University of Alberta

Library Release Form

Name of Author: Kenneth Alfred Anderson

Title of Thesis: Memory and Abstraction used for Heuristics in Single-Agent Search

Degree: Master of Science

Year this Degree Granted: 2007

Kenneth Alfred Anderson
5249 Andrea Dr.
Wescosville, Pennsylvania
United States, 18106

Date: ____________________

University of Alberta

MEMORY AND ABSTRACTION USED FOR HEURISTICS IN SINGLE-AGENT SEARCH

by

Kenneth Alfred Anderson

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2007

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Memory and Abstraction used for Heuristics in Single-Agent Search** submitted by Kenneth Alfred Anderson in partial fulfillment of the requirements for the degree of **Master of Science**.

Prof. Jonathan Schaeffer
Supervisor

Prof. Robert Holte
Co-Supervisor

Prof. Ryan Hayward

Prof. Petr Musilek
External Examiner

Date: ____________________

Abstract

Pattern databases in combination with IDA* search have proved extremely successful in optimally solving a variety of single-agent search problems. Pattern databases use retrograde search and abstraction to produce a memory-based heuristic lookup table. Perimeters similarly use retrograde search, but without abstraction, to produce a memory-based heuristic. While traditionally separate lines of research, this thesis introduces the two approaches, partial pattern databases and compressed partial pattern databases, which combine the approaches of pattern databases and perimeters in a general way. Partial pattern databases and compressed partial pattern databases are tested on the K-pancake puzzle and the fifteen sliding-tile puzzle. Compressed partial pattern databases are shown to lead to performance improvements on both domains.

Acknowledgements

I would like to extend my sincere appreciation for the guidance, support, and dedication provided by my supervisors Rob Holte and Jonathan Schaeffer. Although both were quite busy, they always answered my questions and gave me as much attention as was humanly possible. I know from experience that proof-reading my writing is a tedious chore; they both generously performed this service. Rob and Jonathan provide contrasting views, broad experience, and expertise that complement each others' wonderfully. I could not have asked for better supervision.

Thank you to the numerous people I discussed my research ideas with: friends and colleagues, visiting scholars, and long distance email correspondents. Ariel Felner was especially enthusiastic regarding our research. Of great help were my proof-readers Bruce Fraser and David Thue. My committee members Prof. Ryan Hayward and Prof. Petr Musilek were very generous in donating their time and interest to this work.

Finally, I am eternally grateful for the support provided by my family and friends during this endeavor. Let me tell all incoming graduate students: writing a thesis ain't easy. This collection of pages encapsulates not only the knowledge transfer, research effort, and dedication by Rob, Jonathan, and myself, but hours of complaining, seas of red correcting ink, and uncountable bottles of beer.

The Natural Sciences and Engineering Research Council of Canada (NSERC) provided financial support for my second year of studies in the form of the Canadian Graduate Scholarship. This was supplemented by the Informatics Circle of Research Excellence's (iCORE) Graduate Student Award and the University of Alberta's Walter H. John's Graduate Fellowship. Conference travel funds were provided by the Faculty of Graduate Studies and Research through the J Gordan Kaplan Graduate Student Award and Jonathan Schaeffer. Additionally, Rob Holte has generously supported my work this summer.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Single-agent search is used for a variety of real-world applications and is regularly applied in domains such as the gaming industry, robotics, and bioinformatics. Worldwide, the total revenue of the video game industry is estimated at $30 billion USD [34]. Pathfinding is a single-agent search problem critical to many genres of video games that must be very efficient. For example, real-time strategy games coordinate the movements of hundreds of units and must be able to pathfind within milliseconds.

Robot planning uses single-agent search. NASA's Mars rovers, Spirit and Opportunity, use autonomous pathfinding for navigation over 55 million kilometers away [4]. A little closer to home, the Hubble Telescope uses the planning system, Spike, for scheduling tasks [23]. Approaches for solving single-agent search tasks can be effective in solving planning tasks [10].

In the computational biology community, single-agent search is used for the important problem of sequence alignment [36]. Sequence alignment is used for comparing genes, finding evolutionary linkage, and predicting structure [17].

Single-agent search is a versatile tool for solving problems that can be restructured into an optimal path finding problem on a graph. It is important to improve upon current techniques to solve larger, more challenging applications. Pathfinding, robotic planning, and sequence alignment can all be easily scaled up to exceed the computational limits of today's techniques.

1.2 Problem Definition

The question of how to use memory in single-agent search is multifaceted. Explicitly stored graphs, memory-based search algorithms, and memory-based heuristics all must compete for the same limited resource. With the increasing capacity of memories (1 Gigabyte and more) and refined heuristic techniques, today some previously intractable problems can be solved quickly; but memory will always a critical limiting factor in this field.

One use of memory is to create and store a heuristic or improve an already existing heuristic. In the case of the puzzle domains, *e.g.*, the Sliding-Tile puzzle, K-pancake, Top-Spin, Rubik's Cube, Towers of Hanoi, and Sokoban, virtually all the available memory can be used for the heuristic. There are two reasons for this. First, the puzzle domains have implicit successor functions, thus eliminating explicit storage of the graph in memory. Second, the linear-space IDA* search algorithm has been shown to be effective on puzzles [26]. In fact, IDA* is the primary search algorithm for these domains because of its low-memory requirements, simplicity, and speed, even though it generates more states than some other techniques.

The memory-based heuristic often takes the form of pattern databases. Pattern databases are a proven, powerful, and flexible technique that use available memory as a heuristic lookup table [6]. Domain abstraction reduces the size of the state space by merging multiple states into a smaller number of abstract states [19]. Each abstract state has an entry in the pattern database for its heuristic value. Retrograde search is used to fill in the heuristic values by searching backwards from the abstract goal to all abstract states. The pattern database size is dependent upon the abstraction granularity; finer-grained abstractions require more entries in the database and more memory, but yield better heuristics [19, 40].

Increasing the number of pattern database lookups for each state improves the heuristic quality. Taking the maximum value over the values of multiple pattern database lookups for a single state is one simple and general example [21]. Other approaches take advantage of domain-specific properties such as symmetry, additivity, and duality [7, 14, 50]. This procedure continues pushing the state-of-the-art to new heights, but only for a subset of puzzles.

Alternatively, perimeter search uses no abstraction at all [9]. Retrograde search builds a perimeter around the predetermined goal which is stored in memory. The forward search can use multiple perimeter states to improve the heuristic values, which in turn reduce the size of the search tree. A variety of methods have been presented which further improve the heuristic, but are dependent upon specific properties of the heuristic itself [9, 24, 35].

Pattern databases and perimeter search have traditionally been two disjoint areas of research. Yet these approaches are similar: both use backwards search from the goal, store heuristic information in memory, and use the heuristic to improve the forward search. This thesis examines combining these techniques in a simple and effective way.

1.3 Approach to the Problem

This thesis presents two new heuristic table-lookup techniques that separate the level of abstraction from the memory bound: partial pattern databases and compressed partial pattern databases. Partial pattern databases are closely linked with perimeters. However, where perimeters use states in the original state space, partial pattern databases use abstract states at any abstraction level. The most appropriate abstraction level is determined through testing. Compressed partial pattern databases are more like pattern databases, storing each entry in a memory-efficient manner. However, while the pattern database abstraction level is dependent upon the memory limit, compressed partial pattern databases are free to use any abstraction level.

Our work approaches the problem from a general standpoint instead of utilizing domain-specific properties. Similar to taking the maximum over multiple pattern databases [21], the new techniques presented in this thesis are applicable across all puzzle domains.

While the approaches taken are general enough to be applicable to all possible puzzle domains, this thesis specifically examines the Fifteen Sliding-Tile and K-pancake puzzles. IDA* search is used for the forward search because of its low memory requirements [26, 50]. To determine the scalability of these database techniques, the 8, 10, 12, and 13-pancake puzzles are tested.

1.4 Research Contributions

The contribution of this thesis is three-fold. The first is the introduction of two new types of heuristic lookup tables. Perimeter search and pattern databases have long been separate lines of research. This work examines the similarities between the two methods and attempts to combine the strengths of each. The approach taken is general enough to be applicable to many domains and the memory-based table can be reused over multiple problem instances with a fixed goal.

Secondly, empirical testing demonstrates the benefits and disadvantages of these techniques. Two contrasting puzzle domains are tested: one with a large branching factor and one with a small branching factor. The coarse abstraction of pattern databases allows for more cutoffs shallower in the search. The lack of abstraction of perimeters causes a higher number of cutoffs deeper in the search, which is important due to the duplicate nodes caused by IDA* search. Testing shows the appropriate choice of abstraction granularity for each puzzle can improve performance over that of a full pattern database.

Finally, the worst-case node expansion formula proposed by Korf and Reid for pattern databases can be applied to these new lookup tables [31]. This thesis examines the use of Korf and Reid's formula to predict the worst-case node expansion given the heuristic distribution of a partial pattern database. If accurate, this formula would allow for intelligent database selection without running exhaustive experimentation. Much of this work will appear in proceedings of the Seventh International Symposium on Abstraction, Reformulation, and Approximation in July 2007 [1].

1.5 Outline

This thesis is organized as follows. Chapter 2 introduces the search domains, search techniques including IDA*, and uses of memory in single-agent search. Different variations on perimeters and pattern databases and their applicability to these domains are discussed in depth.

Chapter 3 introduces two new heuristic lookup table techniques and discusses the advantages and disadvantages of each. Chapter 4 and Chapter 5 present a thorough empirical analysis of the performance of each individual technique. In addition, general approaches are combined to yield a stronger algorithm. A theoretical formulation of the worst-case performance is shown to work on one of the tested domains, which provides a tool used for choosing the most appropriate pattern database.

Finally, in Chapter 6, conclusions, limitations, and possible future work are discussed.

Chapter 2

Related Work

2.1 State Space Representations

A single-agent search problem is structured as finding a minimum-cost path between a start node s and a goal node t on a weighted, directed graph G. The *cost* of a path is the sum of the costs of all edges on the path, while the path *length* is the total number of edges on the path. The graph consists of a set of nodes (set V) and directed edges (set E) where each edge $e(u, v) \in E$ has an associated non-negative edge cost $c(u, v)$. A node n' is a *successor* of node n if there exists an edge $e(n, n')$ from n to n'. Similarly, n is the *predecessor* of n'. Given unlimited memory, a graph G can always be represented *explicitly*; all nodes and edges of G are stored in memory. A real-world example of such a graph is the connectivity of the world wide web; each node represents a website and each directed edge represents a link from one website to another. As this example shows, these types of graphs can be very large, so fitting them in memory can be problematic.

The graph structure of a state space is represented *implicitly* in some domains, such as puzzle domains. Puzzle domains are simple and readily understandable, yet demonstrate the important principles of real-world problems [38]. This makes them an ideal and interesting testbed for research. The Arrow Puzzle [25], Sliding-Tile Puzzle, K-pancake Puzzle, Top-Spin, Rubik's Cube, and Towers of Hanoi are examples of such puzzles. This thesis specifically examines the the K-pancake puzzle and the Fifteen Sliding-Tile puzzle (henceforth the Fifteen Sliding-Tile puzzle shall be refer to as the fifteen-puzzle).

The K-pancake puzzle consists of a stack of K pancakes, all of different sizes, numbered 0 to $K - 1$ (Figure 2.1). There are $K - 1$ operators, where operator k ($1 \leq k \leq K - 1$) reverses the order of the top $k - 1$ pancakes. An individual pancake will be referred to as a *tile* and its placement in the stack as a *location*. A *state* is an arrangement of tiles in specific locations.

The fifteen-puzzle is comprised of a four by four grid of 15 distinct tiles, with one location empty. The empty location is called the *blank*. Valid operations include swapping the blank with one of up to four adjacent tiles. Figure 2.1 shows the $goal$ state of the fifteen-puzzle.

For puzzle domains, the search space can be restructured as a graph by mapping each possible

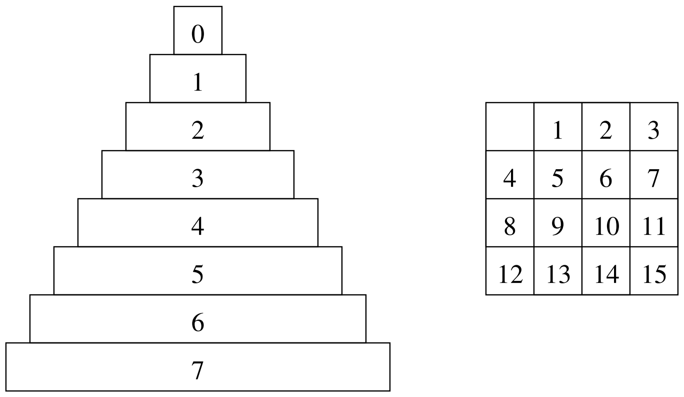

Figure 2.1: Goal states for 8-pancake (left) and fifteen-puzzle (right).

state to a node in graph G, and each applicable operator to an edge of unit weight ($c = 1$). The set of all states is called the *search space*, S. An *operator* transforms a state, s, into its successor state, s', at a specified cost. Not all operators can apply for every state; for example, the fifteen-puzzle cannot move the blank up when it is in the top-most row. A *successor function* generates the set of successor states using the set of possible operators for state s. Similarly, the *predecessor function* generates the set of states with s' as their successor. For puzzle domains, the successor function is compact, consisting only of a set of rules for the applicability of given operators. Also, in puzzle domains, operators have unit-cost ($c = 1$) and are reversible (undirected edges). Our analysis is valid in the more general context of directed, weighted graphs.

Now that there is a compact representation of weighted, directed edges, the graph no longer needs to be explicitly stored. Memory can be used for other purposes. This is crucial for examining large state spaces; the fifteen-puzzle has a state space of $16!/2 \approx 10^{13}$ states, and the K-pancake puzzle is of size $K!$ [41].

2.2 Heuristics in Search

Search through a state space (or graph) is used to find an optimal path from the start state to the goal state. A search algorithm traverses a tree with the start state at the root of the tree. Using the successor function, a state *generates* its child states. A state is *expanded* after generating all the state's children. Search algorithms apply different techniques for generating and expanding states [38].

In uninformed search, state expansion is dependent only upon the visited part of the graph. To speed up search, informed or heuristic search uses domain-knowledge in the form of a heuristic [38]. In single-agent search, heuristics are estimates of the remaining cost from the current state to the goal state. Search procedures utilize this information to prune or cut off states that do not need to be expanded. This makes the search tree much smaller than it otherwise would be. This thesis examines heuristic search.

Heuristics originally were hand-designed using context-sensitive information. The fifteen-puzzle's Manhattan Distance heuristic is an example of this approach [38]. This heuristic examines each tile in the fifteen-puzzle. Each tile has a lower bound on the number of moves (distance) necessary to get this tile to its individual goal location: the change in the x dimension plus the change in the y dimension. Since each tile has to move this minimal amount, simply sum the distance each tile has to move to get a lower bound on the number of moves required to solve a given instance.

$$ManhattanDistance = \sum_{tile=tile1}^{tile15} |loc_x(tile) - goalLoc_x(tile)| + |loc_y(tile) - goalLoc_y(tile)|$$

However, designing heuristics for domains can be tedious and some puzzles, like the K-pancake puzzle, currently have no known strong human-designed heuristic. Section 2.4 will outline some methods for generating heuristics using abstraction.

An *admissible* heuristic does not overestimate the cost to the goal [16]. In many heuristic search algorithms, using an admissible heuristic can guarantee finding an optimal solution [16, 26]. A heuristic is *consistent* if the heuristic difference between any two connected states is less than or equal to the edge weight between those states:

$$|h(s) - h(s')| \leq c(s, s')$$

. This property can be checked locally by examining all successors to all states in the state space. Consistency implies admissibility, but admissibility does not imply consistency [38].

Bidirectional-Pathmax

Search algorithms can improve inconsistent heuristics during search by enforcing consistency locally. If a state s has a successor or predecessor s' where $|h(s') - h(s)| > c(s, s')$, then s's heuristic value can be increased to $h(s') + c(s, s')$. Mero propagates heuristic corrections only to a state's successors (pathmax) [37].

If a depth-first search technique is used with an inconsistent heuristic, the heuristic can be improved through the use of bidirectional-pathmax (BPMX) [14]. BPMX propagates heuristic improvements to both successors and predecessors [14]. This technique uses no extra memory in IDA* and takes very little time.

2.3 Memory in the Heuristic Search Algorithm

Memory is often a limiting factor in single-agent search. There are three primary ways to use memory: (1) explicitly in the graph representation, (2) in the search algorithm, or (3) in the heuristic. The first of these is not directly applicable to puzzle domains, so this thesis concentrates on the latter two. This section presents how memory can be applied to the search algorithm itself in order to improve the performance of heuristic search.

