Title
Front-To-End Bidirectional Heuristic Search

Author
Barker, Joseph Kelly

Publication Date
2015

Peer reviewed|Thesis/dissertation

University of California
Los Angeles

Front-To-End Bidirectional Heuristic Search

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

by

Joseph Kelly Barker

2015

Abstract of the Dissertation

Front-To-End Bidirectional Heuristic Search

by

Joseph Kelly Barker

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2015

Professor Richard E. Korf, Chair

Bidirectional heuristic search is a well-known technique for solving pathfinding problems. The goal in a pathfinding problem is to find paths—often of lowest cost—between nodes in a graph. Many real-world problems, such as finding the quickest route between two points in a map or measuring the similarity of DNA sequences, can be modeled as pathfinding problems.

Bidirectional *brute-force* search does simultaneous brute-force searches forward from the initial state and backward from the goal states, finding solutions when both intersect. The idea of adding a heuristic to guide search is an old one, but has not seen widespread use and is generally believed to be ineffective.

I present an intuitive explanation for the ineffectiveness of front-to-end bidirectional heuristic search. Previous work has examined this topic, but mine is the first comprehensive explanation for why most front-to-end bidirectional heuristic search algorithms will usually be outperformed by either unidirectional heuristic or bidirectional brute-force searches. However, I also provide a graph wherein bidirectional heuristic search *does* outperform both other approaches, as well as real-world problem instances from the road navigation domain. These demonstrate that there can be no general, formal proof of the technique's ineffectiveness.

I tested my theory in a large number of popular search domains, confirming its predictions. One of my experiments demonstrates that a commonly-repeated explanation for the ineffectiveness of bidirectional heuristic search—that it spends most of its time proving solution optimality—is in fact wrong, and that with a strong heuristic a bidirectional heuristic search tends to find optimal solutions very late in a search.

Finally, I introduce state-of-the-art solvers for the four-peg Towers of Hanoi with arbitrary initial and goal states, and peg solitaire, using disk-based, bidirectional algorithms. The Towers of Hanoi solver is a bidirectional brute-force solver which, as my theory predicts, outperforms a unidirectional heuristic solver. The peg solitaire solver is a bidirectional heuristic algorithm with novel heuristics. While my theory demonstrates that bidirectional heuristic search is generally ineffective, the peg solitaire domain demonstrates several caveats to my theory that this algorithm takes advantage of.

The dissertation of Joseph Kelly Barker is approved.

Adnan Youssef Darwiche

Eleazar Eskin

John Mamer

Richard E. Korf, Committee Chair

University of California, Los Angeles

2015

We shall not cease from exploration

And the end of all our exploring

Will be to arrive where we started

And know the place for the first time.

˜ T S Eliot, Four Quartets ˜

Table of Contents

List of Figures

List of Tables

Acknowledgments

My thanks to my adviser, Rich Korf, for many years of advice and support. His high expectations, attention to detail, and dilligence were a model to me in my graduate student career. The example he set of what a researcher can be has made a profound impact on me, greatly increasing the standards to which I hold myself. He has constantly made himself available to me, providing me with support and feedback whenever I needed it. For his time, effort, and guidance, I am grateful.

Thanks also to the members of my committee: Adnan Darwiche, Eleazar Eskin, and John Mamer. They provided me with useful feedback and advice, and helped make my dissertation and research better. Their time and effort spent, in their duties as committee members, their classes I attended, and in many conversations over the years, were invaluable and appreciated.

George Bell, Jonathan Schaeffer, and Ariel Felner took time to help me with research problems, more time than they needed or than I expected. I am grateful for their thoughts, input, and help.

My parents, Nora and Paul, and my sisters, Leila and Eve, have been a constant and endless source of love and support to me. I could not ask for a better family, and I'm glad to have them in my life.

My thanks and appreciation to my partner, Jamie Stash, who has sacrificed and worked to build our relationship even over the challenges of distance. Her love and enthusiasm impress me, constantly surprising me with how good a relationship can be.

Ethan Schreiber has been my friend, collaborator, and companion in the trenches for most of my graduate school career. His encouragement and support helped me deal with challenges and setbacks, and he has been a constant and unwavering friend.

To the many fellow students and officemates in AI and elsewhere: Cesar Romero, Eric Huang, Alex Dow, Teresa Breyer, Trevor Standley, Zhaoxing Bu, and Karthika Mohan. Many good talks and solidarity. Thanks to Arthur Choi for regularly reminding me of the importance of stepping outside for some fresh air.

Thanks to Talia Gruen for the excellent picture in figure 6.5.

Vita

2003	BS (Computer Science), University of Oregon
2003–2007	Software Engineer, On Time Systems, Eugene, Oregon, USA
2010	MS (Computer Science), University of California, Los Angeles
2010-2013	Software Engineering Intern, Google, Seattle, Washington and Los Angeles, California

Publications

Joseph K Barker and Richard E Korf. "Limitations of Front-To-End Bidirectional Heuristic Search." In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015

Joseph K Barker and Richard E Korf. "Solving Peg Solitaire with Bidirectional BFIDA*." In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012

Joseph K Barker and Richard E Korf. "Solving Dots-And-Boxes." In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012

Joseph K Barker and Richard E Korf. "Solving 4x5 Dots-And-Boxes." In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011*, 2011. Extended abstract

CHAPTER 1

Introduction

One of the fundamental realizations in the field of artificial intelligence is that problem solving can be viewed as a search of a large space of possible solutions. We call this abstract space a *problem space* [New80]. A problem space is a graph that encodes states—configurations of the problem to be solved—and how one state can be transformed into another. This powerful idea lets us apply very general algorithms to very different types of problems. Problem solving becomes a search through a space of candidate solutions to find one that solves the original problem.

The field of *heuristic search* is concerned with finding algorithms that can efficiently search these problem spaces. It considers general-purpose algorithms that find solutions to real world problems represented as a problem space. The primary challenge in developing these algorithms is that problem spaces are often quite large. The problem-space graph of the well-known Rubik's Cube puzzle, for example, has $8! \times 3^7 \times (12!/2) \times 2^{11} \approx 4.3 \times 10^{19}$ unique states [SSH09]. If a computer were able to explore ten million nodes per second, it would take over 100,000 years to perform a complete exploration of the Rubik's Cube problem-space graph.

Some heuristic search approaches, known as local-search techniques, try to find good solutions quickly without proving optimality. In this dissertation, however, I am concerned with optimal algorithms, which are guaranteed to find the best possible solution if one exists. To do so, these algorithms must either explore the

entire search space, or prove that parts of the search space need not be explored to find and verify an optimal solution.

Heuristic search research is broadly broken down into three main categories: game playing, constraint-satisfaction problems, and pathfinding problems. Game-playing domains are those in which two or more players compete to win a game, such as chess, checkers, or go. A heuristic-search algorithm for a two-or-more player game tries to find an optimal strategy for one of the players, where a strategy gives the best possible response that player could make to any move their opponent might make. In a constraint-satisfaction problem, the goal is to find a complete assignment of values to a set of variables that does not violate some given constraints. A well-known example is the Sudoku puzzle, which is played on a 9×9 grid, which is further divided into nine 3×3 subgrids. The goal of the game is to fill the grid with the digits one through nine, subject to the constraint that no digit may be repeated in any row, column, or 3×3 subgrid. A heuristic-search algorithm tries to find a state in the problem space representing a complete assignment of values to variables that does not violate the constraints.

The final category, and the one studied in this paper, is that of pathfinding problems. In these problems, the goal is to find a path between two states in a problem space. Examples include the road navigation problem, where the goal is to find a path between two points on a map, and puzzles like the Rubik's Cube, where the goal is to find a sequence of moves that transform a scrambled initial state into a solved state.

Heuristic search algorithms use a number of approaches to reduce the amount of the problem-space graph explored so that the search can be completed in a tractable amount of time. The primary method is through the use of *heuristics*, or informed guesses, which an algorithm can use to restrict search to more promising regions and entirely avoid exploring regions that can provably not be part of a satisfying solution. An example of a heuristic is the use of Euclidean distance in

the road navigation domain. The straight-line distance between two points roughly approximates the distance of a shortest path between two points on a map, even if the true optimal path is less direct.

Another search method, specific to pathfinding problems, is *bidirectional search*. A unidirectional search explores outwards from the initial state and tries to find a path to a goal state. A bidirectional search, however, does simultaneous searches from the initial and goal states, looking for solution paths that are the concatenation of paths generated in the forward and reverse directions. A bidirectional search has the potential to find a solution and stop faster than a unidirectional search, thereby avoid exploring much of the problem-space graph.

The idea of combining these two techniques to form a bidirectional heuristic search was introduced as early as 1971 with the Bidirectional Heuristic Path Algorithm (BHPA) [Poh71]. Like many of the algorithms that were introduced since then, BHPA is a *front-to-end* bidirectional heuristic search. Front-to-end algorithms, discussed more thoroughly in Chapter 3, use the same heuristic evaluation function that would be used in a unidirectional heuristic search, and are the most straightforward implementation of a bidirectional heuristic search. Some bidirectional heuristic algorithms use *front-to-front* heuristic evaluations to increase the accuracy of the heuristic over what would be obtained in a unidirectional search. Bidirectional algorithms using front-to-front heuristic evaluations are the state-of-the-art in some domains, such as the 24 puzzle [FMS10].

Intuitively, one would expect front-to-end bidirectional heuristic search to be an effective search technique. It combines two independent and effective techniques for reducing search complexity: heuristics and bidirectional search. It is also the most straightforward way of combining these techniques. Despite the introduction of BHPA and several others since, however, front-to-end bidirectional heuristic search has not proven to be very effective and is rarely used. While a 1997 result [KK97] proved the ineffectiveness of the algorithm BHPA, the question of

why front-to-end bidirectional heuristic search as a general approach is ineffective has remained an open problem.

1.1 Pathfinding Problems

The goal in a pathfinding problem is to find a path—often of lowest cost—between an *initial* node and a *goal* node in a problem-space graph. The problem-space graph models *states*, which are legal configurations of the problem, and *operators*, which transform one state into another. A path in the graph corresponds to a sequence of operators that transform the initial state into a goal state.

Formally, a problem-space graph for a pathfinding problem consists of *states* connected by *operators*. Nodes in the graph represent states while edges connecting nodes are operators that transform one state into another. An operator between two states is reversible if applying the operator to a state results in a different state and applying it in reverse to the resultant state yields the original state. If an operator is reversible then the edge representing it is undirected. If not, the operator is a directed edge from the state to which it is applied to the state that results. Each edge has an associated *cost* of applying that operator.

An instance of a pathfinding problem defines an initial state and a set of one or more goal states. A pathfinding algorithm then tries to find a path between the initial state and a goal state. The cost of the path is defined as a function of the costs of its constituent edges. In most cases, the cost of a path is the sum of its edge costs, but some domains use different functions, such as the maximum cost of any edge on the path [DK07]. Often, the goal of a pathfinding problem is to find an optimal solution, which is usually defined as a minimum cost path from the initial state to a goal state.

A simple problem that can be represented as a pathfinding problem is the road navigation problem, which is the problem of finding a route between two locations

on a map. The problem space represents the road network as a graph: nodes in the problem-space graph are intersections on a map that a road can occupy, and edges are road segments connecting two adjacent intersections. A path between two nodes in this graph corresponds to a route that a road can take between the two corresponding locations in the actual road network. The cost associated with each edge might be the time taken to traverse the corresponding road segment when traveling at the speed limit. The cost of a path is the sum of these costs, and thus a lowest-cost path between two nodes is a fastest route between two locations on the road network.

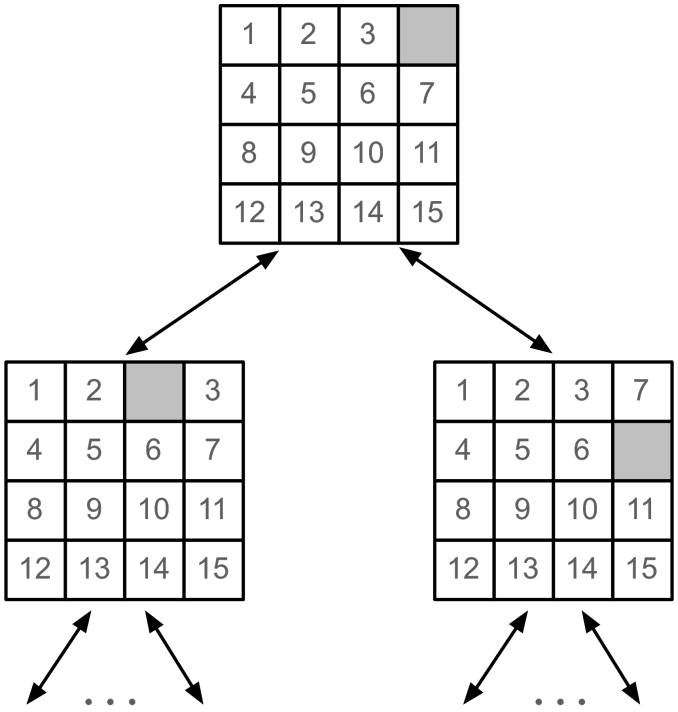

Figure 1.1: Part of the problem-space graph for the 15-puzzle.

Less obviously, puzzle games such as the 15 puzzle can be represented as pathfinding problems. The 15 puzzle is played on a 4×4 grid. Each position on the grid is occupied by tiles numbered from 1 to 15, with one location left empty, called the *blank*. A legal move is to move any tile adjacent to the blank into the blank position, after which the blank is in the position previously occupied

by the moved tile. The goal is to arrange the tiles in ascending numerical order from left to right, top to bottom.

In this kind of problem, nodes in the problem-space graph represent legal configurations of the puzzle, while edges are moves that transform one configuration into another. The goal is then to find a path that connects the initial node (corresponding to a random initial puzzle configuration) to the goal node (the solved puzzle configuration). The sequence of edges traversed on this path in the problem-space graph corresponds to a sequence of moves that would transform the initial randomized puzzle into the goal configuration.

Figure 1.1 shows a small portion of the problem-space graph of the 15 puzzle. The three states shown are unique configurations of the tiles on the board. The double-ended arrows between them are operators that transform one state into another, corresponding to sliding a tile into the adjacent blank position.

1.2 Dissertation Overview

This dissertation is an exploration of front-to-end bidirectional heuristic search. Its main contribution is a theory of why front-to-end bidirectional heuristic search will rarely be effective. This theory is an intuitive explanation which shows that bidirectional search and heuristics in general prevent a pathfinding algorithm from exploring the same regions of the problem-space graph. Thus, combining the two techniques provides no improvement over using just the stronger of the two techniques on a given problem domain.

In addition, I provide an example of a graph in which front-to-end bidirectional heuristic search *is* effective, and show that it can produce an arbitrarily large reduction in search over both unidirectional heuristic search and bidirectional brute-force search done independently. I also show that there exist real-world examples of these pathological cases in instances of the road navigation domain. This shows

that there cannot be a proof that front-to-end bidirectional heuristic search will *never* be effective. An informal theory showing that bidirectional heuristic search will rarely be effective is the strongest general claim that can be made.

I support this theory with extensive evidence from all of the main pathfinding domains studied in the heuristic search literature. This data supports the claims of my theory, demonstrating that it successfully shows why front-to-end bidirectional heuristic search is not usually effective in most search domains.

This data also shows that my theory predicts when unidirectional heuristic search or bidirectional brute-force search will be more effective, as well as when pathological cases may allow a bidirectional heuristic search to outperform either one. I empirically show that even though these pathological cases exist in practice, the overall improvement in performance they provide is small.

In my experiments, I tested a commonly-repeated claim of why front-to-end bidirectional heuristic search is ineffective in practice: that it tends to find an optimal solution very early in search and spend most of its time proving that solution's optimality. My experiments refute this claim, and show that with strong heuristics an optimal solution actually tends to be found quite late in the search. This reinforces the fact that previous explanations for the ineffectiveness of bidirectional heuristic search are incomplete.

My dissertation also presents novel state-of-the-art solvers for two search domains: peg solitaire and the four-peg Towers of Hanoi with arbitrary initial and goal states. Given the weakness of the heuristic used in the Towers of Hanoi, my theory predicts that a bidirectional brute-force search will outperform a unidirectional heuristic search. Indeed, I show that a bidirectional brute-force search is the state-of-the art solver in this domain.

My solver for peg solitaire is a front-to-end bidirectional heuristic search. While my theory predicts that front-to-end bidirectional heuristic search is almost never

effective, peg solitaire demonstrates some caveats to this theory. These caveats combine to make it a search space where front-to-end bidirectional heuristic search can improve over both unidirectional heuristic search and bidirectional brute-force search.

CHAPTER 2

Heuristic Search In Pathfinding

Two example domains discussed in this chapter are the 15 and 24 puzzles, which are variants of the well-known sliding tile puzzles invented by Sam Loyd in the 1870s [Loy59]. The puzzles are played on an $n \times n$ grid. Each position on the grid is occupied by tiles numbered from 1 to $n^2 - 1$, with one location left empty, called the *blank*. A legal move is to move any tile adjacent to the blank into the blank position, after which the blank is in the position previously occupied by the moved tile. The goal is to rearrange the tiles from a scrambled initial configuration into a goal configuration where the tiles are in ascending order from left to right, top to bottom. Figure 2.1 shows the goal positions in both puzzles.

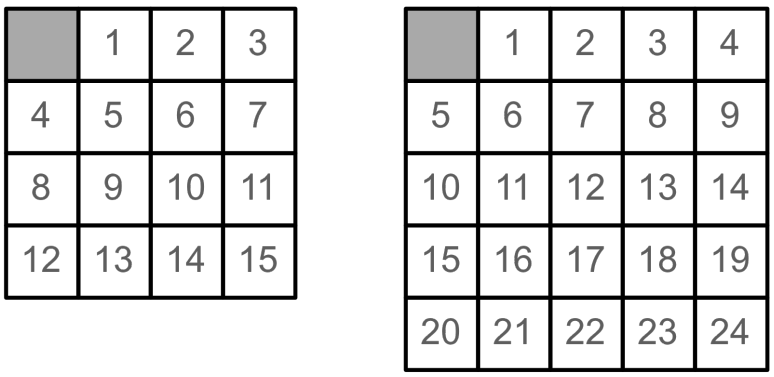

Figure 2.1: Goal positions of the 15 and 24 puzzles.

It is very easy to define problem spaces for domains such as those I consider in this dissertation. The size of the problem spaces described, however, can be extremely large. For example, there are 25!/2 unique reachable configurations of the 24 puzzle, equivalent to approximately $7.75 * 10^{24}$ states in the problem space.

An algorithm that explores the entire problem space to solve a problem would take a prohibitively long time to solve it. The challenge of search algorithms is thus to reduce the portion of the problem space that must be explored in solving a problem.

The simplest algorithms for finding a path through a problem space are *brute-force* searches. A brute-force search does a systematic exploration of the problem space, without using any domain-specific knowledge to guide search, until a path to a goal is found. Two simple and well-known brute-force searches are breadth-first search and depth-first search. A brute-force algorithm to find *any* solution path can terminate as soon as it finds the first path between an initial and a goal state. To find an *optimal* solution, however, a brute-force search must find an optimal solution and prove that no solution of lesser cost exists. To do so, it must explore every possible path from the initial state whose cost is lower than the optimal solution cost, and verify that none of them lead to a goal with a cheaper cost than the best solution found so far.

A heuristic search uses domain specific knowledge to explore less of the problem space than a brute-force search when finding optimal solutions. It uses a heuristic—an estimate of solution cost—to prevent an algorithm from exploring portions of the search space that provably cannot be part of an optimal solutions. Some algorithms, such as A* [HNR68], use the heuristic to queue nodes for expansion after an optimal solution has been found, if they provably cannot be part of an optimal solution. Others, such as IDA* [Kor85], use the heuristic to explicitly prune nodes that cannot be part of an optimal solution. In either case, provably suboptimal nodes are not explored. By doing so these algorithms can solve significantly larger problems than can be addressed by a pure brute-force approach.

2.1 Fundamental Concepts

Because the problem spaces addressed in many search problems are far too large to fit entirely in memory, search algorithms generally do not operate on an explicitly represented graph. Rather, they operate on an *implicit* graph, and construct portions of the graph as they are explored. The act of representing a node of the search graph in memory is called *generating* a node. *Expanding* a node means to generate all of the children of a node. The performance of different search algorithms is often accounted for by the number of nodes expanded or generated, which is independent of the performance characteristics of the computer they are run on.

I use the term *state* to refer to the abstract concept of a problem configuration in a problem space. Each state is unique. A *node* refers to a representation of a state as discovered during search. In general, it is possible to reach a state in a problem-space graph via different paths. For example, if two moves in a puzzle can be made in either order and result in the same puzzle configuration, the corresponding state in the problem space graph can be reached through two different paths. The two different nodes generated at the end of each path correspond to the same state in the problem space. When two or more nodes are generated in a search that correspond to the same state, we call these *duplicate* nodes. If a search algorithm generates a duplicate node without detecting that it is a duplicate, it may do redundant work searching past that node.

When a node is generated, some path to that node has been found from the initial state. The cost of a path to a node is called the g cost, and the function $g(n)$ gives the g cost of a path to a node n.

An optimal solution in a pathfinding problem is a path to a goal with the lowest-possible g cost. The g cost of an optimal path is referred to as C^*.

While the g cost gives the actual cost of a path *to* a node from the initial state, a heuristic function gives an estimate of an optimal cost of a path *from* a node to a goal. A heuristic function of a node n is denoted $h(n)$, and returns the h cost of a node. If a heuristic function never overestimates the cost of the actual lowest-cost path to a goal node—that is, it always returns a lower bound on the optimal path cost to a goal node—we call it an *admissible* heuristic [Pea84].

A simple example of an admissible heuristic comes from the road navigation domain. If we are trying to find the shortest-distance route to a certain location, the Euclidean distance metric provides a lower bound on the optimal cost of the remainder of a solution. The shortest-possible path from a given node to the goal is a straight line. Any drivable path to the goal must be at least as long as the straight line path, and so the euclidean distance is a lower bound on the remaining solution cost from a node. Since it never overestimates this cost, it is an admissible heuristic.

Finally, we define the f cost of a node. Given a node n we define $f(n) = g(n) + h(n)$. $f(n)$ gives the total cost of a node n. With an admissible heuristic, $f(n)$ never overestimates the cost of a complete path to a goal that starts with the current path from the start to n. If $f(n)$ is greater than the optimal solution cost, then the current path from the start to n provably cannot be part of an optimal solution. All admissible heuristic search algorithms use this property to avoid generating nodes with $f(n) > C^*$ when finding an optimal solution.

It is important to understand that, in a complete exploration of a problem space, the h cost of an expanded node is independent of its g cost. The g cost of a node n is the cost of the current path from the initial state to n, while the h cost is the estimate of the remaining cost from n to a goal. The optimal path from n to the goal is independent of the path taken to reach n so far, and the path taken to reach n is independent of the remainder of the path taken from n to a goal. In a heuristic search, g and h costs will be inversely correlated because nodes with

high f costs will not be explored. In a brute-force search which does no pruning, however, the values are independent.

2.2 Consistent Heuristics

An admissible heuristic never overestimates the cost of an optimal path from a node to a goal. A *consistent* heuristic makes a stronger guarantee. If $c(m, n)$ is the cost of the cheapest path from m to n, then a heuristic is consistent if, for all m and n, $h(m) <= c(m, n) + h(n)$ [Pea84]. In other words, the heuristic estimate from m to a goal cannot be greater than the sum of the heuristic estimate from n to the goal plus the cost to reach n from m. This can be seen as a form of the triangle inequality, with $h(m)$, $c(m, n)$, and $h(n)$ the lengths of three different sides of a triangle.

A consistent heuristic is also sometimes referred to as a *monotone* heuristic. This is because the f cost of the children of any node n must be at least as large as the f cost of n. The f costs of nodes on any path from the initial state to a goal are thus monotonically non-decreasing.

A heuristic that is consistent must be admissible, however a heuristic can be admissible and not consistent. While many natural heuristics are consistent, work in the literature has shown that searching with inconsistent heuristics can be effective in practice [ZSH09].

Consistent heuristics have a particular property that is relevant to this dissertation. Many algorithms, such as A*, perform search by expanding nodes in non-decreasing order of f cost. When these kinds of searches are done with a consistent heuristic, the first time a node is expanded it is guaranteed to be expanded with its lowest-possible f cost. In other words, the first time a node is expanded we can guarantee that we have found the lowest-cost path from the initial state to that node. This property allows us to implement several useful improvements

when implementing bidirectional heuristic search with a consistent heuristic. This is discussed further in Chapter 3.

2.3 Breadth-First Heuristic Search

A simple heuristic search algorithm that forms the basis for much of my work in this dissertation is Breadth-First Heuristic Search (BFHS) [ZH06]. BFHS performs a breadth-first search of the problem space, starting at the initial state. While searching, BFHS maintains a global cost cutoff. When BFHS generates a node, it computes that node's f cost and compares it to the cost cutoff. If the f cost exceeds the cutoff, the node is pruned and not stored or considered again during the remainder of that iteration of the search.

The cost cutoff can be initialized to the optimal solution cost C^* if this is known in advance. For example, in the 4-peg Towers of Hanoi puzzle the Frame-Stewart Algorithm [Fra41, Ste41] gives a conjectured-optimal solution, but the algorithm's optimality has not been proven. Finding a provably-optimal solution still requires a search of the problem space. In this domain, the BFHS cutoff can be initialized to the value returned by the Frame-Stewart Algorithm. Any node whose f cost exceeds this cutoff, and the only paths found to a goal are those with f cost less than or equal to this initial cost cutoff. Such a search would find the solution path returned by the Frame-Stewart Algorithm, as well as any cheaper ones that might exist.

In general, however, the solution cost is not known in advance. In these cases, BFHS can initialize its cost cutoff to the cost of some possibly-suboptimal solution found via a non-optimal algorithm. As BFHS is a breadth-first algorithm, it generates paths in increasing order of g cost. As such, the first solution it finds will be of optimal cost. However, it may expand a large number of nodes with $f(n) > C^*$ in doing so, performing unnecessary work. An alternative approach to

searching when the optimal solution cost is not known in advance is discussed in the following subsection.

2.3.1 BFIDA*

Breadth-First Iterative-Deepening A* (BFIDA*) is an iterative algorithm that builds on top of BFHS to find the optimal cost cutoff [ZH06]. It performs a series of BFHS searches with successively larger cutoffs. The first search is done with a cutoff that is provably a lower bound on the optimal solution cost. With an admissible heuristic this can be done by initializing the cutoff to the h cost of the initial state. If a solution is found on the first BFHS iteration, it is optimal. If not, the cutoff is incremented by setting it to the lowest f cost among nodes that were generated but pruned in the completed iteration, and BFHS is repeated. This continues until a solution is found. The first solution found must be of minimal cost, since a solution has not been found when searching with all possible lower cost cutoffs.

In most combinatorial domains, the number of nodes generated increases exponentially with the cutoff used during search [Kor85]. This means that the majority of work done in a BFIDA* search is done during the last iteration. Thus, even though BFIDA* does duplicate work by exploring the same portion of the search space in successive searches, it does not generate significantly more nodes than a single BFHS search with cost cutoff initialized to C^*.

2.4 Disk-Based Search

In problem-space graphs with large numbers of cycles, a search algorithm can generate very large numbers of duplicate nodes. If these duplicate nodes are not detected, then the algorithm will perform significant amounts of duplicate work by repeatedly regenerating the search tree beneath each duplicate node.

Many search algorithms such as A* prevent the regeneration of duplicate nodes by recording the nodes generated during search. Each time a node is generated, it is looked up in the set of nodes already generated and discarded if present. This allows the algorithm to avoid reexpanding nodes, but requires a large amount of memory to store the set of nodes generated. If a problem space is particularly large then this set will be too large to fit into the computer's memory.

Disk-based search avoids this problem by using magnetic disk space to store the set of generated nodes. The obvious way to store this set in memory is using hash tables, which are randomly accessed. Magnetic disks, however, have very poor random-access speed but very high sequential throughput. An effective disk-based search does duplicate detection without requiring random access to the set of previously-generated nodes.

