edge costs, and presented the results of the hardest 20 problem instances. Experimental results show that IDUCHS is slower than A*, but requires less memory than A*, sometimes significantly, on a variety of planning domains.

This work was published in the Fifteenth International Symposium on Combinatorial Search (SOCS), 2022, under the title "Iterative-Deepening Uniform-Cost Heuristic Search" [BK22b].

8.1.4 When to Use Each Algorithm?

Since we have introduced three new algorithms, one immediate question is: given a new problem, how to choose which algorithm to use? In general, given a new problem instance, we can first run A* and count the number of duplicate nodes generated during A*. If A* solves it, then we can stop. Otherwise, if the majority of nodes generated in A* represent unique states, as in the case of the 24-Puzzle, we can try A*+IDA*. If many duplicate nodes are generated in A*, then we can use A*+BFHS or IDUCHS, depending on whether the problem instance comes from a unit-cost domain.

8.1.5 Node Ordering Is Important

Both A*+IDA* and A*+BFHS order their frontier nodes based on the heuristic value of those nodes and expand the frontier node with the smallest heuristic value first in each iteration. In contrast, IDA* orders nodes based on the order of operators being applied while BFIDA* orders nodes according to their depth, which means that the heuristic values of nodes do not contribute to the node ordering, with the exception of pruning nodes based on the current cost bound.

Our experimental results of A*+IDA* and A*+BFHS show how much node ordering can affect the number of nodes generated in heuristic search algorithms. Both IDA* and BFIDA* generate most nodes in their last iteration, while A*+IDA* and A*+BFHS generate most nodes in their penultimate iteration on most domains. The significant savings of nodes in the last iteration is the main reason for the speedups of A*+IDA* over IDA* on the 24-Puzzle, and A*+BFHS over BFIDA* on various planning domains. The explanation for why node ordering works well in A*+IDA* and A*+BFHS is quite intuitive. When the heuristic function is relatively accurate, which means a state's heuristic value is roughly proportional to its optimal cost to the goal state, then when we expand frontier nodes in increasing order of the heuristic values, it is highly likely that we expand the frontier node leading to the optimal goal quite early.

As a result, our work suggests that node ordering should play an important role in algorithm design. If a heuristic search algorithm does not order nodes based on the heuristic values, then there may be a chance to change the node ordering scheme in this algorithm to speed up search.

8.1.6 A Scheme for Combing Heuristic Search Algorithms

Our combination of A* and IDA* provides a general way to combine best-first search algorithms with depth-first search algorithms. We also combined A* with recursive best-first search (RBFS) [Kor93] and observed similar speedups as A*+IDA* on the 24-Puzzle. Similarly, we can also combine IDA* or RBFS with other A* variations like divide-and-conquer frontier A* (DCFA*) [KZ00] to achieve similar speedups.

Our combination of A* and BFHS also provides a general way to combine two memory-intensive search algorithms, which is rare in the literature. Traditionally, when a combination of different heuristic search algorithms is used, usually one memory intensive search algorithm is combined with one linear space search algorithm. However, such combination can still generate a lot of duplicate nodes during the search if only a small portion of nodes in the search space can be stored in memory. Our A*+BFHS shows that when designed properly, two different memory-intensive search algorithms can be combined together to achieve the advantages of both algorithms at the same time.

We anticipate that as heuristic search and planning come of age, it will be more and more common to develop new search algorithms that are a combination of different previously developed search algorithms to achieve the advantages of those combined algorithms while minimizing their disadvantages.

8.2 Ideas for Future Work

Future work based on this research includes several directions. For example, we can test A*+IDA*, A*+BFHS, and IDUCHS on more domains and problem instances. In addition,

we may explore if such algorithm combinations are possible in other related fields like multi-agent path finding, answer set programming, constraint satisfaction problems, and games.

8.2.1 When Will Node Ordering Work?

Currently, we do not know if node ordering in A*+IDA* and A*+BFHS will work well before we actually run both algorithms on a domain. Our current observation is that if the heuristic value returned by a heuristic function is roughly proportional to the state's optimal cost to the goal state, then node ordering should work well in general on this domain, and A*+IDA* and A*+BFHS should generate most nodes in their penultimate iteration.

On the other hand, if many states that are far from the goal state have a similar heuristic value to the states that are close to the goal state, then node ordering may not work well. In this case, the frontier nodes that have a relatively small heuristic value, which are also the nodes expanded early in each iteration, may in fact be far from the goal state, hence A*+IDA* and A*+BFHS may not discover the goal early in their last iteration. A good example is our A*+IDA* results on Rubik's Cube. The states that are 10 to 20 moves away from the goal state all have the similar heuristic values. Therefore, the frontier node on an optimal path was expanded quite late on some test cases because many frontier nodes that are in fact far from the goal state had a smaller heuristic value, and hence were expanded before the goal-led frontier node.