2.3.1 Best-First Search

Best-first search techniques expand the apparent best state (the state most likely to lead to a solution) according of some criteria [38]. Memory is used to keep track of an *open* list and *closed* list. The open list consists of the set of generated but not yet expanded states. The closed list consists of the set of expanded states. At every step, the best apparent state is removed from the open list and expanded. Expanding a state causes all successors to be generated and placed on the open list, unless they have already been visited. The specific details depend on the algorithm used. The expanded state itself is then placed on a closed list. In effect, the open list is the search frontier (leaf states) and the closed list is comprised of the interior, expanded states.

One such method is Dijkstra's algorithm [8], which finds the minimal cost paths to a goal in a graph with non-negative edge weights. States are selected for expansion off the open list based on the lowest-cost path to a state. Each state is only expanded once. This method is *uninformed* because it uses no heuristic [42]. The following algorithm (A*) introduces a heuristic, which, in most cases, improves performance significantly.

A*

A* is a classic algorithm that expands states in order of increasing priority [16]. A* keeps track of states via an open list and closed list. The closed list is composed of states already expanded. It is used to (1) prevent unnecessarily re-visiting these states (via transpositions and cycles) and (2) reconstruct the solution path from the start to the goal. The open list is comprised of generated but not-yet expanded states. It is used to prevent transpositions and choose the next state for expansion.

States on the open list are prioritized by f (low f-values mean high priority).

$$f(s) = g(s) + h(s)$$

where $g(s)$ is the minimal cost of a path from the start to s found so far and $h(s)$ is the heuristic (estimated cost from state s to the goal). $f(s)$ is the estimated cost from the start to goal on a path constrained to go though state s. The state s with the lowest f-value is expanded, whereby (1) s is taken off the open list and placed on the closed list and (2) all successor states of s are either added to the open list if not already on the open or closed list, updated if already on the open list, or updated and added to the open list if already on the closed list. Ties are typically broken in favor of the state with the higher g value.

A* can use a large amount of memory for the closed and open lists; in the worst case (if the heuristic is zero), the space complexity is $O(|S|)$, where $|S|$ is the size of the state space. Because the graph representation is implicit, the state space is generally much larger than the available memory size. A* requires every visited state to remain in memory, either in the closed list or open list. When millions of states are generated per second, the memory can be filled in minutes.

2.3.2 Reducing A*'s Memory Requirements

The high memory requirements of A* is a major deterrent for use in large state spaces. Recent research investigates approaches that reduce the memory requirements of A*.

Frontier Search

Frontier search, by Korf *et al.*, is a search technique that reduces the number of states stored on the closed list [28, 32, 33]. This technique works well for domains with reversible operators (undirected edges) as will be discussed here, but can also be extended to directed graphs [33]. In an undirected graph, when a state is expanded, its successors may be on the closed list. However, a state on the closed list will only be re-generated if one of its successors is on the open list. Therefore, the closed states whose successors are all closed can be eliminated.

To reconstruct a path, an intermediate ancestor state along every potential solution path is stored. All closed states now point to one of these ancestor states. The path is reconstructed by recursively solving the two remaining subproblems: the path from the start to the ancestor state, and the path from the ancestor state to goal [32].

Breadth-First A*

Zhou and Hansen present an adaptation on frontier search to further reduce memory requirements with their Breadth-First A* search algorithm [51, 54]. Instead of expanding the best state according to the f-value, Zhou and Hansen use breadth-first state expansion. The states that exceed a cost bound are discarded and only the states on the frontier are kept in memory. The cost-bound is determined by repeating the search with an increased cost bound (this is called iterative-deepening and is discussed in Section 2.3.3) or by finding an upper bound using weighted A* [39]. This approach can often expand the exact same states as frontier search. In experimental results, breadth-first frontier search stores fewer states in memory than frontier search.

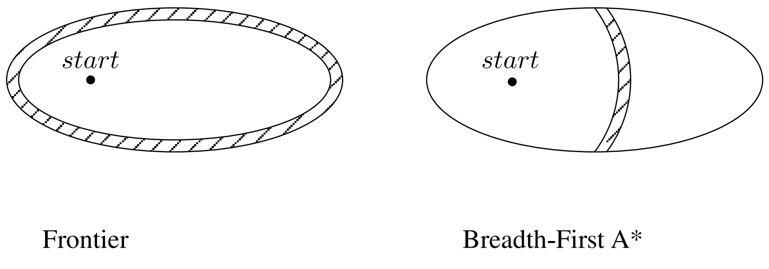

Figure 2.2: Frontier search and breadth-first A* search trees (from [51]).

2.3.3 Memory-Limited Search

Recall that A* and frontier search have memory requirements proportional to the size and width of the search space respectively. The following techniques have memory bounds linear in the solution length, freeing the programmer from the memory limitations of larger problem spaces. However, using such a small amount of memory has drawbacks in the form of state re-expansion. Therefore, numerous techniques utilize user-defined memory bounds to augment this algorithm.

Many of the strategies in the following sections have their roots in two-opponent games. Depth-first iterative deepening has been used for over 30 years in chess-playing programs [46], transposition tables were introduced in 1967 by Greenblatt [15] *et al.*, and retrograde search has been used to create end-game databases (perimeters) to reduce storage and improve heuristics [48].

IDA*

Iterative Deepening A* (IDA*) is a minimal-memory search technique that repetitively uses depth-first, cost-bounded searches while increasing the cost-bound f_{max} [26]. The memory requirements are proportional to the solution length due to the depth-first nature of the search. IDA* has proven to be successful on puzzle domains because of its speed, simplicity, and negligible memory requirements.

After each iteration in which the goal is not found, the bound f_{max} increases to the smallest f-value of the generated but not expanded states. By iteratively deepening the cost-bound in this way, IDA* and A* expand similar states.

However, each of these states may be expanded many more times than A* because of two reasons. First, multiple searches result from iteratively deepening the cost-bound. This is not of much concern for puzzle domains however. In puzzle domains, the number of states generated increases exponentially with the cost bound. Therefore, the computation done at lower-depth levels is dominated by the deeper searches. In addition, all edges (operations) are of unit cost. So every depth increment will be at least one, limiting the number of iterations to at most the solution length.

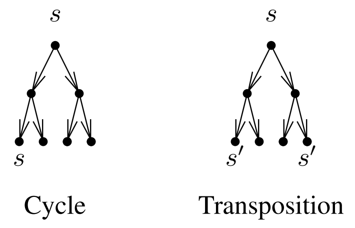

Figure 2.3: Example of a cycle and transposition in a search tree.

Second and more importantly, IDA* cannot detect duplicate states like A* and is therefore susceptible to cycles and transpositions. A *cycle* is a sequence of operators that return to a previously seen state (Figure 2.3). A *transposition* is a state reached from different paths in the search tree (Figure 2.3).

An example of a cycle in an undirected graph is moving from state s to a successor s', then moving back to s. Cycles can be eliminated simply by comparing the current state against the states in the current path (stored on a stack); the memory and time complexity is proportional to the solution length. However, reducing transpositions in IDA* is more difficult than reducing cycles. A variety of approaches to this problem are discussed below.

2.3.4 Memory-Enhanced IDA*

Combining A* and IDA*

On one end of the spectrum, IDA* search uses very little memory but expands states multiple times because of transpositions. On the other end, with a consistent heuristic A* search expands states once, but uses a massive amount of memory. The following techniques combine the two approaches into a limited-memory algorithm.

Sen and Bagchi's algorithm MREC uses iterative depth-first searches, like IDA*, but MREC builds the closed list until a memory limit has been reached [45]. The closed list is statically stored, no states are added or removed after it is built. On every search iteration, the closed list is used to make additional cutoffs.

MA* and SMA* use A* search until a memory limit is reached, keeping the closed and open lists in memory [3, 43]. After this phase, the search continues starting from the open list states. Memory is reallocated to the current search states by retracting the least promising open list states. Additionally, the f-values of states are propagated back through the search tree during the state-retraction step. These techniques efficiently use memory to reduce duplicate states while retaining some previously learned information.

Transposition Tables

As previously mentioned, one of the main difficulties with IDA* is transpositions. One fast and simple way to reduce transpositions is with a memory-based table [41]. Each entry in this hash table stores a state along with the cost bound to which that state has been searched. The data structures are simple, access-time is constant (and fast), and size can be arbitrary, making these approaches very desirable.

Transposition tables are one application of this approach. When a state is expanded during search, the transposition table records the state and the cost bound to which the state has been searched. When the search reaches this state again, the state can be pruned if the cost bound is less-than or equal-to the recorded cost bound. States are only added or updated in the transposition table, not over-written. Therefore any other states mapped to the same location by the hash function are ignored.

Finite State Machines

Another technique for reducing transpositions is by predetermining which operator sequences lead to cycles and transpositions. One simple example is eliminating the operator that leads to the parent state. For instance, in the K-pancake puzzle, applying operator x twice in sequence leads back to the parent state.

Taylor and Korf use a finite state machine (FSM) to eliminate forbidden operators that lead to duplicate states [47]. However, not all operators are applicable to every state. For example, if the blank is in the top row of the fifteen-puzzle, it cannot move up. So an operator is pruned only if it matches specific preconditions.

To create the FSM, an initial breadth-first search generates all states in the search tree until a limit is reached. The aim is to acquire a single optimal sequence of operators to each reached state. Therefore, the duplicate states are identified along with their operator sequences. These operator sequences are added to a forbidden list along with their preconditions.

2.4 Memory Used for a Heuristic

2.4.1 Perimeters

Perimeter search is a type of bidirectional search technique that avoids some of the problems of traditional bidirectional search. As such, perimeter search requires a predecessor function in addition to the successor function. Originally proposed by Dillenburg and Nelson, perimeter search performs two successive searches [9]. The first search proceeds in the backwards direction from the goal, forming a set of expanded, perimeter states P, which encompass the goal. A state s is *on* the perimeter if $s \in P$, *inside* the perimeter if it was expanded during perimeter creation, and *outside* the perimeter if it was not expanded. Every state in P is also inside the perimeter. Any state outside the perimeter must pass through some state on the perimeter to reach the goal. Many techniques can be applied to generate the perimeter (see Figure 2.4): Breadth-first search creates a *constant-depth perimeter* [9, 35]; A* creates a *constant-evaluation perimeter* [9]; and expansion based on heuristic difference creates another kind of perimeter [24].

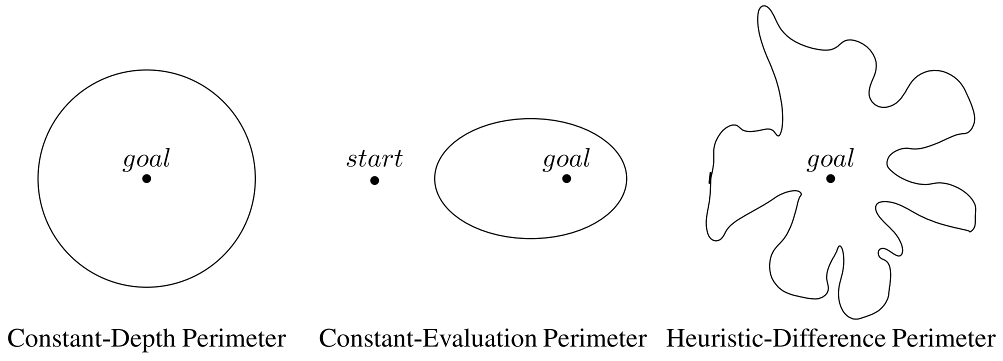

Figure 2.4: Different kinds of perimeters.

If the perimeter is generated for one problem instance, then the backward and forward searches are performed in series. The backward search forms a perimeter and, if the start state is not found, it is followed by the forward search. However, if the perimeter is constructed for use on multiple problem instances with the same goal, then the interior of the perimeter, set A, is stored. When the forward search begins, if the start is in set A, the actual cost to the goal is known so the heuristic is corrected to this value. Otherwise the start is outside the perimeter and the forward search begins.

The second, forward search progresses either from the start to the perimeter, called *front-to-front* evaluation, or from the start to the goal, called *front-to-goal* evaluation. Front-to-front evaluation calculates the heuristic value of a state based on the estimated cost through every state on the perimeter (Figure 2.5). Although larger perimeters provide better heuristic values, the heuristic takes increasingly longer to compute. Additionally, front-to-front evaluation requires for a heuristic estimate to exist between any one state to every state on the perimeter.

By contrast, search using front-to-goal evaluation requires only an existing heuristic to the goal (Figure 2.6). The heuristic values of states found to be inside the perimeter are corrected using the exact cost to the goal. The heuristic values of states outside the perimeter can sometimes be corrected; here are two different approaches using front-to-goal evaluation. Using a depth-limited perimeter where d is the cost-bound, d is a lower bound on the true cost to the goal for all states outside the perimeter [5, 44]. Or given a consistent heuristic, a correction factor is equal to the lowest difference between the actual cost to the goal and the estimated heuristic cost to the goal [24]. The correction factor is now added to the original heuristic if outside the perimeter.

Any search technique will work for the forward search, but IDA* and similar low-memory search techniques are most commonly employed [9, 35, 24]. IDA* is also the search technique used throughout this thesis.

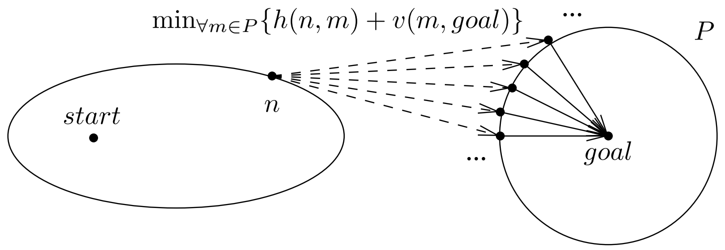

Figure 2.5: Front-to-front heuristic.

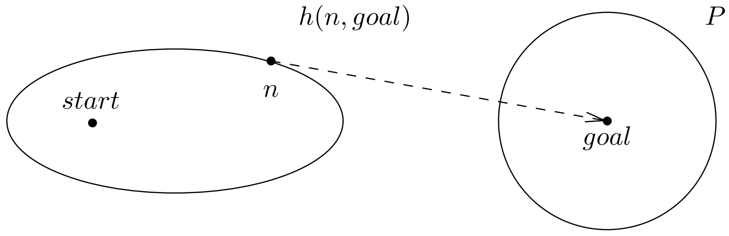

Figure 2.6: Front-to-goal heuristic.

2.4.2 Pattern Databases

Introduced by Culberson and Schaeffer, pattern databases require an abstraction mechanism to reduce the state space [6]. The *original space*, S, consists of the set of states that can reach the goal through a series of operators. An *abstract space* is a set of abstract states, where every state in the original space maps to some corresponding state in the abstract space. The abstraction used in this thesis is *domain abstraction* [21]. The abstraction is created by renaming specific tiles to the same name, x. These re-named tiles are called *don't-care* tiles while the other tiles are *unique tiles*. In the case of the fifteen-puzzle, the blank will always be a unique tile. *Abstraction-N* refers to a specific abstraction with N unique tiles (the actual tiles vary with the domain). Figure 2.7 shows one possible abstraction-6 for the 8-pancake puzzle and fifteen-puzzle. A *coarse-grained* abstraction has fewer unique tiles (and hence has fewer abstract states) than a *fine-grained* abstraction.

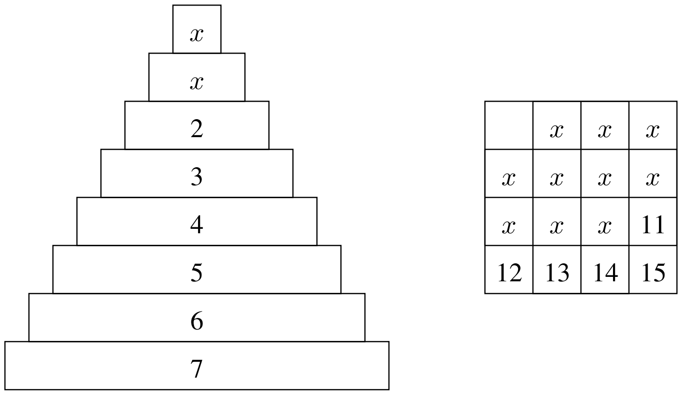

Figure 2.7: Abstraction-6 for 8-pancake puzzle and fifteen-puzzle.

The basic idea of pattern databases is the use of abstraction and retrograde (backwards) search to create a heuristic lookup table [6]. Retrograde search starts from the abstract goal. The search proceeds backwards applying all applicable reverse operators until the abstract space is entirely enumerated. The costs from the abstract goal in the abstract space are recorded in a table and used as a heuristic in the forward search. This produces an admissible and consistent heuristic.