The framework I use for doing disk-based search in my algorithms is Delayed Duplicate Detection (DDD) [Kor08]. DDD is based on a breadth-first search of the problem space, but no duplicate detection is done until an entire level of the breadth-first search has been completed. When generating a breadth-first level, each generated node is written to one or more files on disk. Once an entire level of a breadth-first search is generated, duplicate detection of all of the nodes on that level is done in an efficient manner.

The most straightforward way to do efficient duplicate detection on disk is by storing each entire breadth-first level in a single file. The algorithm sorts the nodes in the file using a disk-efficient sorting technique, such as merge sort. After sorting, duplicate nodes occur next to each other in the file. It then does a simple linear scan of the file, writing out each unique node to a separate file. This new file contains all of the unique nodes on the new level, each of which must be expanded when generating the next level of the breadth-first search.

There are more efficient and complicated methods for duplicate detection, based on partitioning the nodes on a level based on some feature of the state rep-

resentation. Hash tables are then used to do duplicate detection within the nodes of a partition. The details of this approach are discussed thoroughly in [Kor08].

As BFHS and BFIDA* are breadth-first algorithms, they are very easy to implement using DDD. These approaches form the basis of my Towers of Hanoi and peg solitaire solvers.

2.5 Pattern Databases

A pattern database [CS96] is a general technique for generating heuristics in many domains. A pattern database tracks a partial representation of a problem state and stores the optimal cost of solving any given configuration of that partial representation, ignoring constraints placed upon it by the remainder of the state.

For example, a pattern database in Rubik's Cube might record the location and orientation of all of the corner cubies. Given a particular configuration of corner cubies, the pattern database would store the minimum number of moves required to put those cubies into the correct location and orientation for the goal state. This ignores the effect of these moves on the remaining cubies: they are not required to end up in their goal location or orientation.

Another example comes from the sliding tile puzzles. A pattern database in this domain tracks the location of a specific set of tiles. It records the minimum number of moves required to move those tiles into their goal location. The location of the remaining tiles is not considered, and moves taken to move untracked tiles are not counted.

The cost of solving a partial component of a state is clearly a lower bound on the cost of solving the entire state. Any path from a state that solves the entire state must solve all components of the state as well. Thus, the cost of solving any component is a lower bound on the overall solution cost and is an admissible heuristic.

In some cases, it is possible to construct multiple pattern databases for a problem and use the sum of the values returned by each as an admissible heuristic. These are called *additive* pattern databases. This can be done if these pattern databases track disjoint components of the problem, and only count moves made by tracked components in their stored values. In other words, the value they store for the number of moves to solve a partial component only counts moves that affect only that component, and ignores moves that transform other parts of the problem.

For example, figure 2.2 shows four pattern databases for the 24 puzzle. Each pattern database tracks a disjoint set of tiles. For a given configuration of tiles, these pattern databases return the number of moves required to move those tiles into their correct position in the goal state. If the number of moves returned for a given board configuration only counts moves made by tiles tracked by that pattern database, then these pattern databases are additive.

I use pattern database heuristics in the solvers for many of the domains I consider, where they are the state-of-the-art heuristics.

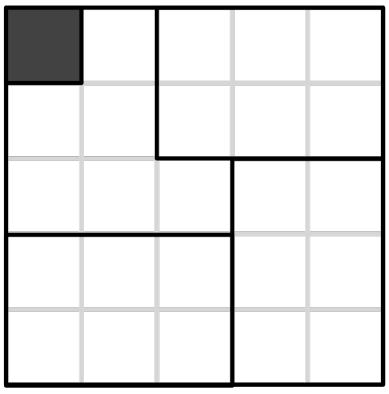

Figure 2.2: Tiles of the 24 puzzle partitioned into four pattern databases. The blank tile is not tracked.

CHAPTER 3

Front-To-End Bidirectional Heuristic Search

Most search algorithms are unidirectional, which means that they begin at the initial state and search towards the goal states. A bidirectional approach, by contrast, searches simultaneously both forward from the initial state and backwards from the goal states. When a node is generated in one direction that has already been generated in the opposite direction, then the concatenation of the paths found in both directions to that node is a solution path. By not doing a complete search in the forward or reverse direction, a bidirectional search has the potential to reduce the number of nodes expanded over a unidirectional search.

3.1 Bidirectional Brute-Force Search

The simplest bidirectional algorithm is one that does brute-force searches in both directions. A bidirectional search must have an explicit representation of the goal states to begin its reverse search. By contrast, a unidirectional search may only define a condition that needs to be satisfied for a state to be a goal, allowing us to test whether a state is a goal without having an explicit representation of a goal state. A solution in a bidirectional search is found when a node that is generated in one direction has already been generated in the previous direction: the concatenation of the two paths to reach the intersecting node from both directions is a path from the start to the goal. The simplest way to test whether this has happened is by keeping an explicit record of the states generated in each direction in memory, in a data structure that supports efficient lookups such as a hash table.

In fact, a generated node can be checked against a subset of the nodes generated in the opposite direction rather than all of them. The *frontier* of one direction of search is all the nodes that have been generated in that direction but not yet expanded. No path to a goal can pass through an expanded node without passing through at least one of its children as well. Thus, to see if the path to a newly-generated node can be concatenated with a path in the opposite direction to generate a solution, we need only test if that node is in the opposite direction's search frontier.

The reverse search of a bidirectional search is done backwards from the goal states towards the initial state, using the standard operators in reverse. Given an operator o that generates state t when applied to state s, the reverse of o is an operator o^r that yields s when applied to t. In some domains, the reverse of an operator is also a legal operator in that domain: the 15 puzzle operators shown in figure 1.1 can be applied in either direction. In other domains, an operator cannot legally be reversed: for example, in a road navigation problem a one-way road can only be driven in one direction. It is not necessary for operators in a domain to be legally reversible for us to do a reverse search, only that given a state, we must be able to determine the predecessors of that state in a forward search. It is sufficient for us to search using the reverse of operators that *are* legal in a forward search.

Once the two frontiers intersect, a solution has been found. The first solution found in a bidirectional brute-force search is not necessarily optimal [Hol10] and search must continue until a solution is proved optimal. The algorithm can stop once the sum of the minimum g costs on both frontiers is at least as large as the cost of the best solution found so far. g costs increase with depth and any solution found concatenates a path found in the forward and reverse directions. So, any solution found after this point must have cost at least as high as the sum of the minimum g costs in each direction, and thus the best solution found at that point is optimal.

A bidirectional brute force search has the potential to expand significantly fewer nodes than a unidirectional brute-force search. Consider a unidirectional brute-force search of an infinite tree where each edge in the tree has unit edge cost. If the average branching factor of a node is b in both directions and the depth of an optimal solution is d, then there are $O(b^{d-1})$ nodes shallower than the goal nodes. A unidirectional brute-force search of the problem space must thus explore $O(b^{d-1})$ nodes to prove that none of these shallower nodes are a goal state, and that the path of length d found to a goal is indeed optimal.

If the branching factor b is the same in both directions, then a bidirectional brute-force search will expand the same number of nodes in both directions to reach a given depth. If the search balances the number of nodes it expands in each direction, then the frontiers will meet at the midpoint, at depth $d/2$. At this point a solution is found and proved optimal. Both directions of the search expand $O(b^{d/2-1})$ nodes to reach the midpoint, and so the cost of the overall search is $O(2*b^{d/2-1})$. A bidirectional search thus expands approximately the square root of the nodes expanded in a unidirectional brute-force search.

3.2 Bidirectional Heuristic Search

A natural modification to this approach is to do heuristic searches in both directions rather than brute-force searches. The hope is that by combining bidirectional search with heuristic search, we can take advantage of the best characteristics of both with an algorithm that outperforms both of them.

The oldest bidirectional heuristic search algorithm is the Bidirectional Heuristic Path Algorithm (BHPA), introduced by Ira Pohl in 1971 [Poh71]. Instead of a brute-force search, BHPA performs the heuristic search A* in both directions. As in a bidirectional brute-force search, a solution is found when the two search

frontiers intersect but the first solution found is not necessarily optimal. BHPA terminates when it proves that the best solution found so far is optimal.

In Pohl's formulation of the algorithm, this happens when the minimum f cost of all of the nodes on one frontier is at least as high as the cost of the best solution found so far. With an admissible heuristic, the f cost of a node n is a lower bound on the cost of extending the current path to n to reach the goal. Any path from the initial state to a goal state must go through both search frontiers, so if the minimum f cost of all nodes on one frontier is at least that of the best solution found so far, that solution is optimal. Bidirectional heuristic search algorithms based on BHPA use this same termination condition.

The termination condition discussed previously for bidirectional brute-force search can also be used to prove solution optimality in a bidirectional heuristic search. However, this condition is not often mentioned in the literature. It may be that BHPA's termination dominates the brute-force termination condition in practice. In a heuristic search, nodes with high f cost are not expanded and so it is possible for nodes with low g and high h cost to remain unexpanded when there are no nodes with low f cost remaining on a search frontier.

In BHPA, as in A*, a node is selected for expansion and its children generated. If the expanded node has also been expanded in the opposite direction, then a path from the initial state to a goal state is found. However, the children of that node are still placed on an open list and will later be expanded, even though they have already been expanded in the opposite direction. As a result, once the two search frontiers intersect they continue searching through each other and the same nodes will be redundantly expanded in both directions. Figure 3.1 gives an idealized example of the effect of this in practice.

When both directions of a bidirectional heuristic search are not done in parallel, the algorithm must decide which node to expand next among both frontiers. Pohl proposed a cardinality principle, where the next node expanded is selected from

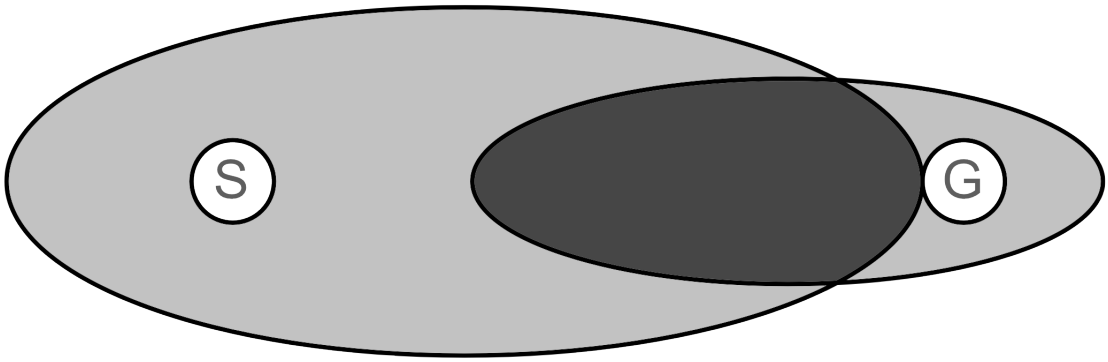

Figure 3.1: An idealized representation of two frontiers in a BHPA search passing through each other in a search from node S to node G. The light gray regions are nodes expanded in either a forward or reverse heuristic search, and the dark gray region are nodes expanded in both.

the smaller of the two search frontiers. The goal of this approach is to balance both directions of search so that they meet as early as possible in the middle of the search space.

Pohl's BHPA was the first published bidirectional heuristic search algorithm. While there were some initially promising results, by and large the algorithm remained unused until a 1997 paper by Kaindl and Kainz showed that its best case performance was only slightly better than the best case performance of unidirectional A* [KK97] (discussed later in section 4.1). In the meantime, several variations and improvements on this original idea were introduced.

3.3 Pruning Frontier Intersections

One of the earliest improvements in bidirectional heuristic search is the algorithm BS* [Kwa89]. Kwa observed that BHPA has the potential to do significant amounts of redundant work by allowing both directions of search to pass through the opposing frontier, re-expanding the same nodes. BS*, like BHPA, performs A* searches both forward from the initial state and backward from the goal states.

However, it incorporates a number of improvements that reduce its memory usage and prevent it from performing redundant work.

Recall that, with a consistent heuristic, when a node is expanded the lowest-cost path to that node has been found. If a node has been expanded in two directions, then, that means that the lowest-cost path to that node has been found both from the initial and the goal states. The optimal path found from the initial state to the goal states through that intersecting node is the concatenation of these two lowest-cost paths. Thus, when a node has been expanded in both directions an optimal path from initial to goal state through that node has been found. The children of a node that has intersected the opposing frontier thus do not need to be generated, and the two search frontiers do not pass through each other.

This is the primary contribution of BS* over BHPA. When searching with a consistent heuristic, this technique allows a bidirectional heuristic search to prevent the expansion of many nodes.

3.4 Front-To-End Heuristic Evaluation

BHPA and many of the bidirectional heuristic searches based on it use a *front-to-end* heuristic evaluation strategy. In a front-to-end heuristic evaluation, the h cost of a node is the heuristic evaluation of the optimal solution cost from that node to the goal states, or to the initial state in a reverse search. This is the exact same heuristic evaluation strategy used in a unidirectional heuristic search, and the simplest possible.

3.5 Alternative Heuristic-Evaluation Techniques

Since the introduction of BHPA, other algorithms have been introduced that do not use front-to-end heuristic evaluation. The most obvious alternative is a *front-to-front* heuristic evaluation [SC77]. In this strategy, the heuristic value of a node n to every node on the opposing search frontier is computed, plus the g cost of reaching those nodes from the goal (or the start, in the reverse direction). The heuristic value of n to the goal is thus given by $h_{min}(n)$, the minimum among all of those values. A path from the node to a goal node must past through the opposing search frontier and then reach the goal, and so $h_{min}(n)$ is an admissible lower bound on the cost of an optimal solution from n to a goal.

A front-to-front heuristic evaluation is more accurate than a front-to-end heuristic evaluation, but comes at the cost of a much more expensive heuristic evaluation. The overhead of calculating heuristic values to each node on the opposing frontier is quite high and as such the technique has not seen much use.

Other techniques have been introduced that use bidirectionality to increase the strength of the heuristic. Single-Frontier Bidirectional Search (SFBDS) [FMS10] breaks an overall pathfinding problem into subtasks of finding the shortest path between two nodes on opposing search frontiers. Heuristic evaluation is then done to the two nodes on the search frontiers, rather than to the initial or goal states.

Perimeter search [DN94, Man95] performs a limited form of front-to-front bidirectional heuristic search. A complete search is done in the reverse direction from the goal nodes to a fixed depth from the goals, and the perimeter of nodes generated at that depth is kept in memory. A forward heuristic search is done and the h cost of each node is the minimum of the heuristic estimates of that node to every node on the perimeter. This can be viewed as a limited form of a front-to-front bidirectional heuristic search, with only a small search to a fixed depth done in one direction.

KKAdd [KK97] uses a bidirectional search to increase heuristic accuracy without the overhead of a front-to-front heuristic evaluation. As with perimeter search, it does a reverse search to some depth from the goal nodes and stores the perimeter. For every node on the perimeter, it has the optimal solution cost from that node to a goal. For each node on the perimeter, its *heuristic error* is the difference between the actual optimal solution cost from that node and the heuristic function's estimate for that node. When searching with a consistent heuristic, the minimum heuristic error among all frontier nodes can be added to the heuristic function in the forward direction without losing admissibility. The proof that this is an admissible technique is in Kaindl and Kainz's paper. While not as accurate as a front-to-front heuristic evaluation, this technique has significantly less performance overhead.

My dissertation considers front-to-end heuristic evaluation only. While front-to-front algorithms have been shown to be effective in some domains, front-to-end algorithms have not. It is also the simplest formulation of a bidirectional heuristic search.

3.6 Memory Requirements of Front-To-End Bidirectional Heuristic Search

Unidirectional heuristic search algorithms have a wide range of memory requirements. Some algorithms, like A* [HNR68], store every node that they generate. Because every node that they generate is stored in memory, these algorithms can always detect when a newly-generated node is a duplicate of a previously-generated node. When such a duplicate node is detected, these algorithms can avoid doing redundant work by not expanding the same node twice. (In some cases, a duplicate node may need to be expanded a second time if, for example, it is found later with lower g cost). While these algorithms can avoid doing redun-

dant work, they are memory bound. They cannot practically solve problems that require the generation of more nodes than can fit in a machine's memory.

Other unidirectional algorithms, like IDA* [Kor85], only store in memory the nodes generated on a single path from the root to the most recently generated nodes. These algorithms are not memory bound. In principle, given enough time, they can solve any solvable pathfinding problem, so long as there is enough available memory to store the nodes on the longest path explored. However, without using additional memory, these algorithms cannot detect most duplicate nodes generated in a search and may do significant redundant work in finding a solution.

Bidirectional heuristic search algorithms have additional constraints beyond unidirectional algorithms: they must be able to detect when both directions of search have intersected. This limits the algorithms that can be implemented in either direction, and puts constraints on the memory requirements of a bidirectional heuristic search. In the two most well-known bidirectional heuristic search algorithms, BHPA [Poh71] and BS* [Kwa89], A* searches are done in either direction of search. In this case, frontier intersections can be easily tested because both frontiers are explicitly stored in memory. However, like A*, the maximum problem size solvable by these algorithms is limited by the amount of memory available.

Other bidirectional heuristic search approaches have been developed that are not memory bound. For example, Kaindl and Kainz introduced an algorithm which they call the *generic approach* to bidirectional heuristic search. The algorithm does a best-first search (such as A*) in one direction until memory is exhausted and then, unless the optimal solution has already been found, does a linear space search (such as IDA*) in the opposite direction. While these algorithms can in principle solve arbitrarily large problems without memory constraints, they are still limited in the work they can do in one direction by available memory. On increasingly

large problems, as a greater fraction of the work is done in the non-memory-limited direction, these algorithms will come to resemble a simple unidirectional linear-space search in that direction.

In specific domains, it may be possible in principle to test for intersections between two directions of a bidirectional search without explicitly representing either search frontier in memory. For example, Fiat et al. introduced an algorithm [FMS89] for solving permutation problems such as the Rubik's Cube puzzle bidirectionally, without storing either frontier in memory. In these domains, it is possible to sequentially generate all nodes at a given depth from the root node in lexicographic order. The bidirectional algorithm generates all nodes at a given depth from the initial and goal states in lexicographic order and scans these two levels of search in parallel as they are generated, looking for an intersection. This is an example of a bidirectional algorithm that does not need to store either search frontier in memory. However, it is only applicable to permutation problems.

In general, a front-to-end bidirectional heuristic search needs to store in memory the frontier of at least one direction of search. There are no general-purpose front-to-end bidirectional heuristic search algorithms that use as little memory as a linear-space unidirectional heuristic search algorithms.

Because of this, a front-to-end bidirectional heuristic search can only outperform a unidirectional heuristic search if it takes less time, expanding fewer nodes. On sufficiently large problems, a bidirectional heuristic search algorithm will be memory bound and unable to solve them due to insufficient available memory. A linear-space unidirectional heuristic search may be able to solve the same problems, however. It may take a very long time to solve them, but the linear-space algorithm will not run out of memory and will eventually terminate successfully.

The primary limitation on the effectiveness of front-to-end bidirectional heuristic search algorithms are their memory requirements. Given sufficient time, in problems without significant numbers of duplicate nodes, linear-space unidirec-

tional heuristic searches can solve larger problems than a bidirectional heuristic search, which will be limited by available memory. When the algorithms are not memory constrained, the relevant question is whether a front-to-end bidirectional heuristic search can solve problems faster than alternative algorithms.

3.7 Current State of Front-To-End Bidirectional Heuristic Search

Since its introduction in 1971, front-to-end bidirectional heuristic search has not seen much success as a search technique, even as related techniques have become the state-of-the-art in other domains. The two main algorithms BHPA and BS* are occasionally discussed in the literature, for example in [GH05, AK04]. However, no major results have been reported using these algorithms. As an example, [GH05] conducts experiments on road navigation using BS* and finds it to be not significantly better than other approaches considered.

The idea of front-to-end bidirectional heuristic search has repeatedly failed to be effective in practice, despite frequent discussion in the literature. However, there has not yet been a comprehensive discussion of why this promising technique has not proven effective in practice. Kaindl and Kainz's proof of the ineffectiveness of BHPA, discussed further in section 4.1, provides a partial explanation but does not generalize to other algorithms.

My dissertation presents the first comprehensive theory of why, despite 40 years of study, front-to-end bidirectional heuristic search has not met with success.

CHAPTER 4

Previous Analyses Of Bidirectional Heuristic Search

In the more than 40 years since its introduction, front-to-end bidirectional heuristic search has not been widely successful, despite many attempts to use it. The technique's ineffectiveness has been repeatedly observed in the literature but, while many enhancements and modifications have been proposed, the underlying question of *why* the technique does not work has remained largely unaddressed.

Two works in the literature have addressed this question directly. The first, a 1997 paper by Hermann Kaindl and Gerhard Kainz, provides a formal proof of the ineffectiveness of BHPA [KK97]. The proof is limited to BHPA, however, and leaves open the possibility that other front-to-end algorithms might be effective.

Stefan Schrödl and Stefan Edelkamp's 2011 textbook *Heuristic Search* [ES12] has a brief section that attempts to address the question more broadly. They present an intuitive theory to explain the ineffectiveness of all such algorithms. Unfortunately, their theory makes assumptions about which nodes are expanded in a heuristic search that turn out to be incorrect in practice. They support their theory with a very small amount of experimental evidence. Their core idea has some merit and was a useful inspiration in the formulation of my theory. However, their theory itself is not a sufficient explanation of the ineffectiveness of front-to-end bidirectional heuristic search in general.

4.1 Kaindl And Kainz 1997

Kaindl and Kainz [KK97] performed one of the first analyses of bidirectional heuristic search, focusing specifically on BHPA. The then-accepted explanation for its ineffectiveness—stated in Ira Pohl's original paper on the topic—was that the two frontiers of search pass by each other without intersecting. In this theory, the two directions of search penetrate very deeply towards their respective goals without intersecting the opposing search frontier. Until they intersect, a solution is not found. As a result, a lot of early work on bidirectional heuristic search focused on directing the two frontiers to encourage them to meet as early in search as possible. Figure 4.1, taken from Pohl's paper, graphically shows the theory of the two search frontiers passing without intersecting until very late.

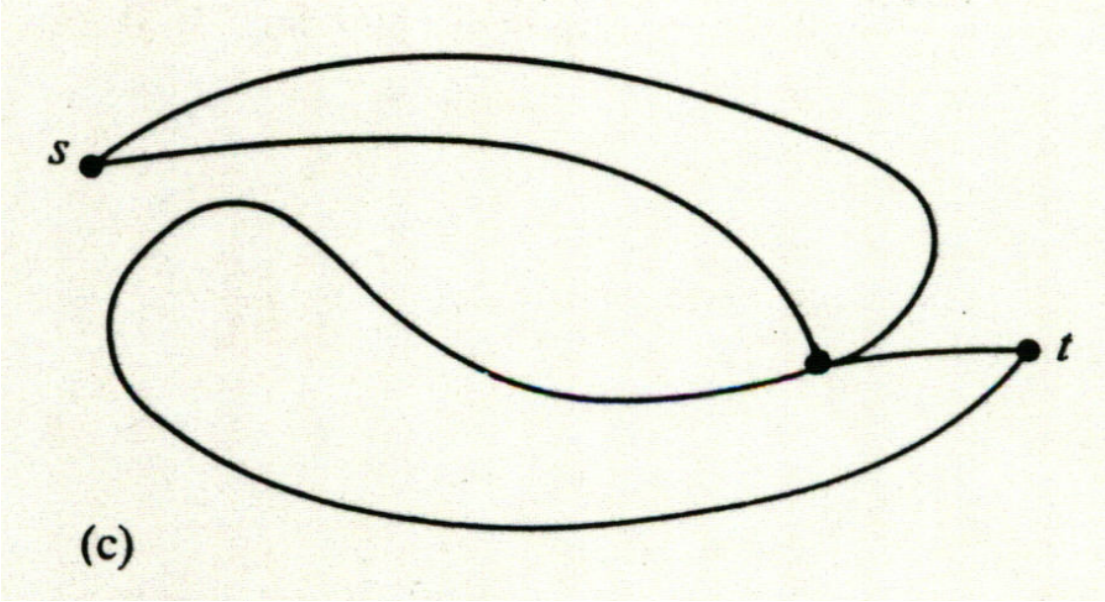

Figure 4.1: Figure from [Poh71] showing two search frontiers passing each other. s is the start node and t is the goal node.

Kaindl and Kainz's work showed that this theory is incorrect. The first and primary contribution of their paper was a proof that the best case of BHPA is identical to the best case of A*. A key point of this observation is that, while

the algorithm checks for frontier intersections to see if a solution has been found, both search frontiers in BHPA are allowed to pass through each other. As a result, the order of nodes expanded in one direction of a BHPA search is the same as an A* search in the same direction. A* terminates when a goal node is selected for expansion, at which point there are no nodes with f cost less than C^* on the open list. BHPA's termination condition is almost identical: it terminates when there are no nodes with f cost less than C^* on one of the open lists. BHPA does not need to select a goal node for expansion for this to occur, however, if it has already found a solution path before the termination condition is satisfied.

As a result, the best case for BHPA is identical to the best case for A* when done in the cheaper of the two directions.

In the average case, both algorithms may expand some number of nodes with f cost equal to C^*. BHPA may find an optimal solution well before its termination condition, while A* only finds its optimal solution immediately before terminating. If this happens, BHPA will expand no nodes with f cost equal to C^* while A* will. On the other hand, BHPA needs to pay the overhead cost of an additional reverse search while A* does not. As such, Kaindl and Kainz's proof only applies to the best case of both algorithms, and does not apply to the expected case.

This proof showed why BHPA specifically would not expect to do any better than a unidirectional heuristic search. However they explicitly do not consider the case of an algorithm like BS* that prevents the two search frontiers from passing through each other.

In addition, they conducted a number of experiments to test the conjecture that bidirectional heuristic searches tend to find their optimal solutions very late in search. In their experiments, they found that an optimal solutions actually tended to be found quite *early* in a search, and most search time was spent proving solution optimality. This fact has been cited a number of times as Kaindl and

Kainz's explanation for the ineffectiveness of bidirectional heuristic search [ES12, FMS10, LEF12], but it is not.

In sum, Kaindl and Kainz's paper showed that BHPA does not significantly outperform A* in the best case of both. It does not address the expected case of either algorithm, however, and explicitly does not consider the effectiveness of other, related algorithms.

4.2 Schrödl And Edelkamp 2011

In *Heuristic Search: Theory and Applications* [ES12], Edelkamp and Schrödl have a brief section proposing a more general theory for the ineffectiveness of bidirectional heuristic search. The central claim of their argument is that in a unidirectional heuristic search, the majority of the nodes are expanded at roughly the depth of the solution midpoint. We would expect both directions of search to meet near this solution midpoint and thus require twice the memory to store the open lists over a unidirectional heuristic search. While it contains some good intuitions, the overall claim of this argument is incorrect.

First, they note that the number of nodes expanded at each depth of a unidirectional heuristic search tends to increase up to a certain point and then decrease again until a goal is reached. This effect is shown in figure 4.2, taken from their textbook, which shows the distribution of nodes expanded at each depth on a single instance of the 15 puzzle. As they note, the number of nodes expanded tends to increase exponentially with depth near the initial state. They also observe that the number of nodes tends to *decrease* exponentially near the goal state. While this does happen in practice, their explanation—that the accuracy of the heuristic increases with proximity to a goal—is not correct. Rather, as I explain in Chapter 5, it is because the f cost of nodes tends to increase with depth and nodes with high f cost are not expanded in a heuristic search.