How can we know if the heuristic value returned by a heuristic function is roughly proportional to the state's optimal cost to the goal state? To get a good estimation, we may have to generate random states at various depths and then compute their heuristic values and check the heuristic value distribution. However, when we randomly generate a state, we do not know its optimal cost to the goal state, and the only way to find that out is to actually solve it. Therefore, more ideas are needed here.

8.2.2 Disk-based Search

We can also use external memory such as magnetic disk or flash memory in A*+BFHS and IDUCHS to solve even harder problems. For example, in A*+BFHS, instead of allocating 1/10 of RAM for the A* phase, we can first run A* until RAM is almost full, then store both Open and Closed nodes in external memory and remove them from RAM. Then in the BFHS phase, we load back the set of frontier nodes for each call to BFHS from external memory. This A*+BFHS version would never perform worse than A*, since it is identical to A* until memory is exhausted, at which point the BHFS phase would begin.

8.2.3 Delayed Closed Node Deletion

On directed graphs, both A*+BFHS and IDUCHS cannot detect all duplicate nodes. To mitigate this issue, we can modify both algorithms such that they only remove a node from memory when the memory is almost full. For example, in IDUCHS, instead of removing a node from memory after expanding all nodes up to its d_g-value, we may just keep all nodes stored in memory until the RAM is almost full, at which time we start to remove nodes according to their d_g-values. In this way, we will be able to detect more duplicate nodes on directed graphs, hence speeding up the search.

8.2.4 Computing Cost Bounds Between Iterations

On domains with non-unit edge costs, if we only increase the cost bounds between iterations by 1, then we may end up having hundreds or thousands of iterations with each iteration being only slightly larger than the previous one. Therefore, we proposed Equation 7.3 to calculate the cost bonds between iterations, so that each iteration is much larger than its previous iteration.

For IDUCHS, we could refine Equation 7.3 to achieve more accurate results in calculating the cost bounds. For example, we may take the previous three or more iterations into consideration.

Furthermore, as can be seen in Table 6.1, sometimes each iteration in BFIDA* and A*+BFHS is also only slightly larger than its previous iteration on unit-cost domains. For instance, the number of nodes generated in BFIDA*'s last iteration on *termes18* 05 is only 5.9% of the total nodes generated. As a result, BFIDA* and A*+BFHS generated more than an order of magnitude more nodes than A* on this problem instance. Therefore, we could also apply Equation 7.3 on unit-cost domains to reduce the number of iterations, hence speeding up BFIDA* and A*+BFHS on those problems. For example, we may first increase the cost bound by only one. After several iterations, if we see each iteration is only slightly larger than its previous one, then we can use Equation 7.3 to compute the future cost bounds.

8.2.5 A*+BFHS

Other ideas that are specific to A*+BFHS include the following. First, investigating what is the best memory threshold for the A* phase. Second, determining the optimal number of calls to BFHS in each iteration. Third, finding other ways to partition the frontier nodes besides the current depth-based approach. If a set of frontier nodes is too large, we may split it into multiple smaller sets and make one call to BFHS on each such smaller set. This approach may reduce the maximum number of stored nodes but may generate more duplicate nodes. In addition, when we make each call to BFHS on frontier nodes at multiple depths, we may consider the number of frontier nodes at each depth so each call to BFHS is on a different number of depths instead of a fixed number.

8.2.6 IDUCHS

Future work that is specific to IDUCHS includes the following. First, adjust the d_g-values automatically to reduce the duplicate nodes of IDUCHS on directed graphs. For example, at the beginning of IDUCHS, we can make a few calls to UCHS, each with the same cost bound but a different d_g-value adjustment to find out if the problem is a directed graph and whether the d_g-value adjustment works. Second, we can optimize the code to reduce the memory usage.

8.2.7 Searching on Directed Graphs

As shown in [KZT05] and in our Section 4.8, frontier search in general cannot detect all duplicate nodes on directed graphs, with the exception in the case where we know all the predecessor nodes of each node in advance. Our experimental results of A*+BFHS and IDUCHS also show that they can generate many more nodes than A* on directed graphs. This shows that compared to undirected graphs, it is harder to develop algorithms than can work well on directed graphs. Therefore, we could also explore this direction and see if we can come up with a new algorithm for directed graphs.

