Abstraction-Based Heuristics with True Distance Computations

Ariel Felner
Information Systems Engineering
Ben-Gurion University
Be'er-Sheva, Israel 85104
felner@bgu.ac.il

Nathan Sturtevant and **Jonathan Schaeffer**
Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{nathanst, jonathan}@cs.ualberta.ca

Abstract

Pattern Databases (PDBs) are the most common form of memory-based heuristics, and they have been widely used in a variety of permutation puzzles and other domains. We explore the *true-distance heuristics* (TDHs) (also appeared in [Sturtevant *et al.*, 2009]) which are a different form of memory-based heuristics, designed to work in problem states where there isn't a fixed goal state. Unlike PDBs, which build a heuristic based on distances in an abstract state space, TDHs store distances which are computed in the actual state space. We look in detail at how TDHs work, providing both theoretical and experimental motivation for their use.

1 Introduction

In the area of heuristic search, a major research direction has been finding optimal or suboptimal solutions in state spaces where the number of states grows exponentially with the solution depth. Examples for such *exponential domains* are the different permutation puzzles (e.g., the tile puzzles, Topspin, Pancake puzzle and Rubik's cube) as well as other forms of combinatorial problems (e.g. scheduling, SAT, CSP etc.).

There are, however, many problems where the number of states grow quadratically or polynomially with the solution depth. Examples for *quadratic-* or *polynomial domains* are two- and three-dimensional pathfinding problems or the sequence alignment problem, where a number of DNA sequences must be aligned with minimum cost.

Pattern databases (PDBs), a common method for building memory based heuristics, have shown great success in building heuristics for exponential domains but their general effectiveness in polynomial domains is limited. In addition, PDBs are goal specific and may not work if a path between *any* two states is needed.

In this paper we describe a class of memory-based heuristics called *true distance heuristics* (TDHs). They are useful in any undirected graph, even for applications where traditional PDBs are not efficient such as polynomial domains. Unlike traditional PDBs, which store distances in an abstract state space, TDHs can be seen as using abstract states but storing information about true distances in the original state space.

A perfect heuristic for any pair of start and goal states could be achieved by computing and storing *all-pairs-shortest-path* distances (e.g., using the Floyd-Warshall algorithm). However, due to time and memory limitations this is not practical. TDHs compute and store only a small part of this information based on an abstraction of the state space.

We first provide a new analysis of TDHs which better relates the TDH idea to previous abstraction-based work on heuristic search such as hierarchical A* and PDBs. Based on this we develop a theoretical explanation of why PDBs work well in exponential domains but are not as effective in polynomial domains. We then present the main forms of TDHs and relate them to previous work on abstraction, showing why TDHs are better-suited to polynomial domains. For example, TDHs work well for domains such as map-based searches (common in GPS navigation, computer games, and robotics), where paths often must be found very quickly due to their realtime nature. Finally, we provide experimental results that show the benefits of TDHs on a number of domains.

TDHs were also introduced in [Sturtevant *et al.*, 2009] and the two papers have some overlap for completeness. The portions that overlap are clearly denoted. However, the focus of the two papers are different as this paper aims to understand these heuristics and their relation to abstractions and PDBs on different domain settings. In addition, this paper introduces the border heuristic variant of TDHs and provides new results on the 8-puzzle and on the 4-peg Towers of Hanoi puzzle.

Forms of TDHs have already appeared before. For example [Björnsson and Halldórsson, 2006], used the exact distances between some of the states in the domain. Also, [Goldberg and Harrelson, 2005] independently introduced DH heuristics (see below). Our paper is the first to deeply explore these heuristics and provide theoretical analysis and thorough experimental results.

2 Background: Heuristics from Abstractions

Heuristics are naturally generated by abstracting a problem and then using the distances from the abstract space in the original space. Early analysis on how abstractions could be used for search [Valtorta, 1984] showed that in some classes of problems it isn't possible to compute an effective abstraction-based heuristic online during search. An example would be edge supergraphs [Gasching, 1979; Pearl, 1984] which arise when constraints are removed from a problem definition, resulting in more edges being added to the graph. In such cases the size of the abstract search space

is no smaller than the size original search. Thus in general, the cost of building the heuristic (based on the abstract graph) will equal that of doing a brute-force search.

2.1 Homomorphic Abstractions

Homomorphic abstractions use abstract state spaces that are much smaller than the original state space. The main idea is to merge groups of nodes from the original graph G into one abstract node in an abstract graph G'. There is an *edge* between two different abstract nodes n_1 and n_2 in G' if there was an edge between two nodes in G that are abstracted to n_1 and to n_2 respectively. Homomorphic abstractions preserve locality, so that nodes that are close to each other in G are also close to each other in G'.