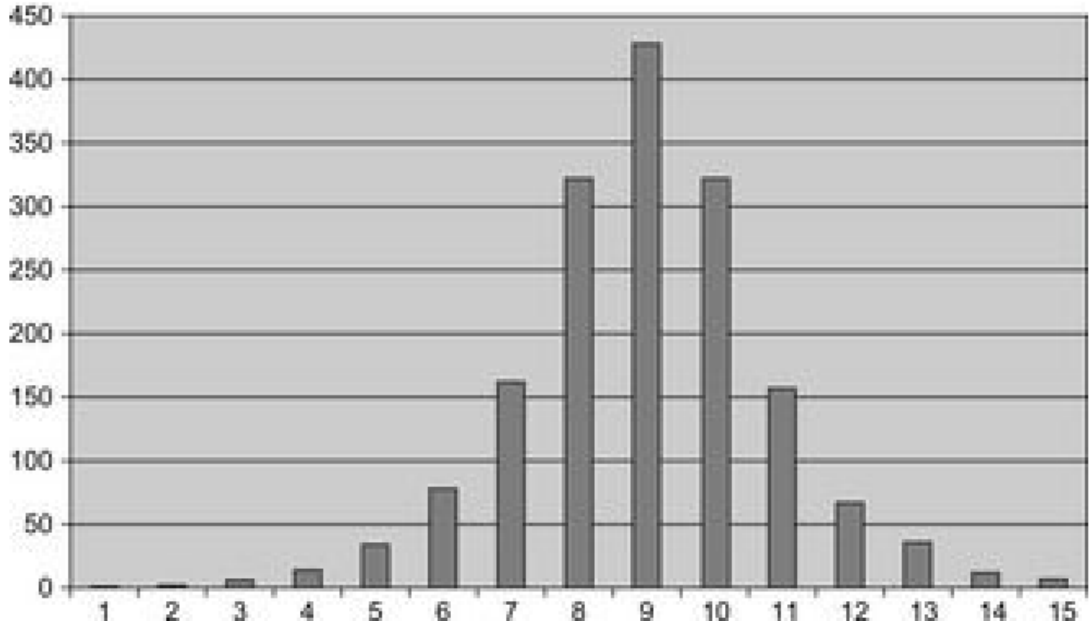

Figure 4.2: Distribution on the number of generated nodes in A* with respect to a given search depth. [Caption taken from *Heuristic Search*]

From this, they claim that we can expect the majority of nodes to be expanded near the solution midpoint in both directions of a bidirectional heuristic search. If both directions of search meet near this midpoint, then the size of the open lists in both directions will be approximately the same as the maximum size of the open list in a unidirectional heuristic search. A bidirectional heuristic search would thus use approximately twice as much memory.

This argument is incorrect, however. As I show in Chapter 6, the majority of nodes in a heuristic search are *not* expanded near the solution midpoint. Their claim appears to be based on data from a single, non-representative instance of the 15-puzzle. In my experiments with a much larger number of instances from many more domains, often the vast majority of nodes are expanded at a depth significantly less than the solution midpoint.

Furthermore, the size of the open lists at the point of intersection is irrelevant to the performance of bidirectional heuristic search. In fact, I show that perfor-

mance can be expected to be quite poor even if the number of nodes expanded at the depth of solution intersection is quite small. Regardless of where the majority of nodes are expanded in a bidirectional heuristic search, I show that it would not be expected to perform much better than the better of unidirectional heuristic search or bidirectional brute-force search performed independently.

As Edelkamp and Schrödl suspected, the distribution of depths of nodes expanded in a heuristic search is a useful tool for explaining the ineffectiveness of bidirectional heuristic search. This is a valuable contribution to my argument. The rest of their argument, however, is incorrect and I provide a correct explanation in Chapter 5.

CHAPTER 5

Ineffectiveness Of Front-To-End Bidirectional Heuristic Search

This chapter outlines my theory explaining the general ineffectiveness of front-to-end bidirectional heuristic search. Chapter 6 provides empirical data that supports this theory.

The main intuition of this theory is that, compared to a unidirectional brute force search, unidirectional heuristic search and bidirectional brute-force search improve performance in the same manner. Namely, they reduce the number of nodes expanded with high g cost. Since both techniques prevent the expansion of the same set of nodes from a unidirectional brute-force search, the performance improvement they provide is redundant. Depending on the strength of the heuristic in a domain, one of the two techniques will be more effective than the other. The improvements of that technique will dominate the improvements of the other, and so combining the two techniques provides no improvement over using the stronger of the two individually.

The structure of my argument consists of four steps. First, I show how a bidirectional brute-force search prevents the expansion of nodes with high g cost. Second, I show how unidirectional heuristic search also reduces the number of nodes expanded with high g cost. Third, I show that adding a weak heuristic to a bidirectional brute-force search to turn it into a bidirectional heuristic search does not reduce the number of nodes expanded. Finally, I show that with a strong

heuristic, adding a reverse search to a unidirectional heuristic search to make it a bidirectional heuristic search *increases* the number of nodes expanded.

My theory is not a proof but rather an intuitive explanation of the ineffectiveness of bidirectional heuristic search. In fact, in section 5.6, I demonstrate a pathological case where bidirectional heuristic search *does* outperform each of the other techniques. Section 6.8.3 gives examples of real problem instances where this occurs, as well, although the performance improvement is modest. These show that, in fact, no proof of the ineffectiveness of bidirectional heuristic search is possible without making more assumptions than my theory, or restricting it to only certain problem spaces.

In this theory, I talk about *strong* and *weak* heuristics. These are defined by the distribution of g costs among nodes expanded in a unidirectional heuristic search using that heuristic. With a strong heuristic, the majority of nodes expanded in a unidirectional heuristic search have g cost less than $C^*/2$. With a weak heuristic, the majority of nodes expanded have g cost *greater* than $C^*/2$.

When there is ambiguity, in this chapter I use $g_f(n)$ to refer to the g cost of a node when expanded in a forward search, and $g_r(n)$ for nodes expanded in a reverse search.

5.1 Nodes Expanded In A Bidirectional Brute-Force Search

In a bidirectional brute-force search, two brute force searches are done: one forward from the initial state towards the goal states and one in the reverse direction. The two searches meet somewhere between the initial and goal state and the search terminates once one direction of search has generated all nodes on the same level as the intersecting node [Hol10]. No nodes are expanded deeper than this point of intersection

If a bidirectional brute-force search expands approximately the same number of nodes with each g cost in both directions, we call it a search of a *balanced* search space. Approximately the same number of nodes will be expanded in either direction to reach a given g cost, and both directions of search intersect at approximately the solution midpoint with cost $C^*/2$. Nodes deeper than this in either direction will not be expanded, and so a balanced bidirectional brute-force search expands no nodes with $g(n) > C^*/2$.

So long as both directions of search are not very unbalanced, they will meet near the solution midpoint and few nodes will be expanded with $g(n) > C^*/2$. Section 5.7.3 discusses the case of a greatly imbalanced search.

Figure 5.1 shows this property graphically. It plots the number of nodes expanded at each g cost in unidirectional and bidirectional brute-force searches of the 16-disk, four-peg Towers of Hanoi puzzle. The instance shown is the standard instance, where all disks start and end on a single peg. The x axes show g cost in both directions. The y axis shows the number of nodes expanded with that g cost. The dashed lines are the distributions of g costs expanded in forward and reverse unidirectional brute-force searches. The dark gray nodes are those expanded in a bidirectional brute-force search. Most of the nodes expanded in the unidirectional searches occur past the solution midpoint, with $g(n) > C^*/2$. The vast majority of these nodes are not expanded in a bidirectional brute-force search.

5.2 Nodes Expanded in a Unidirectional Heuristic Search

The effect of a heuristic in a heuristic search is also to prevent the expansion of nodes with high g cost. A heuristic search improves over a brute-force search by preventing the expansion of nodes with higher f cost, which are also those nodes with higher g cost.

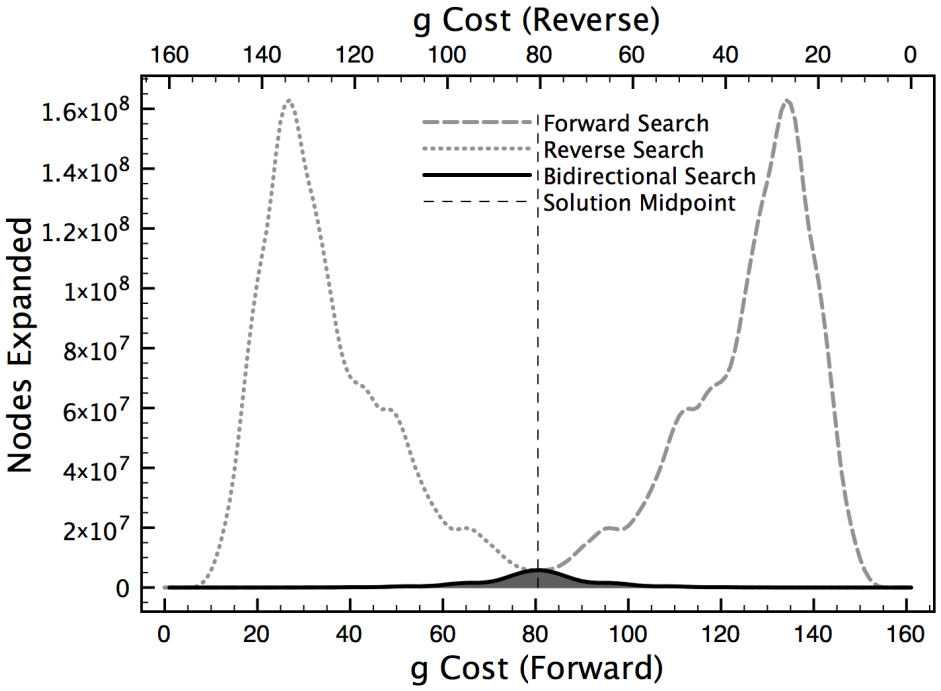

Figure 5.1: Unidirectional and bidirectional brute-force searches of the standard 16-disk Towers of Hanoi instance.

The f cost of a node is the sum of the g cost to reach that node and the h cost estimating the remaining cost from that node to a goal. When using an admissible heuristic, the h cost is a lower bound on the remaining cost of an optimal solution. Thus, the f cost of a node n is a lower bound on the total cost of extending the path to n to reach the goal.

If the f cost of a node n exceeds C^*, then the current path to n cannot be extended to reach the goal without exceeding the optimal solution cost. A heuristic search can thus prove that a solution is optimal without exploring beneath that node. Heuristic searches use this property to prevent the expansion of nodes. In algorithms like A*, nodes are expanded in non-decreasing order of f cost, so nodes with an f cost greater than the optimal solution cost C^* will not be expanded before the optimal solution has been found. Other algorithms, like IDA*, maintain a cost cutoff and explicitly prune nodes whose f cost exceeds this cutoff.

In either case, a heuristic search expands fewer nodes than a brute force search by preventing the expansion of nodes whose f cost exceeds C^*.

In complete brute force searches of a problem space, a node's g and h costs tend to be independent. For a given goal node, each node in the graph has a fixed h cost, which estimates its cost to the goal. An admissible heuristic never overestimates the cost of an optimal solution to a goal state, and so with a heuristic that returns non-zero values h costs tend to be lower in nodes closer to goal states. The h cost of a node is not a function of the cost of a path to that node from the initial state.

The g cost of a node, meanwhile, is a measure of the cost of a path to that node from the initial state, and is not a function of that node's distance from the goal. In a brute-force search from the initial state, g costs of nodes increase with depth. In paths that go closer toward the goal, h costs will tend to decrease while g costs increase. In paths that go further from the goal, h costs will tend to *increase* while g costs increase. Overall, the g cost of nodes does not predict the h cost of those nodes relative to the h cost of the initial state. This is not a formal theorem but, for example, this assumption forms the basis of an extremely accurate predictive theory of the number of nodes generated at each depth in an IDA* search [KRE01].

As g and h costs are independent, nodes with high g cost tend to have higher f costs. Nodes with higher f cost are those whose expansion is prevented by a heuristic search (as compared to a brute-force search). Thus, the nodes whose expansion is prevented by a heuristic search tend to have high g costs.

Figure 5.2 shows this graphically with increasingly strong heuristics on a Towers of Hanoi instance. Dashed lines are unidirectional searches with heuristics of different strengths. The algorithm used is BFHS with the cutoff set to C^*. I used increasingly large pattern databases to strengthen the heuristic. As expected, fewer nodes are expanded with successively stronger heuristics, and the nodes not

expanded are those with the highest g cost. The vast majority of those nodes whose expansion is prevented have $g(n) > C^*/2$.

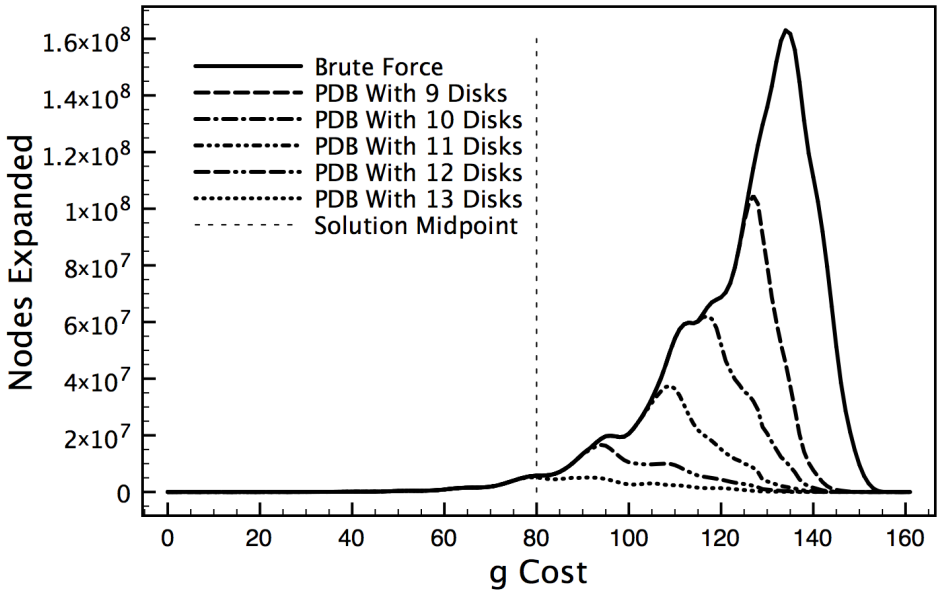

Figure 5.2: Unidirectional heuristic searches of the standard 16-disk Towers of Hanoi instance with heuristics of varying strengths.

5.3 Searching With Weak Heuristics

Adding a weak heuristic to a bidirectional brute-force search to produce a bidirectional heuristic search does not prevent the expansion of any additional nodes.

In a bidirectional brute-force search, both directions of search intersect when the same node is generated in both directions, with some g cost from the initial or goal state. No nodes in either direction are expanded with greater g cost than this. If this is a search of a balanced search space, then the two directions will meet at the solution midpoint and no nodes will be expanded with $g(n) > C^*/2$. Even if the two directions of the search space are somewhat imbalanced, both directions will intersect near the midpoint and no nodes with high g cost will be expanded.

Adding a heuristic to this bidirectional brute-force search presents the possibility of preventing the expansion of even more nodes. However, with a sufficiently weak heuristic, the only nodes whose expansion is prevented are those with $g(n) > C^*/2$. No such nodes are generated in a bidirectional brute-force search, as these would be generated deeper than the point at which the two search frontiers intersect. If the search space is somewhat imbalanced, a bidirectional brute-force search will only expand nodes with g cost a little higher than $C^*/2$. A sufficiently weak heuristic will not prevent the expansion of any of these nodes, either. Thus, any nodes whose expansion would be prevented by a heuristic would not even be generated in a bidirectional brute-force search.

Figure 5.3 graphically shows the effect of adding weak heuristics to a bidirectional brute-force search. The dark gray region plots the distribution of g costs of nodes expanded in a bidirectional brute-force search of a Towers of Hanoi instance. The dashed lines show unidirectional heuristic searches in both directions, using heuristics of varying strengths. The light gray regions are the forward searches. The nodes whose expansion is prevented by the addition of a heuristic occur past the point at which both directions of the bidirectional search intersect. Thus, they would not be expanded in a bidirectional brute-force search, either.

5.4 Searching With Strong Heuristics

With a strong heuristic, combining forward and reverse heuristic searches to make a bidirectional heuristic search expands *more* nodes than a unidirectional heuristic search.

With a sufficiently strong heuristic, the majority of the nodes expanded in a forward unidirectional heuristic search have $g_f(n) < C^*/2$, because the heuristic prevents the expansion of most nodes with $g_f(n) > C^*/2$. A reverse heuristic search will also have $g_r(n) < C^*/2$.

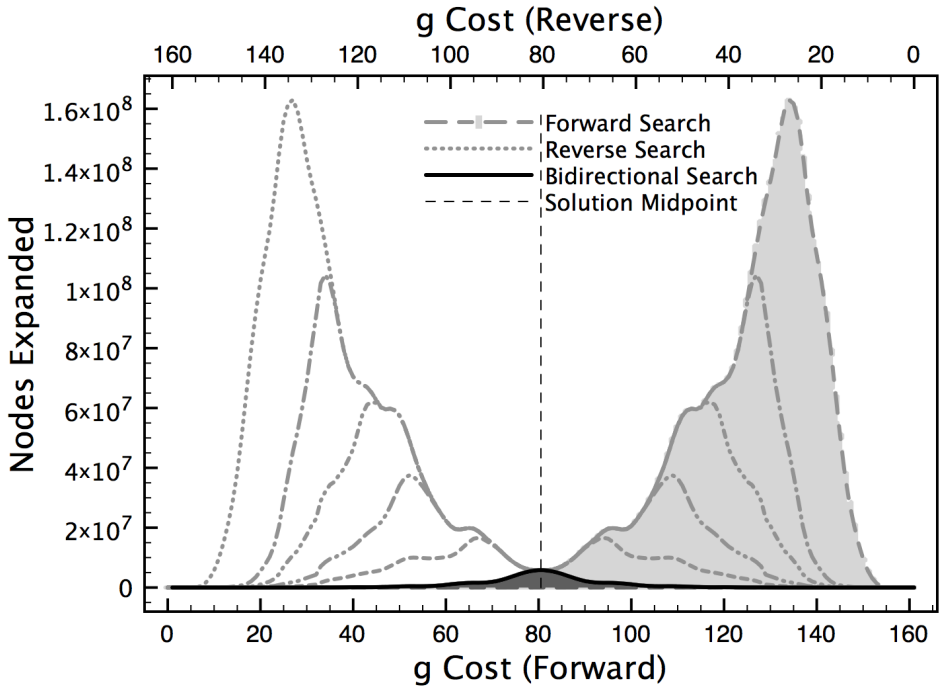

Figure 5.3: Bidirectional brute-force search of a Towers of Hanoi instance overlaid with unidirectional heuristic searches of increasing strengths.

If a node is expanded with $g_r(n) < C^*/2$ in a reverse heuristic search, then it can only be reached with $g_f(n) > C^*/2$ in a forward search. The converse is true of nodes expanded with $g_f(n) < C^*/2$ in a forward search. This is because, for a given node n, $g_f(n) + g_r(n) >= C^*$. If the majority of the nodes in a reverse search are expanded with $g_r(n) < C^*/2$, then most of these nodes would not be expanded in a forward search, which *also* expands most nodes with $g_f(n) < C^*/2$. Similarly, if the majority of nodes in a forward heuristic search are expanded with $g_f(n) < C^*/2$, then most of those nodes would not be expanded in a reverse search.

If these two searches are combined to form a bidirectional heuristic search, they will expand *more* nodes than a unidirectional heuristic search, because both directions expand some number of nodes that are not expanded in the opposite direction. The stronger the heuristic, the greater the fraction of nodes expanded

with $g_f(n) < C^*/2$ and $g_r(n) < C^*/2$, and the greater the additional work of a bidirectional heuristic search over a unidirectional heuristic search.

Figure 5.4 demonstrates this graphically. It shows the depth of nodes expanded by heuristic searches of a 24 puzzle instance using IDA* and the pattern databases described in [KF02] (these were shown previously in figure 2.2). The light gray nodes are expanded in a forward search and the dark gray nodes are expanded in a reverse search. The hashed region is expanded in both. A bidirectional heuristic search expands the nodes in both gray regions, while a unidirectional heuristic search expands only one gray region.

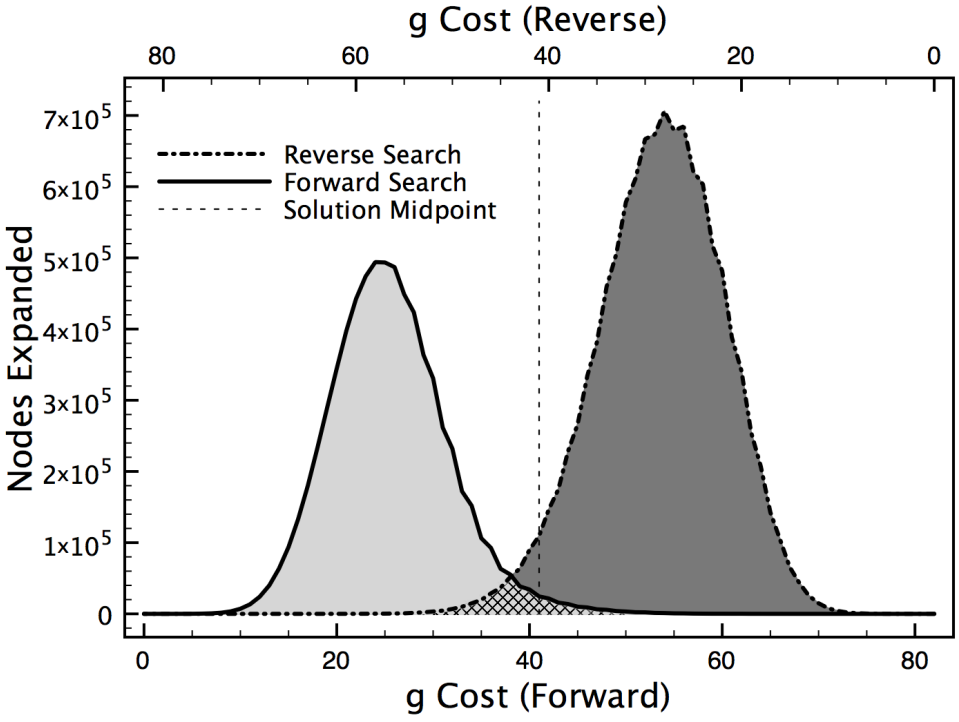

Figure 5.4: g costs of nodes expanded in forward and reverse searches on a 24 puzzle instance.

5.5 Searching With A Medium Strength Heuristic

For a given search instance, a heuristic may be exactly strong enough that approximately half of all nodes expanded in a forward unidirectional heuristic search have $g_f(n) < C^*/2$. In this case, a reverse heuristic search will also expand approximately half its nodes with $g_r(n) < C^*/2$. The forward search will thus expand as many nodes with $g_f(n) > C^*/2$ as the reverse will expand with $g_r(n) < C^*/2$.

In this case, if we do a bidirectional heuristic search with a breadth-first search ordering and balance the number of nodes expanded in each direction, both directions of search will meet near the solution midpoint. At this point, the forward direction will have expanded all nodes with $g_f(n) < C^*/2$ that would be expanded in a forward heuristic search, and the reverse direction will have expanded all nodes with $g_r(n) < C^*/2$ that would be expanded in a reverse heuristic search.

This search would prevent the expansion of approximately half of the nodes of a forward heuristic search—those with $g_f(n) > C^*/2$—while requiring the expansion of approximately the same number of additional nodes in the reverse direction—those with $g_r(n) < C^*/2$. In this case, a bidirectional heuristic search would expand approximately the same number of nodes as a unidirectional heuristic search.

A bidirectional heuristic search might not use a breadth-first search ordering, however. If so, then it is possible for each direction of search to expand some number of nodes with g cost greater than $C^*/2$ before it expands other nodes with lower g cost. In particular, it would be possible for the forward direction of a bidirectional heuristic search to expand nodes very close to a goal before it has expanded all of the nodes that a unidirectional heuristic search would expand (and vice versa).

Section 5.6 shows how this can result in pathological cases where a bidirectional heuristic search *can* outperform a unidirectional heuristic and bidirectional brute-

force search. Section 6.8.3 shows cases where this occurs in practice and provides bidirectional heuristic search with a slight performance benefit. These real-world examples occur in a domain with a medium-strength heuristic.

Figure 5.5 shows the distribution of g costs on forward and reverse heuristic searches of a road navigation problem. 52% of nodes expanded in the forward direction have $g_f(n) < C^*/2$ and 53% in the reverse direction have $g_r(n) < C^*/2$. Because the g costs in road navigation are real-valued, the data is given as a histogram with 50 buckets. Due to irregularities in the search graph, the distribution of g costs is very irregular and not symmetric. In this problem instance, both the initial and goal nodes are near urban areas, but the optimal path between them is a remote route with few branching paths. This explains the bimodal distribution of this graph.

Despite this irregularity, approximately half of the nodes expanded in each direction have $g(n) < C^*/2$. A bidirectional heuristic search with a breadth-first search order would meet near the solution midpoint, expanding approximately as many nodes as either direction of a unidirectional heuristic search.

5.6 A Pathological Counterexample

My theory does not prove that bidirectional heuristic search can never be effective, but rather just shows why that is unlikely. It is still possible for it to outperform both unidirectional heuristic and bidirectional brute-force search. This section gives a concrete example of how this can happen, and section 6.8.3 shows that these cases can occur in practice.

Figure 5.6 is a pathological graph where a bidirectional heuristic search outperforms both unidirectional heuristic and bidirectional brute-force search. In the forward direction, each node is labeled with its minimum g cost from the initial state and the h cost to a goal, as well as its f cost. The f, g, and h costs in the

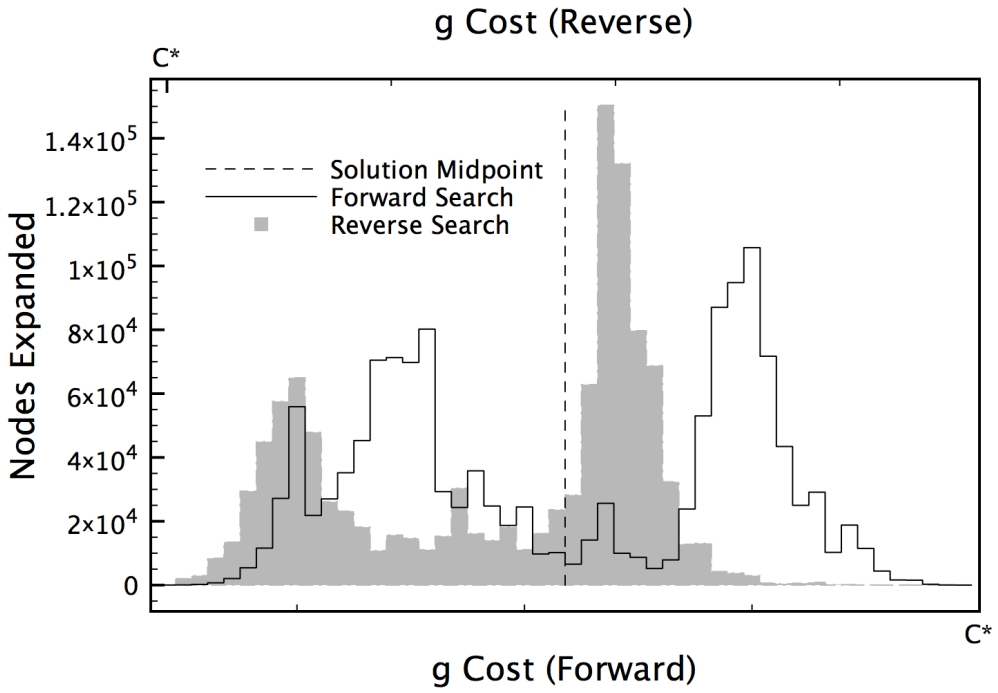

Figure 5.5: g costs of nodes expanded in forward and reverse heuristic search of a road navigation problem instance. Because g costs are real valued, data is given as a histogram with 50 buckets.

reverse direction are symmetrical. The graph has unit edge costs. The h cost of two sibling nodes never differs by more than one, so the heuristic is consistent.

Often, the only nodes that have an h cost of zero in a domain are the goal nodes. In this example, however, several other nodes have h cost of zero as well. It is possible to construct other examples where this is not the case, but this requires larger graphs.