Enhancements

Maxing is a general technique, initially presented by Culberson and Schaeffer, then further developed by Holte *et al.* [6, 21]. Maxing uses the maximum heuristic value of a state over two separate heuristics. Any two admissible heuristics can be maxed into an admissible heuristic. Holte *et al.* have found that that maxing over two smaller pattern databases can yield better search performance than a single larger pattern database.

Domain-specific knowledge has subsequently been applied to take advantage of specific properties. Some examples include symmetry [6], additivity [12], and duality [14, 50]. These techniques can dramatically improve performance by using multiple heuristic evaluations.

Holte *et al.* have investigated generating and caching parts of pattern databases during search in [20, 22]. Similarly, Zhou and Hansen have demonstrated a technique whereby provably unnecessary parts of the pattern database are not generated (given an initial consistent heuristic and upper bound on solution length) [52]. Felner and Adler further built upon on this procedure using *instance dependent pattern databases* [11]. This thesis approaches this problem from the opposite position; can part of a pattern databases and/or a perimeter be created and used over multiple problem instances with the same goal state? Holte *et al.* also consider this approach in [20].

Disk-Based Pattern Databases

In general, larger pattern databases yield better heuristics; this in turn reduces the number of states generated in solving a problem [27]. Unfortunately, the pattern database size is limited by the size of memory. Zhou and Hansen approach this problem by building pattern databases that are larger than memory, then loading in the relevant portions of the pattern database during search [53]. During the forward search, states on the open list are sorted. Then the required portions of the pattern database are loaded into memory in a predetermined order, minimizing the amount of I/0.

2.4.3 Perimeter and Pattern Database Comparison

Perimeters and pattern databases are similar in many respects. Both perimeters and pattern databases require a predecessor function. The predecessor function enables retrograde search from the goal. Both procedures can also be improved by using domain-specific properties: perimeters can use an existing heuristic function for front-to-front heuristic improvement; and pattern databases can use symmetry [6], additivity [30, 12], and duality [14, 50].

The two techniques also differ in critical ways. First, pattern databases require a state abstraction mechanism, which perimeters avoid. This allows perimeters to be applied to domains without any known abstraction. Also, pattern databases have a table entry for every abstract state (Figure 2.8(a)), while perimeters only have a table entry for part of the state space (Figure 2.8(b)). As a result, each state in the perimeter must store a state identifier as well as the cost to the goal. In general, any partial set of the original or abstract space requires extra memory to store the state identifier information (Figure 2.9(b)). Perimeters fall into this category, as do instance-dependent pattern databases. On the other hand, because pattern databases cover the entire abstract space, heuristic values may be indexed by their $stateID$ (the state need not be stored for each entry) (Figure 2.9(a)).

2.4.4 Combining Perimeters with Pattern Databases

Perimeter search works well for correcting pre-existing heuristics [9, 35, 24], while pattern databases work well in domains where no human-generated heuristic exists [6]. In Chapter 3, a new, general method for combining these two techniques into a single lookup table is proposed and investigated.

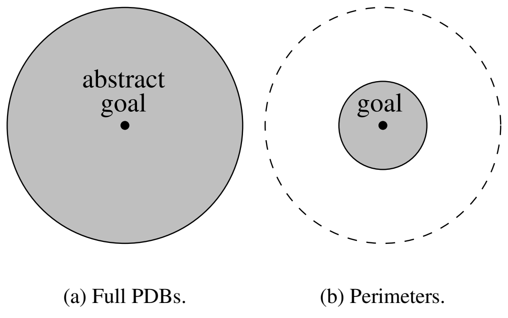

Figure 2.8: Coverage of original space by lookup tables.

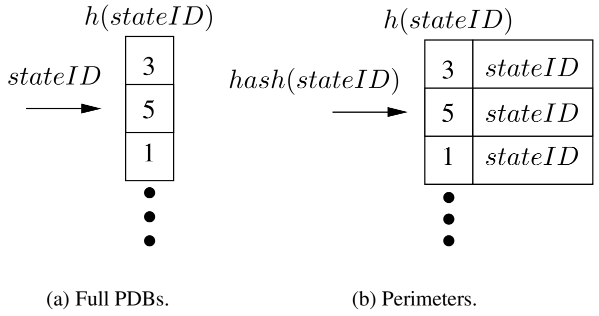

Figure 2.9: PDB storage strategies.

Culberson and Schaeffer use a pattern database as a heuristic simultaneously with a perimeter (the perimeter was called an end-game database) [5]. Because the pattern database only provides a heuristic to the goal state, perimeter search must use a front-to-goal evaluation technique. If a state is in the perimeter, the actual cost to the goal is known and is used for the heuristic value. A perimeter with cost bound d has the following property: d is a lower bound on the cost to the goal of states outside the perimeter. This means that for any state s not inside this perimeter, $h(n) \geq d$ and will be corrected accordingly. In fact, as long as d is defined as above, this can be applied to any shaped perimeter.

Ariel Felner uses a perimeter to *seed* a pattern database [13]. The pattern database is built using the perimeter as the goal state. This represents an alternative procedure for combining the techniques of perimeter search with pattern databases, but differs significantly from the new approach presented in Chapter 3.

For every state on the perimeter, Kaindl and Kainz track the difference between the actual cost to the goal and the heuristic value [24]. The minimal difference value for all states on the perimeter will be called the *heuristic correction factor*. The perimeter is built by expanding the state with the smallest difference, to increase the heuristic correction factor. If the heuristic is consistent, they admissibly add the heuristic correction factor to every state outside the perimeter. A pattern database is a consistent heuristic, so this technique is applicable. However, Felner has shown the following is true for the fifteen-puzzle using our abstraction method: given an abstract state a with an abstract cost c to the abstract goal, there exists a state in the original space mapping to a with a cost c from the goal [13]. The same is true for the K-pancake puzzle. Consider using a perimeter built by expanding states with the lowest heuristic value [24]. To obtain a correction factor equal to one, the perimeter must have at least as many entries as the pattern space. Because of the additional memory required by perimeters to store the $stateID$ (see Figure 2.9(b)), this method is impractical for puzzle domains.

2.5 Conclusions

From the introduction of IDA* in 1985, to today (22 years), search efficiency on hard problems has improved by four orders of magnitude [12]. Most of this improvement has attributable to the efficient use of memory to improve the search heuristic(s). First, the use of perimeters with the Manhattan Distance heuristic gave improvements on the fifteen puzzle. Then, the invention of pattern databases has proven effective on many domains. Finally, domain-specific information has been incorporated to improve search efficiency of the fifteen-puzzle, the K-pancake puzzle, and the Towers of Hanoi, allowing for solutions to harder and harder problems. Research presented in this thesis builds upon this continuing trend of improving memory-based heuristics.

Chapter 3

New Lookup-Tables

3.1 Motivation

Recently, pattern database techniques have incorporated perimeter ideas to use finer-grained abstractions rather than full pattern databases. Specifically, *instance-specific* pattern databases and hierarchical search [52, 11, 20] store subsets of full pattern databases, allowing the use of finer-grained abstractions that could not otherwise fit in memory. Typically pattern databases are precomputed for a fixed goal. Thus the build time can be amortized over multiple search instances. However, instance-specific PDBs and hierarchical search recompute some parts of the pattern database for every search instance. The technique proposed in this thesis (partial pattern databases) is not instance-specific; it reuses the same database over multiple search instances with the same goal (like full pattern databases).

Based on Korf and Reid's work, Holte *et al.* conjecture that increasing the low heuristic values that are seen during search has more of an impact than the consequential lowering of high heuristic values [31, 21]. Examine this conjecture further. Figure 3.1 shows how this approach can affect IDA* search. Assume the following: the start state is far away from the goal state, this is the last iteration of IDA* search, and all states in the last iteration are expanded (worst-case analysis). The states are grouped by g-value on the x-axis. The y-axis depicts the number of states on a logarithmic scale. The solution length for this instance is 12.

IDA* expands states in a depth-first manner, but this analysis groups states with the same g for analysis. The start state is the only state with $g = 0$. If all states with the same g-value are expanded, this is called the *brute-force expansion phase*. Brute-force expansion starts at $g = 0$ and continues until the lowest g-value encounters a cutoff. At this point the heuristic causes additional cutoffs at every higher g-value, reducing the effective branching factor. This is called the *cutoff phase*.

In Figure 3.1 the brute-force expansion phase occurs at states with small g values, following exactly the brute-force branching factor. The cutoff phase occurs in states with higher g values. On the last iteration, f_{max} is equal to the cost from the start to the goal. The start state has $g(start) = 0$, and $h(start) \leq f_{max}$. Therefore, $f(start) = g(start) + h(start) \leq f_{max}$. Since a state s in the

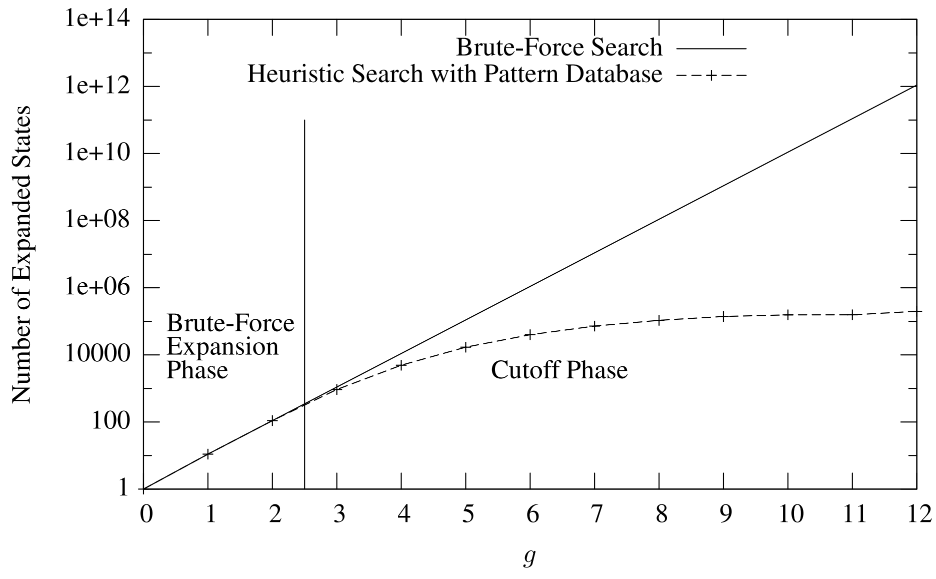

Figure 3.1: Phases of search

search tree is cutoff iff $f(s) > f_{max}$, the start state is expanded; all its children are generated.

To cause a cutoff for low g-values, the heuristic must be high. In Figure 3.1, when g is 3, then h must be greater than 9 to cause a cutoff. However, in this example, the total number of states in the brute-force expansion phase is smaller than the number of states in the cutoff phase. Maxing improves the low heuristics, reducing the number of states expanded in the cutoff phase. This is done at the cost of lowering the high heuristic values, increasing the number of states in the brute-force expansion phase. Think of this as extending the brute-force expansion phase. Because the brute-force expansion phase has few states, the extra state expansions are outweighed by the improved cutoff phase.

The approach proposed in Section 3.2 stores part of a fine-grained full pattern database (one that is larger than could normally be used), while all states not held inside the database get assigned some default heuristic value. Like maxing [21], this approach also increases low heuristic values while decreasing high heuristic values. This is accomplished with only one database, which means only one table lookup, as opposed to multiple lookups. This approach has the potential to speed up search because of the reduced time spent on heuristic evaluation and because improving the low heuristic values could cause enough cutoffs to reduce the number of states generated by IDA*.

3.2 Partial Pattern Databases

A *partial pattern database* (PPDB) consists of a set of abstract states A and their cost to the abstract goal, where A contains all states in S with cost to the abstract goal less than d. d is a lower bound on the cost of any abstract states not contained in A. In essence, a partial pattern database is a perimeter in the abstract space (with the interior states stored). Any state n in the original space has a heuristic estimate to the goal: if the abstraction of a state is in the partial PDB, return the recorded cost; otherwise, return d. This heuristic is both admissible and consistent.

Admissibility. A partial PDB consists of a subset of abstract states, A, from a full PDB of the same abstraction. All states in A have a heuristic value less than or equal to d and the heuristic value in the full PDB is the same as in the partial PDB. All states outside A have a heuristic value greater than or equal to d in the full PDB and equal to d in the partial PDB. Because the partial PDB always returns a value equal to or less than the full PDB and the full PDB is an admissible heuristic, the partial PDB is also admissible. □

Consistency. The full PDB is a consistent heuristic, therefore the states in A are consistent between themselves. The states outside A are consistent between themselves because every state has the same heuristic value (d). For every state in A with a successor outside of A, named s and s' respectively, the following condition holds: $h_{fullPDB}(s') \geq d \geq h(s)$. The following condition holds for the full PDB because it is a consistent heuristic: $|h(s) - h(s')| \leq c(s, s')$. For the partial PDB, the consistency property holds because $h(s) \geq d$: $|h(s) - d| \leq c(s, s')$. All states outside A with a successor inside A follow similarly. □

Building a partial PDB is similar to building a perimeter, only in the abstract space. Retrograde search is performed from the abstract goal, recording heuristic values. When a memory limit is reached, the perimeter building stops and heuristic values are used for the forward search. d is the minimum cost of all abstract states outside the perimeter. Note that all abstract states in A with cost equal to d can be removed from the perimeter to save memory. This will not affect the heuristic.

Partial PDBs occupy the middle ground between perimeters and pattern databases. At one extreme, a partial PDB with no abstraction reverts to exactly a perimeter (with the interior states stored). At the other extreme, a partial PDB with a coarse-grained abstraction covers the entire abstract space and performs exactly like a full PDB. However, the main drawback to partial PDBs is that a partial PDB cannot store the data as efficiently as a full PDB.

3.2.1 Memory Requirements

A full PDB encompasses the entire abstract space (Figure 3.2(a)); every state visited during the forward search has a corresponding heuristic value in the lookup table. The PDB abstraction level is determined by the amount of available memory. Finer-grained abstraction levels are not possible because the memory requirements increase exponentially with finer abstractions.

Partial PDBs generally do not cover the entire abstract space (Figure 3.2(b)). During the forward search, if a state is not contained in the partial PDB lookup table, d is returned. Partial PDBs add flexibility over full PDBs by allowing the use of any abstraction level. However, there are drawbacks with respect to memory requirements.

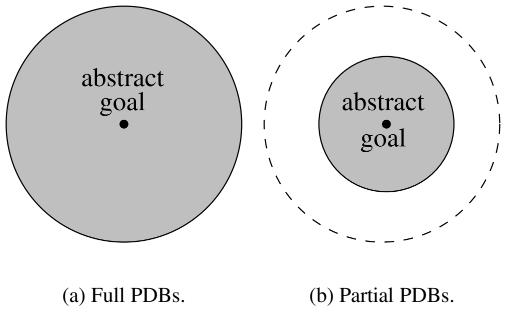

Figure 3.2: Coverage of original space by lookup tables.

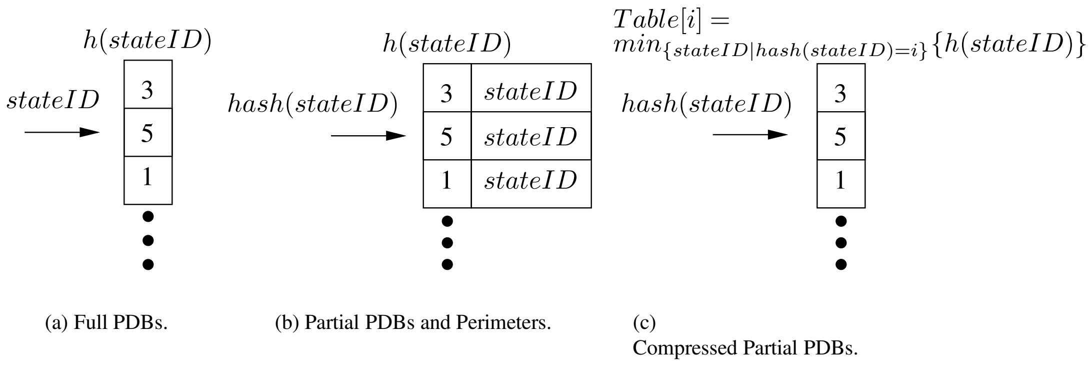
$h(stateID)$
$stateID$
3
5
1
$h(stateID)$
$hash(stateID)$
3 $stateID$
5 $stateID$
1 $stateID$
$Table[i] = min_{\{stateID|hash(stateID)=i\}}\{h(stateID)\}$
$hash(stateID)$
3
5
1
(a) Full PDBs.
(b) Partial PDBs and Perimeters.
(c) Compressed Partial PDBs.