8.3 Conclusions

Heuristic search and planning have existed as sub-fields of artificial intelligence for several decades, during which an average computer's RAM size has increased from multiple KB to 16 GB or even more. In addition to the hardware evolution, new heuristic search algorithms and heuristic functions like pattern databases [CS98] have enabled us to optimally solve hard problems that we could not solve before, like Rubik's Cube [Kor97] and the 24-Puzzle [KF02].

Due to the nature of exponential domains, RAM size will never be enough for hard problems. As a result, even A* cannot solve 24-Puzzle instances due to memory limitations on modern computers. In contrast, IDA*'s linear space requirement makes it possible to solve any hard problem in theory. However, many domains in both heuristic search and planning have the property that many distinct paths exist between the same pair of nodes. As a result, most nodes generated by IDA* on those domains will be duplicate nodes, hence making IDA* run an exceptionally long time on such problems.

In light of above observations, we present in this dissertation three new algorithms for solving problems that cannot be solved by A* due to memory limitations, nor IDA* due to the existence of many distinct paths between the same pair of nodes. Our algorithms also illustrate the importance of node ordering in algorithm design and we provide a scheme for

building new algorithms by combining existing algorithms to realize their advantages. We hope our algorithms provide new insights and directions for solving hard problems. Instead of testing new algorithms on easy domains that have already been solved, such as the 15-Puzzle, we encourage researchers and future readers of this dissertation to focus on solving hard problems that could not be solved before.

Bibliography

[AF16] Masataro Asai and Alex S. Fukunaga. "Tiebreaking Strategies for A* Search: How to Explore the Final Frontier." In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pp. 673–679. AAAI Press, 2016.

[AHK98] Constructions Aeronautiques, Adele Howe, Craig Knoblock, ISI Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, David Wilkins SRI, Anthony Barrett, Dave Christianson, et al. "PDDL — The Planning Domain Definition Language." *Technical Report, Tech. Rep.*, 1998.

[AK04] Andreas Auer and Hermann Kaindl. "A Case Study of Revisiting Best-First vs. Depth-First Search." In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pp. 141–145. IOS Press, 2004.

[AKF10] Yuima Akagi, Akihiro Kishimoto, and Alex Fukunaga. "On Transposition Tables for Single-Agent Search and Planning: Summary of Results." In *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010, Stone Mountain, Atlanta, Georgia, USA, July 8-10, 2010*. AAAI Press, 2010.

[BK15] Joseph Kelly Barker and Richard E. Korf. "Limitations of Front-To-End Bidirectional Heuristic Search." In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pp. 1086–1092. AAAI Press, 2015.

[BK19] Zhaoxing Bu and Richard E. Korf. "A*+IDA*: A Simple Hybrid Search Algorithm." In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pp. 1206–1212. ijcai.org, 2019.

[BK22a] Zhaoxing Bu and Richard E. Korf. "A*+BFHS: A Hybrid Heuristic Search Algorithm." In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pp. 10138–10145. AAAI Press, 2022.

[BK22b] Zhaoxing Bu and Richard E. Korf. "Iterative-Deepening Uniform-Cost Heuristic Search." In Lukás Chrpa and Alessandro Saetti, editors, *Proceedings of the Fifteenth International Symposium on Combinatorial Search, SOCS 2022, Vienna, Austria, July 21-23, 2022*, pp. 20–28. AAAI Press, 2022.

[BSF14] Zhaoxing Bu, Roni Stern, Ariel Felner, and Robert Craig Holte. "A* with Lookahead Re-Evaluated." In Stefan Edelkamp and Roman Barták, editors, *Proceedings*

of the Seventh Annual Symposium on Combinatorial Search, SOCS 2014, Prague, Czech Republic, 15-17 August 2014. AAAI Press, 2014.

[CGA89] P. P. Chakrabarti, Sujoy Ghose, Arup Acharya, and S. C. De Sarkar. "Heuristic Search in Restricted Memory." *Artif. Intell.*, **41**(2):197–221, 1989.

[CS98] Joseph C. Culberson and Jonathan Schaeffer. "Pattern Databases." *Comput. Intell.*, **14**(3):318–334, 1998.

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

[Ede16] Stefan Edelkamp. "External-Memory State Space Search." In *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pp. 185–225. 2016.

[ES95] Jürgen Eckerle and Sven Schuierer. "Efficient Memory-Limited Graph Search." In *KI-95: Advances in Artificial Intelligence, 19th Annual German Conference on Artificial Intelligence, Bielefeld, Germany, September 11-13, 1995, Proceedings*, volume 981 of *Lecture Notes in Computer Science*, pp. 101–112. Springer, 1995.