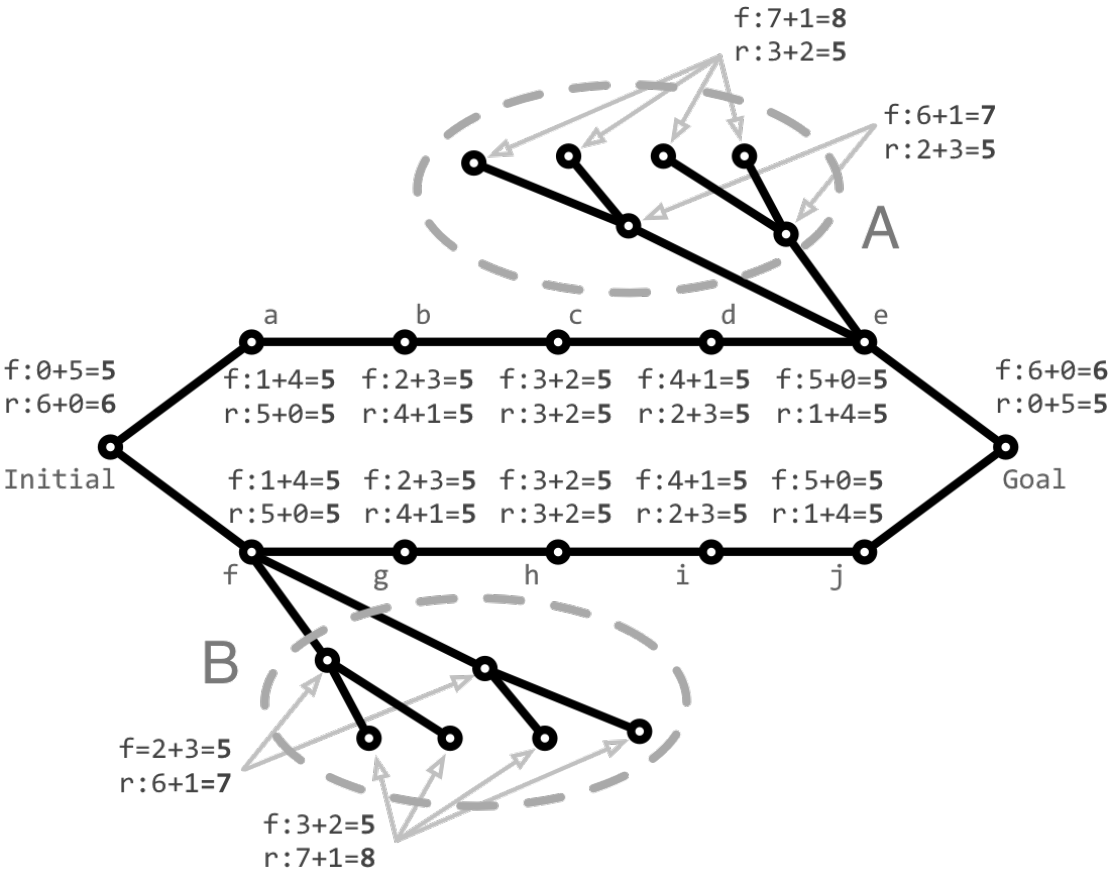

Figure 5.6: A pathological graph where bidirectional heuristic search outperforms both unidirectional heuristic and bidirectional brute-force search. Nodes are labeled with g-cost + h-cost = f-cost, in the forward (f) and reverse (r) direction.

The f and g costs of this graph are carefully constructed so that nodes in the subtrees labeled B are expanded in a forward heuristic search but not in the reverse. Nodes in the subtrees labeled A will be expanded in a reverse heuristic

search but not in the forward direction. A bidirectional brute-force search would expand at least some of the nodes in A and B. With the right ordering of search, bidirectional heuristic search expands neither. The children of the nodes in A and B follow a pattern of increasing g costs while decreasing h costs. By extending these subtrees using this pattern and increasing the depth of an optimal solution, we can construct graphs where bidirectional heuristic search has an arbitrarily large performance advantage over the other two algorithms.

A forward unidirectional heuristic search of this graph expands all nodes with f cost less than the optimal solution cost, which is six. There are 17 such nodes: nodes a through j, the nodes in B, and the start. A reverse heuristic search also expands only 17 nodes: nodes a through j, the nodes in A, and the goal.

A bidirectional brute-force search expands all nodes in both directions up to depth two in both directions. At this point, an intersection is generated at depth three. All nodes at depth two must be expanded to prove that there is no shorter path. A bidirectional brute-force search thus expands 14 nodes: nodes a, b, d, e, f, g, i, j, the roots of the subtrees in A and B, and the start and goal.

Bidirectional heuristic search can expand fewer nodes if it generates children in left-to-right order and breaks f cost ties in favor of lower h cost (which is a standard technique). Node e will be expanded in the forward direction before the reverse, and node f will be expanded in the reverse direction before the forward. The forward search will never expand the children of e because their f cost (seven) exceeds C^*. The same is true of the children of f in the reverse direction.

When node e is considered for expansion in the reverse direction, it can be discarded without expansion because it has already been expanded in the forward direction. This is possible because we are using a consistent heuristic. (See section 3.3). Similarly, the forward search will not expand the children of node f because f has already been expanded in the reverse direction.

As a result, the nodes in A and B will not be expanded in either direction. A bidirectional heuristic search will thus expand 12 nodes: nodes a through j, the start, and the goal. Thus, bidirectional heuristic search outperforms both unidirectional heuristic and bidirectional brute-force search on this graph.

The nodes in A are subtrees that follow a pattern in their h values: as $g_r(n)$ increases by one $h_r(n)$ *decreases* by one, and so the f_r costs of each node in this tree are equal, with $f_r(n) < C^*$. $h_f(n)$ *increases* by one as $g_f(n)$ increases by one, and so $f_f(n) > C^*$ for every node in A. If the optimal solution depth is deep enough, we can extend subtree A to be arbitrarily large while still requiring that all nodes in it must be expanded in a reverse heuristic search but not a forward heuristic search. Similarly, we can extend B into an arbitrarily large subtree whose nodes must be expanded in a forward heuristic search.

With a large enough optimal solution depth, a bidirectional brute force search can similarly be made to expand an arbitrarily large number of nodes in A and B. Meanwhile, a bidirectional heuristic search need not expand either A or B and can thus expand arbitrarily fewer nodes than either other algorithm. This counterexample shows that no formal proof of the ineffectiveness of bidirectional heuristic search is likely.

An intuitive example of how this might happen is a road navigation problem finding the shortest path between the initial and goal states. Subtrees A and B might represent cul-de-sacs or neighborhoods with no outlets. Subtree A would represent roads that go in the direction of the initial state but never reach it due to dead ends. In a reverse search, A would look like a promising route to the initial state as it moves in the general direction of the start state. In the forward search, however, A branches off from the optimal route and moves back towards the initial state. This would look like an unpromising route, as it moves away from the goal even as search gets very near to the goal. Similarly, B would look promising to a forward search but not a reverse search.

Such a pathological case is unlikely to occur often or provide much benefit in a real search problem. This case relies on both directions of search being able to search very close to their respective goal node before they have expanded many other nodes. For this to happen, there must be nodes deep in search with the same f cost as shallow nodes. However, in general, f costs of nodes in a search space increase with depth, as the g cost increases. This is particularly true with a consistent heuristic, which results in f costs that are monotonically non-decreasing with depth. As such, we would expect shallower regions to generally be expanded before deeper regions, and for neither direction of search to intersect the other before most nodes with lower depth have been expanded.

As such, I expect pathological cases like that of figure 5.6 to be unlikely in practice, and to provide little performance benefit when they do occur. Section 6.8.3 gives evidence to support this claim.

This particular pathological case relies on specific tie-breaking behavior for bidirectional heuristic search to outperform the other two algorithms. It is possible to construct graphs where a bidirectional heuristic search is the strongest algorithm without relying on this implementation detail, but they require longer optimal solution paths and are somewhat more complicated to describe. This example graph, however, shows that the best case of bidirectional heuristic search can be better than the best case of the other two algorithms.

5.7 Caveats

Here I identify some caveats to my theory.

5.7.1 Strengthening Heuristic Evaluation

My theory only considers front-to-end heuristic functions, where the heuristic used in a bidirectional search is identical to the heuristic used in a unidirectional search.

A bidirectional search can use bidirectionality to increase the accuracy of the heuristic function used. Several examples of how to do this using a front-to-front heuristic evaluation were previously discussed in section 3.4. By strengthening the heuristic, the two directions of a bidirectional heuristic search can be made to expand fewer nodes than in a unidirectional heuristic search, even when the two frontiers do not intersect.

A related example of how a bidirectional search can reduce the number of nodes expanded comes from my peg solitaire solver, discussed further in section 6.10.6. Peg solitaire has a number of techniques that can be used to prove that certain states can never be solved. Normally, one would test whether it is possible to solve a board configuration by seeing if it is possible to reach the goal state. In my solver, I instead test whether it is possible to reach any node on the opposing frontier, which may be some distance away from the goal state. This allows my solver to tighten pruning constraints, which could not be done in a unidirectional search.

5.7.2 Improved Tie-Breaking

Kaindl and Kainz's theorem showed that the best case of BHPA is only slightly better than the best case of A*. Both algorithms must expand essentially the same number of nodes with $f(n) < C^*$. Counting only nodes with $f(n) < C^*$, A* will always expand the same number of nodes, while BHPA's expected case can be worse than the best case.

However, this leaves open the possibility that BHPA (and other bidirectional heuristic search algorithms) may expand fewer nodes with f cost *equal to* C^*. In other words, BHPA must expand as many nodes as A* to *prove* that a solution is optimal, but it may *find* the optimal solution faster. In the best case, both algorithms expand the same number of nodes to prove that a solution is optimal,

but A* may expand many nodes that do not need to be expanded when finding that solution.

One way in which this might happen is if a unidirectional heuristic search has particularly bad tie-breaking properties. In other words, the algorithm may expand a very large number of nodes with $f(n) = C^*$. A bidirectional search may be able to find an optimal solution earlier, while still expanding at least as many nodes with $f(n) < C^*$.

By way of analogy, consider the algorithm IDA*. IDA* does repeated iterations of depth first searches with a cost cutoff. Any node whose f cost exceeds the cutoff of the current iteration is pruned and not considered. If an optimal solution is not found on a given iteration the cutoff is incremented and another iteration is performed. To prove solution optimality, IDA* must complete every iteration of search with cutoff less than C^*, to prove that no cheaper iteration exists. On the *last* iteration, however, IDA* can terminate as soon as it finds a solution, which it can guarantee has lowest cost.

The order of search of the last iteration of IDA* can have a huge impact on performance. One search ordering may require IDA* to explore every node whose f cost does not exceed the cutoff before finding a solution, while a different ordering may immediately march down an optimal path towards a solution. An ordering that reliably finds the optimal solution faster on the final iteration could result in large performance improvements, even though the number of nodes expanded on each previous iteration is identical.

A similar effect happens in my peg solitaire solver, as discussed further in section 6.10. Due to the size of the problem space size and the large number of duplicates, the obvious unidirectional algorithm for solving this problem is the algorithm BFIDA*. BFIDA* has very bad tie-breaking properties, expanding almost all nodes with f cost equal to C^*. My bidirectional algorithm, by contrast,

is able to find an optimal solution very early in search and has essentially perfect tie-breaking.

5.7.3 Unbalanced Directions Of Search

Finally, my theory assumes that forward and reverse searches will be roughly symmetrical, and that the distribution of g costs of nodes expanded in both directions will be similar. This may not be true in all domains, however. Due to irregularities in the search graph, a unidirectional heuristic search may expand significantly fewer nodes in one direction than another.

It is possible for a search in one direction to be significantly cheaper than the other. This may happen if, for example, the branching factor in that direction is significantly smaller than the other direction. If one direction of search is consistently significantly cheaper than the other then it makes sense to do a unidirectional heuristic search in that direction only, regardless of the strength of the heuristic. The cheaper direction may expand the majority of its nodes with $g(n) > C^*/2$, but this may be dwarfed by the number of nodes expanded in the opposite direction.

If the cheaper direction cannot be predicted in advance, however, a bidirectional heuristic search may be effective even with a very strong heuristic. This is because by keeping the number of nodes expanded in each direction balanced, a bidirectional search will search deeper in the cheaper of the two directions. It will thus find the majority of an optimal solution path in the cheaper of the two directions. Finding the majority of an optimal path in the more expensive direction would require expanding significantly more nodes.

This is analogous to doing simultaneous forward and reverse heuristic searches and stopping as soon as the first one completes. Some overhead is paid for doing a search in the direction that does not find a solution, but this prevents us from

doing a complete search in the more expensive of the two directions. If solving an instance in the harder direction reliably requires expanding at least twice as many nodes as the cheaper, this can speed up search.

Again, this caveat manifests itself in peg solitaire where, due to a large imbalance in branching factors, the reverse searches are significantly more expensive than forward searches.

CHAPTER 6

Empirical Studies Of Bidirectional Heuristic Search

I conducted several experiments to substantiate my theory with empirical data. I tested nine domains in my experiments: the 15 puzzle, the 24 puzzle, the pancake problem, peg solitaire, Rubik's Cube, Top Spin, the four-peg Towers of Hanoi, pairwise sequence alignment, and road navigation. I tested each domain with the state-of-the-art unidirectional heuristic solver in each, using the best available heuristics.

For each domain, I conducted a number of different experiments. In all domains I examined the distribution of g costs, computing the fraction of nodes expanded with $g(n) <= C^*/2$. I tested if this accurately predicts whether unidirectional heuristic or bidirectional brute-force search will be more effective on that domain.

For many of the solvers, I implemented bidirectional heuristic search to verify that, indeed, it is generally an ineffective search technique. In these domains, I identified a few problem instances where bidirectional heuristic search is effective, but show that the performance improvement is minimal, as I predict. The one exception is in peg solitaire, where bidirectional heuristic search reliably provides tie-breaking improvements. And finally, I conducted experiments to find how early in a bidirectional heuristic search an optimal solution tends to be found.

For two of these domains, four-peg Towers of Hanoi and peg solitaire, I developed new state of the art solvers using bidirectional search. The section for those domains describes the techniques used in those solvers and their performance.

Each domain and the results of the experiments I conducted on it are described in a separate section. Chapter 7 synthesizes the results of these experiments and makes overall observations.

6.1 Experiments Performed

6.1.1 g Cost Distributions Among Many Domains

My first set of experiments calculate the distribution of nodes generated with each g cost in many instances of each problem domain. I then computed the fraction of nodes expanded with $g(n) <= C^*/2$ in each search instance. For each domain, I recorded the mean of this fraction among all problem instances, as well as the minimum and maximum fraction of nodes expanded with $g(n) <= C^*/2$ on a given instance. I only consider nodes expanded with $f(n) < C^*$, as these are the nodes that must be expanded in any heuristic search. In my IDA* implementations, I only consider the last complete search iteration, to avoid multiple counting of nodes on different iterations.

My theory predicts that in algorithms where the majority of nodes are expanded with $g(n) <= C^*/2$, a unidirectional heuristic search will be the state of the art, and if the majority are expanded with $g(n) >= C^*/2$, then a bidirectional brute-force search will win.

6.1.2 Bidirectional Heuristic Search Tested

The absence of a published state-of-the-art bidirectional algorithm in the literature does not prove that one does not exist. As such, I implemented bidirectional

heuristic search in the five domains where it is feasible and not the state of the art: the 15 puzzle, pairwise sequence alignment, the pancake problem, Rubik's Cube, and road navigation. I compared the performance of my bidirectional solver to unidirectional heuristic search, which is the predicted state of the art in these domains. I also introduced a new bidirectional heuristic solver for peg solitaire, which is the new state of the art.

6.1.3 Work Spent Proving Optimality

Finally, in the domains where I implemented bidirectional heuristic search, I calculated the point at which optimal solutions are found. In their experiments, Kaindl and Kainz found that BHPA tended to find an optimal solution very early and then spent most of its search proving optimality [KK97]. This observation is often erroneously given in the literature as Kaindl and Kainz's explanation for why bidirectional heuristic search is ineffective [ES12, FMS10, LEF12]. These experiments test this claim.

6.2 The 15 Puzzle

The 15 puzzle is a version of the well-known sliding tile puzzle invented by Sam Loyd in the 1870s [Loy59]. The game is played on a 4×4 grid. Each position on the grid is occupied by tiles numbered from 1 to 15, with one location left empty, called the *blank*. A legal move is to move any tile adjacent to the blank into the blank position, after which the blank is in the position previously occupied by the moved tile. The goal is to rearrange the tiles from a scrambled initial configuration into a goal configuration where the tiles are in ascending numerical order from left to right, top to bottom.

Figure 6.1 gives the goal position of the 15 puzzle on the left. Moves in the 15 puzzle have unit edge cost. The branching factor of nodes in the problem space

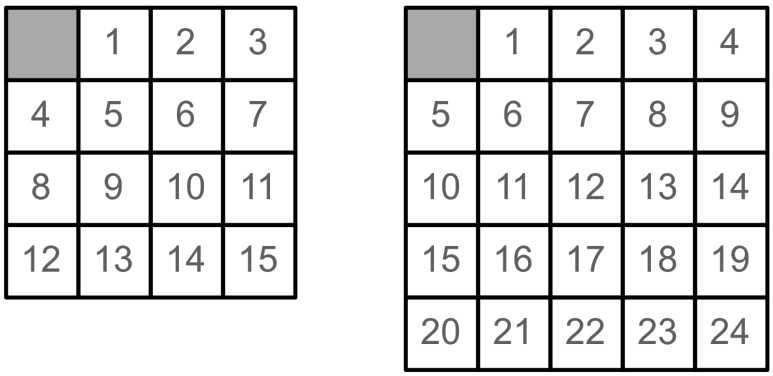

Figure 6.1: Goal positions of the 15 and 24 puzzles.

follows a predictable pattern: all nodes have a branching factor of 4, 3, or 2, and the branching factors of the children of a node are entirely determined by the branching factor of the parent.

6.2.1 Search Technique Used

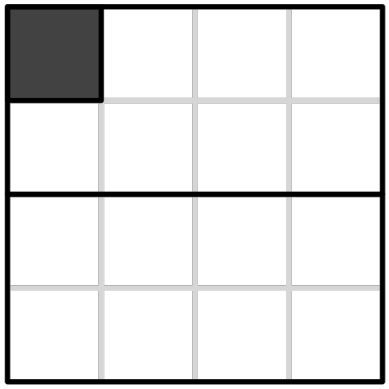

Figure 6.2: The two pattern databases used in the 15 puzzle

For the 15 puzzle, I use IDA* with the pattern databases (PDBs) described in [KF02]. These are shown in figure 6.2. I also use the reflection of these pattern databases across the diagonal, as described in the same paper. To avoid generating some duplicate nodes, the search algorithm prevents regeneration of the parent of

a node n as one of its children by not applying the reverse of the operator that generated n.

6.2.2 Distribution Of g Costs

I generated 100 random instances of the 15 puzzle by making 1000 random moves from the goal state. I solved each instance using IDA*. In these, I calculated the fraction of nodes that are expanded with $g(n) <= C^*/2$. Among these instances, the mean fraction of nodes expanded with $g(n) <= C^*/2$ was 93%. The instance with the smallest fraction expands 63% of its nodes with $g(n) <= C^*/2$, and the instance with the largest fraction expands 99%.

These results show that the 15 puzzle has a strong heuristic and predicts that a unidirectional heuristic search would be the dominant algorithm. And, indeed, I am unaware of a bidirectional heuristic search or a bidirectional brute-force search that is the state-of-the-art on this domain.

6.2.3 Bidirectional Heuristic Search Tested

I implemented bidirectional heuristic search in the 15 puzzle to compare it to the unidirectional heuristic search. I implemented BS* as a canonical representative of bidirectional heuristic search, and implemented A* for unidirectional heuristic search. A* is not the state of the art heuristic search algorithm in the 15 puzzle, but it is the unidirectional algorithm most similar to BS*. Hence, comparing A* to BS* makes the most direct comparison of the contribution of bidirectionality. Using the optimal unidirectional algorithm would only improve the performance of a unidirectional heuristic search.

I compared the two algorithms on 1,000 randomly-generated instances. Among these, A* expanded 20% fewer nodes than BS* and took 37% less time. BS* outperformed A* on 18% of the instances tested. In these instances, BS* expanded

at least as many nodes with $f(n) < C^*$ as did A*. The performance advantage in these instances comes from improved tie breaking, as discussed in section 5.7.2. Overall, as predicted, unidirectional heuristic outperforms bidirectional heuristic search.

6.2.4 Work Spent Proving Optimality

In the 1,000 instances tested using BS* the final solution was found after 90% of the nodes in an instance had been generated, on average. Unlike in Kaindl and Kainz's experiments, optimal solutions tends to be found quite late in search. In their experiments on the 15 puzzle, they used the Manhattan distance heuristic. The pattern databases I used are a much stronger heuristic.

6.3 The 24 Puzzle

The 24 puzzle is another sliding tile puzzle, like the 15 puzzle studied in the previous section. It is played on a 5×5 grid occupied by tiles numbered from 1 to 24, with one location left blank. The goal of the puzzle and the legal moves are identical to the 15 puzzle. Figure 6.1 gives the goal position of the 24 puzzle on the right. Moves in the 24 puzzle have unit edge cost. All nodes have a branching factor of 2, 3, or 4. A node can be placed in one of six classes based on the location of the blank tile on the board. All nodes in the same class have the same branching factor, and the classes of the children of a node are entirely determined by the class of the parent.

6.3.1 Search Technique Used

For the 24 puzzle, I use IDA* with the pattern databases (PDBs) described in [KF02]. These are shown in figure 6.3. I also use the reflection of these pattern

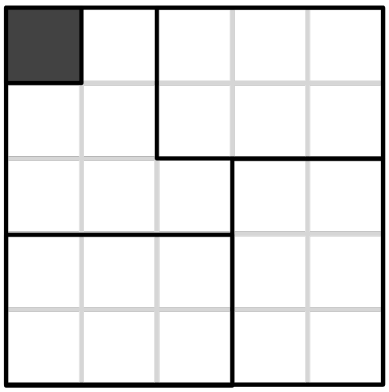

Figure 6.3: The four pattern databases used in the 24 puzzle (this image was shown earlier in figure 2.2).

databases across the diagonal, as described in the same paper. To avoid generating some duplicate nodes, the search algorithm prevents regeneration of the parent of a node n as one of its children by not applying the reverse of the operator that generated n.

6.3.2 Distribution Of g Costs

I solved the 50 24 puzzle instances described in [KF02] using IDA* and calculated the fraction of nodes that are expanded with $g(n) <= C^*/2$. Among these instances, the mean fraction of nodes expanded with $g(n) <= C^*/2$ was 96%. The instance with the smallest fraction expands 80% of its nodes with $g(n) <= C^*/2$, and the instance with the largest fraction expands almost 100%.

These results show that the 24 puzzle has a strong heuristic and predicts that a unidirectional heuristic search will outperform a bidirectional heuristic search. The state-of-the-art algorithm for the 24 puzzle is in fact a *front-to-front* bidirectional heuristic search algorithm, Single-Frontier Bidirectional Search (SFBDS) [FMS10], but this falls outside the scope of my theory. I am unaware of a state-of-the-art front-to-end bidirectional heuristic search in this domain.

The 24 puzzle search space requires too much memory to solve with BS* or A*, so I did not perform the remaining two experiments in this domain.

6.4 Rubik's Cube

The popular Rubik's Cube puzzle was invented in 1974 by Erno Rubik [SSH09]. The puzzle is a $3 \times 3 \times 3$ cube. A $3 \times 3 \times 1$ face of the cube is made up of nine smaller cubes, called *cubies*. Eight of these cubies are located on corners of the cube, and are called *corner cubies*. 12 cubies are located on the edge of the cube, between two corner cubies, and are called *edge cubies*. The remaining eight cubies are called *face cubies*.

Each face can be independently rotated by 90, 180, or 270 degrees in a single move, while the remainder of the cube stays fixed. Each visible face of a cubie is colored with one of six colors. In the goal position, all nine cubie faces that make up each face of the larger cube are all the same color. A move consists of rotating a single face of the cube, reorienting all nine cubies on it simultaneously. The goal of the puzzle is to take a scrambled initial state and transform it into the goal state.

Operators in the Rubik's Cube have unit cost, and all nodes in the problem state have the same branching factor.

6.4.1 Search Technique Used

I solved Rubik's cube with an IDA* solver using a PDB that tracks all eight corner cubies and a PDB that tracks nine edge cubies. For the edge cubies, I do multiple lookups of the edge pattern database using transformational lookups. Because the Rubik's Cube has 24 symmetries, it is possible to transform the cube into a symmetric state by relabeling cubies and look up the symmetric state in the pattern database as well, possibly giving a different heuristic estimate. I use

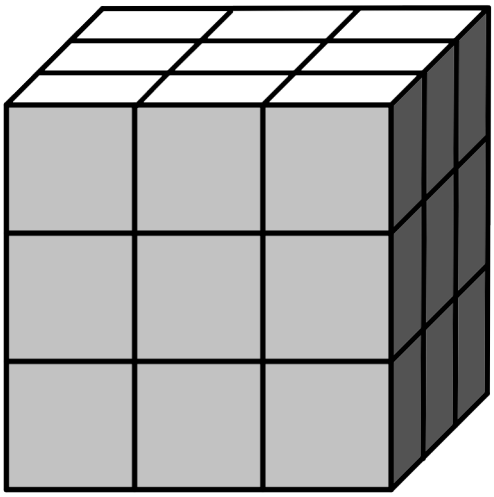

Figure 6.4: A Rubik's Cube.

this technique to do six separate transformation lookups into the edge cubie PDB. This technique is described more fully in [FZH11].

To avoid generating duplicate nodes, I apply a number of pruning techniques. I do not allow multiple consecutive turns of a single face, as the resulting node can also be generated by a single turn of the same face. Two moves that rotate opposite sides of the cube can be made in either order to generate the same state. To avoid generating duplicate nodes in this manner, I assign a number to each face of the cube. For each pair of opposite faces that have disjoint sets of cubies, moves by the lower-numbered face cannot be made immediately after moves of the higher-numbered face.

6.4.2 Distribution Of g Costs

I generated 25 random solvable Rubik's Cube instances by making 100 random moves from the goal state. I solved each instance using IDA*. In these, I calculated the fraction of nodes that are expanded with $g(n) <= C^*/2$. Among these instances, the mean fraction of nodes expanded with $g(n) <= C^*/2$ was 90%. The

instance with the smallest fraction expands 84% of its nodes with $g(n) <= C^*/2$, and the instance with the largest fraction expands 98%.

These results show that the Rubik's Cube heuristics I used are strong, and predicts that a unidirectional heuristic search would be the dominant algorithm. As in the 24 puzzle, the state of the art algorithm is SFBDS, a front-to-front bidirectional heuristic search algorithm, which is beyond the scope of this paper. I am unaware of a front-to-end bidirectional search in the literature that outperforms the best unidirectional heuristic search, and so this result agrees with my theory.

6.4.3 Bidirectional Heuristic Search Tested

I implemented a BS* solver for Rubik's Cube and compared its performance to an A* solver. I compared the two algorithms on 1,000 randomly-generated instances created by making 16 random moves. Instances requiring more than 16 moves to solve require too much memory to be solved by either algorithm.

Among these instances, A* expanded 15% fewer nodes than BS* and took 74% less time. BS* outperformed A* on 29% of the instances tested. In these instances, BS* expanded at least as many nodes with $f(n) < C^*$ as did A*. The performance advantage in these instances comes from improved tie breaking. Overall, as predicted, unidirectional heuristic search outperforms bidirectional heuristic search.

6.4.4 Work Spent Proving Optimality

In the 1,000 instances tested using BS* the final solution was found after 96% of the nodes in an instance had been generated, on average. Optimal solutions tend to be found quite late in my bidirectional heuristic searches of Rubik's Cube.

6.5 Top Spin

Ferdinand Lammertink developed the Top Spin puzzle in 1989 [Lam89]. The puzzle consists of a circular track filled with numbered *tokens*. The tokens can be simultaneously rotated around the track while maintaining the same order. On the track is a *turnstile* that holds some number of tokens. The turnstile can be rotated by 180 degrees, reversing the order of the tokens that are in the turnstile while leaving the remaining tokens in the same order. A single move in Top Spin consists of moving the tokens by any distance on the track followed by a single rotation of the turnstile. The goal is to take some random initial ordering of tokens and reorder them in ascending numerical order.