Figure 3.3: PDB storage strategies.

Because pattern databases cover the entire abstract space, the heuristic values in the lookup table are indexed by their unique state identifier, $stateID$. Therefore, a table of the exact size of the abstract space can be used and there is no need to store the $stateID$ (Figure 3.3(a)). Memory is only used to store the abstract cost to the goal. On the other hand, partial PDBs only cover part of the space. As a result, each state in the partial PDB must store the $stateID$ in addition to the abstract cost to the goal (Figure 3.3(b)). This requires extra memory for every table entry.

In the studied domains, the fifteen-puzzle and the K-pancake puzzle, partial pattern database entries require nine times more memory than full pattern database entries due to the cost of storing the $stateID$. This is a severe limitation on the effective use of partial pattern databases.

3.3 Compressed Partial PDBs

For perimeters and partial pattern databases, the added cost of storing a state's identification information is an expensive use of memory. Also, if a hash table is used, it should have a reasonable fill-factor to maintain efficiency; but space is inevitably wasted space on empty table positions. Therefore, this thesis presents a compressed version of a partial PDB that does not store the $stateID$

and can be filled to any convenient fill-factor. This is called a compressed partial pattern database (CPDB).

Given an abstraction granularity, the hash function maps each abstract state to a location. For all abstract states mapped to the same location, the minimum heuristic value is stored (to preserve admissibility) (Figure 3.3(c)). The heuristic value returned for all states hashed to this location is this stored minimum. This heuristic value is guaranteed to be admissible, but it may be inconsistent.

Admissibility. Every entry in a full PDB, PDB, maps to some entry in a compressed partial PDB, $CPDB$, of the same abstraction. For every heuristic lookup from $CPDB$, the returned value is either the same as the PDB lookup, less than PDB because some other abstract state with a smaller heuristic mapped to the same location, or d (which is less than the PDB lookup). Therefore, the $CPDB$ heuristic is admissible. □

Possible Inconsistency. Suppose a state s has a state s' as a successor where $c(s, s') = 1$, PDB is a full PDB, and $CPDB$ is a compressed partial PDB with the same abstraction as PDB. $h_{PDB}(s) = h_{CPDB}(s) = 5$. If s' has the same hash value as the goal state then $h_{CPDB}(s') = 0$ because s' and the goal collide in the hash table. If $CPDB$ were consistent, then $4 \leq h_{CPDB}(s') \leq 6$. This condition does not hold; therefore, this example $CPDB$ is not consistent. □

The creation of compressed partial PDBs occurs as a preprocessing step. Therefore, this step can use extensive computing resources, for example using machines with more memory or using disk-based algorithms. One technique is to build a full PDB at a fine-grained level using a machine with a large amount of memory. Full pattern databases can be computed very efficiently in terms of memory and time by using iterative-deepening depth-first search. On every iteration, once a state gets expanded, a visited flag is set and the state is only expanded again if the distance to the goal is less than the current value. On the next iteration all flags are reset. Search is stopped when an iteration makes no new changes to the pattern database. Every heuristic value in the database is then hashed to the compressed partial PDB, where each entry is the minimum heuristic value. This is the approach used in Chapter 5. The main drawback to this approach is that it does not scale well to very fine-grained abstractions. However, the results in Chapter 4 indicate that very fine-grained abstractions are not necessary.

A second technique is to use iterative-deepening depth-first search from the abstract goal, filling the partial PDB directly. A table entry is updated if it is lower than the existing entry. However, unlike with normal PDBs, if a state's heuristic value is larger than the entry in the table, it cannot be cut-off without breaking admissibility. Search can be stopped when an iteration does not improve an entry in the table or when filled past a predetermined threshold. Therefore, this *depth-first construction* method must use a complementary technique to remove transpositions; this thesis uses a transposition table [41]. This is the technique used in Chapter 4.

A third technique is to use breadth-first search. This removes transpositions, but uses a large amount of memory for construction. However, delayed duplicate-detection is an efficient disk-based algorithm that can be used to do this [29]. In the interest of simplicity, this thesis does not attempt this technique.

One item worth consideration is the hashing function. As the author has found out, a simple modular hashing scheme can introduce regularity in the table. In the worst case, a fine-grained compressed partial PDB can revert to exactly the coarser-grained version. Also, depending on the hash function, the table may not fill to 100%. Therefore, when building the compressed PDB, the stopping condition should not be based solely on the fill-factor.

3.4 Conclusions

Partial PDBs and compressed partial PDBs separate the abstraction granularity from the memory limit. The hope is that varying the abstraction granularity will lead to better search performance, as maxing does [21]. Compressed partial PDBs are much more memory-efficient than partial PDBs, but are more difficult to build. Both techniques are admissible, but only partial PDBs are guaranteed to be consistent.

Chapter 4

Experiments on the K-Pancake Puzzle

The abstractions used for the K-pancake puzzle have don't-care tiles as the low indexed tiles. For example, abstraction-7 for the 12-pancake puzzle refers to the abstraction 'x x x x x 5 6 7 8 9 10 11,' with x as a don't-care. Note that abstraction-11 is the finest-grained abstraction and is the same as the original space because there are 12 distinct tiles.

All tests are run using IDA*. The two move cycle (returning to the parent state) is eliminated by disallowing the reverse operator. BPMX is implemented when explicitly stated, but will only have an effect when using inconsistent heuristics. In our studies, the compressed partial pattern database is the only inconsistent heuristic.

4.1 Performance with a Constant Number of Entries

This section reports the average number of states generated during the forward search as our metric. Each data point is an average of 100 random starting states. Partial pattern databases of various abstractions are used for the heuristic. The number of entries in the partial PDB remains constant (specified in the results tables), while the level of abstraction varies. Note that this is only a comparison between heuristic lookup-tables with the same number of entries; normal and compressed pattern databases make much more efficient use of memory than partial pattern databases. Section 4.2 cross-compares these techniques when memory is fixed.

4.1.1 Partial PDBs

Table 4.1 shows the average number of states generated on the 8, 10, 12, and 13-pancake puzzles using partial PDBs. The 14-pancake puzzle is not reported due to the excessive time required to build the compressed PDBs. The partial PDB is built to a cost bound d, shown in parentheses. Each puzzle fixes the number of entries in the partial PDB to exactly the number of entries of a full PDB with $\lfloor K/2 \rfloor$ unique tiles. For instance, the 12-pancake puzzle partial PDBs each have 665,280 entries (six unique tiles). The top data point of each column generates the same number of states as a full pattern database, while the bottom data point generates the same number of states as a perimeter. The purpose of this experiment is to determine if changing the granularity of a PDB can improve performance (while keeping the number of entries constant).

With the same number of entries, but using a finer-grained partial PDB, the average number of states generated reduces for the 8, 10, and 12-pancake puzzles. Note however, that in the 12-pancake puzzle, abstraction-7 produced fewer states than abstraction-8. Search performance is very sensitive to the cost-bound of the partial pattern database. The increase in states generated from abstraction-7 to abstraction-8 is because abstraction-7 has a larger cost-bound d than abstraction-8.

Consider two partial pattern databases with the same cost-bound d, but one being based on a coarser granularity than the other. The finer-grained database will *dominate* the coarser-grained one *i.e.* for every state n in the space, $h_{fine}(n) \geq h_{coarse}(n)$. Abstraction-8, 9, 10, and 11 on the 12-pancake puzzle demonstrate this principle.

This trade-off between abstraction granularity and database cost-bound will be seen throughout the results. Think of this intuitively as follows: making a partial PDB finer-grained improves the heuristic value of states inside the database at the cost of states outside the database if d gets smaller. At some point the heuristic outside the partial PDB becomes so inaccurate that the performance suffers. Refining the abstraction further from this point generally produces worse performance. Analogous results are documented in [21], where increasing small heuristic values improves performance, but only up to a point.

4.1.2 Compressed Partial PDBs

Tables 4.2 and 4.3 show the average number of states generated over 100 search instances using compressed partial PDBs. The compressed database is filled to 70% full, after which the current iteration is finished and the remaining untouched entries are filled in. The 8, 10, 12, and 13-pancake puzzles are examined. Table 4.2 uses IDA* without BPMX and Table 4.3 uses IDA* with BPMX. The purpose of these tables is to analyze performance compared to partial PDBs of the same abstraction and number of entries, and to determine the effect of BPMX on search efficiency.

The top entry of each column of Tables 4.2 and 4.3 match closely with the performance of a full pattern database (shown at the top of the corresponding columns in Table 4.1). At this abstraction level, each entry corresponds to one abstract state; if the compressed partial PDB were filled to 100% and had no hash collisions, then it would be identical to a normal PDB. However, these conditions do not hold in general, so the top entries in each table do not match exactly.

Examine two corresponding entries in Tables 4.1 and 4.2, the entry for 12-pancake at abstraction-7. For the partial pattern database (Table 4.1), there is an average of 178,464 states generated. For the compressed partial PDB without BPMX (Table 4.2), there are 635,649 generated states on average. Keep in mind that both databases have the same number of entries, but the entries themselves are

Number of Unique Tiles	8-Pancake 1,680 entries (cost-bound d)	10-Pancake 30,240 entries (cost-bound d)	12-Pancake 665,280 entries (cost-bound d)	13-Pancake 1,235,520 entries (cost-bound d)
4	2,065 (7)			
5	682 (5)	48,408 (9)		
6	403 (5)	14,268 (6)	1,316,273 (11)	25,833,998 (12)
7	**335 (5)**	8,251 (6)	178,464 (8)	**3,106,345 (8)**
8		7,370 (6)	183,172 (7)	4,097,683 (7)
9		**7,242 (6)**	167,584 (7)	3,851,260 (7)
10			165,390 (7)	3,820,667 (7)
11			**164,951 (7)**	3,816,931 (7)
12				3,816,546 (7)

Table 4.1: Average number of states generated using a single partial PDB on K-pancake puzzle.

Number of Unique Tiles	8-Pancake 1,680 entries (cost-bound d)	10-Pancake 30,240 entries (cost-bound d)	12-Pancake 665,280 entries (cost-bound d)	13-Pancake 1,235,520 entries (cost-bound d)
4	2,065 (7)			
5	**1,399 (6)**	48,414 (8)		
6	1,891 (6)	**28,839 (8)**	1,316,284 (10)	25,834,132 (10)
7	2,366 (6)	40,558 (7)	**635,649 (9)**	**12,021,692 (9)**
8		54,026 (7)	861,365 (9)	16,216,581 (9)
9		68,488 (7)	1,096,865 (8)	23,152,643 (9)
10			1,381,920 (8)	32,159,310 (8)
11			1,708,840 (8)	44,332,028 (8)
12				51,689,205 (8)

Table 4.2: Average number of states generated using a single compressed partial PDB filled to 70% on the K-pancake puzzle.

Number of Unique Tiles	8-Pancake 1,680 entries (cost-bound d)	10-Pancake 30,240 entries (cost-bound d)	12-Pancake 665,280 entries (cost-bound d)	13-Pancake 1,235,520 entries (cost-bound d)
4	2,065 (7)			
5	**1,024 (6)**	48,414 (8)		
6	1,227 (6)	**18,026 (8)**	1,316,284 (10)	25,834,132 (10)
7	1,564 (6)	22,117 (7)	**358,585 (9)**	**6,481,829 (9)**
8		29,867 (7)	379,655 (9)	6,599,913 (9)
9		39,080 (7)	520,648 (8)	10,142,002 (9)
10			677,805 (8)	15,214,336 (8)
11			841,576 (8)	22,068,332 (8)
12				27,498,382 (8)

Table 4.3: Average number of states generated using a single compressed partial PDB filled to 70% on K-pancake puzzle. Using BPMX.

different.

The compressed partial PDB generates more states than the normal partial PDB for three reasons. First, the table is only filled to 70% full, but this has a small effect because the unreached entries are filled with d, limiting the heuristic error. Second, the heuristic values in the perimeter are degraded by taking the minimal value of all states hashed to the same location. Third and most importantly, the heuristic correction of d is not applied to all states outside the perimeter. Because of collisions in the heuristic table, states outside the perimeter overlap with states inside the perimeter. Thus, the heuristic correction factor, d, is not applied to any states outside the perimeter that collide with a state inside the perimeter.

However, because of the construction method, the partial pattern PDB does not necessarily dominate the compressed partial PDB. Overlapping states cause the table to fill more slowly than the partial PDB. Thus the final cost-bound d may be greater in the compressed partial PDB than the partial PDB. This is seen in our example with the 12-pancake puzzle at abstraction-7: the partial PDB is built to $d = 8$ while the compressed partial PDB is built to $d = 9$.

In Table 4.2, as the granularity becomes finer, the point of diminishing returns is reached. In all examples, this occurs after adding one more unique tile to the original PDB abstraction. For the 8, 10, 12, and 13-pancakes, the optimal granularity is 5, 6, 7, and 7 unique tiles respectively. Further refining of the abstraction only causes the number of generated states to increase. This is due to the poor high-valued heuristics.

BPMX is an important improvement when using the inconsistent compressed partial PDBs. On the 12-pancake puzzle, with abstraction-7, using BPMX (Table 4.3), there are 358,585 generated states on average. This is an 44% improvement over not using BPMX (635,649 states generated).

The *improvement factor* is the improvement over the original abstraction (top entry in Table 4.1). This factor increases with puzzle size. Without BPMX, the 8, 10, 12, and 13-pancake puzzles' best improvement factors are 32%, 40%, 52%, and 53%; indicating that savings may scale favorably to larger problems. With BPMX, the best improvement factors are 50%, 63%, 73%, and 75%. This re-enforces the usefulness of incorporating BPMX into the search algorithm.

4.2 Performance with Constant Memory

As stated, partial pattern databases need extra memory to store the $stateID$. In the case of the 10-pancake puzzle, the amount of memory used to store the $stateID$ is about nine times larger than the heuristic value that is stored. To directly compare partial PDBs with full PDBs and compressed partial PDBs, the memory must be kept constant. So the number of entries in the partial PDB is limited appropriately.

The implementation of partial pattern databases uses the C++ standard template library, and as such gives little control over memory allocation. Therefore, the following formula is used to calculate the approximate number of entries that can fit into the memory limit if implemented as a hash table. Each full PDB entry consists of one byte, as does each compressed partial PDB entry. Each partial PDB entry consists of nine bytes (one for the heuristic value and eight for the $stateID$). Additionally, extra room is required to keep the hash table operating efficiently; we allow a 70% fill factor. Therefore, the approximate number of entries for the partial PDB that fits into the designated number of bytes is calculated using the formula: $entries_{partialPDB} = (bytes/9) * 0.7$.

10-Pancake Puzzle			
Memory Limit (bytes)	Normal PDB (unique tiles)	Best Partial PDB (unique tiles)	Best Compressed Partial PDB (unique tiles)
3,628,800	**40 (9)**	2,003 (7)	**40 (9)**
1,814,400	200 (8)	4,628 (7)	**70 (9)**
604,800	1,216 (7)	16,147 (6)	**417 (8)**
151,200	7,756 (6)	78,073 (5)	**2,695 (7)**
30,240	48,408 (5)	511,794 (5)	**18,026 (6)**
5,040	337,021 (4)	3,299,716 (4)	**135,352 (5)**

Table 4.4: Average number of states generated using a single PDB of the best abstraction granularity. The tests are performed on the 10-pancake puzzle while keeping memory constant.

Table 4.4 directly compares the performance of the three heuristic techniques (full PDBs, partial PDBs, and compressed partial PDBs) on the 10-pancake puzzle while keeping memory constant. The purpose of this experiment is to determine which technique makes the best use of available memory. The partial pattern databases have the number of entries calculated to fit into an efficient hash table of the allocated memory. The compressed partial pattern databases are filled to 70%. Each data entry is the average number of generated states over 100 random instances. For the partial PDB and compressed partial PDB, all different number of unique tiles are tested; this table shows only the best result. The associated number in parentheses depicts the number of unique tiles used to generate the data point. For each memory limit, the best-performing database is shown in bold. BPMX is used in combination with the compressed partial PDB.

Partial PDBs by themselves are not an efficient use of memory and as a result, performance suffers. In all tested cases, partial PDBs generate at least an order of magnitude more states than normal PDBs. However, the compressed partial PDBs are an efficient use of memory. The top row covers the entire space at the finest granularity; this is a perfect heuristic. In this case the normal PDB has slightly better performance because the compressed partial PDB is only filled to 70% (this is not apparent in the table because of averaging). For every row except the first, the improvement factor is between 60% and 65%, indicating similar performance gains when using finer-grained abstractions.

4.3 Analysis of the 12-Pancake Puzzle

This section analyzes the performance of partial pattern databases on the 12-pancake puzzle. Section 4.3.1 examines the distribution of heuristic values in the pattern database and during runtime.