[ES00] Stefan Edelkamp and Stefan Schrödl. "Localizing A^*." In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, pp. 885–890. AAAI Press / The MIT Press, 2000.

[ES11] Stefan Edelkamp and Stefan Schroedl. *Heuristic search: theory and applications.* Elsevier, 2011.

[Fel05] Ariel Felner. "Finding optimal solutions to the graph partitioning problem with heuristic search." *Ann. Math. Artif. Intell.*, **45**(3-4):293–322, 2005.

[FKM07] Ariel Felner, Richard E. Korf, Ram Meshulam, and Robert C. Holte. "Compressed Pattern Databases." *J. Artif. Intell. Res.*, **30**:213–247, 2007.

[FLB18] Santiago Franco, Levi HS Lelis, Mike Barley, Stefan Edelkamp, Moises Martines, and Ionut Moraru. "The Complementary2 planner in the IPC 2018." *IPC-9 planner abstracts*, pp. 28–31, 2018.

[FMS10] Ariel Felner, Carsten Moldenhauer, Nathan R. Sturtevant, and Jonathan Schaeffer. "Single-Frontier Bidirectional Search." In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

[FN71] Richard Fikes and Nils J. Nilsson. "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving." *Artif. Intell.*, **2**(3/4):189–208, 1971.

[FTL17] Santiago Franco, Álvaro Torralba, Levi H. S. Lelis, and Mike Barley. "On Creating Complementary Pattern Databases." In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pp. 4302–4309. ijcai.org, 2017.

[FZS05] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert C. Holte. "Dual Lookups in Pattern Databases." In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pp. 103–108. Professional Book Center, 2005.

[HBH07] Patrik Haslum, Adi Botea, Malte Helmert, Blai Bonet, and Sven Koenig. "Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning." In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, pp. 1007–1012. AAAI Press, 2007.

[HD09] Malte Helmert and Carmel Domshlak. "Landmarks, Critical Paths and Abstractions: What's the Difference Anyway?" In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*. AAAI, 2009.

[Hel06] Malte Helmert. "The Fast Downward Planning System." *J. Artif. Intell. Res.*, **26**:191–246, 2006.

[HLL19] Malte Helmert, Tor Lattimore, Levi H. S. Lelis, Laurent Orseau, and Nathan R. Sturtevant. "Iterative Budgeted Exponential Search." In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pp. 1249–1257. ijcai.org, 2019.

[HNF04] Robert C. Holte, Jack Newton, Ariel Felner, Ram Meshulam, and David Furcy. "Multiple Pattern Databases." In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, pp. 122–131. AAAI, 2004.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Trans. Syst. Sci. Cybern.*, **4**(2):100–107, 1968.

[Hol10] Robert C. Holte. "Common Misconceptions Concerning Heuristic Search." In *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010, Stone Mountain, Atlanta, Georgia, USA, July 8-10, 2010*. AAAI Press, 2010.

[KF02] Richard E. Korf and Ariel Felner. "Disjoint pattern database heuristics." *Artif. Intell.*, **134**(1-2):9–22, 2002.

[KK94] Hermann Kaindl and Aliasghar Khorsand. "Memory-Bounded Bidirectional Search." In *Proceedings of the 12th National Conference on Artificial Intelligence,*

Seattle, WA, USA, July 31 - August 4, 1994, Volume 2, pp. 1359–1364. AAAI Press / The MIT Press, 1994.

[KKL95] Hermann Kaindl, Gerhard Kainz, Angelika Leeb, and Harald Smetana. "How to Use Limited Memory in Heuristic Search." In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, pp. 236–242. Morgan Kaufmann, 1995.

[Kor85] Richard E. Korf. "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search." *Artif. Intell.*, **27**(1):97–109, 1985.

[Kor93] Richard E. Korf. "Linear-Space Best-First Search." *Artif. Intell.*, **62**(1):41–78, 1993.

[Kor97] Richard E. Korf. "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases." In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*, pp. 700–705. AAAI Press / The MIT Press, 1997.

[Kor04] Richard E. Korf. "Best-First Frontier Search with Delayed Duplicate Detection." In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pp. 650–657. AAAI Press / The MIT Press, 2004.

[KRE01] Richard E. Korf, Michael Reid, and Stefan Edelkamp. "Time complexity of iterative-deepening-A*." *Artif. Intell.*, **129**(1-2):199–218, 2001.