The number of tokens in a Top Spin puzzle can be varied to make the problem easier or harder, and the turnstile can be resized to hold different numbers of tokens. The 18-4 puzzle shown in figure 6.5 has 18 tokens and a turnstile that reverses the order of four tokens.

All operators in Top Spin have unit cost, and all nodes have the same branching factor.

6.5.1 Search Technique Used

I solved Top Spin using IDA*. The instances I solved had 21 tokens and a turnstile of size four, and so I used three seven-token, additive PDBs. By relabeling tiles I used these databases to do three separate lookups, taking the max. These techniques for doing PDB lookups in Top Spin are described in [YCH08]. To avoid generating some duplicate nodes, the search algorithm prevents regeneration of the parent of a node n by not rotating the turnstile twice in a row with the same set of tokens. Moves affecting disjoint sets of tiles can also be applied in either order to generate the same node. To prevent the generation of duplicate nodes through this method, the solver does not allow consecutive moves that affect dis-

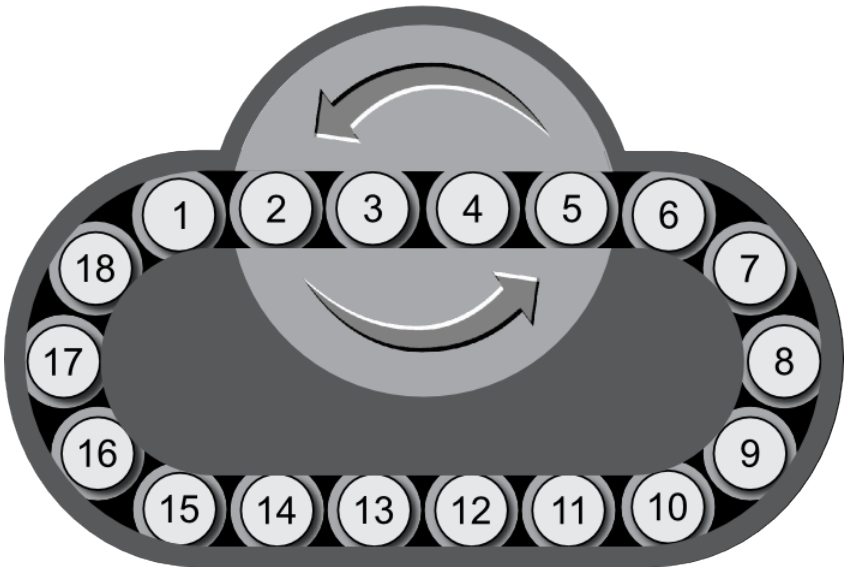

Figure 6.5: The 18-4 Top Spin puzzle

joint sets of tokens when the index of the first token moved by the first move is greater than the index of the first token moved by the second move.

6.5.2 Distribution Of g Costs

I solved 20 randomly generated instances of Top Spin with 21 tokens and a four-token tray. In these, I calculated the fraction of nodes that are expanded with $g(n) <= C^*/2$. Among these instances, the mean fraction of nodes expanded with $g(n) <= C^*/2$ was 98%. The instance with the smallest fraction expands 86% of its nodes with $g(n) <= C^*/2$, and the instance with the largest fraction expands almost 100%.

These results show that the Top Spin heuristics I used are strong, and predicts that a unidirectional heuristic search would be the dominant algorithm. And, indeed, I am unaware of a state-of-the-art bidirectional heuristic search algorithm in this domain.

Top Spin instances require too much memory to be solved with BS*, so I did not implement the remaining two experiments.

6.6 Pancake Problem

The pancake problem is a very simple combinatorial problem and the subject of the only academic paper written by Bill Gates [GP79]. We are given a stack of n pancakes, all of different sizes. The goal is to rearrange the pancakes in order of descending size with the largest pancake on the bottom. A legal operation is to flip the top k pancakes, reversing their order on the top of the stack. Each flip of any number of pancakes counts as a single move, and the objective is to produce a sorted stack in the minimum number of moves possible. Figure 6.6 shows an example initial and goal state for the pancake problem with six pancakes.

All operators in the pancake problem have unit edge cost, and every node in the problem space has the same branching factor, which is one less than the number of pancakes in the search instance. (There is no point in flipping the single topmost pancake.)

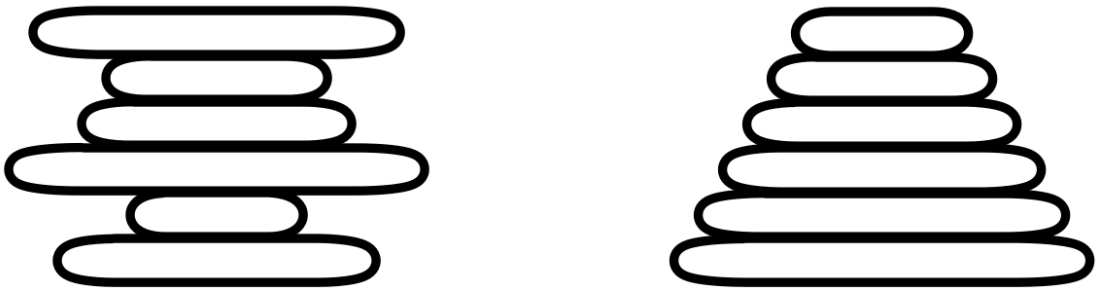

Figure 6.6: A random pancake stack and its sorted goal state.

6.6.1 Search Technique Used

My solver for the pancake problem uses IDA* with the gap heuristic [Hel10]. The gap heuristic considers cases in the current stack where two pancakes are adjacent that are not adjacent in the goal stack. Each such case requires at least one flip to fix, and so the algorithm returns the total number of such cases in the current stack. To avoid generating duplicate nodes, the search algorithm prevents regeneration of the parent of a node by not flipping the top k pancakes twice in a row.

6.6.2 Distribution Of g Costs

I solved 60 random instances of the pancake problem with 80 pancakes using IDA*. Among these instances, the mean fraction of nodes expanded with $g(n) <= C^*/2$ was 57%. The instance with the smallest fraction expands 49% of its nodes with $g(n) <= C^*/2$, and the instance with the largest fraction expands 66%.

As an interesting observation, the gap heuristic for the pancake problem is extremely strong: in all instances I tested very few nodes expanded had $f(n) < C^*$, and often *none* did. The gap heuristic is thus very strong, and so we would expect the vast majority of nodes to be expanded with low g cost. However, in my experiments, a small majority of nodes were in fact expanded with $g(n) <= C^*/2$.

This is because the gap heuristic is almost perfect. With a perfect heuristic a search algorithm is able to expand only the nodes on a single optimal solution path to a goal. The mean g cost of expanded nodes would thus be $C^*/2$. While the gap heuristic is not perfect, it is close enough that the distribution of g costs of nodes expanded is almost flat.

The pancake problem heuristic is very strong, despite the deceptively small fraction of nodes expanded with $g(n) <= C^*/2$. As my theory predicts, unidirectional heuristic search is the state of the art in the pancake problem [Hel10].

6.6.3 Bidirectional Heuristic Search Tested

I implemented and tested BS* and A* on the pancake problem and compared them to each other. I compared them on 1,000 randomly-generated instances with 50 pancakes, the hardest solvable within my memory limitations. While the gap heuristic is very strong, it is not perfect and some number of nodes are expanded that are not on the optimal solution path.

Among these instances, A* expanded 11% fewer nodes than BS* and took 32% less time. BS* outperformed A* on 16% of the instances tested. In these instances, BS* expanded at least as many nodes with $f(n) < C^*$ as did A*. The performance advantage in these instances comes from improved tie breaking. Overall, as predicted, unidirectional heuristic outperforms bidirectional heuristic search.

6.6.4 Work Spent Proving Optimality

In the 1,000 instances tested using BS* the final solution was found after 86% of the nodes in an instance had been generated, on average. Optimal solutions tend to be found quite late in my bidirectional heuristic searches of the pancake problem.

6.7 Pairwise Sequence Alignment

Pairwise sequence alignment is typically applied to the problem of determining the similarity of two strands of DNA or of two proteins. In this dissertation I considered only DNA sequences. A DNA molecule consists of a very large number of small molecules, known as nucleotides connected to a backbone structure. There are four types of nucleotides which are represented typographically by the letters A, C, T, and G. A DNA sequence is thus represented by a very long string of those

characters. One application is to determine the evolutionary similarity between two strands of DNA (which may be subsequences of larger DNA molecules).

The problem defines three operations that can be used to transform one strand into another. A nucleotide on a sequence can be *substituted* by replacing it by a different nucleotide. One or more nucleotides can be *deleted*, in which case they are removed from the sequence entirely and the nucleotides immediately preceding and succeeding the gap become adjacent. And finally, an arbitrary sequence of nucleotides can be *inserted* between two adjacent nucleotides on a sequence. Each of these operations correspond to biologically plausible mechanisms by which DNA sequences can mutate or recombine.

Applying each operation incurs a cost, and the goal is to find the lowest-cost set of operations that can be used to transform one of the two sequences into the other. This gives an estimate of the two strings' similarity. A substitution of one nucleotide with another incurs a fixed cost. Inserting or deleting a nucleotide subsequence incurs a cost that is proportional to the length of the given subsequence. In my solver I add an additional fixed cost for opening a new gap or inserting a subsequence. This is known as an *affine gap cost*, and reflects the fact that it is biologically more plausible to have a few long insertions or deletions than many short ones. Inserting a subsequence into another sequence is equivalent to deleting that subsequence from the latter sequence, and so has the same associated cost.

Pairwise sequence alignment is modeled as a pathfinding problem by placing one sequence on the X axis of a grid and the other on the Y axis. I call these sequences S_X and S_Y, respectively. The goal is to find the lowest-cost path from the top-left corner of the grid, representing the start of both sequences, to the bottom-right corner, representing the end of both sequences. Each entry on the grid maps to a single nucleotide each from S_X and S_Y. The path can move from one position in the grid to an adjacent position, either immediately right, downward, or diagonally down-right.

Any path taken through this grid corresponds to a sequence of operations that will transform one sequence into the other. A single move to an (x, y) position in the grid from an adjacent position corresponds to one of the three possible operations. A diagonal move means that the nucleotide at index x in $S_X$is substituted with the nucleotide at index y in S_Y, and vice versa. If these are the same nucleotides then they are not changed. A horizontal move corresponds to the insertion of the nucleotide at index x into S_X, or equivalently the insertion of a gap into S_Y. A vertical move is the insertion of the nucleotide at index y into S_Y, or the insertion of a gap into S_X.

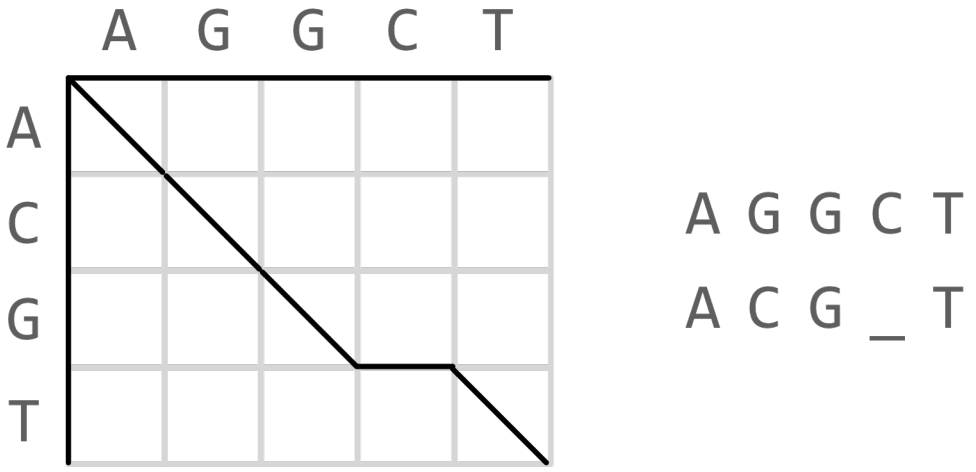

Figure 6.7: A path through a sequence-alignment grid, and the corresponding alignment. The "_" character represents a gap inserted in the sequence.

The scoring function in pairwise sequence alignment is unusual in that it allows for negative edge costs. While there are several well-accepted cost functions for protein alignment problems, such as the PAM and BLOSUM matrices [DS78, HH92], there does not appear to be an accepted standard for DNA sequence alignment. There appears to be a consensus that aligned nucleotides (i.e., a subsitution of one nucleotide for the same one) should be given a positive score, and that all other substitutions, insertions, and deletions should be given negative scores. Unlike most other domains, then, the goal is to maximize the

score of a path. Because there is no consensus opinion on a reasonable scoring matrix, I used a custom scoring function in my code. The cost of substitutions is given in table 6.1, and is taken from the BLASTN aligner as described in [SGA91]. I could not find accepted values for gap costs, and so used a cost of -20 to open a gap and -8 to continue a gap by one nucleotide.

	A	C	T	G
A	5	-4	-4	-4
C	-4	5	-4	-4
T	-4	-4	5	-4
G	-4	-4	-4	5

Table 6.1: The cost of substituting one nucleutide with another in my sequence alignment solver.

Sequence alignment does not have unit costs for operators, however all operator costs are integer. In addition, the domain has the unusual property that there are both positive and negative edge costs. As a result, the g cost of a child node can be lower or higher than the cost of the parent, whereas in most studied domains g costs of children cannot be lower than that of their parent. However, because the heuristic used by my algorithm is consistent, it is still the case that the f cost of a child can be no greater than the f cost of its parent. (Remember that, unlike other domains, the goal in sequence alignment is to *maximize* the cost of a solution.)

As the problem space in pairwise sequence alignment is a grid, the structure of the graph is very regular. Nodes at the edge of the grid (corresponding to the end of one or both sequences) have a branching factor of two. All other nodes have a branching factor of three.

6.7.1 Search Techniques Used

My solver for pairwise sequence alignment is LBD-Align, an algorithm specifically designed for this domain [Dav01]. LBD-Align is based on dynamic programming. It organizes the dynamic programming matrix with the top left of the matrix corresponding to the beginning of both sequences, or the origin, and the bottom right corresponding to the end of both, or the goal. It fills in the matrix by exploring successive diagonal levels that are perpendicular to the main diagonal extending from the origin to the goal.

Unlike in standard dynamic programming, LBD-Align uses a heuristic to avoid filling in matrix values that provably cannot be part of an optimal alignment. It first generates a cost cutoff by using a cheap search to find a suboptimal solution. During search, it does heuristic evaluations of the first and last cells filled in a diagonal. If their f cost exceeds the cutoff, their children are not filled on the next level. The first and last nodes of a diagonal tend to have the highest f costs, and many children reached through these nodes cannot be reached through any other nodes on that level. This technique means that LBD-Align explores a small number of provably-suboptimal nodes, but this is more than made up for by the time savings of only doing a small number of heuristic evaluations.

To the best of my knowledge, LBD-Align is the state-of-the art solver for optimal pairwise alignment. I use the heuristic suggested in [Dav01], which returns the minimum gap size required to align the remainder of the two sequences and assumes that any remaining nucleotides can be aligned optimally.

I designed and implemented a bidirectional heuristic search for pairwise sequence alignment based on LBD-Align, as well. In this algorithm, LBD-Align is run both forward from the initial state and backwards from the goal state. An entire level is filled in at a time. At each point, the algorithm fills the next level in the direction that has the fewest filled cells on its most-recently-filled level. When

both directions fill the same level from different directions, all possible solutions can be found. For each cell in the level the costs to reach that cell from both directions are merged, and the best-cost cell gives the optimal solution cost.

6.7.2 Distribution Of g Costs

I generated test instances by constructing a sequence of random nucleotides, and then created two test sequences as "descendants" of that sequence by randomly perturbing it using substitutions, gaps, and deletions. Using this technique I generated 100 random sequence pairs of approximately 500,000 nucleotides each.

LBD-Align is based on dynamic programming, and so does not search nodes in order of their f or g cost, but rather in order of increasing depth from the initial state. A bidirectional brute-force search of the problem space would not meet when expanding nodes with g costs of $C^*/2$, but rather at half the depth of an alignment. As such, I compute the fraction of nodes expanded with half the depth of an optimal solution, rather than with $g(n) <= C^*/2$.

Among these instances, the mean fraction of nodes expanded with less than half the solution depth was 57%. The instance with the smallest fraction expands 51% of its nodes at less than half the solution depth, and the instance with the largest fraction expands 60%.

While not as strong as the other heuristics I considered, the sequence alignment heuristic is strong enough that my theory predicts a unidirectional heuristic search would dominate. And indeed, so far as I know, unidirectional LBD-Align is the strongest search algorithm for pairwise sequence alignment.

6.7.3 Bidirectional Heuristic Search Tested

I compared LBD-Align to my bidirectional LBD-Align implementation on the same 100 random instances discussed in the previous section. Among these in-

stances, LBD-Align expanded 9% fewer nodes than bidirectional LBD-Align and took 11% less time. Bidirectional LBD-Align did not outperform unidirectional on any instances.

6.7.4 Work Spent Proving Optimality

Because my bidirectional LBD-Align algorithm uses a breadth-first search order, the last nodes generated in either direction are those on the single intersecting level. The optimal solution is found on this level, after all other levels of search have been completely generated. The problem instances I tested had on the order of 1,000,000 levels, so the optimal solution is found after the vast majority of nodes have been generated: over 99% of all the nodes in each problem instance I tested.

6.8 Road Navigation

Road navigation is the problem of finding routes between two points on a map. It is perhaps the most well-known pathfinding problem, encountered by many people on a daily basis when they use GPS to find the quickest driving route. In road navigation, a map is represented by a problem-space graph by modeling road intersections as nodes and road segments connecting those intersections as edges connecting nodes. Depending on the specific optimization goal, edge costs can represent different real-world properties. My experiments solved the problem of finding the fastest route between two points on a map (as opposed to, for example, the shortest). As such, the cost of an edge is the time taken to traverse that road segment while driving at the speed limit.

More complex implementations of road navigation can incorporate other goals into their costs, such as a preference for not driving down undeveloped roads or modeling the costs of making turns. I chose a very simple model for my work,

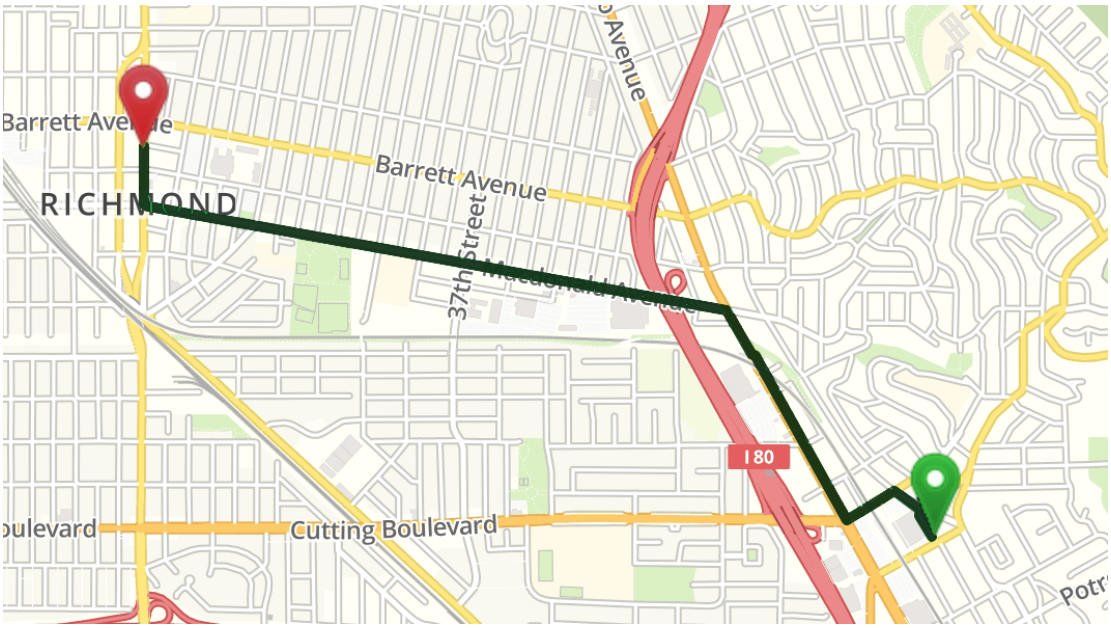

Figure 6.8: Example pathfinding problem showing the route between two points on a map. Graphics are from OpenStreetMap [Ope15] and route is generated by GraphHopper [Gra15].

where we assume that a road can always travel at the speed limit on a road and that turns are free. Figure 6.8 is an example of a real road route found by a pathfinding algorithm.

Like sequence alignment, road navigation does not have unit operator costs. Unlike all of the other domains I studied, operator costs are non integer and very few operators in the problem space have identical edge costs. As a result, there are a very large number of unique f and g costs encountered in a search of the problem space: very few nodes have duplicate costs. The most notable implication of this is that no algorithm in this domain has the benefit of improved tie-breaking, since only a very small number of nodes will have f cost equal to C^*.

Unlike the other domains I study, the structure of the road navigation problem-space graph is very irregular. The graphs of all of the other domains considered are very regular: most nodes have the same branching factor, or there is a small

number of unique branching factors whose distribution in the graph follow regular patterns. Additionally, in the case of sequence alignment even though there are non-unit edge costs, edges with the same costs do not tend to be clustered in the search graph. In road navigation, however, the branching factor of nodes does not follow a simple pattern, as the graph models a road network. Also, values of edge costs in the graph tend to be clustered. For example, regions of the graph that represent a city will tend to have many edges with low cost in close proximity, while rural regions will tend to have edges with higher g cost.

Finally, the problem-space graph for road navigation is quite small and is explicitly represented. The entire map of California can be represented with half a gigabyte of RAM. All of the other domains studied are represented implicitly and generated as the algorithm explores it.

6.8.1 Search Technique Used

For the road navigation problem I used the GraphHopper solver [Gra15], an open-source routing engine for OpenStreetMap data [Ope15]. The state-of-the-art solvers for road navigation preprocess the graph using an expensive technique known as contraction hierarchies [GSS08] and then uses a bidirectional form of Dijkstra's algorithm [Dij59] on the preprocessed graph. Since my interest is in general-purpose heuristic search algorithms, I instead use GraphHopper's A* implementation. I use the Euclidean distance heuristic.

6.8.2 Distribution Of g Costs

I generated test instances by picking two random latitude/longitude points in a map of California and picking the closest nodes in the graph to those points. Some of these problem instances may be unsolvable because there is no connecting route

between the two points (for example, if only one of the points is on an island). I discarded any unsolvable pairs of points, and constructed 500 solvable instances.

Among these instances, the mean fraction of nodes expanded with $g(n) <= C^*/2$ was 53%. The instance with the smallest fraction expands 4% of its nodes with $g(n) <= C^*/2$, and the instance with the largest fraction expands 96%.

The distribution of g costs has high variance, which is due to the irregularity of the road navigation search graph. Road navigation is unique among the domains I studied in having a very irregular search graph: different regions of the map have very different densities of nodes. The high variance of g cost distributions is due to this irregularity.

In instances where a small fraction of nodes are expanded with low g cost, the initial state is found in a remote, rural area of the map and the goal is found in a dense area (such as a city). In a brute force search of the problem space, vanishingly few nodes would be expanded near the initial state and an extremely large number near the goal. Many of those expanded near the goal are very close to the goal state and would have f costs close to optimal. Even though the heuristic prevents the expansion of some large number of nodes near the goal, there are so many more nodes near the goal state that a heuristic search still expands the majority of its nodes with high g cost.

The converse is true for instances where a huge fraction of nodes are expanded with low g cost. These problem instances have a initial state in a city and a goal state in a remote area. The number of nodes expanded with very low g cost in a brute force search is very large, and adding a heuristic ensures that the vast majority of nodes expanded in a unidirectional heuristic search have low g cost.

I verified that this is the correct explanation by performing reverse searches on problem instances with extremal values. For those instances where the vast majority of nodes expanded had low g cost, the vast majority of nodes expanded

in a reverse search had *high* g cost. The same held for reverse searches on instances where the majority of nodes expanded had high g cost.

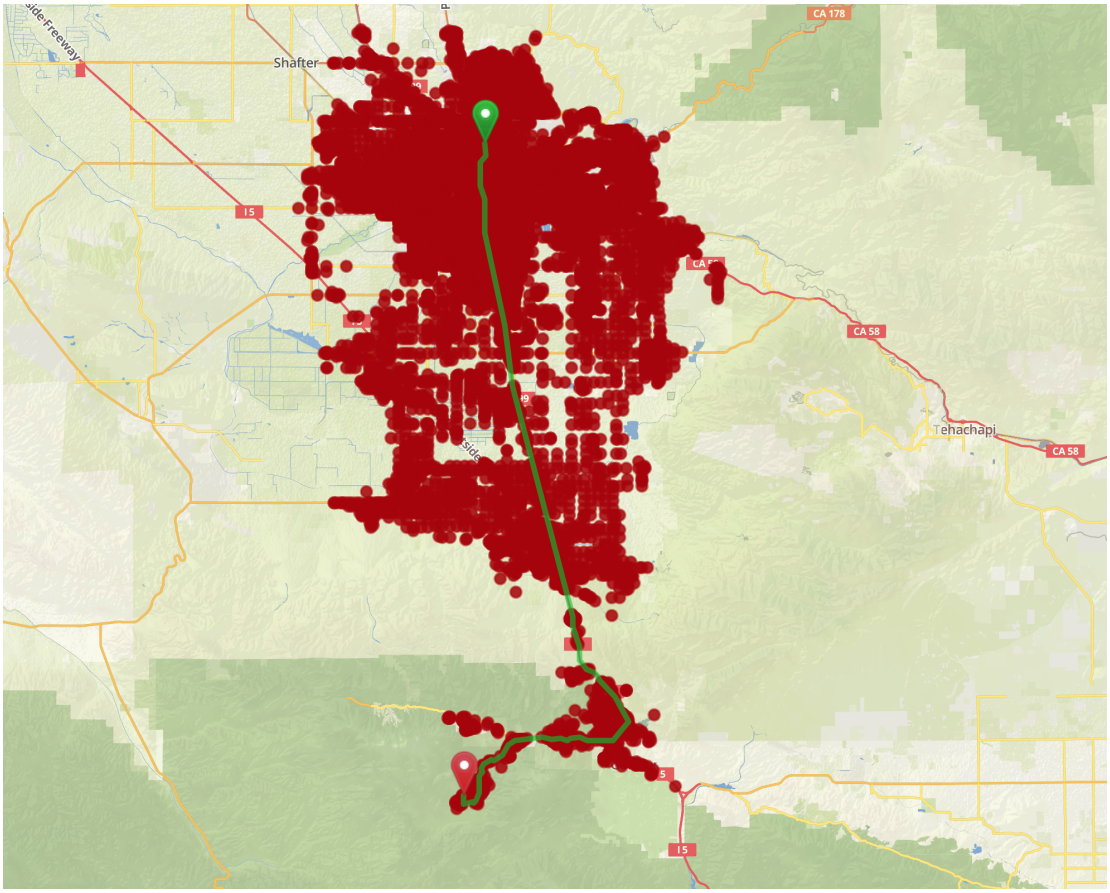

Figure 6.9: Nodes expanded in a forward A* search of a road navigation instance. The start state is the topmost location marker.