A single search instance is examined in Section 4.3.2 and Korf and Reid's estimates on worst-case performance are shown to hold in Section 4.3.3.

4.3.1 Static and Runtime Heuristic Distribution

Holte *et al.*'s analysis in [21] explains that improved lower valued heuristics can improve search performance. To show this, they examine static and runtime heuristic distributions. This section examines similar results to verify this claim.

Table 4.5 shows the distribution of entries in partial PDBs for the 12-pancake puzzle. Each column represents a partial PDB and each data point shows the number of entries in the partial PDB with the specified heuristic value (left column). Each partial PDB has no more than 665,280 entries. A partial PDB can have fewer entries than the limit because all entries with a heuristic value of d can be removed without affecting the heuristic. The purpose of this table is to show the heuristic distribution of entries contained in the PPDB and to differentiate between the distribution of heuristic values over the entire state space (Table 4.6).

Every abstraction has only one entry with a heuristic value of zero. This corresponds to the abstract goal state. The partial PDB is filled until it reaches 655,280 entries. The coarser-grained abstractions fill more slowly, while the finer-grained abstractions fill more quickly. After the memory limit has been reached, all entries with a value of d are removed to save memory.

h	Number of Unique Tiles					
	11	10	9	8	7	6
0	1	1	1	1	1	1
1	11	10	9	8	7	6
2	110	100	90	80	70	60
3	1,099	989	863	727	587	449
4	9,883	8,553	7,038	5,488	4,023	2,733
5	77,937	65,245	50,690	36,421	23,885	13,917
6	533,397	426,630	307,149	197,996	111,831	52,898
7	-	-	-	-	391,115	137,041
8	-	-	-	-	-	216,065
9	-	-	-	-	-	173,590
10	-	-	-	-	-	62,359
11	-	-	-	-	-	6,161
d =	7	7	7	7	8	12
sum =	622,438	501,528	365,840	240,721	531,519	665,280

Table 4.5: Heuristic distribution of entries in each PPDB on the 12-pancake puzzle with 665,280 entries (or less).

Table 4.6 shows the distribution of heuristic values over the entire state space for a PPDB of 665,280 entries. This is *not* the number of entries; this is the total number of states in the entire state space that have a specific h value. Notice that as the abstraction level becomes more fine, fewer states in the state space are contained in the PPDB. More states are missed, causing more states to have the heuristic value of d. For example, in abstraction-11, 99.9% of the states have a heuristic value of d, while abstraction-6 has 0% of the states having a heuristic value of d.

Coarser abstraction levels, on the other hand, allow for higher heuristic values because of deeper backwards search. In abstraction-6, $d = 12$, but in abstraction-11, $d = 7$. Coarser abstraction groups some states with smaller heuristic values together. The backwards search quickly covers lower-valued states, but retains memory for larger heuristic values. Correspondingly, this increases the average heuristic value.

h	Number of Unique Tiles					
	11	10	9	8	7	6
h						
0	1	2	6	24	120	720
1	11	20	54	192	840	4,320
2	110	200	540	1,920	8,400	43,200
3	1,099	1,978	5,178	17,448	70,440	323,280
4	9,883	17,106	42,228	131,712	482,760	1,967,760
5	77,937	130,490	304,140	874,104	2,866,200	10,020,240
6	533,397	853,260	1,842,894	4,751,904	13,419,720	38,086,560
7	**478,379,162**	**477,998,544**	**476,806,560**	**473,224,296**	46,933,800	98,669,520
8	0	0	0	0	**415,219,320**	155,566,800
9	0	0	0	0	0	124,984,800
10	0	0	0	0	0	44,898,480
11	0	0	0	0	0	**4,435,920**
$d =$	7	7	7	7	8	11
$h_{avg} =$	7.00	7.00	6.99	6.99	7.82	8.03
$sum =$	479,001,600	479,001,600	479,001,600	479,001,600	479,001,600	479,001,600

Table 4.6: Static heuristic distribution over entire state space for PDB with various abstraction levels on 12-pancake puzzle.

Table 4.7 shows the heuristic values of the states generated during search (runtime). As noted in [21], the distribution of runtime heuristic values for abstraction-6, which is equivalent to the full PDB, is more heavily influenced by low values. The high heuristic values cause cutoffs early in the search tree, when the g-values are still small, and do not get visited often. The low heuristic values cause later cutoffs and get visited much more often. This is why the average heuristic value at runtime is so small.

For finer abstractions, this distribution is much different; most heuristic values are clustered around d. For the low g valued states, there are fewer cutoffs and all heuristic values are d. When the cutoffs start occurring (at high g values), the heuristic is more accurate, and fewer states are generated with small heuristic values (search phases are further explained in Section 4.3.2). The combination of these two effects makes the average heuristic value (avg h val) large for finer abstractions and small for coarse abstractions.

h	Number of Unique Tiles					
	11	10	9	8	7	6
h						
0	1	9	152	1,617	16,835	245,663
1	1	40	468	3,440	25,486	272,765
2	6	45	503	3,890	31,448	359,354
3	6	46	512	3,961	30,914	326,102
4	6	46	507	3,766	27,616	271,302
5	5	44	473	3,349	22,933	196,050
6	6	43	422	2,744	16,752	119,149
7	309,335	309,356	309,541	310,478	10,348	54,030
8	0	0	0	0	33,290	17,393
9	0	0	0	0	0	2,878
10	0	0	0	0	0	206
11	0	0	0	0	0	3
d =	7	7	7	7	8	11
avg h val =	7.00	7.00	6.96	6.74	3.92	2.84
sum =	309,367	309,629	312,577	333,247	215,622	1,864,895

Table 4.7: Runtime heuristic distribution of generated states for PDB with various abstraction levels on 12-pancake puzzle.

Note that the average heuristic value across the state space is largest for the coarse-grained abstraction-6 (Table 4.6). The increased d in abstraction-6 pulls up the average heuristic, while abstraction-11 is dominated by its lower d. This is interesting, as the best-performing PPDB is of abstraction-7, which has neither the lowest or highest average heuristic values. During runtime, the opposite happens; abstraction-6 has the lowest average heuristic and the fine-grained abstractions have the highest average heuristic (Table 4.7). In abstraction-6, most generated states have low h-values, while in the finer-grained abstractions, most generated states have $h = d$. Once again, abstraction-7 has a heuristic value in between the extremes. This indicates that for partial pattern databases, the average heuristic value (both static and runtime) is not a good indicator of search performance.

4.3.2 Examination of a Single Instance of Search

This section examines the states expanded on a single iteration of IDA* in-depth for abstraction-6, 7, and 8. The domain used is the 12-pancake puzzle. The purpose of this analysis is to quantify what is happening during search. The solution length for this problem is 12 and only the last iteration ($f_{max} = 12$) is examined. This iteration is fully expanded (worst-case analysis). In-depth tables are located in Appendix B (Tables B.1-B.6). Figure 4.1 summarizes the results of these tables.

The PPDB for abstraction-6 produces the exact same heuristic values as a PDB of the same abstraction level; this is why the PPDB and PDB searches match identically (Figure 4.1). The initial heuristic estimate is 9. Cutoffs start occurring fairly quickly at a g equals 3, previous to which the search is following brute-force search. The brute-force branching factor on the 12-pancake puzzle is 11 for the first iteration, and 10 for all others.

Abstraction-7 starts with an initial estimate of 8. The heuristic values tend to stay mostly around 8, until cutoffs start to occur at $g = 5$. Note that because cutoffs occurred at states with larger g-values, more states were generated with smaller g-values. However, after the first round of cutoffs, the number of states drops dramatically. This is because the heuristic value of 8 (where much of the heuristic distribution is located) now causes cutoffs. At high g-values, the search tree follows precisely the same as a search with the full PDB search.

Abstraction-8 takes this to a further extreme. Because the majority of heuristic values equal 7, the brute-force section of search extends a full ply further. However, because the heuristics contained in the PPDB are more accurate, the number of states drops significantly.

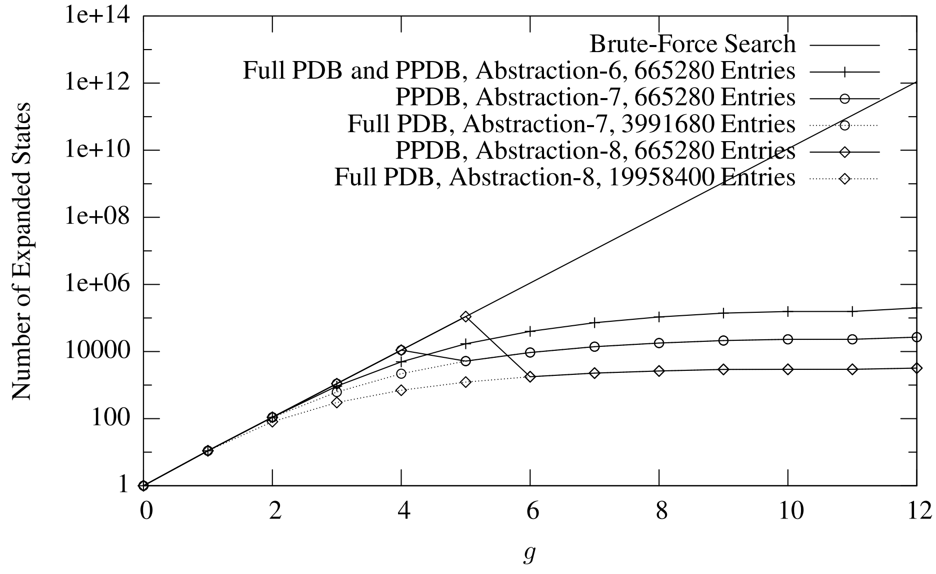

Figure 4.1: Comparison of PDBs vs. PPDBS. Number of expanded states at each depth g for a single search instance and single iteration ($f_{max} = 12$) run to completion.

Phases

Compare a PDB and PPDB, where the unique tiles of the PDB are a subset of the unique tiles of the PPDB. An example is a full PDB using abstraction-6 and a PPDB using abstraction-7. State expansion can be thought of in three phases:

1. Brute-force expansion for full PDB and partial PDB,

2. Cutoffs for full PDB, brute-force expansion for partial PDB, and

3. Cutoffs for full PDB and partial PDB. The finer-grained abstraction partial PDB will have more cutoffs.

In the first phase, all possible states are expanded because an h value that results in a cutoff has not yet been found. The branching factor is equal to brute-force search. This phase ends when the full PDB returns a heuristic sufficiently large enough to cause a cutoff.

The second phase happens when the full PDB causes cutoffs, but the PPDB is still expanding according to the brute-force tree. This happens because the PPDB may only be able to store a subset of the heuristic values in the full PDB. The values stored are the low ones, which cause cutoffs later in search.

The third phase starts when heuristic values in the PPDB start causing cutoffs. For PPDBs, since the majority of heuristic values equals d, a large decrease in states occurs when $g + d > f_{max}$. For abstraction-6, this occurs when $g = 3$; for abstraction-7, when $g = 4$; and for abstraction-8, when $g = 5$. During this last phase, state expansion using the PPDB once again exactly follows the state expansion using the full PDB of the same abstraction.

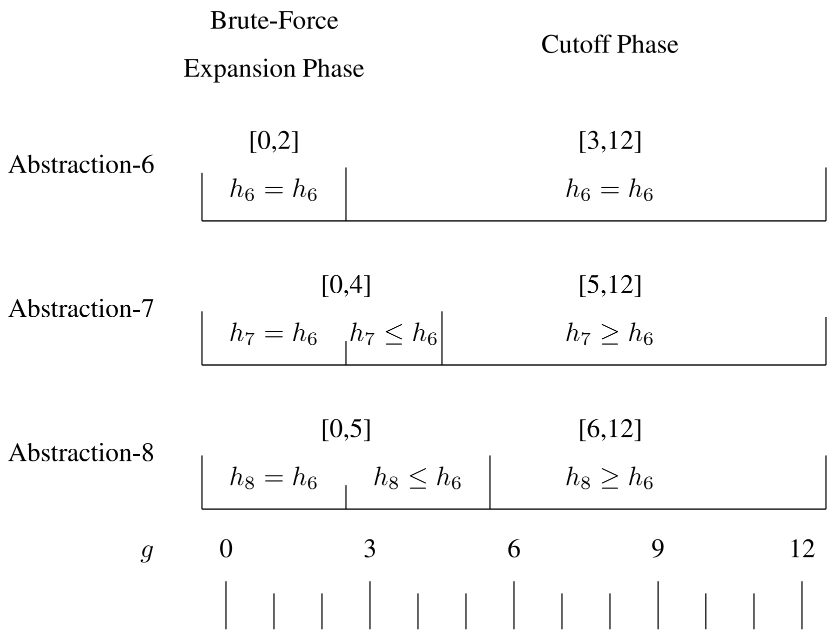

Figure 4.2: Phases of search for abstraction-6, 7, and 8.

This is empirically shown in Figure 4.1 which plots the brute-force search, search using PPDBs, and search using the corresponding full PDBs with the same abstraction levels. Each search with a PPDB expands states following the brute-force expansion curve exactly. After cutoffs start occurring, the curve drops to follow the full PDB curve with the same abstraction level.

Figure 4.2 illustrates where each phase occurs. The partial PDBs correspond to the three partial PDBs from Figure 4.1. Each partial PDB is compared to a full PDB of abstraction-6. The first line is the partial PDB at abstraction-6. This database has exactly the same entries as the full PDB at abstraction-6 and therefore, the same brute-force and cutoff phases. In the second line, abstraction-7 is compared to abstraction-6. The beginning of the brute-force search phase, from g equals 0 to

2, abstraction-7 expands the same states as the abstraction-6. However, abstraction-7's brute-force phases lasts longer, and has inferior heuristic values during the second phase (from g equals 3 to 4). The last phase, from g equals 5 to 12, abstraction-7's heuristic values are better than abstraction-6's heuristic values.

A similar trend is seen for abstraction-8 in the third line. The brute-force expansion phase lasts longer. when $3 \geq g \geq 5$, abstraction-8 has inferior heuristic values than abstraction-6. But the cutoff phase has better heuristic values. As the abstractions get more fine-grained, the low g-values are sacrificed for the higher g-values.

4.3.3 Korf and Reid Estimates

This section examines Korf and Reid's formula for predicting the worst-case number of state expansions on a single iteration of IDA* [31]. This number of states expanded E is approximated by

$$E(N, f_{max}, P) = \sum_{g=1}^{f_{max}+1} N_g P(f_{max} - g + 1)$$

where N_g is the number of states expanded in brute-force search, f_{max} is the cost bound, g is the cost so far, and $P(x)$ is the fraction of total states in the search space whose heuristic is less than or equal to x. Because the K-pancake puzzle is so regular, N_g can be estimated without defining the equilibrium fractions used in [31]. The 12-pancake puzzle has a branching factor of 11 for the first move, and 10 for every following move because the reverse operator is eliminated. Thus

$$N_g = \begin{cases} 11^g & \text{if } g \leq 1 \\ 11 * 10^{g-1} & \text{if } g > 1 \end{cases}$$

$P(x)$ is approximated using the static distribution of heuristic values over the state space to estimate P (calculated from Table 4.6).

Table 4.8 through Table 4.10 compare the predicted number of expanded states with the actual number of expanded states. This is for a single iteration of IDA* where $f_{max} = 12$, run until the level is complete. Two tests are run. The first test pays no attention to a start state's actual solution length ($solLength \geq 0$); the second test only examines the start states which have a solution length greater than or equal to 12. The results are averaged over 1000 unique starting states when the solution lengths when ignoring the solution length. Of the 1000 starting states, there are 269 with a solution length greater-than or equal-to 12. The results for the last column are averaged over these 269 starting states. The number of entries in each PPDB is held constant at 665,280. The purpose of this table is to determine whether the worst-case approximation formula accurately predicts the actual performance.