[KSS18] Michael Katz, Shirin Sohrabi, Horst Samulowitz, and Silvan Sievers. "Delfi: Online planner selection for cost-optimal planning." *IPC-9 planner abstracts*, pp. 57–64, 2018.

[KT96] Richard E. Korf and Larry A. Taylor. "Finding Optimal Solutions to the Twenty-Four Puzzle." In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 2*, pp. 1202–1207. AAAI Press / The MIT Press, 1996.

[Kwa89] James B. H. Kwa. "BS*: An Admissible Bidirectional Staged Heuristic Search Algorithm." *Artif. Intell.*, **38**(1):95–109, 1989.

[KZ00] Richard E. Korf and Weixiong Zhang. "Divide-and-Conquer Frontier Search Applied to Optimal Sequence Alignment." In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, pp. 910–916. AAAI Press / The MIT Press, 2000.

[KZT05] Richard E. Korf, Weixiong Zhang, Ignacio Thayer, and Heath Hohwald. "Frontier search." *J. ACM*, **52**(5):715–748, 2005.

[Man95] Giovanni Manzini. "BIDA: An Improved Perimeter Search Algorithm." *Artif. Intell.*, **75**(2):347–360, 1995.

[MME18] M Martinez, I Moraru, S Edelkamp, and S Franco. "Planning-PDBs planner in the IPC 2018." *IPC-9 planner abstracts*, pp. 63–66, 2018.

[PK89] Curt Powley and Richard E. Korf. "Single-Agent Parallel Window Search: A Summary of Results." In *Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989*, pp. 36–41. Morgan Kaufmann, 1989.

[RM94] Alexander Reinefeld and T. Anthony Marsland. "Enhanced Iterative-Deepening Search." *IEEE Trans. Pattern Anal. Mach. Intell.*, **16**(7):701–710, 1994.

[Rus92] Stuart J. Russell. "Efficient Memory-Bounded Search Methods." In *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*, pp. 1–5. John Wiley and Sons, 1992.

[SA83] David J Slate and Lawrence R Atkin. "Chess 4.5—the Northwestern University chess program." In *Chess skill in Man and Machine*, pp. 82–118. Springer, 1983.

[SB89] Anup K. Sen and Amitava Bagchi. "Fast Recursive Formulations for Best-First Search That Allow Controlled Use of Memory." In *Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989*, pp. 297–302. Morgan Kaufmann, 1989.

[SCG91] U. K. Sarkar, P. P. Chakrabarti, Sujoy Ghose, and S. C. De Sarkar. "Reducing Reexpansions in Iterative-Deepening Search by Controlling Cutoff Bounds." *Artif. Intell.*, **50**(2):207–221, 1991.

[SDR13] Thorsten Schütt, Robert Döbbelin, and Alexander Reinefeld. "Forward Perimeter Search with Controlled Use of Memory." In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pp. 659–665. IJCAI/AAAI, 2013.

[Sie18] Silvan Sievers. "Merge-and-Shrink Heuristics for Classical Planning: Efficient Implementation and Partial Abstractions." In *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden - 14-15 July 2018*, p. 99. AAAI Press, 2018.

[SKF10] Roni Stern, Tamar Kulberis, Ariel Felner, and Robert Holte. "Using Lookaheads with Optimal Best-First Search." In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010.* AAAI Press, 2010.

[SOH12] Silvan Sievers, Manuela Ortlieb, and Malte Helmert. "Efficient Implementation of Pattern Database Heuristics for Classical Planning." In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*. AAAI Press, 2012.

[SWH14] Silvan Sievers, Martin Wehrle, and Malte Helmert. "Generalized Label Reduction for Merge-and-Shrink Heuristics." In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, pp. 2358–2366. AAAI Press, 2014.

[SWH16] Silvan Sievers, Martin Wehrle, and Malte Helmert. "An Analysis of Merge Strategies for Merge-and-Shrink Heuristics." In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016*, pp. 294–298. AAAI Press, 2016.

[ZFH06] Uzi Zahavi, Ariel Felner, Robert Holte, and Jonathan Schaeffer. "Dual Search in Permutation State Spaces." In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pp. 1076–1081. AAAI Press, 2006.

[ZFH08] Uzi Zahavi, Ariel Felner, Robert C. Holte, and Jonathan Schaeffer. "Duality in permutation state spaces and the dual search algorithm." *Artif. Intell.*, **172**(4-5):514–540, 2008.

[ZH04] Rong Zhou and Eric A. Hansen. "Breadth-First Heuristic Search." In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, pp. 92–100. AAAI, 2004.