Figures 6.9 and 6.10 show this graphically. These images show nodes expanded in forward and reverse A* searches of a road navigation instance. The initial state in the problem instance is in the middle of a city and the goal state is in a remote rural area. The forward search expands significantly more nodes than the reverse search, and 90% of the nodes it expands have $g(n) < C^*/2$. This is because it explores most of the city in the its search but there exist few nodes to explore near the goal state. The reverse search expands significantly fewer nodes, and only 43% of the nodes it expands have $g(n) < C^*/2$. This is because there are few

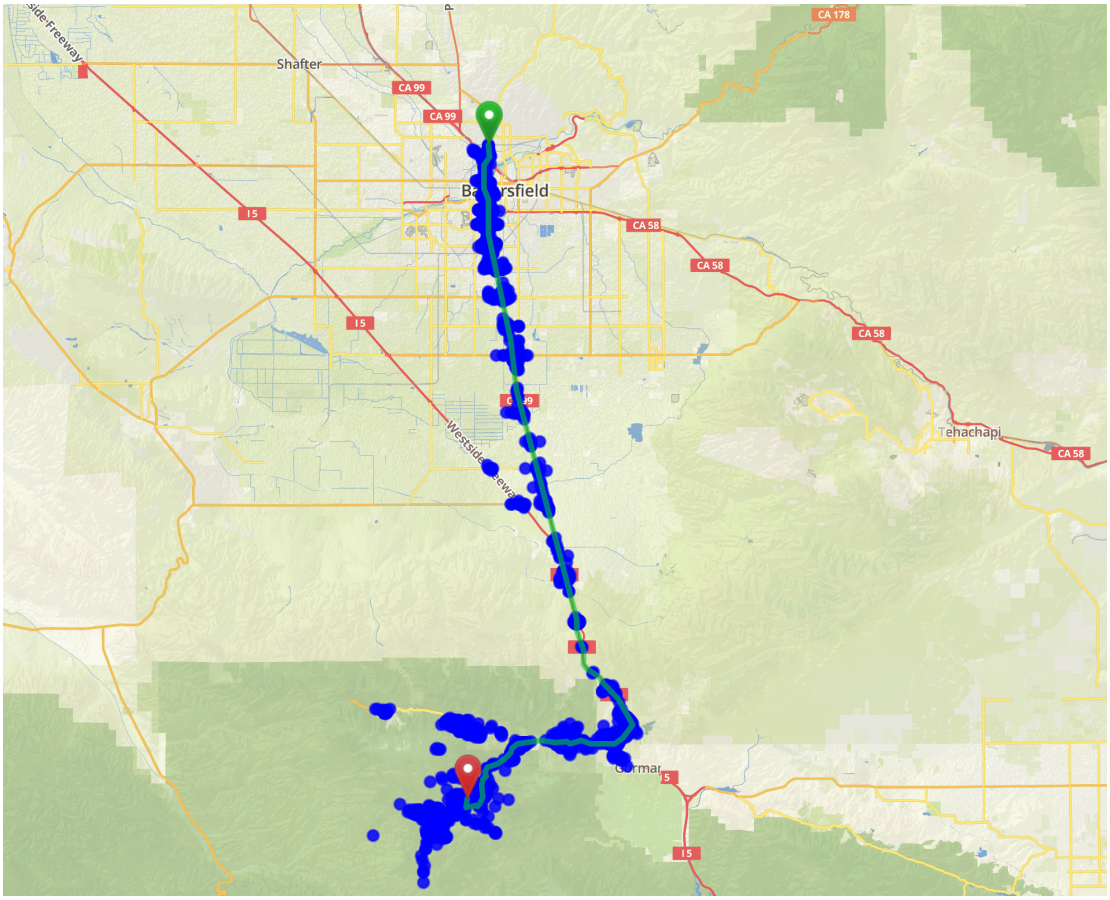

Figure 6.10: Nodes expanded in a reverse A* search of a road navigation instance. The start state is the topmost location marker.

nodes to search near the goal state, and by the time the search gets close to the initial state the heuristic is accurate enough that the search expands few nodes off of the optimal path.

Because of this potential for asymmetric search difficulty, I would expect a unidirectional heuristic search to be more effective in one direction than the other. Since slightly more than half of the nodes expanded have g cost less than $C^*/2$, my theory predicts that unidirectional heuristic search will outperform bidirectional brute-force search. However, because the heuristic is only of medium strength and a unidirectional heuristic search may perform better in one direction of search than the other, a bidirectional heuristic search may be at least somewhat effective (see section 5.7.3).

6.8.3 Bidirectional Search Algorithms Tested

I compared A* to a bidirectional implementation of the brute-force Dijkstra's algorithm [Dij59] on all 500 instances described previously. Because the majority of nodes are expanded with $g(n) <= C^*/2$, A* outperforms Dijkstra's algorithm. A* expands 221,404,936 solving all instances, while Dijkstra expands 353,964,832. A* performs better in 369 instances, while bidirectional Dijkstra performs better in the remaining 131.

I also implemented BS* and compared it against A* on the same 500 instances. Among these instances, BS* expanded 9% fewer nodes than A* and took 5% less time. BS* outperformed A* on 57% of the instances tested. In other words, BS* outperforms A* by a small margin.

As noted previously, A* searches in road navigation can be much cheaper in one direction than the other. To account for this, I also compared BS* to an A* search in the cheaper of the two directions for each instance. When this is done, the cheaper A* expands 20% fewer nodes that BS* on average, and takes 27% less

time. BS* beats A* only 19% of the time. This confirms that the major advantage of BS* over A* is due to the imbalance of search difficulty in both directions, as discussed in section 5.7.3.

Of all of these instances, BS* outperforms both A* *and* bidirectional Dijkstra on 77, or 15% of all instances tested. In particular, it expands fewer nodes with $f(n) < C^*$, so the performance advantage is not just from improved tie-breaking. Rather, these are examples of pathological cases of the form discussed in section 5.6. The road navigation heuristic is of medium strength, which is where I predicted these types of pathological cases might occur. There are relatively few of them overall, however, and on average BS* only expanded 8% fewer nodes than the strongest of the two other algorithms, a relatively small improvement.

6.8.4 Work Spent Proving Optimality

In the 1,000 instances tested using BS* the final solution was found after 42% of the nodes in an instance had been generated, on average. This is much earlier, on average, than the optimal solutions were found in the other domains I tested. The heuristic used in this domain is also weaker than that of the other domains studied.

6.9 Four-Peg Towers of Hanoi

The Towers of Hanoi puzzle is played with n disks of different sizes, which are distributed across some number of pegs. In this dissertation I consider the four-peg variant. In the standard version of this problem, all disks start off stacked on a single peg (in descending order of size with the largest on the bottom) and the goal is to move them all to some other single peg. Only one disk may be moved at a time, from the top of the stack on one peg to another peg, with the constraint that a larger disk cannot be placed on top of a smaller disc.

The standard, three-peg version of this peg has a provably-optimal constructive solution. There is a constructive solution to the four-peg version known as the Frame-Stewart Algorithm [Fra41, Ste41] which has been shown to be optimal up to 31 disks [Kor08], however its optimality in general has not been proven. As such, search is currently the only way to prove solution optimality for a given number of disks.

With the standard initial and goal states, all disks start on the first peg and end on the last peg. At the halfway point of an optimal solution to the standard problem, all disks but the largest are distributed across the middle two pegs [Hin97] and the largest disk remains on the first peg. The largest disk is then moved to the last peg, and the labels of the first peg and the last peg are swapped. Reversing the sequence of moves that reached the middle state will place the smaller disks back on top of the largest disc.

Because the form of the middle state of an optimal solution is known, searching with the standard initial and goal states can be done much more quickly. We can generate all possible midpoint states where all but the largest disk are distributed across the middle two pegs and the largest disk is on the initial peg, and define those as our goal states. Then a search can be done from the initial state to the set of middle states, rather than to the goal state of the original problem. This cuts the search depth in half and dramatically reduces the work in finding an optimal solution.

6.9.1 Arbitrary Start And Goal States

The standard problem instance of the four-peg Towers Of Hanoi has been solved for up to 31 disks [Kor08]. In my dissertation, however, I consider the general case of the Towers of Hanoi where disks in the initial and goal states are distributed

arbitrarily across any of the four pegs, subject to the restriction that no disk can be on top of a smaller one.

One implication of this general case is that I consider a significantly larger number of problem instances, rather than the single problem instance of the standard games. This allows for a much larger number of test instances.

A second, more subtle implication is that in this version of the problem it is not possible to search to a set of known middle states, as previously discussed. Since we cannot search to a known midpoint state, we must complete a search to full depth to find a solution to a problem instance. This greatly reduces the maximum number of disks in a tractably-solvable problem instance.

Finally, reverse searches in the standard problem instance are significantly more expensive than forward searches. Reverse searches are done backwards from the set of candidate middle states towards the goal state. For a given number of disks d, there are 2^{d-1} candidate middle states, in which all but the largest disk are distributed across the middle two pegs. With even 21 disks a reverse search is seeded with over a million goal states. The number of nodes expanded with depth increases exponentially, making the reverse search far more expensive than the forward search.

This makes a bidirectional search pointless, despite the weakness of the heuristic, as a forward search is so much cheaper than a reverse serach (see section 5.7.3). This was the observation that first led me to consider the general four-peg Towers of Hanoi case. In the general case, however, a problem instance is not symmetric and we cannot search to a set of candidate middle states. This means that a bidirectional search is a viable strategy.

6.9.2 Search Techniques Used

I implemented two algorithms for Towers of Hanoi, a unidirectional heuristic search and a bidirectional brute-force search.

My unidirectional solver is based on Korf's solver from [Kor08]. That solver uses BFHS with delayed-duplicate detection (described in sections 2.3 and 2.4, respectively) to solve the standard problem instance with disks of varying sizes. Because there is a presumed-optimal solution given by the Frame-Stewart conjecture, BFHS can be run with the cost of this presumed-optimal solution as its cost cutoff. In theory, we could run BFHS with a cutoff of one less than the presumed-optimal solution cost to simply verify that there exist no cheaper solutions. Using the actual cost allows us to find the Frame-Stewart solution as well, though, and verifying the correctness of our implementation.

The solver uses two PDBs as a heuristic. The databases track two disjoint sets of disks. As these sets are disjoint and only one disk moves at a time, the heuristic values returned by these two databases can be added together to make a stronger admissible heuristic.

I modified this solver for the general case with arbitrary start and goal states. Because the goal state differs with each instance, the same PDB cannot be reused. As an initial step, then, my solver generates instance-specific PDBs. The cost of this step is included in my timing numbers for solving a given instance. Since the Frame-Stewart conjecture cannot be used to give a conjectured-optimal solution for a given instance in advance, we cannot use the cost of such a solution as the cutoff for BFHS. Instead, I first find the cost of an optimal solution using my bidirectional algorithm and then provide this cost to BFHS as its cutoff. This means that my unidirectional solver is an idealized one that knows the cost of an optimal solution in advance. A real solver would not have this advantage and would thus perform worse.

I use bidirectional breadth-first search as my brute-force bidirectional solver. Disk-based breadth-first searches are performed from the initial and goal states towards a point of solution intersection. As this is a brute-force search, it does not require a preparatory step to generate PDBs. In addition, since breadth-first search does not use a cost cutoff, the solver does not need to know the cost of an optimal solution in advance.

6.9.3 Distribution Of g Costs

I solved 100 randomly generated instances of the four-peg Towers of Hanoi with 20 disks. I calculated the fraction of nodes expanded with $g(n) <= C^*/2$ using my unidirectional solver with two pattern databases tracking 15 disks and five disks, respectively. This solver requires knowing the optimal solution cost in advance, which I found by first solving each instance with my bidirectional brute-force solver.

Among these instances, the mean fraction of nodes expanded with $g(n) <= C^*/2$ was 31%. The instance with the smallest fraction expands 13% of its nodes with $g(n) <= C^*/2$, and the instance with the largest fraction expands almost 59%.

The heuristic I used for this domain is quite weak, and as such I expect a bidirectional brute-force search to outperform the a unidirectional heuristic search on this domain. The following section shows that this is the case.

The four-peg Towers of Hanoi domain has many duplicate states and, with many disks, has very large search spaces. Solving these instances requires the use of disk-based techniques to store nodes for duplicate detection. While there is some discussion in the literature of how to do best-first searches using disk-based techniques [Kor08], to my knowledge this discussion remains theoretical

and no such algorithms have actually been implemented. As such, I only compare a unidirectional heuristic search and a bidirectional brute-force search.

6.9.4 Bidirectional Brute-Force Search Tested

I compared my unidirectional heuristic solver against my bidirectional brute-force solver on many random instances with 20 and 21 disks. Because the heuristic for this domain is quite weak, I predicted that a bidirectional brute-force search will outperform a unidirectional heuristic search.

The heuristic search solver must construct a new PDB for each instance. While I found that the savings of the heuristic function outweighs the cost of generating the PDB, this introduces a degree of freedom: the size of the PDB to generate for a given instance.

For N=21 disks, I solved 10 random instances with both algorithms. I considered three different PDB sizes for the heuristic search. The weakest PDB configuration partitions the disks into two sets of disks, one of size 15 and one of size 6; the strongest partitions them into sets of size 17 and 4. I solved each instance with my unidirectional heuristic solver using each PDB configuration, as well as with my bidirectional brute-force solver.

The results are given in table 6.2. The first row gives the time and number of nodes required to build the PDB, which applies only to unidirectional searches and does not vary by instance. The remaining rows are the nodes and times required by each solver for each instance, including PDB creation. The best performance overall is bolded (always bidirectional search), and the best unidirectional search is underlined. Times are in seconds and node counts are in billions.

Note that the average time per node expanded is not constant across each problem instance. In particular, in larger problem instances fewer nodes are expanded per second. This is because these solvers use disk storage, which introduces

some level of unpredictability in timing numbers. In smaller problem instances, the files generated may be quite small, and so the operating system may never in fact write these files to disk. In larger problems, the files are larger and actual disk access is required.

Additional factors affect timing numbers of node expansions of the two algorithms. The bidirectional search has to test for frontier intersections while the unidirectional algorithm does not. Conversely, the unidirectional algorithm does pattern-database lookups for its heuristic while the bidirectional solver does not. Because of all of these differences in overhead, my comparison of these algorithms is based on wall-clock time rather than number of nodes expanded.

	Bidir.		Unidir. 15/6		Unidir. 16/5		Unidir. 17/4	
	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes
PDB	N/A		83s	1	328s	4	1,380s	17
Instance 1	**3,707s**	60	51,186s	291	20,770s	174	*8,315s*	97
2	**125s**	4	*202s*	6	411s	5	1,445s	17
3	**1,299s**	34	9,385s	116	3,338s	65	*1,858s*	36
4	**300s**	13	*461s*	17	474s	9	1,461s	20
5	**589s**	19	1,223s	36	*1,057s*	27	1,650s	31
6	**7,283s**	88	88,349s	430	50,967s	300	*24,261s*	174
7	**158s**	6	*272s*	9	466s	9	1,415s	18
8	**4,104s**	66	60,981s	331	26,803s	208	*7,660s*	96
9	**414s**	17	869s	30	*693s*	22	1,539s	24
10	**445s**	17	2,168s	47	*1,022s*	28	1,577s	24

Table 6.2: Instances of 21-disk Towers of Hanoi. Node counts are in billions.

Bidirectional brute-force search outperformed the best unidirectional heuristic search by an average factor of 1.5-3. In some instances, like number three, unidirectional is faster when ignoring PDB creation; when counted, however, bidirectional

brute-force search is always fastest. If bidirectional brute-force search is compared to each unidirectional search independently, rather than against the strongest of the three for a given instance, the relative performance is often much stronger, sometimes by over an order of magnitude. As one would expect, the weaker heuristic does relatively worse on harder instances and the stronger heuristic does relatively worse on easier instances (due to PDB creation cost). We cannot know the difficulty of an instance in advance, and thus do not have the luxury of guaranteeing we will always use the best PDB.

In addition, I solved 100 random instances with 20 disks. For the unidirectional search, I selected the heuristic by solving the first 10 instances with different PDBs and finding the configuration with the best overall performance. This was a partitioning into two PDBs of 15 and 5 disks. I then solved all 100 instances using this strongest configuration, as well as with my bidirectional brute-force solver.

Table 6.3 summarizes these results. The first two columns give the solution time for each algorithm, and the third gives the relative slowdown of unidirectional heuristic search over bidirectional brute-force search. The first row gives PDB creation time, the second gives the mean solution time over all 100 instances, and the third and fourth give the values for the instances with the worst and best relative performance of bidirectional brute-force search. All times are in seconds. Again, bidirectional brute-force search outperforms unidirectional heuristic search in all instances, by an average factor of 3.65.

	Bidir.	Unidir. 15/5	Unidir. Slowdown
PDB Generation	N/A	83s	N/A
Mean Results	**217s**	793s	3.65
Best for BiDir.	**347s**	3939s	11.35
Worst for BiDir.	**85s**	126s	1.48

Table 6.3: Timing results on 100 instances of 20-disk Towers of Hanoi.

6.9.5 New State-Of-The-Art Algorithm

Based on the results of my experiments in section 6.9.3, we would expect a bidirectional brute-force search to outperform a unidirectional heuristic search. My experiments confirmed this expectation. Even with an idealized implementation of unidirectional heuristic search, bidirectional brute-force search outperforms it by over a factor of three. These results reinforce the predictions made by my theory. These are the first conducted in this domain with arbitrary initial and goal states, and the bidirectional brute-force solver I introduce is the state of the art approach for solving these types of instances.

6.10 Peg Solitaire

Peg solitaire is a simple puzzle game. I introduced the state-of-the-art solver for this domain in a 2012 paper [BK12b], which involves many novel techniques. The solver that I implemented in my experiments is much more involved than the others described in this chapter, and this section is correspondingly much longer.

Peg solitaire is a one-player puzzle played on a board which has a number of holes, some of which are occupied by pegs. Any peg can jump over an adjacent peg to land in an empty hole, and the jumped-over peg is removed from the board. The objective of the game is to make a sequence of jumps that leave the board in a specified goal configuration. While not required, the initial position generally has a single empty hole and the goal board has a single peg remaining, often left in the initially-empty hole.

While peg solitaire can be played on a number of different board types, I consider only games played on a rectangular grid. On these, pegs can jump vertically or horizontally, but not diagonally. Figure 6.11 shows solvable opening states on the four primary boards studied in this paper.

The most basic question in peg solitaire is whether there exists *any* sequence of jumps that transforms the initial state into a goal state. In my work, however, I consider the optimization problem, where consecutive jumps by the same peg count as a single move and we try to find the fewest number of moves required to generate a goal state. To disambiguate, I use the word "move" to apply exclusively to one or more single jumps by the same peg.

Figure 6.11 shows examples of solvable states on the four boards studied in my work: the English, French, Diamond(5), and Wiegleb boards. In an initial state for the English board, having the center hole empty is a solvable state. In the three remaining boards, though, the puzzle cannot be solved if only the center hole is empty in the initial state.

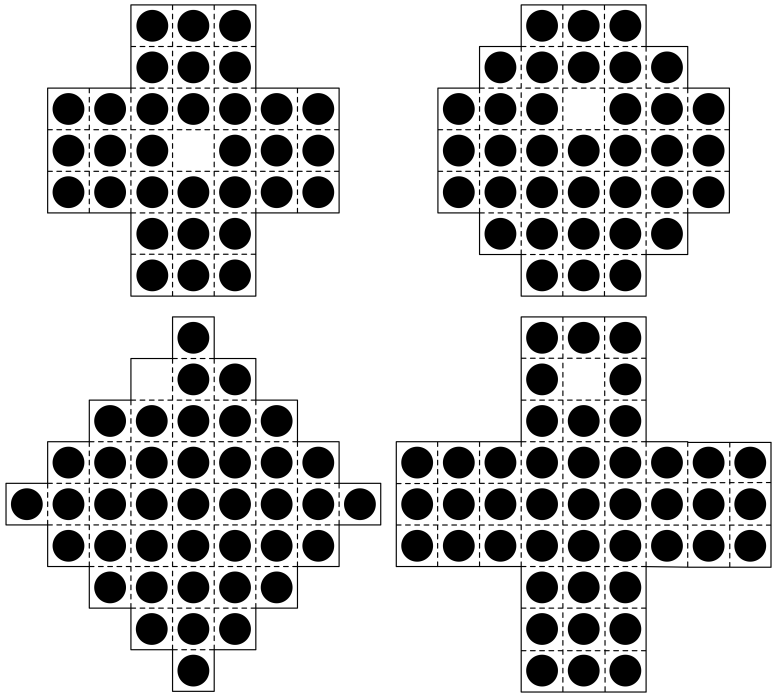

Figure 6.11: Solvable initial states on the English (top left), French (top right), Diamond(5) (bottom left), and Wiegleb boards (bottom right).

6.10.1 Previous Work

Most of the previous work on computational search in peg solitaire has been done by George Bell, using both brute-force and heuristic techniques. [Bel07] introduces a heuristic that he uses in unidirectional heuristic search, which he used on the English board. [Bel12] gives Bell's timing numbers for a bidirectional brute-force solver on the French and Diamond(5) boards, where his heuristic is not effective. To the best of my knowledge, these are the previous states of the art on this domain.

6.10.2 Domain-Specific Techniques

This section describes several domain-specific techniques that are used in my solver. I did not discover these ideas, but understanding them is crucial to understanding the techniques I developed and so I introduce them here.

6.10.2.1 Pruning Constraints

There are a number of techniques used by my solver that can prove that certain board positions can never be solved. If a search algorithm encounters a board that violates one of these pruning constraints, it can ignore that board as provably unsolvable without doing any further work. [Bea85] and [Bel12] describe three techniques used to prove certain board states unsolvable that I use in my solver.

The first of these is the concept of *position classes*. Every board configuration can be put into one of a set of position classes, and every child has the same class as its parent. Thus, if the initial and goal states are of different classes, a problem instance can be discarded as unsolvable without performing any search. My implementation of position classes is taken from the description in [Bel12], and is not my own original contribution.

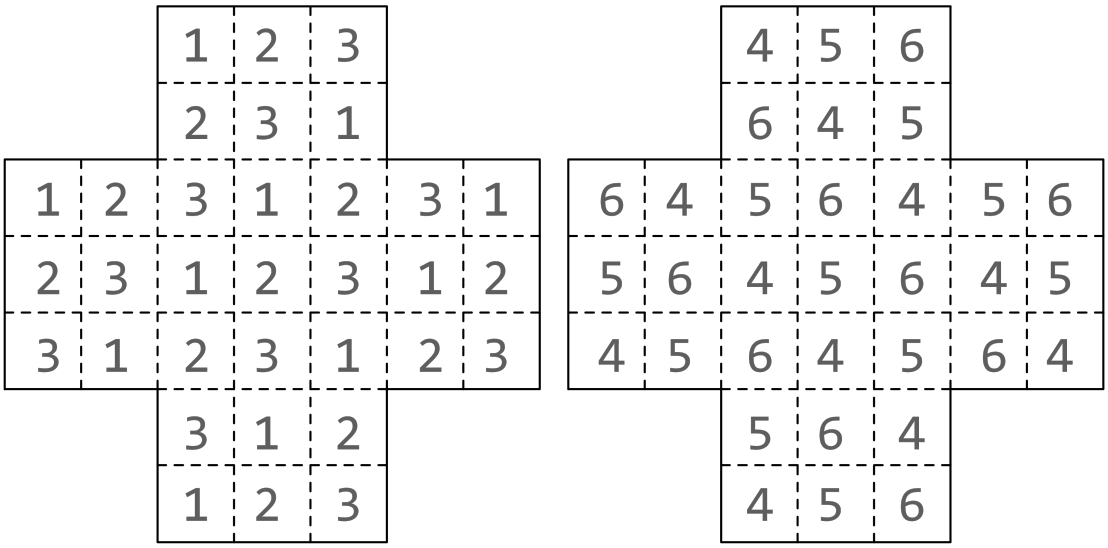

Figure 6.12: Labels of cells used to define position classes on the English board.

To define a position class, we label each hole with the numbers one through six. The numbers one through three are assigned such that each hole in a diagonal from the top-right to the bottom-left is labeled with the same number, with numbers written in ascending order from left to right (starting over at one after a three). The numbers four through six are assigned to the same holes, but assigned such that each hole in a diagonal from the top-left to the bottom-right is labeled with the same number, with numbers written in ascending order from left to right (starting over at four after a six). Each hole is thus labeled with two numbers. Figure 6.12 gives an example of this labeling on the English Board.

Any jump removes pegs from two holes and puts a peg in a third. Under these two labelings, the three holes affected are labeled with one each of the numbers one, two, and three, and one each of the numbers four, five, and six. These are the only two labelings (ignoring renumbering) for which every possible jump affects holes labeled with three different numbers.

Given a particular board, T is the total number of pegs on that board, and N_i is total the number of pegs occupying holes labeled i. A jump in peg solitaire

removes one peg from the board, decreasing T by one. For each of N_1, N_2, and N_3, a jump either increases or decreases the number of pegs with that label by one as a peg is placed or removed in a hole with that label, respectively. Similarly, a jump either increments or decrements the values N_4, N_5, and N_6 by one. Thus, for each label i, the value $T - N_i$ has the same even-odd parity before and after a jump is made.

A position class is described by the parity of each $T - N_i$. All boards for which these values are identical are in the same position class. As a jump does not change the parity of any of these values, a jump does not change a board's position class. My solver tests if the initial and goal states are in the same position class and if not, knows that the initial state is not solvable without needing to search.

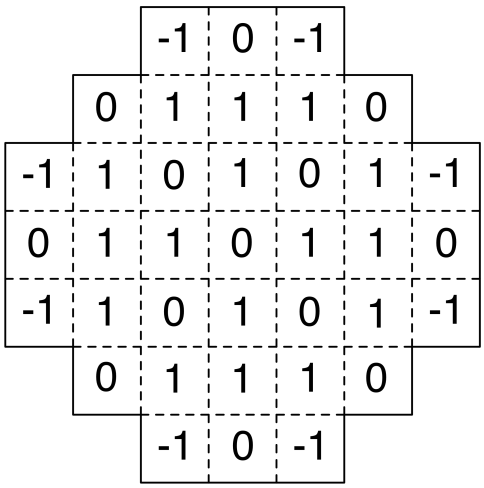

Figure 6.13: Pagoda values for the French board

A *pagoda function* can be used to prune certain states as unsolvable. To define one, each hole on the board is assigned a number subject to the following constraints: for any three horizontally- or vertically-adjacent holes with values x, y, and z we have that $x \leq y + z$ and $z \leq y + x$. The pagoda value of a board state is given by the sum of the values of every occupied hole. Since a single jump removes pegs from two adjacent holes and places a peg in a third consecutive hole, a board's pagoda value can only stay the same or decrease after a jump is made.

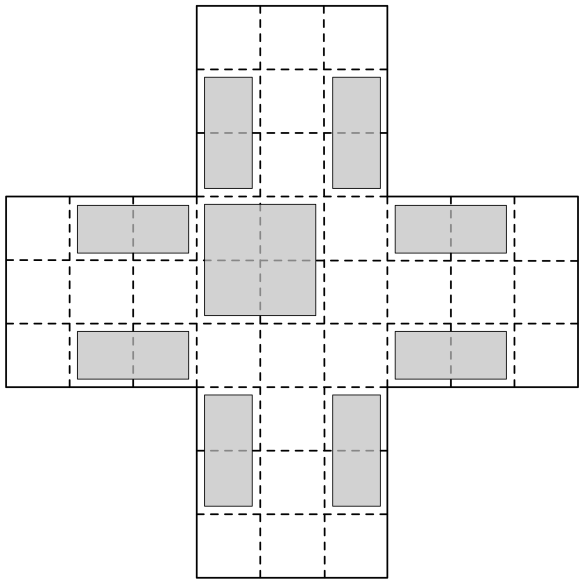

Figure 6.14: Example Merson regions on the Wiegleb board.

Thus, if we reach a state whose pagoda value is less than that of the goal, we know the state is a dead end and we cannot reach the goal. Figure 6.13 gives values for an effective pagoda function on the French board.

All pegs on a board are classified into four distinct *types*. A peg's type is defined by the parity of the row and column that it appears in, so any 2×2 square would contain one peg of each type. Since a jump moves a peg either two holes horizontally or vertically, the type of a peg never changes during a game. Thus, the number of pegs of a given type never increases and if a state has less of one peg type than the goal state does, then that state is unsolvable. Also note that pegs can only jump other pegs of certain types. In particular, peg types are divided into two classes: pegs that have both row and column of the same parity, and pegs that do not. Pegs in one class can only jump pegs in the opposite class. This fact is used by my solver's heuristic function.