When the solution lengths are ignored ($solLength \geq 0$), the Korf and Reid formula predicts the actual number of states extremely closely for all partial pattern database abstractions. Recall that abstraction-6 performs the same as a full pattern database. This is because the branching factor is

g	Brute-Force	estimated	actual ($solLength \geq 0$)	actual ($solLength \geq 12$)
0	1	1	1	1
1	11	11	11	11
2	110	109	110	**110**
3	$1,100$	987	986	**849**
4	$11,000$	6,997	7,042	**4,762**
5	$1.1 * 10^5$	34,244	34,582	**17,355**
6	$1.1 * 10^6$	115,847	117,010	**43,838**
7	$1.1 * 10^7$	283,829	286,457	**83,452**
8	$1.1 * 10^8$	537,202	540,852	**128,716**
9	$1.1 * 10^9$	853,175	859,024	**174,848**
10	$1.1 * 10^{10}$	1,107,804	1,116,678	**202,741**
11	$1.1 * 10^{11}$	1,157,407	1,165,713	**205,248**
12	$1.1 * 10^{12}$	1,653,439	1,663,942	**269,667**
total:	$1,222 * 10^9$	5,751,052	5,792,408	**1,131,600**

Table 4.8: Estimated worst-case state expansion compared to the actual worst-case state expansion when $f_{max} = 12$ for abstraction-6 on the 12-pancake puzzle.

known exactly and the many heuristics with value d allow for a large, predictable reduction in states when $g + h > f_{max}$. This result is unrealistic, though, because 80% of the solution lengths are less than 12, so we are counting extra states that are not normally expanded.

When we examine only states with solutions greater than or equal to 12, the approximation is very inaccurate. In the case of abstraction-7, the estimated total number of expanded states overestimates by a factor of eight. The brute-force phase of the search is estimated accurately, but the cutoff phase (in bold) grossly overestimates the number of states. In abstraction-6, 7, and 8, the Korf and Reid formula overestimates the number of expanded states with $g = 12$ by a factor of 5, 8, and 16 respectively. Holte has previously observed this prediction inaccuracy when start state's solution lengths are taken into account [18].

This error in the approximation is caused by limiting the analysis to starting states with solutions greater than or equal to the depth examined. This is reasonable to do because IDA* does not continue to a deeper depth if the solution is found. What happens in the search instances where the solution length is less than 12 is that the goal is found through many paths, but the states around the goal continue to be expanded repeatedly. Because of transpositions, these states can be expanded an impressive number of times before being cutoff. This increases the number of states expanded with large g-values.

4.4 Conclusions

On the K-pancake puzzle, partial pattern databases are not an efficient use of memory. This is because they store the $stateID$. Compressed partial PDBs are an efficient use of memory and

g	Brute-Force	estimated	actual ($solLength \geq 0$)	actual ($solLength \geq 12$)
0	1	1	1	1
1	11	11	11	11
2	110	110	110	110
3	$1,100$	1,100	1,100	1,100
4	$11,000$	11,000	11,000	11,000
5	$1.1 * 10^5$	14,647	14,467	**4,346**
6	$1.1 * 10^6$	38,692	38,005	**8,231**
7	$1.1 * 10^7$	78,740	76,842	**12,837**
8	$1.1 * 10^8$	129,189	125,174	**17,270**
9	$1.1 * 10^9$	183,256	176,675	**21,139**
10	$1.1 * 10^{10}$	214,947	206,327	**22,865**
11	$1.1 * 10^{11}$	220,459	211,080	**22,958**
12	$1.1 * 10^{12}$	275,573	262,691	**27,064**
total:	$1,222 * 10^9$	1,167,724	1,123,484	**148,932**

Table 4.9: Estimated worst-case state expansion compared to the actual worst-case state expansion when $f_{max} = 12$ for abstraction-7 on the 12-pancake puzzle.

search performance can be further improved by using BPMX to exploit their inconsistency. The best abstraction level for compressed partial PDBs is slightly more fine-grained that the full PDB.

Careful analysis of single search instances shows that by sacrificing high-valued heuristics for improved low-valued heuristics, search performance can be improved. Korf and Reid's formula can be used to predict the worst-case performance of partial pattern databases on the 12-pancake puzzle. However, if solution lengths are taken into account, the prediction can be significantly inaccurate.

g	Brute-Force	estimated	actual ($solLength \geq 0$)	actual ($solLength \geq 12$)
0	1	1	1	1
1	11	11	11	11
2	110	110	110	110
3	$1,100$	1,100	1,100	1,100
4	$11,000$	11,000	11,000	11,000
5	$1.1 * 10^5$	110,000	110,000	110,000
6	$1.1 * 10^6$	13,267	13,367	**1,567**
7	$1.1 * 10^7$	23,548	23,867	**2,141**
8	$1.1 * 10^8$	34,744	35,312	**2,620**
9	$1.1 * 10^9$	44,974	45,731	**2,986**
10	$1.1 * 10^{10}$	49,052	49,767	**3,081**
11	$1.1 * 10^{11}$	49,603	50,076	**3,086**
12	$1.1 * 10^{12}$	55,115	55,180	**3,366**
total:	$1,222 * 10^9$	392,525	395,521	**141,070**

Table 4.10: Estimated worst-case state expansion compared to the actual worst-case state expansion when $f_{max} = 12$ for abstraction-8 on the 12-pancake puzzle.

Chapter 5

Experiments on the Fifteen-Puzzle

The following tests on the fifteen-puzzle are run with IDA*. This domain contrasts with the K-pancake puzzle by having a smaller branching factor (2.1 on average) and an existing heuristic. The heuristic is the maximum of the Manhattan Distance heuristic and the pattern database. The two-move cycle is eliminated by disallowing reverse operators. Results with partial PDBs are not reported because they were extremely poor for this domain.

5.1 Small Databases

The tests in this section use abstractions with don't-care tiles as the low-indexed tiles. The blank is always one of the unique tiles. For example, abstraction-7 for the fifteen-puzzle refers to the abstraction 'b x x x x x x x x x 10 11 12 13 14 15,' with x as a don't-care. Relatively small compressed partial PDBs are built (about 33MB) in this section to examine the effects of changing the parameters. The compressed partial PDBs are built using depth-first search and a transposition table. BMPX is not used on these tests unless specified.

5.1.1 Varying Abstraction Granularity

Figure 5.1 varies the abstraction granularity of the compressed partial PDB. The size of the compressed partial PDB is 33,554,432 bytes. The purpose of this test is to determine which abstraction granularities give the best performance, given our memory limit. The database is filled to 99%.

The total number of states generated over the 100 Korf problem set [26] is listed on the y-axis. The abstraction (number of unique tiles) is listed on the x-axis. On the secondary x-axis (top) is listed the total number of abstract states in the abstract state space.

On the left side of the graph, the size of the abstract state space is less than the size of the database. Therefore, only a small portion of the database is even being used and the performance is dominated by the Manhattan Distance heuristic. Moving to the right, the abstraction becomes more fine-grained. Abstraction-15 has no abstraction at all and acts like a perimeter. The abstractions at the center of the graph provide the best performance. The abstraction whose full PDB size

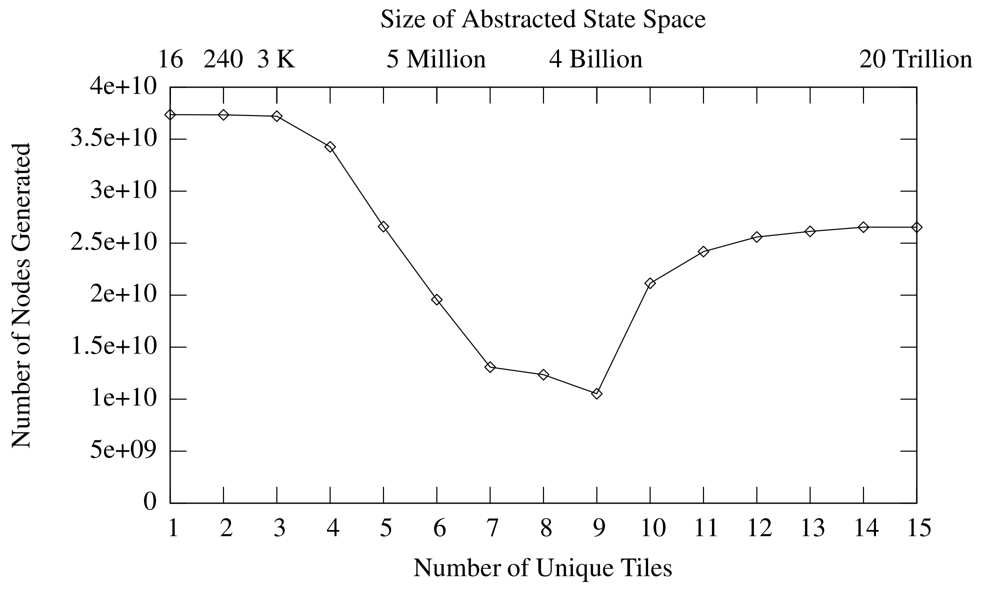

Figure 5.1: Varying abstraction granularity on the fifteen-puzzle (DB=33MB).

most closely matches the 33 million entries is abstraction-7 with 57,657,600 abstract states. One might expect abstraction-7 to perform the best. However, the finer-grained abstractions-8 and 9 outperform abstraction-7. This shows that finer granularity on the fifteen-puzzle can also provide better search performance.

5.1.2 Varying d

Figure 5.2 varies the cost bound d used when building the compressed partial PDB using no abstraction. The size of the compressed partial PDB is 33,554,432 bytes. The heuristic is the maximum of the Manhattan Distance heuristic and the compressed partial pattern database. The purpose of this test is to understand (1) how the database fills up and (2) how search performance is affected by the fill of a compressed partial PDB.

The total number of states generated over the 100 Korf problem set is listed on the y-axis. The compressed partial PDB is built to a cost bound of d, shown on the x-axis. The fill percentage is listed on the secondary y-axis (right side). Abstraction-15 (no abstraction) is used.

The compressed partial PDB is at most 33 million entries and the branching factor of the fifteen-puzzle is only 2.1. Therefore, it takes a large d before the database fill becomes noticeable. We see an exponential growth caused by the branching factor up to about 50% full, then the fill tapers off. This is because states begin overlapping significantly with already-existing states in the database.

The performance improves with the larger d and correspondingly larger fill factor. This is expected, as increasing d increases the heuristic values. But similarly, at the highest d, improvement in performance slows because the heuristics haven't changed very much. In addition, building a compressed partial PDB in a depth-first manner can take exponentially longer to increase d by one.

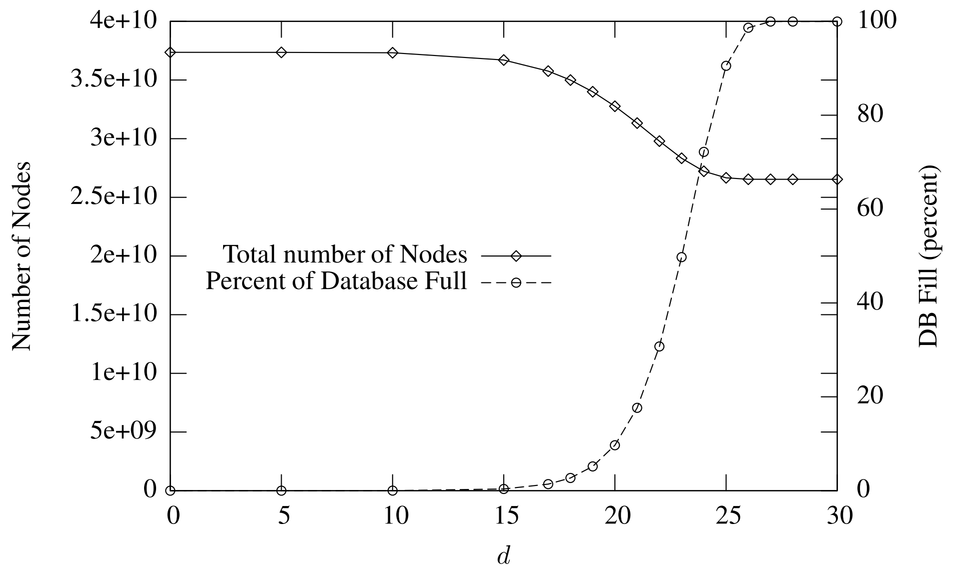

Figure 5.2: Varying d on the fifteen-puzzle (DB=33MB). No abstraction is used.

This approach shows that getting one last ply deeper is not always useful if your fill is already high.

5.2 High-Performance Tests

The abstractions used for the fifteen-puzzle are as follows: abstraction-8 is the *fringe* [6], ‘b x x 3 x x x 7 x x x 11 12 13 14 15’; and abstraction-9 adds one more unique tile, ‘b x x 3 x x x 7 x x 10 11 12 13 14 15’ (Figure 5.3). The heuristic used is the maximum of Manhattan Distance and the PDB or compressed partial PDB lookup.

	x	x	3
x	x	x	7
x	x	x	11
12	13	14	15

	x	x	3
x	x	x	7
x	x	10	11
12	13	14	15

Figure 5.3: Abstraction-8 (left) and abstraction-9 (right) used for the fifteen-puzzle.

Each test is run over all 100 Korf problem instances [26]. The databases compared are the full pattern database with abstraction-8 (PDB_8) and the compressed partial pattern database using abstraction-9 ($CPDB_9$). $CPDB_9$ is created from the full PDB using abstraction-9. Both PDB_8 and $CPDB_9$ are of size 518,918,400 bytes. IDA* is used with bidirectional pathmax (BPMX) in order to take advantage of the inconsistency in the compressed partial PDBs. The two-move cycle is eliminated by disallowing reverse operators.

The columns of Table 5.1 are as follows:

PDB	BPMX	Fill	d	avg. small	avg. medium	avg. large	total
PDB_8	DC	100	64	283,309	6,929,803	80,291,808	1,067,439,170
$CPDB_9$	Yes	50	38	792,425	14,465,508	123,546,247	1,840,334,929
$CPDB_9$	Yes	60	39	651,152	12,181,920	101,463,007	1,525,898,811
$CPDB_9$	Yes	70	40	554,045	10,529,666	86,096,055	1,304,112,453
$CPDB_9$	Yes	80	41	487,551	9,361,520	75,459,248	1,149,450,912
$CPDB_9$	Yes	90	43	409,084	7,943,644	62,933,272	965,285,000
$CPDB_9$	Yes	98	66	330,510	6,473,794	50,611,367	780,456,393
$CPDB_9$	No	98	66	539,065	9,917,842	82,560,198	1,242,278,013

Table 5.1: Number of states generated on the fifteen-puzzle using a single PDB technique and Manhattan Distance while keeping memory constant.

- The *PDB* is the type of pattern database used: either the full pattern database PDB_8 or the compressed partial pattern database $CPDB_9$.
- *BPMX* tells whether bidirectional pathmax is used: *Yes* it is used, *No* it is not used, and *DC* (don't care) means that BPMX has no effect.
- *Fill* shows the percentage of memory that is expanded when creating the pattern database. Because of the hashing scheme, however, even when filled as much as possible, 2% of $CPDB_9$ remains unused.
- For PDB_8, d is the largest value in the database. For $CPDB_9$, d is the cost bound (as defined in Chapter 3).
- The *small* problems are the problems that result in searches with less than 1,000,000 generated states when solved with PDB_8. *Medium* problems require between 1,000,000 and 31,999,999 generated states. The *hard* problems require greater than or equal to 32,000,000 generated states. There are 43 small problems, 48 medium problems, and 9 hard problems. The average number of states expanded in each of the three problem sets are reported.
- *total* is the total number of generated states over the 100 Korf problem instances. This is not the average number of generated states.

At less than 90% full, using BPMX, $CPDB_9$ generates more total states than PDB_8. $CPDB_9$ generates slightly fewer states when at 90% full, and at 98% full the total number of generated states is decreased by 27%. However, this result can be slightly misleading, since the total is dominated by the largest searches (which generated over three orders of magnitude more states). With and without BPMX, the small problems have worse performance using $CPDB_9$ than PDB_8. However, the large problems have improved performance when using BPMX and filled to 80% or more. When $CPDB_9$ is 98% full, the small, medium, and large problems have -95%, 6%, and 36% improvement respectively, in average number of states generated. The use of CPDBs results in poor performance on small problems, but the performance improves with the problem difficulty.

The last row shows the importance of using BPMX in the fifteen-puzzle to improve performance. Using BPMX leads to a 37% reduction in the total number of states generated when $CPDB_9$ is 98% full. This pushes the performance ahead of PDB_8. Small, medium, and large instances benefit equally from BPMX, improving the number of generated states by 39%, 35%, and 39% respectively. This indicates that the influence of BPMX may not depend on instance difficulty.

5.3 Conclusions

Compressed partial pattern databases are shown to be effective on the fifteen-puzzle. Like the K-pancake puzzle, the best abstraction granularity is slightly more fine-grained than the full PDB that best matches the memory limit. Also, the performance improvement is shown to closely match the fill of the compressed partial PDB. However, total reduction in states generated on high-performance tests is only 37%, which is significantly smaller than the improvement on the K-pancake puzzle.

Chapter 6

Conclusions

6.1 Summary

Search in the puzzle domains with a minimal-memory search technique, such as IDA*, allows all available memory to go toward the heuristic. Pattern databases and perimeters are memory-based heuristic techniques (described in Chapter 2). They both require a predecessor function to perform retrograde search for the purpose of improving or creating a heuristic. However, pattern databases additionally require an abstraction mechanism. The similarities between the two techniques allows us to combine approaches in a general way.