6.10.2.2 Merson Regions

A *Merson region* is a region on a peg solitaire board that allows us to put a lower bound on the number of moves required to solve a board configuration. A Merson region is a contiguous region of the board which has the property that, if all holes in that region are occupied, a peg in that region can only be captured by a jump that originates in that same region. In other words, pegs in a fully-occupied Merson region cannot be captured by jumps made by pegs from outside that region. If a Merson region is fully occupied in the current state and not in the goal state, then a new move (rather than the continuation of a move) is required to remove a peg from that region. This property is used in my solver's heuristic. Figure 6.14 gives examples of Merson regions on the Wiegleb board.

6.10.3 Breadth-First Search By Jumps

One contribution I used in my solver is a novel method for representing levels in BFIDA* searches of peg solitaire. This technique is used by my unidirectional and bidirectional solvers and significantly improves duplicate detection.

The traditional implementation of BFIDA* does a breadth-first search, where all nodes on a level have the same g cost. In peg solitaire, however, this presents some problems. Nodes on a level are the result of a completed move from a node on the previous level. Generating all successors to a node thus involves a secondary search of all legal jump sequences, adding complexity. In addition, doing a secondary search of moves on different states may result in duplicate intermediate states. Detecting and pruning these duplicates is non-trivial. In addition, moves of different lengths remove different numbers of pegs. Thus, not all nodes on the same level of search have the same number of pegs and duplicate states appear at different levels of search. We must then either store multiple levels of the problem space for duplicate detection or pay the cost of additional, duplicate work.

In my solvers, however, each level consists of all nodes reachable by performing a single *jump* from some node on the preceding level. Generation of children is trivial and quick: the successors to a node on the next level are simply all nodes reachable by a single jump. In addition, all nodes at the same depth now have the same number of pegs, so my solver can trivially detect all duplicates by only looking at nodes on the current level.

The downside is that this involves additional bookkeeping complexity. Since not all nodes on the same level have been generated by the same number of moves, we need to store with each node its associated g cost. In addition, we need to store the identity of the last peg moved so that the solver can determine whether a jump made by a peg starts a new move. This imposes a small additional space overhead. Finally, care must be taken when deciding when to prune a node, since it may be possible to extend the move that generated a node and reduce its h cost. We cannot prune a node until the move that generated it is complete.

At the cost of some additional bookkeeping space, however, we can now guarantee that all duplicate nodes will occur at the same level of search. This greatly simplifies duplicate detection.

6.10.4 Improved Heuristic

I developed an improved heuristic for peg solitaire. Importantly, it is a consistent heuristic. It is composed of three component functions: $h_c(n)$, $h_t(n)$, and $h_m(n)$.

The first component, $h_c(n)$ counts the number of corner holes that are occupied in the current state but not the goal. Figure 6.15 labels corners using a "C" on the French board. Each such corner hole must be vacated to find a solution. Since pegs in those holes cannot be captured, they can only be removed from their hole with a move starting at that hole. Thus, $h_c(n)$ is a lower bound on the optimal

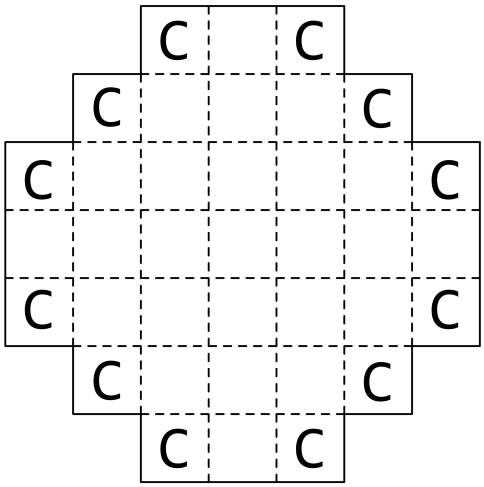

Figure 6.15: Corners on the French board.

solution cost. Note that $h_c(n)$ counts the number of moves required to remove pegs from certain holes, not to remove them from the board entirely.

$h_t(n)$ counts the number of moves required to remove pegs of certain *types* (see section 6.10.2.1) from the board. There are a maximum number of pegs of each type that can be removed from the board in a single move. For example, consider the peg type that can only occupy even-parity rows and columns on the English board. As shown in figure 6.16, a maximum of four such pegs can be removed by a single move. A total of 12 even-row, even-column pegs can be on the board at the same time: four in the holes jumped over in figure 6.16, and eight in the eight corner holes. Since a maximum of four such pegs can be removed in a single move, removing all twelve from the board would require at least three separate moves.

For each peg type, $h_t(n)$ computes the minimum number of moves needed to remove excess pegs of that type. It returns the maximum of those values. Since moves that vacate corner holes will also capture pegs, this conflicts with $h_c(n)$. To ensure that these functions are independent, I calculate $h_t(n)$ based only on peg types that cannot be captured by a move originating in a corner.

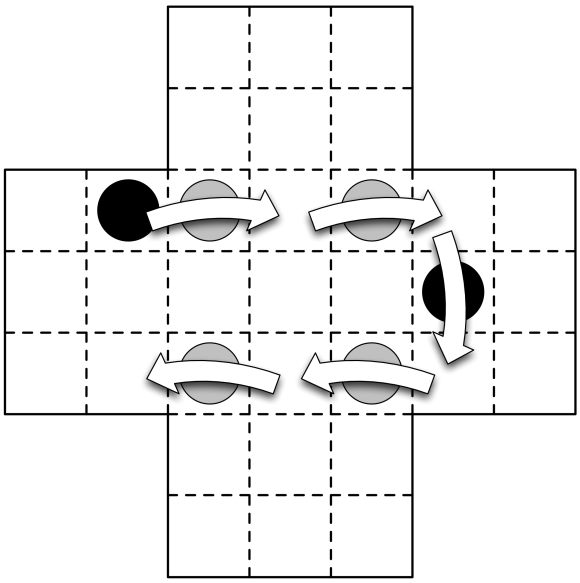

Figure 6.16: A move that removes four even-row, even-column parity pegs on the English board, shaded gray, the largest number removable in one move.

$h_m(n)$ counts the number of Merson regions (see section 6.10.2.2) that are fully occupied in the current state but not the goal. By definition, if every hole in a Merson region is occupied, the only way to remove a peg from that region is with a new move made by a peg from the same region. If a Merson region is fully occupied in the current state and not the goal, then one move is required to vacate that region in a solution path to the goal. $h_m(n)$ defines Merson regions on the board (excluding corners) and returns the number of regions filled in the current state but not the goal.

$h_t(n)$ and $h_m(n)$ can count moves made by the same peg, as a move vacating a Merson region might take pegs of any type. As such, $h(n)$ cannot return the sum of both values and must use the maximum of both. A peg moved from a corner hole, however, will not remove a peg tracked by $h_t(n)$ or $h_m(n)$, and so the value of $h_c(n)$ can be added to the value returned by these two functions. Thus, my complete heuristic function is $h(n) = h_c(n) + max(h_t(n), h_m(n))$.

This heuristic builds upon and improves the heuristic introduced by Bell in [Bel07]. It is also a consistent heuristic. Any move changing the value of $h_c(n)$ does not change the value of the other two functions. A single move can affect both $h_t(n)$ and $h_m(n)$, but I take the max of both. A move can reduce the value returned by each of these components by at most one, so $h(n)$ never changes by more than one for a given move. Thus, $h(n)$ is monotone and consistent.

6.10.5 Bidirectional BFIDA*

I developed a novel bidirectional heuristic search algorithm based on BFIDA*. It performs two independent BFIDA* searches, one forward from the initial state and one backward from the goal. I define my peg solitaire instances as a pair of a single initial state and a single goal state, but in a problem domain with multiple goal states the reverse search would be seeded with all of them. Since BFIDA* expands its nodes in a breadth-first order, it is particularly easy to implement using disk-based search techniques. My implementation uses delayed-duplicate detection (DDD, described in section 2.4) to allow it to solve the largest problem instances.

Both directions of search have an independent cutoff. The algorithm does iterations of BFIDA* in one direction at a time, completing an iteration in one direction before starting a new iteration in either direction. After an iteration is completed in one direction, the cutoff for that direction is increased. To keep work in each direction balanced, I choose whether to perform a forward or backward iteration next based on which direction had the fewest number of nodes expanded on the previous iteration.

As we do an iteration of BFIDA* with a given cutoff, we maintain a frontier of all nodes which have been expanded but had a child pruned for exceeding that cutoff. Coming from the opposite direction, we cannot reach the goal (or

initial) state without passing through this frontier. Since we are searching with a consistent heuristic, we can guarantee that we have found an optimal path to a node once it is expanded. By keeping only expanded nodes on the frontier, we guarantee that we have found the optimal path to every node on the frontier.

The frontier is stored on a level-by-level basis: each level of the frontier contains only nodes with the same number of pegs. To eliminate duplicate states on disk using DDD we efficiently sort each breadth-first layer on disk and then do a linear scan to detect duplicates. When expanding nodes, any node that has had a child pruned for exceeding the cost cutoff is written to a sorted file of nodes, containing only nodes from that breadth-first level. It would be possible to implement the hash-based duplicate detection technique discussed in section 2.4, instead, but I did not do so.

During search, we check each node selected for expansion against the opposing frontier. If the frontiers intersect, we have a candidate solution and we keep track of the lowest-cost solution found so far. Finding intersections between the frontiers can be efficiently done, even on disk. The current level of the breadth-first search is stored in a sorted file of nodes, and any nodes on the opposing frontier with the same number of pegs are also stored in a single sorted file. The file encoding the opposite search frontier is scanned in parallel with the file encoding the current breadth-first level, advancing each as appropriate and looking for duplicate nodes. Looking for intersections with the opposing search frontier thus only requires one additional file read per breadth-first level.

In addition, following from the ideas of BS* discussed in section 3.3, we need not explore the children of any node that has intersected the opposing frontier. Since our heuristic is consistent and we have expanded the same node from both directions, we must have an optimal path from the initial state to the goal state through this node. Thus, we cannot improve upon this solution by exploring children beneath it.

The fact that frontier intersections provide us with candidate solutions leads to one of the main benefits of this algorithm. If we are searching for a single optimal solution we can stop searching once we prove that the best solution found so far is optimal. Any time we expand a node that also exists on the opposite frontier we have a candidate solution, and, since this is the intersection of two searches coming from opposite directions, its cost will likely exceed the current f cutoff. In all of my experiments, I found an optimal solution on an iteration with a smaller-than-optimal cost cutoff. As soon as we *start* an iteration whose cutoff is the cost of the best solution found so far, we can stop as that solution has been proved optimal. This avoids performing the last BFIDA* search iteration.

Unidirectional BFIDA* has the disadvantage that it expands a large number of nodes with $f(n) = C^*$ [ZH06]. A search algorithm must expand all nodes with $f(n) < C^*$ to prove that that there are no paths cheaper than the optimal solution past through those nodes. However, it does not need to expand any nodes with $f(n) = C^*$ after it has found an optimal solution. Because BFHS is a breadth-first algorithm, it expands nodes in increasing order of g cost, and so does not find an optimal solution until it has generated all nodes with $g(n) < C^*$ on its last iteration. At this point, it has generated almost all nodes with $f(n) = C^*$.

By potentially avoiding the last iteration of BFIDA*, my bidirectional algorithm expands *no* nodes with $f(n) = C^*$, and has best-case tie-breaking properties. This is the caveat to my theory identified in section 5.7.2, and in peg solitaire allows my bidirectional heuristic search to be effective.

6.10.6 Bidirectional Constraint Propagation

In my bidirectional solver, I leverage the fact that the frontiers do not pass through each other to improve its ability to prune unsolvable states. As described in section 6.10.2.1, pagoda values and counts of pegs of a certain type are properties of

a puzzle that can never increase during play. This allows us to prove that certain states are unsolvable. For example, if the pagoda value of a state is less than that of the goal, then that state cannot be solved.

My bidirectional algorithm increases the pruning capability of these techniques by considering the pagoda and peg-type values of all states on the search frontiers, rather than just those of the initial and goal states. In forward search, any path from a node to the goal must pass through the reverse search frontier. Thus, the pagoda value of the current node must be at least as great as the minimum pagoda value of all nodes on the reverse frontier. This value may be higher than the pagoda value of the goal state by itself. The same technique can be used for peg-type constraints, as well.

When searching in reverse, we keep track of the minimum pagoda value and peg type counts of all nodes added to the final frontier. These minimum values are stored and used in forward search to prune unsolvable states. The same is done for nodes added to the search frontier in forward search.

Any node that violates any of these constraints cannot be part of a solution. These nodes can be discarded as soon as they are found to violate a constraint, and do not need to be stored on the search frontiers.

6.10.7 Distribution Of g Costs

I computed the fraction of nodes expanded with $g(n) <= C^*/2$ on the 35 solvable instances of the English, French, and Diamond(5) boards. Among these instances, the mean fraction of nodes expanded with $g(n) <= C^*/2$ was 6%. The instance with the smallest fraction expands 2% of its nodes with $g(n) <= C^*/2$, and the instance with the largest fraction expands 14%.

These numbers show that my peg solitaire heuristic is extremely weak, meaning we would predict that a bidirectional brute-force search will win. However, as I

show in the next section, my bidirectional heuristic search algorithm is in fact the state of the art. Section 6.10.9 discusses why this is the case.

6.10.8 Bidirectional Heuristic Search Tested

I compared my bidirectional heuristic search solver to the previous state of the art solvers, as well as my own implementation of a unidirectional heuristic search. My bidirectional heuristic solver is described in the previous section, and the previous state-of-the-art solvers are described in section 6.10.1. I also implemented unidirectional BFIDA* using my improved heuristic and doing breadth-first searches by jumps.

Tables 6.4, 6.5, and 6.6 summarize the results of these experiments on the English, French, and Diamond(5) boards. Each row is an instance on that board. The first column is the location of the empty hole in the initial state and the second column is the location of the final peg in the goal state. Peg locations are given as row/column coordinates on the smallest box that encloses the board, starting at (0, 0) in the top-left corner. For example, (3, 3) is the center-most hole on the English and French boards.

All experiments were run on a 3.33 GHz Intel Xeon with 48 GB of RAM. I find the same minimum path length as Bell in all cases.

Bell graciously provided me with his solver, which I used to regenerate his results on my machine. Recall that these results represent the state of the art on minimal-length peg solitaire solutions. The column labeled "Bell (2012)" in tables 6.5 and 6.6 shows the timing results of the brute-force bidirectional search algorithm used in [Bel12]. This was previously the strongest solver on these boards. Note that this column has two numbers for each instance on the Diamond(5) board. The first gives the timing numbers of the default algorithm. The second number, in parentheses, gives the timing numbers with stronger pagoda

Start hole	End peg	Bell (2007)	BFIDA*		BD-BFIDA*		BD-BFIDA* (no tightening)	
		Time	Time	Nodes	Time	Nodes	Time	Nodes
(3, 3)	(3, 3)	46.6s	† 3.0s	4.07	***2.9s***	3.90	4.0s	5.55
(0, 3)	(3, 3)	49.8s	† ***3.5s***	4.79	5.2s	6.90	7.8s	10.79
(0, 3)	(0, 3)	49.3s	27.6s	32.83	***3.2s***	4.41	4.5s	6.44
(0, 3)	(6, 3)	51.9s	27.7s	32.83	***3.2s***	4.41	4.5s	6.44
(3, 3)	(0, 3)	9.2s	7.0s	9.57	***0.3s***	0.50	0.4s	0.59
(2, 3)	(2, 3)	4.9s	12.5s	16.28	***1.3s***	1.84	1.9s	2.70
(0, 2)	(3, 2)	6.8s	5.0s	6.78	***0.3s***	0.38	***0.3s***	0.38
(0, 2)	(3, 2)	8.2s	6.7s	9.10	***0.5s***	0.73	***0.5s***	0.73
(1, 3)	(4, 3)	8.1s	5.4s	7.21	***1.4s***	1.65	***1.4s***	1.66
(1, 3)	(1, 3)	452.4s	† ***39.6s***	45.90	52.2s	62.68	55.0s	68.13
(2, 3)	(5, 3)	80.2s	† ***5.0s***	6.78	6.4s	8.78	6.3s	8.93
(0, 2)	(3, 5)	107.7s	† ***6.7s***	9.10	12.0s	16.46	13.0s	18.41
(0, 2)	(0, 2)	17.5s	13.3s	18.06	***1.9s***	2.73	***1.9s***	2.74
(2, 3)	(2, 0)	29.6s	19.8s	26.13	1.6s	2.21	***1.5s***	2.24
(0, 2)	(6, 2)	17.8s	13.3s	18.06	***1.9s***	2.73	***1.9s***	2.75
(1, 3)	(4, 0)	31.7s	20.9s	27.46	***1.8s***	2.57	***1.8s***	2.72
(2, 2)	(2, 2)	5.0s	4.0s	5.54	***1.4s***	1.98	***1.4s***	1.98
(1, 2)	(3, 3)	2.6s	2.3s	3.24	1.4s	1.69	***1.3s***	1.69
(1, 2)	(1, 2)	43.5s	16.1s	21.05	1.1s	1.54	***1.0s***	1.54
(2, 2)	(2, 5)	128.7s	49.3s	59.95	3.9s	5.56	***3.7s***	5.56
(1, 2)	(4, 5)	43.2s	16.1s	21.05	1.1s	1.54	***1.0s***	1.54
Total:		00:19:53	00:05:05		00:01:45		00:01:55	

Table 6.4: Timing values and numbers of nodes expanded for all solvable instances on the English board. Node counts are given in millions.

Start hole	End peg	Bell (2012)	BFIDA*		BD-BFIDA*		BD-BFIDA* (no tightening)	
		Time	Time	Nodes	Time	Nodes	Time	Nodes
(0, 2)	(0, 4)	9,804s	174s	191.08	16s	16.80	***15s***	17.46
(0, 2)	(3, 1)	5,515s	92s	100.95	***7s***	8.11	***7s***	8.11
(0, 2)	(3, 4)	3,825s	85s	92.11	***5s***	5.42	***5s***	5.43
(0, 2)	(6, 4)	9,908s	173s	191.08	***15s***	16.86	16s	17.49
(2, 3)	(4, 0)	7,530s	1,959s	1,744.32	***363s***	354.75	370s	386.90
(2, 3)	(1, 3)	4,033s	562s	521.45	107s	109.43	***105s***	109.62
(2, 3)	(4, 3)	2,410s	542s	495.58	80s	82.30	***79s***	83.00
(1, 3)	(2, 0)	5,666s	384s	396.32	***22s***	25.32	23s	27.13
(1, 3)	(5, 3)	3,125s	102s	106.49	***5s***	5.84	***5s***	5.84
(1, 3)	(2, 3)	2,031s	96s	100.43	***5s***	5.40	***5s***	5.41
Total:		14:57:27	01:09:29		00:10:25		00:10:30	

Table 6.5: Timing values and numbers of nodes expanded for all solvable instances on the French board. Node counts are given in millions.

Start hole	End peg	Bell (2012)	BFIDA*		BD-BFIDA*		BD-BFIDA* (no tightening)	
		Time	Time	Nodes	Time	Nodes	Time	Nodes
(1, 3)	(1, 5)	8,855s (703s)	3,013s	2,445.92	***63s***	61.57	484s	469.85
(4, 6)	(1, 5)	8,585s (1,215s)	6,202s	4,864.44	***272s***	250.68	1,064s	1,009.64
(1, 3)	(4, 2)	8,585s (1,272s)	414s	356.47	***74s***	72.23	164s	151.10
(4, 6)	(4, 2)	1,282s (1,268s)	472s	395.24	73s	70.59	***72s***	70.59
Total:		07:35:07 (01:14:16)	02:48:21		00:08:02		00:29:44	

Table 6.6: Timing values and numbers of nodes expanded for all solvable instances on the Diamond(5) board. Node counts are given in millions. Parenthesized timing numbers are for Bell's solver with Bell's manually found pruning constraints.

constraints. In his work, Bell manually found that one could derive stronger, admissible pagoda constraints than simply using the pagoda constraints of the start and goal states. This is analogous to the stronger constraints derived through my technique described in section 6.10.6.

The column labeled "Bell (2007)" in table 6.4 shows the timing results of the BFIDA* algorithm of [Bel07]. This solver is most effective on the English board, and so I only show results there. I have not generated results for the other boards, but according to Bell they do not improve on a brute-force search. His solver only performs a single iteration of BFIDA* at a time, so the numbers given are for the final iteration with the optimal cost cutoff. A complete run would require slightly more time for the initial iterations.

The column labeled "BFIDA*" gives the numbers for time spent and nodes expanded by my implementation of unidirectional BFIDA*. As mentioned earlier, with a sufficiently weak heuristic, unidirectional BFIDA* can find an optimal solution on an iteration with a cutoff less than the optimal solution cost. And, indeed, this occurs in five instances. These instances are marked with a dagger. Ignoring these cases, my BFIDA* solver compares favorably to Bell's and generates comparable or slightly faster results in most cases.

My heuristic results in significantly better performance than the heuristic given in [Bel07] on the French and Diamond(5) boards. While the previous heuristics barely improve upon brute-force search, my implementation can easily solve all of these instances using simple unidirectional BFIDA*. In the case of the French board, my unidirectional BFIDA* implementation is often over an order-of-magnitude faster than the previously-optimal solver. On the Diamond(5) board, my solver does not benefit from Bell's manually-derived tightened constraints, but still comfortably beats his default solver.

The column labeled "BD-BFIDA*" gives the numbers for time spent and nodes expanded by my bidirectional BFIDA* algorithm. These are my strongest results.

In all cases, my solver was able to find an optimal solution on an iteration before the last. My solver is significantly faster than the previous state-of-the-art, generally beating it by one or two orders of magnitude. In the most extreme example, the third problem of the French board, my solver is able to reduce the solution time from over an hour down to five seconds of run time. While some of this improvement is due to the improved heuristic, my bidirectional solver is itself able to significantly improve over my own unidirectional approach, often by an order of magnitude.

As noted previously, my unidirectional solver occasionally finds optimal solutions before the last iteration (on those instances marked with a dagger). Significantly, unidirectional search was only able to outperform my bidirectional approach in these cases, due to the overhead in the latter of additional iterations of backward search. This emphasizes that the main value of my algorithm is from improved tie breaking by avoiding performing the last iteration of a BFIDA* search.

The column labeled "BD-BFIDA* (no tightening)" gives the numbers for time spent and nodes expanded by bidirectional BFIDA*, but without propagation of pagoda and peg-type constraints. That is, I only place constraints on nodes using the pagoda and peg-type values of the initial and goal states, not values calculated from the search frontiers. In most problem instances this technique does not affect search much, producing only modest reductions in node counts at the cost of slightly more expensive node expansions. On three of the Diamond(5) instances, however, this technique was able to find the tighter constraints that Bell enforces manually. This dramatically speeds up run time. Since the technique usually has little negative impact and has the capacity to dramatically improve performance, I believe that it should be used in general.

Finally, I solved all instances of the Wiegleb board, pictured in figure 6.14. Bell's bidirectional brute-force solver took three months to solve all 35 solvable

instances on this board. My solver, meanwhile, took just a week and a half. I benefited strongly from disk-based techniques on this problem: on larger instances my solver uses up to a terabyte of disk storage. These instances could not be solved by an in-memory solver.

6.10.9 Why Is Bidirectional Heuristic Search Effective?

The solver I introduced for peg solitaire is in fact a bidirectional heuristic search algorithm. This is despite the predictions in Chapter 5 that bidirectional heuristic search will rarely outperform both unidirectional heuristic search or bidirectional brute-force search. There is an obvious question as to why this happens.

The reason is that the peg solitaire domain illustrates two of the caveats discussed in section 5.7. The first is that forward and reverse searches in peg solitaire are not balanced. Reverse searches are significantly more expensive than forward searches. This is because, on an orthogonal board, there are a maximum of four moves that can be made on a board with a single hole, as in the starting state. There are a much larger number of moves that can be made that leave only a single peg left on the board, as in the goal state. Because of this, the branching factor of nodes near the goal state is significantly higher than those of nodes near the start state, and a reverse search is more expensive.

On the English board, reverse unidirectional BFIDA* searches expand on average 47 times as many nodes as forward searches to solve a problem instance. Figure 6.17 shows the distribution of g costs of nodes expanded in one solvable instance on the English board.

Because of the very large disparity in search costs of forward and reverse searches, we would not expect a bidirectional search to be effective. In figure 6.17, the reverse search expands as many nodes with $g(n) <= 7$ as the forward search expands in total. In other words, a balanced bidirectional search of this problem

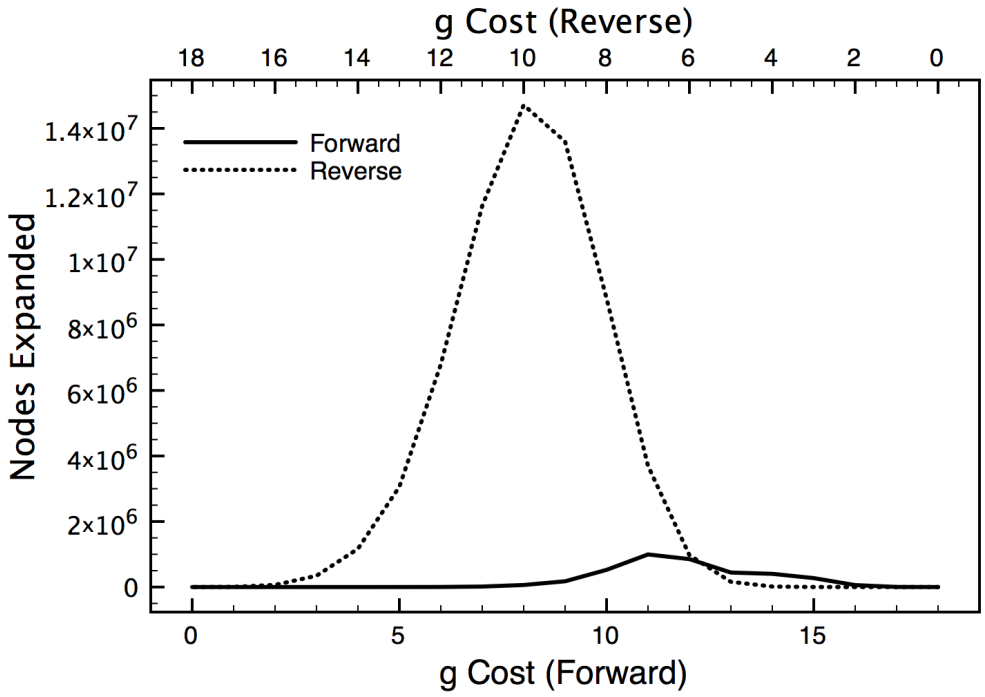

Figure 6.17: Distribution of g costs of nodes expanded in forward and reverse searches of a peg solitaire instance. The instance shown is the first instance of table 6.4, with an optimal solution depth of 18.

instance would meet very near the goal state and could only possibly prevent the expansion of a very small number of nodes over a forward search. This would come at the expense of significant bookkeeping overhead.

Bidirectionality helps in peg solitaire for two reasons, however. The first is that it allows us to implement the constraint-tightening techniques discussed in section 6.10.6. On the instances of the Diamond(5) board, in particular, these techniques result in significant performance improvements. These techniques cannot be implemented in a unidirectional solver. These techniques are very similar to the idea of a front-to-front heuristic evaluation and arguably fall outside the scope of my theory. However, another factor contributes to the effectiveness of bidirectional heuristic search in this domain.

Secondly, and more significantly, bidirectionality improves the tie-breaking performance of my algorithm. Peg solitaire problem spaces can be very large and have many duplicates; this is why my solver uses disk-based techniques. The specific algorithm I used as the basis of my solver, BFIDA*, has very bad tie-breaking properties: it expands virtually all nodes in a search graph with f cost equal to C^* [ZH06]. These nodes are all expanded in the last BFIDA* iteration.

The benefit of my bidirectional approach is that it can find an optimal solution *before* the last iteration. It can then terminate the last iteration without expanding nodes, thereby not expanding *any* nodes with f cost equal to C^*. My solver never finds a solution while doing a reverse search. In fact, the highest cutoff ever used in a reverse search is usually significantly lower than the optimal solution cost. The benefit of this reverse search, rather, is that it searches deep enough that the two frontiers can intersect and find the optimal solution before the last iteration of search.

As confirmation that improved tie breaking is the main benefit provided by my bidirectional algorithm, consider the five instances in table 6.4 where my unidirectional BFIDA* solver found an optimal solution on an iteration before the last.

These numbers are marked with a dagger. In these cases the unidirectional solver terminates without completing the last iteration and expands no nodes with f cost equal to C^*. And in four of these five cases, where the unidirectional solver has best-case tie breaking, it outperforms my bidirectional solver.

The lone exception is the first instance, where my bidirectional solver still outperforms the unidirectional search. In this case, the constraint-tightening techniques of section 6.10.6 allow the bidirectional solver to win. When these are turned off, as shown in the last column, my unidirectional solver wins.

This makes it clear that there are two primary benefits of my solver: strengthening the domain-specific pruning constraints and improving tie breaking behavior, caveats discussed in section 5.7. Otherwise, due to the very unbalanced difficulty of searches in either direction, we would expect a unidirectional heuristic search to be most effective in this domain.

CHAPTER 7

Synthesis Of Results

This chapter summarizes the experimental results of the previous section and provides high-level analysis.

7.1 g Cost Distributions Among Many Domains

For every domain studied in Chapter 6, I generated the distribution of nodes expanded with $g(n) <= C^*/2$. The intent of this experiment is to test the predictions of my theory about whether the strength of the heuristic predicts the effectiveness of unidirectional heuristic search compared to bidirectional brute-force search.

Table 7.1 summarizes these experiments. Each column shows the fraction of nodes expanded with $g(n) <= C^*/2$, except for sequence alignment, where it shows the fraction of nodes expanded more shallowly than the depth of the solution midpoint. The first column is the mean fraction among all instances, and the last two are the value for the instances with the minimum and maximum fractions.

7.1.1 Domains with Weak Heuristics

In peg solitaire and Towers of Hanoi, a mean of 6% and 31% of the nodes expanded have shallower depth than the midpoint, respectively; as such, a bidirectional brute-force search should outperform a unidirectional heuristic search. As I showed in section 6.9, bidirectional brute-force search does indeed outperform a unidirectional heuristic search on the four-peg Towers of Hanoi.

Domain	Mean	Min	Max
Top Spin	0.98	0.86	1.00
24 puzzle	0.96	0.80	1.00
15 puzzle	0.93	0.63	0.99
Rubik's Cube	0.90	0.84	0.98
Sequence Alignment	0.61	0.53	0.66
Pancake problem	0.57	0.49	0.66
Vehicle Navigation	0.53	0.04	0.96
Towers of Hanoi	0.31	0.13	0.59
Peg solitaire	0.06	0.02	0.14

Table 7.1: Fraction of nodes expanded with with $g(n) <= C^*/2$ for many instances of several search domains.

In peg solitaire, however, my state-of-the-art solver is in fact a bidirectional heuristic search algorithm. This is because peg solitaire is an unusual domain and illustrates several of the caveats of my theory. In particular, forward searches in peg solitaire are significantly cheaper than reverse searches, and because of the very large number of duplicates in the domain, the preferred algorithm to solve it, BFIDA*, has very bad tie-breaking behavior. Because of these issues it turns out that a bidirectional heuristic search is effective in this domain. This was discussed thoroughly in section 6.10.9.

7.1.2 Domains with Strong Heuristics

In all of the remaining domains the majority of nodes expanded have $g(n) <= C^*/2$, so I expect unidirectional heuristic search to outperform a bidirectional brute-force or heuristic search. And, indeed, I am unaware of a front-to-end bidi-

rectional search in the literature that outperforms the best unidirectional search on these domains.

The one exception is in road navigation, where only a small majority of nodes have $g(n) <= C^*/2$ and front-to-end bidirectional heuristic search has a small performance advantage over A* search done in the forward direction. However, the performance of both algorithms are roughly equivalent, as we would expect given the strength of the heuristic. In this domain, bidirectional heuristic search outperforms both unidirectional heuristic search and bidirectional brute-force search in 15% of the instances I tested, and provided an 8% reduction in nodes expanded. These are examples of the pathological examples discussed in section 5.6. As predicted, these cases are uncommon and provide minimal performance improvement.

7.2 Bidirectional Heuristic Search Tested

In the domains where it was feasible, I implemented bidirectional heuristic search and compared its performance to unidirectional heuristic search. The results of these experiments are summarized in table 7.2. Column two shows the mean reduction in nodes expanded by unidirectional heuristic search over bidirectional heuristic search. Column three shows the mean reduction in time of unidirectional search. Column four shows the percentage of the instances in which bidirectional heuristic search outperforms unidirectional heuristic search.

In road navigation, A* searches can be done cheaper in one direction than the other, as discussed previously. To account for this, I also compared BS* to an A* search in the cheaper of the two directions for each instance. This data is shown in the row labeled "Road Navigation (Best A*)".

As predicted, in the first four domains, where a strong majority of nodes are expanded with $g(n) <= C^*/2$, a unidirectional heuristic search outperforms a bidirectional heuristic search. In the case of road navigation, where only approx-

Domain	Unidir. Node Reduction	Unidir. Time Reduction	Bidir. Wins
15 Puzzle	20%	37%	18%
Road Navigation (Best A*)	20%	27%	19%
Rubik's Cube	15%	74%	29%
Pancake Problem	11%	32%	16%
Sequence Alignment	9%	11%	0%
Vehicle Navigation	-9%	-5%	57%
Peg Solitaire	-90%	-92%	89%

Table 7.2: Unidirectional heuristic search vs bidirectional heuristic search in six domains.

imately half of nodes are expanded with $g(n) <= C^*/2$, BS* provides a small reduction in time and nodes expanded over A*. However, when BS* is compared to an A* search in the cheaper of the two directions, A* outperforms it on most instances.

In the case of peg solitaire, bidirectional heuristic search outperforms unidirectional heuristic search because of its improved tie breaking properties and ability to automatically improve pruning constraints.

On most of the domains, bidirectional heuristic search is the stronger algorithm on some instances. In the 15 puzzle, the pancake problem, peg solitaire, and the Rubik's Cube, this only happens because of tie breaking. Bidirectional heuristic search expands at least as many nodes with $f(n) < C^*$ as does unidirectional heuristic search. In other words, bidirectional heuristic search only expands fewer nodes by not expanding some nodes whose f cost is equal to C^*; this is the caveat identified in section 5.7. In peg solitaire, this happens reliably and provides a large performance improvement. In the other domains, there are not many of these cases and they do not significantly help BS*'s performance.

In the case of road navigation, BS* outperforms A* in both directions 19% of the time. Among these cases, there are many instances in which BS* actually outperforms *both* A* and a bidirectional brute-force algorithm. These are pathological cases, and were discussed in section 6.8.3.

These results conform overall to the predictions of my theory. When the heuristic is particularly strong, unidirectional heuristic search unambiguously outperforms bidirectional heuristic search. With a medium-strength heuristic in road navigation, there are a few pathological cases that provide marginal benefit, but overall bidirectional heuristic search is comparable to unidirectional heuristic search.

The only clear exception to my theory is peg solitaire, where bidirectional heuristic search strongly outperforms unidirectional heuristic search. This is because the bidirectional algorithm provides reliably better tie-breaking behavior.

7.3 Work Spent Proving Optimality

In the domains where I implemented bidirectional heuristic search, I calculated the point at which optimal solutions are found. In their experiments, Kaindl and Kainz found that BHPA tended to find an optimal solution very early and then spent most of its search proving optimality [KK97]. They tested BS* on instances of the 15 puzzle, finding that the optimal solution was found on average after 22.4% of the nodes in an instance has been generated. This observation is often erroneously given in the literature as Kaindl and Kainz's explanation for why bidirectional heuristic search is ineffective [ES12, FMS10, LEF12]. These experiments test this claim.

In my experiments with bidirectional heuristic search with strong heuristics, however, I found that an optimal solution was actually found quite late: on average after 90% of the nodes had been generated in the 15 puzzle, after 86% had

been generated in the pancake problem, after 96% had been generated in Rubik's Cube, and after 99% had been generated in sequence alignment. This is the exact opposite of Kaindl and Kainz's observation.

In road navigation, where the heuristic is *not* strong (only approximately 50% of nodes are expanded with $g(n) <= C^*/2$), an optimal solution is found much earlier, after approximately 42% of all nodes have been generated. There is very high variance in this number, however. At the earliest, the solution was found after 11% of nodes had been generated and at the latest after 90% had.

These results make sense. The heuristics I used in my experiments are much stronger than those used by Kaindl and Kainz. In their experiments on the 15 puzzle, they used the Manhattan distance heuristic, while my searches on the 15 puzzle used much stronger PDB heuristics. The heuristics used in the other domains I tested are similarly strong. The effect of a heuristic is to prevent the expansion of nodes with high g cost, and there are thus fewer opportunities for frontier intersections. When a weaker heuristic is used, such as in the road navigation problem, there are more nodes expanded with high g cost and the two frontiers have more opportunities to intersect.

This shows that Kaindl and Kainz's observations of when an optimal solution is found in bidirectional heuristic search do not explain the algorithm's ineffectiveness.

This result also reinforces the expectation that pathological cases such as the one in figure 5.6 will rarely occur in practice. If both frontiers tend to not interact until very late in search, then frontier intersections will usually not be able to prevent the expansion of many nodes in either direction. When they do occur, it will be in domains like road navigation where the heuristic is not particularly strong.

7.4 Confirmation of Explanatory Theory

The prediction of my theory is that bidirectional heuristic search will rarely be useful. This is borne out by my experiments. Of the 3,635 problem instances I tested in section 7.2, bidirectional heuristic search outperformed unidirectional heuristic search only 767 times, and outperformed unidirectional heuristic search in the cheaper of the two directions only 124 times. The node reduction afforded by bidirectional heuristic search in these instances is also quite small.

As predicted, there are very few instances in which bidirectional heuristic search expands fewer nodes with $f(n) < C^*$ than a unidirectional heuristic or bidirectional brute-force search. This happens in 77 of the 3,635 instances I tested in section 7.2, and only reduces the number of nodes expanded by approximately 8%.

When bidirectional heuristic search does outperform the other algorithms, it is usually due to improved tie breaking and, except in the case of peg solitaire, this is not predictable and provides only minor improvements.

There are a few instances in road navigation where bidirectional heuristic search provides benefit beyond just tie breaking. As my theory predicts, these occur in domains with a medium strength heuristic and do not provide a significant performance improvement.

My theory predicts that when a heuristic is strong and a strong majority of nodes are expanded with g cost less than $C^*/2$, then a unidirectional heuristic search is the preferred algorithm. This is borne out by my experiments. Conversely, it predicts that when a heuristic is weak and a majority of nodes are expanded with g cost greater than $C^*/2$, then a bidirectional brute force search will dominate. This is also borne out by my experiments.

CHAPTER 8

Miscellaneous Observations

In the course of my research on bidirectional heuristic search, I made two other interesting observations. While these have both been tangentially mentioned in the literature, no paper has dealt with them directly.

8.1 Inconsistency of 15 Puzzle PDBs

In developing my bidirectional solver for the 15 puzzle, I observed that the standard additive PDBs used in this domain are inconsistent. This is because the standard technique with these PDBs is to not store the location of the blank tile, which loses information. When generating a PDB for a given set of tracked tiles it is possible for the same configuration of tiles to occur multiple times but with the blank in a different location—this may result in different heuristic estimates for the same configuration of tracked tiles. The standard practice is to save space by "compressing" away the blank: ignoring the blank, mapping all such configurations to a single entry in the PDB, and storing the minimum heuristic value among them.

As a result of this loss of information, two adjacent states in the problem space can map to PDB entries whose value differs by more than one. This makes the heuristic inconsistent. Thus, it cannot be used as a heuristic in a bidirectional search if we want to prevent the two search frontiers from passing through each other.

The fact that compressed PDBs are inconsistent has been noted in the literature [FKM07, KF02]. It has also been briefly noted in the literature that the 15 puzzle heuristic is inconsistent [BK10]. However, it appears that this fact is not widely known. For example, [FZH11] deals with the topic of searching with inconsistent heuristics and uses the 15 puzzle as one of its test domains. It overlooks the fact that the existing PDBs are inconsistent and in their experiments they additionally compress the PDBs in an attempt to make them inconsistent.

As my experiments require a consistent heuristic, I avoided this problem by using PDBs that do not compress away the location of the blank. This greatly increases the memory requirements of the PDBs but reduces the overall node expansions in a given search.

8.2 Search Imbalance In The 24 Puzzle

In my experiments I conducted forward and reverse searches on the 24 puzzle. For the same problem instances, the time required to solve the same instance could be dramatically harder in one direction than the other. In one problem instance, a reverse search is over two orders of magnitude slower than a forward search.

This property has been observed before in the literature, albeit indirectly. Dual search is a search technique in which permutation problems can be solved by searching in both the standard problem space and a "dual" problem space where variables and values are transposed. It was observed that searches in this dual space are often significantly different in cost than searches in the original space [ZFH08]. It was later discovered that the dual search formulation is in effect a reverse search in the original problem space [FMS10], and so this result implies that forward and reverse searches are of different difficulty. However, this connection has not been explicitly made. More importantly, I conducted experiments to try to determine *why* there is such a great imbalance in search.

I eliminated several potential explanations of the cause of this imbalance. The most obvious explanation, which is not covered by the data in [ZFH08], is that it is simply due to chance. The algorithm used, IDA*, finds an optimal solution in the last of several iterations of depth-first search. If it is lucky, it can find an optimal solution very early in this iteration. One possibility is that forward and reverse searches were finding solutions at different points in this last iteration, and the imbalance observed is simply random variation due to tie-breaking. I eliminated that possibility by requiring both directions of search to find *all* optimal solutions. Doing so requires that the last search iteration be completed, and eliminates the effect of tie-breaking. The search imbalance remains even in this case, showing that random variation does not explain the difference.

The second possibility is that the imbalance is due to different brute-force branching factors in the forward and reverse directions. This could result in a much larger brute-force search tree in one direction and hence require more work. I accounted for this possibility by considering only problem instances where the blank is in the same location in both the initial and goal states. In these cases, the forward and reverse branching factors are identical. Considering only these cases, the difference in search difficulty remains, and so this does not explain the imbalance.

The only remaining possibility is that the heuristics used are stronger in one direction than the other. One possible reason for this is due to different interactions of tiles tracked in the forward and reverse pattern databases. The heuristic used in the 24 puzzle consists of four pattern databases, each of which tracks tiles that end up in a set of locations in the goal state. The configuration of these pattern databases used for the standard goal state is shown in figure 6.3. The final locations of tiles tracked in the forward databases are the same as the final locations of tiles tracked in the reverse databases. However, the specific *tiles* tracked are different. Consider a forward and reverse pattern database that track tiles which end up in

the same locations in their respective goals. For a given problem instance, because these databases track different tiles, the interactions of those tiles will be different.

As a result, the distribution of h costs returned by these pattern databases is different in the forward direction than the reverse. In one direction of search, a pattern database may return many high values deep in search, allowing us to prune nodes. In the reverse direction, these high values may be returned early in search, where they will be added to low g costs and not allow us to prune many nodes. As a result, the heuristic can allow us to prune more nodes in one direction than the other, explaining the imbalance of search difficulty.

Given that one direction of search is often significantly cheaper than the other, and that which direction is cheaper is not predictable, it is tempting to ask if we can leverage these properties to create a faster solver than a simple unidirectional search. One such approach is suggested in section 5.7.3: we could do forward and reverse searches on the same problem instance in parallel and terminate when the faster of the two searches finds a solution. This incurs a penalty of doing twice as much work as the cheapest direction of search would do by itself. However, if the more expensive of the two searches takes more than twice as long as the cheaper on average, then this approach will do less work than a simple unidirectional search.

This is because the cheaper of the two directions cannot be predicted in advance, and so over a large number of instances the work done by a simple unidirectional search will be the average of the work done independently by forward and reverse searches of the same instance. If the cheaper of the two directions is less than half of the average cost, then this simple bidirectional approach will be faster than a unidirectional search.

In the 24 puzzle, however, the average number of nodes expanded by a simple unidirectional heuristic search is fewer than would be expanded in this proposed bidirectional search. In the 100 instances I tested, a unidirectional heuristic search expands 7.4×10^{13} nodes, while the bidirectional approach would expand $9.9 \times$

10^{13}, or 33% more. If both directions of search were balanced, this bidirectional technique would expand exactly twice as many nodes as a unidirectional heuristic search. Unfortunately, while the two directions or search are greatly unbalanced, they are not unbalanced enough for this technique to be effective.

CHAPTER 9

Conclusions And Future Work

9.1 Summary of Contributions

I presented an intuitive theory explaining the ineffectiveness of front-to-end bidirectional heuristic search. I showed that both unidirectional heuristic search and bidirectional brute-force search provide the same improvement over a unidirectional brute-force search: they prevent the expansion of nodes with high g cost. With a weak heuristic, a bidirectional heuristic search expands no fewer nodes than a bidirectional brute-force search. With a strong heuristic, a bidirectional heuristic search expands *more* nodes than a unidirectional heuristic search.

I also showed that there can be no general proof that bidirectional heuristic will never be effective, by showing that there exist pathological cases where it outperforms both unidirectional heuristic and bidirectional brute-force search. The best that can be hoped for is an intuitive theory, like the one I provide, or a proof that it can only be effective in restricted circumstances. I also showed that such pathological cases do occur in practice, finding real-world examples in the road navigation domain. However, bidirectional heuristic search only shows a very modest performance improvement in these cases.

I implemented a new state-of-the-art solver for the general form of the four-peg Towers of Hanoi, with arbitrary initial and goal states. I found that, as my theory predicts, a bidirectional brute-force search outperforms a unidirectional heuristic search in this domain.

I also implemented a new state-of-the-art solver for peg solitaire, which uses a bidirectional heuristic search. The peg solitaire domain illustrates two caveats of my theory that make bidirectional heuristic search effective. There is a very large imbalance in the cost of doing forward and reverse searches, and due to the large size of the search space and large number of duplicate states, a unidirectional solver will have very bad tie-breaking behavior. However, my solver uses bidirectional search to achieve best-case tie-breaking, and can significantly outperform a unidirectional heuristic search.

Finally, I made two miscellaneous observations relevant to bidirectional heuristic search. I found that the standard pattern databases used in the 24 puzzle are inconsistent. While this has been noted in the literature before, it appears to not be widely known and has been overlooked in papers. I also found that forward and reverse searches of the 24 puzzle tend to have very different costs. A previous result in the literature noted this effect using a different search algorithm, but did not describe it in terms of forward and reverse searches. In addition, I conducted the first experiments to explain this behavior, determining that it is due to an imbalance in the values returned by heuristics in both directions.

9.2 Ideas For Future Work

The counterexample shown in section 5.6 shows that a general proof of the ineffectiveness of front-to-end bidirectional heuristic search is unlikely. However, it may be possible to prove its ineffectiveness in more restricted cases. For example, maybe front-to-end bidirectional heuristic search can only be effective on graphs that have certain properties. It may then be possible to prove the technique is ineffective on domains that lack this property.

My explanatory theory explains the ineffectiveness of front-to-end bidirectional heuristic search by classifying heuristics as “weak” or “strong”. This classification

relies on the average number of nodes expanded with g cost less than $C^*/2$ in large numbers of unidirectional heuristic searches of a problem domain. To classify a heuristic requires implementing a heuristic search in a domain and running a very large number of tests, which is impractical if trying to determine whether unidirectional heuristic search or bidirectional brute-force search should be used. It would be useful to develop methods to predict which of these two techniques will be more effective without having to actually implement both algorithms.

9.3 Conclusion

Front-to-end bidirectional heuristic search has had a long and influential history. While never itself a very effective algorithm, its introduction in 1971 led directly to the development of many other important algorithms that are still in use today. The fact that it is still discussed is a testament to the simplicity and power of the idea.

This simplicity belies its ineffectiveness, as it seems intuitively obvious that combining two powerful techniques should only improve over either individually. Yet this intuition is wrong, and the work in this dissertation helps contribute to an understanding of why this is so.

While use of the technique has never taken off, neither has it faded from awareness. I hope that the work of this dissertation helps future researchers focus their work on more productive avenues.

References

[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*, pp. 141–145, 2004.

[Bea85] John Beasley. *The Ins And Outs Of Peg Solitaire*. Oxford University Press, 1985.

[Bel07] George I Bell. "Diagonal Peg Solitaire." *Integers: Electronic Journal Of Combinatorial Number Theory*, **7**(G1):20, 2007.

[Bel12] George I Bell. "George Bell's Peg Solitaire Page." `http://home.comcast.net/ gibell/pegsolitaire`, January 2012.

[BK10] Teresa Maria Breyer and Richard E Korf. "1.6-Bit Pattern Databases." In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010*, 2010.

[BK11] Joseph K Barker and Richard E Korf. "Solving 4x5 Dots-And-Boxes." In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011*, 2011. Extended abstract.

[BK12a] Joseph K Barker and Richard E Korf. "Solving Dots-And-Boxes." In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[BK12b] Joseph K Barker and Richard E Korf. "Solving Peg Solitaire with Bidirectional BFIDA*." In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[BK15] Joseph K Barker and Richard E Korf. "Limitations of Front-To-End Bidirectional Heuristic Search." In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[CS96] Joseph C Culberson and Jonathan Schaeffer. "Searching with Pattern Databases." In *Advances in Artificial Intelligence, 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence, AI 1996, Proceedings*, pp. 402–416, 1996.

[Dav01] Aaron Davidson. "A Fast Pruning Algorithm for Optimal Sequence Alignment." In *2nd IEEE International Symposium on Bioinformatics and Bioengineering, Proceedings*, pp. 49–56, 2001.

[Dij59] Edsger W Dijkstra. "A Note on Two Problems in Connexion with Graphs." *Numerische Mathematik*, **1**(1):269–271, 1959.

[DK07] P Alex Dow and Richard E Korf. "Best-First Search for Treewidth." In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pp. 1146–1151, 2007.

[DN94] John F Dillenburg and Peter C Nelson. "Perimeter search." *Artificial Intelligence*, **65**(1):165–178, 1994.

[DS78] Margaret O Dayhoff and Robert M Schwartz. "A model of evolutionary change in proteins." In *Atlas of Protein Sequence and Structure*, 1978.

[ES12] Stefan Edelkamp and Stefan Schrödl. *Heuristic Search: Theory and Applications*. Academic Press, 2012.

[FKM07] Ariel Felner, Richard E Korf, Ram Meshulam, and Robert C Holte. "Compressed Pattern Databases." *Journal of Artificial Intelligence Reseach*, **30**:213–247, 2007.

[FMS89] Amos Fiat, Shahar Moses, Adi Shamir, Ilan Shimshoni, and Gábor Tardos. "Planning and Learning in Permutation Groups." In *30th Annual Symposium on Foundations of Computer Science*, pp. 274–279. IEEE, 1989.

[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*, 2010.

[Fra41] J S Frame. "Solution to Advanced Problem 3918." *American Mathematical Monthly 48*, pp. 216–217, 1941.

[FZH11] Ariel Felner, Uzi Zahavi, Robert Holte, Jonathan Schaeffer, Nathan Sturtevant, and Zhifu Zhang. "Inconsistent Heuristics in Theory and Practice." *Artificial Intelligence*, **175**(9):1570–1603, 2011.

[GH05] Andrew V Goldberg and Chris Harrelson. "Computing the Shortest Path: *A** Search Meets Graph Theory." In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pp. 156–165, 2005.

[GP79] William H Gates and Christos H Papadimitriou. "Bounds for Sorting by Prefix Reversal." *Discrete Mathematics*, **27**(1):47–57, 1979.

[Gra15] GraphHopper Contributers. "Graphhopper Route Planner." `https://graphhopper.com/`, 2015.

[GSS08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. "Contraction Hierarchies: Faster and Simpler Hierarchical

Routing in Road Networks." In *Experimental Algorithms, 7th International Workshop, WEA 2008, Proceedings*, pp. 319–333, 2008.

[Hel10] Malte Helmert. "Landmark Heuristics for the Pancake Problem." In *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010*, 2010.

[HH92] Steven Henikoff and Jorja G Henikoff. "Amino Acid Substitution Matrices from Protein Blocks." *Proceedings of the National Academy of Sciences*, **89**(22):10915–10919, 1992.

[Hin97] Andreas M Hinz. "The Tower of Hanoi." *Algebras and Combinatorics: Proceedings of ICAC97*, pp. 227–289, 1997.

[HNR68] P E Hart, N J Nilsson, and B Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *Systems Science and Cybernetics, IEEE Transactions on*, **4**(2):100–107, July 1968.

[Hol10] Robert C Holte. "Common Misconceptions Concerning Heuristic Search." In *Proceedings of the Third Annual Symposium on Combinatorial Search, SOCS 2010*, pp. 46–51, 2010.

[KF02] Richard E Korf and Ariel Felner. "Disjoint Pattern Database Heuristics." *Artificial Intelligence*, **134**(1):9–22, 2002.

[KK97] Hermann Kaindl and Gerhard Kainz. "Bidirectional Heuristic Search Reconsidered." *Journal of Artificial Intelligence Research*, **7**:283–317, 1997.

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

[Kor08] Richard E Korf. "Linear-Time Disk-Based Implicit Graph Search." *Journal of the ACM*, **55**(6):26, 2008.

[KRE01] Richard E Korf, Michael Reid, and Stefan Edelkamp. "Time Complexity of Iterative-Deepening-A*." *Artificial Intelligence*, **129**(1):199–218, 2001.

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

[Lam89] Ferdinand Lammertink. "Puzzle or game having token filled track and turntable.", October 3 1989. US Patent 4,871,173.

[LEF12] Marco Lippi, Marco Ernandes, and Ariel Felner. "Efficient Single Frontier Bidirectional Search." In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012*, 2012.

[Loy59] Sam Loyd. *Mathematical Puzzles of Sam Loyd*. Dover, New York, 1959. Compiled and edited by Martin Gardner.

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

[New80] Allen Newell. "Reasoning, Problem Solving and Decision Processes: The Problem Space as a Fundamental Category." *Attention and Performance*, **VIII**:693–718, 1980.

[Ope15] OpenStreetMap Contributers. "OpenStreetMap." `https://www.openstreetmap.com/`, 2015.

[Pea84] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[Poh71] Ira Pohl. "Bi-directional search." *Machine Intelligence*, **6**:127–140, 1971.

[SC77] Lenie Sint and Dennis de Champeaux. "An Improved Bidirectional Heuristic Search Algorithm." *Journal of the ACM (JACM)*, **24**(2):177–191, 1977.

[SGA91] David J States, Warren Gish, and Stephen F Altschul. "Improved Sensitivity of Nucleic Acid Database Searches Using Application-Specific Scoring Matrices." *Methods*, **3**(1):66–70, 1991.

[SSH09] J. Slocum, D. Singmaster, W.H. Huang, D. Gebhardt, and G. Hellings. *The Cube: The Ultimate Guide to the World's Best-Selling Puzzle: Secrets, Stories, Solutions*. Black Dog & Leventhal Publishers, 2009.

[Ste41] B M Stewart. "Solution To Advanced Problem 3918." *American Mathematical Monthly 48*, pp. 217–219, 1941.

[YCH08] Fan Yang, Joseph C Culberson, Robert Holte, Uzi Zahavi, and Ariel Felner. "A General Theory of Additive State Space Abstractions." *Journal of Artificial Intelligence Research*, **32**:631–662, 2008.

[ZFH08] Uzi Zahavi, Ariel Felner, Robert C Holte, and Jonathan Schaeffer. "Duality in Permutation State Spaces and the Dual Search Algorithm." *Artificial Intelligence*, **172**(4):514–540, 2008.

[ZH06] Rong Zhou and Eric A Hansen. "Breadth-First Heuristic Search." *Artificial Intelligence*, **170**(4):385–408, 2006.

[ZSH09] Zhifu Zhang, Nathan R Sturtevant, Robert C Holte, Jonathan Schaeffer, and Ariel Felner. "A* Search with Inconsistent Heuristics." In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pp. 634–639, 2009.