In Chapter 3, this thesis presents two new heuristic lookup table techniques, partial pattern databases and compressed partial pattern databases, which merge the ideas of front-to-goal perimeter search and full pattern databases. Partial pattern databases act similarly to perimeters but with abstraction; partial PDBs store a subset of abstract states around the goal to use as a heuristic. Compressed partial PDBs act more like a normal pattern database; each entry stores the minimum value of all abstract states mapping to the same location.

Both techniques decouple the abstraction granularity from the memory limit, allowing the programmer to choose the best available abstraction for a given domain and memory limit. They are both applicable to domains with a predecessor function and a space abstraction technique; can be used for multiple search instances with a single goal; and a heuristic calculation requires one simple table lookup. Both partial PDBs and compressed partial PDBs are admissible, but only partial PDBs are guaranteed consistent.

Partial pattern databases however are not an efficient use of memory; every heuristic entry must additionally record the $stateID$. This limitation is undesirable. Thus compressed partial PDBs remove the $stateID$, storing heuristics in a memory-efficient manner (like normal PDBs). As a result, compressed partial PDBs lose information which is partially recovered when using IDA* with bidirectional pathmax.

Chapter 4 shows that partial PDBs are not memory efficient in either of our domains, the K-pancake puzzle or the fifteen-puzzle. In the K-pancake puzzle, a three-fold improvement in the average number of states generated was achieved. Chapter 4 also explains why our approach yields fewer states, increasing low heuristic values can have a positive effect. The Korf and Reid formula for predicting worst-case performance of IDA* for full PDBs does not work well for partial PDBs on the 12-pancake puzzle. Therefore, this may not be a viable approach for choosing the best abstraction granularity. The fifteen-puzzle performance does not produce as large an improvement (Chapter 5). A finer-grained CPDB outperforms the full PDB by 37% when filled to 98%. For both test domains, BPMX has proven useful in correcting inconsistency caused by compressed partial PDBs.

6.2 Limitations and Future Work

One immediate conclusion that might draw from this thesis is that abstraction should always be used with perimeters. However, the perimeter approach used in this thesis (partial pattern databases without abstraction) provides only a front-to-goal heuristic. Traditional perimeter search requires multiple heuristic lookups to create a front-to-front heuristic that further improves search performance [9, 35]. Because partial pattern databases only provide a heuristic estimate to the goal, they cannot in general use multiple lookups to improve the heuristic. However, Section 6.2.2 presents one domain where this might be possible.

Building compressed partial PDBs requires careful attention. Using a modular hash function can sometimes result in a heuristic lookup table that is exactly the same as a coarser-grained normal PDB. There are various methods with which to build a compressed partial PDB, but to do so efficiently requires extra memory or disk. The construction of full PDBs is much more efficient in comparison.

The optimal level of abstraction in Chapter 4 was chosen by exhaustive experimentation. Chapter 4.3 presents a possible way for choosing the most appropriate abstraction granularity based on the heuristic distribution of the PDB. However, this is not a general approach. If this technique is useful for other domains, choosing the best abstraction may prove problematic.

There are a fair number of possible extensions that are not tested in this thesis. The most interesting ones are discussed here.

6.2.1 Domain-Specific Improvements

This thesis presents, implements, and tests partial PDBs in as general terms as possible. This technique can be further incorporated with other general methods. For example, the maximum can be taken over multiple heuristic lookup tables,whether they be PDBs, partial PDBs, or compressed partial PDBs [21]. As well, domain-specific adaptations and improvements can be integrated into this framework. Two glaring examples are additivity in the 15-puzzle and duality in the pancake puzzle (Section 6.2.2).

On the 15-puzzle, partial PDBs can always be made into additive versions by ignoring specific tile movements [30]. Fan Yang, Joseph Culberson, and Robert Holte generalized the definition of additivity and applied it with great effect to the K-pancake puzzle [49]. Compressed partial PDBs can effectively compress larger full additive pattern databases into memory-efficient versions. However, it is suspected that this approach may not prove very advantageous, as additivity was originally invented to combine heuristic values of two large pattern databases. This approach on the fifteen-puzzle will increase the granularity of one pattern database at the cost of decreasing granularity of another pattern database. Or two databases might be used instead of three. It is doubtful that the improvement in heuristic values will offset the loss caused by this approach.

6.2.2 Front-to-Front Heuristics

This thesis examined the use of a simple front-to-end heuristic lookup table. Recall, however, that perimeters proved useful when combined with a heuristic that could estimate the distance between any two states in the state space. This is called front-to-end heuristic search.

On the pancake puzzle, any pattern database technique to get a heuristic to the goal can be used. By using the *general duality principal* [14], it is possible to get an admissible heuristic between any two states. First, map the operator sequence Π between two states onto the goal to get state n^d. Then, look up $h(n^d)$ from the PDB. This allows for front-to-front heuristic improvement using a partial pattern database. The perimeter search techniques described by Dillenburg and Nelson [9], and Manzini [35] would be applicable.

6.2.3 Breadth-First A* and swapping

Section 4.3 shows graphically how cutoffs start occurring at specific g-values. Regular PDBs generally have higher heuristic values and cause more cutoffs during search on the states with a low g-value, while finer-grained partial PDBs have lower heuristic values and cause the search to cutoff more states with a high g-value. The benefits of both PDBs can be gained by taking the maximum value. However, both PDBs still have to fit in memory and would require two lookups.

IDA* expands states in a depth-first manner, meaning that the current state being examined could have any g-value at any time. However, if states are expanded in a breadth-first manner, using breadth-first A* [54], then states are expanded by increasing g-value. The states with a low g occur at the beginning of the search while states with a large g occur near the end. For the states at the beginning of the search (with a low g-value), only high valued heuristics will cause a cutoff.

Examine partial pattern databases. Compare two partial PDBs with abstractions where the coarse-grained abstraction has unique tiles that consist of a subset of the unique tiles from the fine-grained abstraction. The coarse-grained PDB dominates the fine-grained PDB on states with low g-values, but the fine-grained PDB dominates the coarse-grained for states with high g-values. Since states are expanded in increasing order of g, only the dominating partial PDB for the current value of g is needed. At some time the current g-value will increase to the point where the other partial PDB now dominates. Swap out the first partial PDB and use the second one exclusively, for the period that the second partial PDB dominates.

The search time now becomes additionally dependent on the time it takes to swap partial PDBs in and out of memory. Therefore, this strategy would be useful only on large search instances. Preliminary testing shows that swapping tables of 500 megabytes takes 14 seconds, so this strategy should be viable.

Breadth-first search takes up a certain amount of memory itself, keeping track of states on the open list. Using a PDB for the heuristic will require the sharing of memory between the heuristic and the search technique. By using a partial PDB for the heuristic, memory can be periodically freed for other purposes (the search technique) by eliminating the portions of the partial PDB that no longer have an impact on the search. For example, if minimum g-value increases by one, all the highest entries in the partial PDB can be removed and h_{max} decreased by one, while still causing all the same cutoffs.

Taking this a step further, if two partial PDBs are kept in memory at the same time, the partial PDB not in use can be swapped out of memory while the search continues. This approach is similar to double buffering of the screen in graphics applications.

This technique should work well with partial PDBs because they are predictable. If two partial PDBs are defined as previously mentioned, one partial PDB will dominate for a consecutive set of g-values. The second partial PDB will dominate for the other g-values. We can determine the exact time to swap one partial PDB for the other.

6.2.4 Binary Decision Diagrams

As this thesis has shown, partial pattern databases utilize memory inefficiently. They must store the full $stateID$ with the heuristic value. Almost always the $stateID$ takes much more memory than the heuristic, in the tested domains, about nine times more memory. This thesis combats this problem by eliminating the state information and taking the smallest heuristic value in compressed partial PDBs. However, compressed partial PDBs lose a lot of information. Is it possible to losslessly compress partial PDBs?

Marcel Ball and Rob Holte are investigating the use of binary decision diagrams and algebraic decision diagrams to losslessly compress partial pattern databases. The hope is that the $stateID$ can be compacted such that the performance improvement will overtake the extra memory required for a partial PDB as compared to a full PDB. Current results for the K-pancake puzzle and fifteen-puzzle show that the compression enables the increase of d by one or two. This causes a minor improvement in the number of states expanded by IDA*. However, additional time is required for quering the binary and algebraic decision diagrams, resulting in more time overall [2].

6.3 The Final Word

There has been a lot of interest in recent years regarding the use of perimeters in combination with pattern databases. There are problems with combining the techniques in a general way. This thesis presented a few approaches in Chapter 2. This is a tricky problem, therefore much research has exploited domain-specific properties to get to maximal performance gain.

The main contribution of this thesis was the new approach to combining perimeters and pattern databases. Partial pattern databases and compressed partial PDBs are general enough to be applied to any single-agent search domain with a predecessor function and an abstraction mechanism, and can be used over multiple problem instances with the same goal. Compressed partial PDBs combined with BPMX prove most effective on the K-pancake puzzle, and less successful on the fifteen-puzzle.

The second contribution of this thesis is the new analysis technique used on the 12-pancake puzzle. States in the search tree are grouped by their g value. This allows analysis on how the heuristic distribution will affect the search tree. This also explains why maxing can sometimes prove very desirable.

Much interesting research has been done over the past 15 years in speeding single-agent search with abstraction. This thesis presents a technique whereby the abstraction used is more fine-grained than a full pattern database. One future possibility is the use of multiple abstraction granularities for improved performance and reduced memory requirements. This thesis frees the single-agent search community to use the different abstraction levels to best find solutions.

Bibliography

[1] Kenneth Anderson, Robert Holte, and Jonathan Schaeffer. Partial pattern databases. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, 2007. (To appear).

[2] Marcel Ball and Robert Holte. Personal correspondence, 2007.

[3] Partha P. Chakrabarti, Subrata Ghose, Arup Acharya, and S. C. de Sarkar. Heuristic search in restricted memory (research note). *Artificial Intelligence*, 41(2):197–222, 1989.

[4] Charles Q. Choi. Mars rovers get four upgrades. http://www.space.com/missionlaunches/070214_smart_rovers.html, 2007.

[5] Joseph C. Culberson and Jonathan Schaeffer. Efficiently searching the 15-puzzle. Technical Report TR 94-08, Department of Computing Science, University of Alberta, 1994.

[6] Joseph C. Culberson and Jonathan Schaeffer. Searching with pattern databases. In *Canadian Conference on AI*, pages 402–416, 1996.

[7] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

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

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

[10] Stefan Edelkamp. Planning with pattern databases. In *ECP: European Conference on Planning*, pages 13–34, Toledo, 2001.

[11] Ariel Felner and Amir Adler. Solving the 24 puzzle with instance dependent pattern databases. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, pages 248–260, 2005.

[12] Ariel Felner, Richard E. Korf, and Sarit Hanan. Additive pattern database heuristics. *JAIR: Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[13] Ariel Felner and Nir Ofek. Combining perimeter search and pattern database abstractions. In *Symposium on Abstraction Reformulation and Approximation (SARA)*, 2007.

[14] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert Holte. Dual lookups in pattern databases. In *IJCAI*, pages 103–108, 2005.

[15] Richard D. Greenblatt, Donald E. Eastlake, and Stephen D. Crocker. The Greenblatt chess program. In *Fall Joint Computer Conference 31*, volume 73, pages 801–810, 1967.

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

[17] Heath Hohwald, Ignacio Thayer, and Richard E. Korf. Comparing best-first search and dynamic programming for optimal multiple sequence alignment. In *IJCAI*, pages 1239–1245, 2003.

[18] Robert Holte. Personal correspondence, 2006.

[19] Robert Holte and István T. Hernádvölgyi. Experiments with automatically created memory-based heuristics. In *Symposium on Abstraction, Reformulation and Approximation*, pages 281–290, 2000.

[20] Robert C. Holte, Jeffery Grajkowski, and Brian Tanner. Hierarchical heuristic search revisited. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, pages 121–133, 2005.

[21] Robert C. Holte, Jack Newton, Ariel Felner, Ram Meshulam, and David Furcy. Multiple pattern databases. In *ICAPS*, pages 122–131, 2004.

[22] Robert C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI*, pages 530–535, 1996.

[23] Space Telescope Science Institute. Spike. http://www.stsci.edu/resources/software_hardware/spike/, 2007.

[24] Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

[25] Richard E. Korf. Toward a model of representation changes. *Artificial Intelligence*, 14(1):41–78, 1980.

[26] Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[27] Richard E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the Workshop on Computer Games (W31) at IJCAI*, pages 21–26, Nagoya, Japan, 1997.

[28] Richard E. Korf. A divide and conquer bidirectional search: First results. In *IJCAI*, pages 1184–1191, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[29] Richard E. Korf. Delayed duplicate detection: Extended abstract. In *IJCAI*, pages 1539–1541, 2003.

[30] Richard E. Korf and Ariel Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134(1-2):9–22, 2002.

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

[32] Richard E. Korf and Weixiong Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 910–916, 2000.

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

[34] Plunkett Research Ltd. Entertainment & media industry. http://www.plunkettresearch.com, 2007.

[35] Giovanni Manzini. BIDA*: An improved perimeter search algorithm. *Artificial Intelligence*, 75(2):347–360, 1995.

[36] Matthew McNaughton, Paul Lu, Jonathan Schaeffer, and Duane Szafron. Memory-efficient A* heuristics for multiple sequence alignment. In *AAAI/IAAI*, pages 737–743, 2002.

[37] Laszlo Mero. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, 23(1):13–27, 1984.

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

[39] Ira Pohl. First results on the effect of error in heuristic search. In *Machine Intelligence*, volume 5, pages 219–236, 1970.

[40] Christopher Raphael. Coarse-to-fine dynamic programming. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(12):1379–1390, 2001.

[41] Alexander Reinefeld and T. Anthony Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(7):701–710, 1994.

[42] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 2003.

[43] Stuart J. Russell. Efficient memory-bounded search methods. In *10th European Conference on Artificial Intelligence (ECAI)*, pages 1–5, Vienna, Austria, 3–7 August 1992. Wiley.

[44] Jonathan Schaeffer. Personal correspondence, 2007.

[45] Anup K. Sen and Amitava Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *IJCAI*, pages 297–302, 1989.

[46] David J. Slate and Lawrence R. Atkin. *Chess Skill in Man and Machine*, chapter Chess 4.5- The Northwestern University Chess Program, pages 82–118. Springer-Verlag, 1977.

[47] Larry A. Taylor and Richard E. Korf. Pruning duplicate nodes in depth-first search. In *AAAI/IAAI*, pages 756–761, 1993.

[48] Ken Thompson. Retrograde analysis of certain endgames. *International Computer Chess Association Journal*, 8(3):131–139, 1986.

[49] Fan Yang, Joseph Culberson, and Robert Holte. A general additive search abstraction. Technical Report TR07-06, Computing Science, University of Alberta, 2007.

[50] Uzi Zahavi, Ariel Felner, Robert Holte, and Jonathan Schaeffer. Dual search in permutation state spaces. In *AAAI/IAAI*, pages 1076–1081, 2006.

[51] Rong Zhou and Eric A. Hansen. Breadth-first heuristic search. In *ICAPS*, pages 92–100, 2004.

[52] Rong Zhou and Eric A. Hansen. Space-efficient memory-based heuristics. In *AAAI/IAAI*, pages 677–682, 2004.

[53] Rong Zhou and Eric A. Hansen. External-memory pattern databases using structured duplicate detection. In *AAAI/IAAI*, pages 1398–1405, 2005.

[54] Rong Zhou and Eric A. Hansen. Breadth-first heuristic search. *Artificial Intelligence*, 170(4):385–408, 2006.

Appendix A

Number of States Generated on the K-pancake Puzzle with various abstractions

The tables reported in Chapter 4 were built by taking the best possible abstraction level. This Appendix shows the testing of each abstraction level that was used to produce the tables in Chapter 4. The average number of generated states over 100 test problems are reported.

The abstraction level and the number of entries are varied. The abstractions used for the K-pancake puzzle have don't-care tiles as the low indexed tiles. For example, abstraction-7 for the 12-pancake puzzle refers to the abstraction 'x x x x x 5 6 7 8 9 10 11,' with x as a don't-care. When using a compressed partial pattern database, it is filled to 70% full.

All tests are run using IDA*. The two move cycle (returning to the parent state) is eliminated by disallowing the reverse operator. BPMX is used when explicitly stated, but will only have an effect inconsistent heuristics. In our case, this is the compressed partial pattern database.

A.1 8-Pancake Puzzle

abstraction	0	1	2	3	4	5	6	7
abstract state space size	1	8	56	336	1,680	6,720	20,160	40,320
entries ↓								
40,320								25
20,160							110	
6,720						427		
1,680					2,065			
336				10,477				
56			58,192					
8		365,793						
1	2,490,999							

Table A.1: 8-pancake puzzle with full pattern database.

abstraction	0	1	2	3	4	5	6	7
abstract state space size	1	8	56	336	1,680	6,720	20,160	40,320
entries ↓								
40,320	2,490,999	365,793	58,192	10,477	2,065	427	110	25
20,160	2,490,999	365,793	58,192	10,477	2,065	427	110	30
6,720	2,490,999	365,793	58,192	10,477	2,065	427	149	71
1,680	2,490,999	365,793	58,192	10,477	2,065	682	403	335
336	2,490,999	365,793	58,192	10,477	3,593	2,227	1,990	1,933
56	2,490,999	365,793	58,192	19,392	12,848	11,763	11,582	11,540
8	2,490,999	365,793	107,798	74,821	70,100	69,348	69,227	415,168
1	2,490,999	2,490,999	2,490,999	2,490,999	2,490,999	2,490,999	2,490,999	2,490,999

Table A.2: 8-pancake puzzle with partial pattern database.

abstraction	0	1	2	3	4	5	6	7
abstract state space size	1	8	56	336	1,680	6,720	20,160	40,320
entries ↓								
40,320	2,490,999	365,793	58,192	10,477	2,065	427	110	25
20,160	2,490,999	365,793	58,192	10,477	2,065	427	110	110
6,720	2,490,999	365,793	58,192	10,477	2,065	428	428	428
1,680	2,490,999	365,793	58,192	10,477	2,065	2,065	2,065	2,065
336	2,490,999	365,793	58,192	10,477	10,477	10,477	10,477	10,477
56	2,490,999	365,793	58,192	58,192	58,192	58,192	58,192	58,192
8	2,490,999	365,793	365,793	365,793	365,793	365,793	365,793	365,793
1	2,490,999	2,490,999	2,490,999	2,490,999	2,490,999	2,490,999	2,490,999	2,490,999

Table A.3: 8-pancake puzzle with compressed partial pattern database (70% full) without BPMX.

abstraction	0	1	2	3	4	5	6	7
abstract state space size	1	8	56	336	1,680	6,720	20,160	40,320
entries ↓								
40,320	2,490,999	365,793	58,192	10,477	2,065	427	110	25
20,160	2,490,999	365,793	58,192	10,477	2,065	427	110	47
6,720	2,490,999	365,793	58,192	10,477	2,065	428	213	257
1,680	2,490,999	365,793	58,192	10,477	2,065	1,024	1,227	1,564
336	2,490,999	365,793	58,192	10,477	5,626	6,318	8,533	10,017
56	2,490,999	365,793	58,192	36,926	38,600	42,254	47,652	55,212
8	2,490,999	365,793	289,086	216,675	311,838	313,607	382,233	382,233
1	2,490,999	1,433,689	1,433,689	1,433,689	1,433,689	1,433,689	1,433,689	1,433,689

Table A.4: 8-pancake puzzle with compressed partial pattern database (70% full) with BPMX.

A.2 10-Pancake Puzzle

abstraction	3	4	5	6	7	8	9
abstract state	720	5,040	30,240	151,200	604,800	1,814,400	3,628,800
space size							
entries ↓							
3,628,800							40
1,814,400						200	
604,800					1,216		
151,200				7,756			
30,240			48,408				
5,040		337,021					
720	2,428,225						

Table A.5: 10-pancake puzzle with full pattern database.

abstraction	3	4	5	6	7	8	9
abstract state	720	5,040	30,240	151,200	604,800	1,814,400	3,628,800
space size							
entries ↓							
3,628,800	2,428,225	337,021	48,408	7,756	1,216	200	40
1,814,400	2,428,225	337,021	48,408	7,756	1,216	200	49
604,800	2,428,225	337,021	48,408	7,756	1,216	298	145
151,200	2,428,225	337,021	48,408	7,756	2,026	1,072	930
30,240	2,428,225	337,021	48,408	14,268	8,251	7,370	7,242
5,040	2,428,225	337,021	100,594	64,077	58,655	57,883	57,773

Table A.6: 10-pancake puzzle with partial pattern database.

abstraction	3	4	5	6	7	8	9
abstract state	720	5,040	30,240	151,200	604,800	1,814,400	3,628,800
space size							
entries ↓							
282,240		337,021	48,408	7,756	2,003	2,713	4,170
141,120		337,021	48,408	7,954	4,628	7,845	10,356
47,040		337,021	48,408	16,147	24,682	35,266	45,264
11,760		337,021	78,073	95,245	138,749	182,124	219,469
2,352		516,027	511,794	691,011	914,429	1,129,918	1,312,288
392	3,864,429	3,299,716	4,057,074	5,055,689	5,891,946	6,844,385	7,496,125

Table A.7: 10-pancake puzzle with partial pattern database. This table uses different amounts of memory that Table A.6 for direct memory comparison.

abstraction	3	4	5	6	7	8	9
abstract state	720	5,040	30,240	151,200	604,800	1,814,400	3,628,800
space size							
entries ↓							
3,628,800							40
1,814,400						200	119
604,800	2,428,225	337,021	48,408	7,756	1,216	717	1,131
151,200	2,428,225	337,021	48,408	7,757	4,780	7,508	9,190
30,240	2,428,225	337,021	48,414	28,839	40,558	54,026	68,488
5,040	2,428,225	337,050	211,858	257,578	369,384	466,507	571,404

Table A.8: 10-pancake puzzle with compressed partial pattern database (70% full) without BPMX.

abstraction	3	4	5	6	7	8	9
abstract state	720	5,040	30,240	151,200	604,800	1,814,400	3,628,800
space size							
entries ↓							
3,628,800							40
1,814,400						200	70
604,800	2,428,225	337,021	48,408	7,756	1,216	417	542
151,200	2,428,225	337,021	48,408	7,757	2,695	3,845	4,738
30,240	2,428,225	337,021	48,414	18,026	22,117	29,867	39,080
5,040	2,428,225	337,050	135,352	147,512	218,664	292,115	370,064

Table A.9: 10-pancake puzzle with compressed partial pattern database (70% full) with BPMX.

A.3 12-Pancake Puzzle

abstraction	6	7	8	9	10	11
abstract state	665,280	3,991,680	19,958,400	79,833,600	239,500,800	479,001,600
space size						
entries ↓						
479,001,600						59
239,500,800					599	
79,833,600				3,127		
19,958,400			19,946			
3,991,680		164,348				
665,280	1,316,273					

Table A.10: 12-pancake puzzle with full pattern database.

abstraction	6	7	8	9	10	11
abstract state	665,280	3,991,680	19,958,400	79,833,600	239,500,800	479,001,600
space size						
entries ↓						
665,280	1,316,273	178,464	183,172	167,584	165,390	164,951

Table A.11: 12-pancake puzzle with partial pattern database.

abstraction	6	7	8	9	10	11
abstract state	665,280	3,991,680	19,958,400	79,833,600	239,500,800	479,001,600
space size						
entries ↓						
3,991,680	1,316,273	164,349	78,191	118,818	151,325	190,427
665,280	1,316,284	635,649	861,365	1,096,865	1,381,920	1,708,840

Table A.12: 12-pancake puzzle with compressed partial pattern database (70% full) without BPMX.

abstraction	6	7	8	9	10	11
abstract state	665,280	3,991,680	19,958,400	79,833,600	239,500,800	479,001,600
space size						
entries ↓						
3,991,680	1,316,273	164,349	40,333	51,059	66,813	86,964
665,280	1,316,284	358,585	379,655	520,648	677,805	841,576

Table A.13: 12-pancake puzzle with compressed partial pattern database (70% full) with BPMX.

A.4 13-Pancake Puzzle

abstraction	6	7	8	9	10	11	12
abstract state	1,235,520	8,648,640	51,891,840	259,459,200	1,037,836,800	3,113,510,400	6,227,020,800
space size							
entries ↓							
1,235,520	25,833,998						

Table A.14: 13-pancake puzzle with full pattern database.

abstraction	6	7	8	9	10	11	12
abstract state	1,235,520	8,648,640	51,891,840	259,459,200	1,037,836,800	3,113,510,400	6,227,020,800
space size							
entries ↓							
1,235,520	25,833,998	3,106,345	4,097,683	3,851,260	3,820,667	3,816,931	3,816,546

Table A.15: 13-pancake puzzle with partial pattern database.

abstraction	6	7	8	9	10	11	12
abstract state	1,235,520	8,648,640	51,891,840	259,459,200	1,037,836,800	3,113,510,400	6,227,020,800
space size							
entries ↓							
1,235,520	25,834,132	12,021,692	16,216,581	23,152,643	32,159,310	44,332,028	51,689,205

Table A.16: 13-pancake puzzle with compressed partial pattern database (70% full) without BPMX.

abstraction	6	7	8	9	10	11	12
abstract state	1,235,520	8,648,640	51,891,840	259,459,200	1,037,836,800	3,113,510,400	6,227,020,800
space size							
entries ↓							
1,235,520	25,834,132	6,481,829	6,599,913	10,142,002	15,214,336	22,068,332	27,498,382

Table A.17: 13-pancake puzzle with compressed partial pattern database (70% full) with BPMX.

Appendix B

Runtime Histograms on 12-Pancake Puzzle

This appendix shows tables depicting the final search iteration, completely expanded, on the 12-pancake puzzle using various pattern databases. The solution length is 12. The g-value is displayed along the x axis and the h-value is displayed along the y axis. The number of states expanded with these specific h and g values form the table entries.

B.1 Partial PDBs

g h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	total
0	0	0	0	0	0	0	0	0	0	17	544	12,082	199,395	0	212,038
1	0	0	0	0	0	0	0	0	17	459	9,532	144,425	0	0	154,433
2	0	0	0	0	0	0	0	17	459	9,600	145,751	0	0	0	155,827
3	0	0	0	0	0	0	17	441	8,770	129,523	0	0	0	0	138,751
4	0	0	0	0	0	16	366	6,829	97,867	0	0	0	0	0	105,078
5	0	0	0	0	13	272	4,755	64,902	0	0	0	0	0	0	69,942
6	0	0	0	8	183	2,729	34,780	0	0	0	0	0	0	0	37,700
7	0	0	6	101	1,229	13,991	0	0	0	0	0	0	0	0	15,327
8	0	2	34	315	3,508	0	0	0	0	0	0	0	0	0	3,859
9	1	6	44	505	0	0	0	0	0	0	0	0	0	0	556
10	0	3	26	0	0	0	0	0	0	0	0	0	0	0	29
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
expanded	1	11	110	929	4,933	17,008	39,918	72,189	107,113	139,599	155,827	156,507	199,395	0	893,540

Table B.1: Histogram Results for partial PDB with abstraction-6 on 12-pancake puzzle.

h \ g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	total
0	0	0	0	0	0	0	0	0	0	0	56	1,427	26,831	0	28,314
1	0	0	0	0	0	0	0	0	0	56	1,203	21,683	0	0	22,942
2	0	0	0	0	0	0	0	0	56	1,203	21,851	0	0	0	23,110
3	0	0	0	0	0	0	0	52	1,108	19,997	0	0	0	0	21,157
4	0	0	0	0	0	0	47	962	16,721	0	0	0	0	0	17,730
5	0	0	0	0	0	42	794	12,905	0	0	0	0	0	0	13,741
6	0	0	0	0	38	577	8,506	0	0	0	0	0	0	0	9,121
7	0	0	0	29	338	4,564	0	0	0	0	0	0	0	0	4,931
8	1	11	110	1,071	10,624	0	0	0	0	0	0	0	0	0	11,817
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
expanded	1	11	110	1,100	11,000	5,183	9,347	13,919	17,885	21,256	23,110	23,110	26,831	0	152,863

Table B.2: Histogram Results for partial PDB with abstraction-7 on 12-pancake puzzle.

h \ g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	total
0	0	0	0	0	0	0	0	0	0	0	2	138	3,206	0	3,346
1	0	0	0	0	0	0	0	0	0	2	132	2,812	0	0	2,946
2	0	0	0	0	0	0	0	0	2	132	2,816	0	0	0	2,950
3	0	0	0	0	0	0	0	2	132	2,794	0	0	0	0	2,928
4	0	0	0	0	0	0	2	122	2,503	0	0	0	0	0	2,627
5	0	0	0	0	0	2	112	2,150	0	0	0	0	0	0	2,264
6	0	0	0	0	2	93	1,656	0	0	0	0	0	0	0	1,751
7	1	11	110	1,100	10,998	109,905	0	0	0	0	0	0	0	0	122,125
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
expanded	1	11	110	1,100	11,000	110,000	1,770	2,274	2,637	2,928	2,950	2,950	3,206	0	140,937

Table B.3: Histogram Results for partial PDB with abstraction-8 on 12-pancake puzzle.

B.2 Full PDBs

h \ g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	total
0	0	0	0	0	0	0	0	0	0	17	544	12,082	199,395	0	212,038
1	0	0	0	0	0	0	0	0	17	459	9,532	144,425	0	0	154,433
2	0	0	0	0	0	0	0	17	459	9,600	145,751	0	0	0	155,827
3	0	0	0	0	0	0	17	441	8,770	129,523	0	0	0	0	138,751
4	0	0	0	0	0	16	366	6,829	97,867	0	0	0	0	0	105,078
5	0	0	0	0	13	272	4,755	64,902	0	0	0	0	0	0	69,942
6	0	0	0	8	183	2,729	34,780	0	0	0	0	0	0	0	37,700
7	0	0	6	101	1,229	13,991	0	0	0	0	0	0	0	0	15,327
8	0	2	34	315	3,508	0	0	0	0	0	0	0	0	0	3,859
9	1	6	44	505	0	0	0	0	0	0	0	0	0	0	556
10	0	3	26	0	0	0	0	0	0	0	0	0	0	0	29
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
expanded	1	11	110	929	4,933	17,008	39,918	72,189	107,113	139,599	155,827	156,507	199,395	0	893,540

Table B.4: Histogram Results for PDB with abstraction-6 on 12-pancake puzzle.

h \ g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	total
0	0	0	0	0	0	0	0	0	0	0	56	1,427	26,831	0	28,314
1	0	0	0	0	0	0	0	0	0	56	1,203	21,683	0	0	22,942
2	0	0	0	0	0	0	0	0	56	1,203	21,851	0	0	0	23,110
3	0	0	0	0	0	0	0	52	1,108	19,997	0	0	0	0	21,157
4	0	0	0	0	0	0	47	962	16,721	0	0	0	0	0	17,730
5	0	0	0	0	0	42	794	12,905	0	0	0	0	0	0	13,741
6	0	0	0	0	38	577	8,506	0	0	0	0	0	0	0	9,121
7	0	0	0	29	338	4,564	0	0	0	0	0	0	0	0	4,931
8	0	0	21	150	1,809	0	0	0	0	0	0	0	0	0	1,980
9	0	7	39	440	0	0	0	0	0	0	0	0	0	0	486
10	1	4	49	0	0	0	0	0	0	0	0	0	0	0	54
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
expanded	1	11	109	619	2,185	5,183	9,347	13,919	17,885	21,256	23,110	23,110	26,831	0	143,566

Table B.5: Histogram Results for PDB with abstraction-7 on 12-pancake puzzle.

h \ g	0	1	2	3	4	5	6	7	8	9	10	11	12	13	total
0	0	0	0	0	0	0	0	0	0	0	2	138	3,206	0	3,346
1	0	0	0	0	0	0	0	0	0	2	132	2,812	0	0	2,946
2	0	0	0	0	0	0	0	0	2	132	2,816	0	0	0	2,950
3	0	0	0	0	0	0	0	2	132	2,794	0	0	0	0	2,928
4	0	0	0	0	0	0	2	122	2,503	0	0	0	0	0	2,627
5	0	0	0	0	0	2	112	2,150	0	0	0	0	0	0	2,264
6	0	0	0	0	2	93	1,656	0	0	0	0	0	0	0	1,751
7	0	0	0	2	68	1,135	0	0	0	0	0	0	0	0	1,205
8	0	0	2	45	632	0	0	0	0	0	0	0	0	0	679
9	0	2	22	255	0	0	0	0	0	0	0	0	0	0	279
10	1	7	57	0	0	0	0	0	0	0	0	0	0	0	65
11	0	2	0	0	0	0	0	0	0	0	0	0	0	0	2
12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
expanded	1	11	81	302	702	1,230	1,770	2,274	2,637	2,928	2,950	2,950	3,206	0	21,042

Table B.6: Histogram Results for PDB with abstraction-8 on 12-pancake puzzle.