A pioneering work using homomorphic abstractions is Hierarchical A* (HA*) [Holte *et al.*, 1996b]. HA* begins with a pre-computed hierarchy of abstract spaces. HA* performs an A* search in the original search space. Whenever a heuristic value is needed, it is computed recursively using A* search in one or more abstracted state spaces. At the highest abstract level, a breadth-first search is performed between the start and the goal. The distances in these abstract graphs are then used as heuristic estimates for distances in a less abstract graphs or the original search space. Hierarchical A* showed some improvement over simple breadth-first search, however the performances gains were not significant.

Similar techniques were applied in cooperative pathfinding. [Silver, 2005] abstracted away one dimension of a three-dimensional cooperative search to build a better heuristic and greatly improved performance. [Sturtevant and Buro, 2006] applied homomorphic abstractions to further improve performance on this domain. However, they used inadmissible heuristics and were not looking for optimal solutions.

2.2 Explicit Homomorphic Abstractions.

There are two ways to build homomorphic abstractions - explicit and implicit. An *explicit abstraction* is built by completely traversing the original graph and explicitly deciding which groups of states to merge. Explicit abstractions are generally only used in domains that fit in memory. A large variety of these abstractions have been analyzed [Sturtevant and Jansen, 2007]. An example of a homomorphic abstraction is the *star* abstraction [Holte *et al.*, 1996a], also called a radius abstraction, where all nodes in a fixed radius are abstracted together. An example of *explicit* homomorphic abstraction is shown in Figure 1. Part (a) shows a map from a commercial video game. Part (b) shows a portion of the graph induced by the map with 16,544 nodes. Part (c) shows the results of applying a homomorphic abstraction to this portion after which there are only 5,121 nodes in the map.

2.3 Pattern Databases (PDBs)

A second way to build homomorphic abstractions is to use general *implicit* rules for deciding which nodes to merge. Pattern databases [Cullberson and Schaeffer, 1998] are a special case of such implicit homomorphic abstractions. States in a search space are often represented using a set of *state variables*. An implicit abstraction of the search space, called the *pattern space*, can be defined by only considering a subset of the state variables (called the *pattern variables*). A *pattern* is a state of the pattern space which has an assignment of values to the pattern variables while ignoring the values of the other variables. A *pattern database* (PDB) stores the distance of each pattern to the goal pattern which is a lower bound on the corresponding distances in the original space. Thus PDBs serve as admissible heuristics for searching in the original search space.

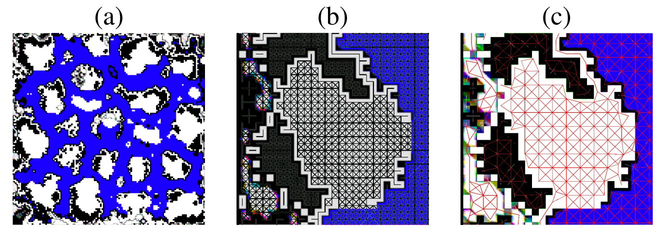

Figure 1: Sample map abstractions.

Despite their large success, PDBs have some limitations. First, PDBs are goal-specific; they only provide heuristics for a single goal state. While some domains have properties (e.g., duality [Felner *et al.*, 2005]) that allow a given PDB to be used for many goal states, this is not a general property.

Second, PDBs store abstract distances between states. This guarantees that the distances are lower bounds on distances in the original domain, but if good abstractions are not available, then the estimates will be poor. PDBs work very well for domains where a state can be described by assigning values to set of variables (e.g., locations to tiles, as in the sliding-tile puzzle). Replacing some of the assignments with a *don't care* value can yield an effective abstraction. But, in map-based pathfinding problems a state is just an x/y coordinate. Replacing the x or y coordinate by a *don't care* yields an abstraction that is too general to be effective. In addition, as will be mathematically suggested in the next section, PDBs are effective in exponential search spaces such as combinatorial puzzles but their applicability in quadratic search spaces such as maps is questionable. True distance heuristics (TDHs) provide alternative ways to build abstraction based heuristics.

3 Effectiveness of PDBs

Before describing TDHs in detail we first provide some insights into the effectiveness of PDBs.

Let us first analyze the effectiveness of a traditional PDB on exponential domains such as Rubik's cube or the sliding tile puzzle. This analysis is similar in nature to previous analysis [Korf, 1997]. First, assume that the maximum solution depth in the original state space is d and that the asymptotic branching factor is b. Then, assume that we have memory to build a PDB which is some fraction $1/f$ of the full problem size ($N = b^d$). If we build a PDB in which the abstract state space has the same branching factor as the original state space, then we can approximately compute the radius of the abstract space, w, as $b^w = 1/f \cdot b^d$. Thus, $w \cdot log(b) = d \cdot log(b) - log(f)$ and $w = d - log_b(f)$.

The maximum heuristic value that we expect in a PDB is w. This is smaller than the radius of the original problem. But, with a reduction of a factor of (f) in memory over the full problem size the distance estimates of the resulting heuris-

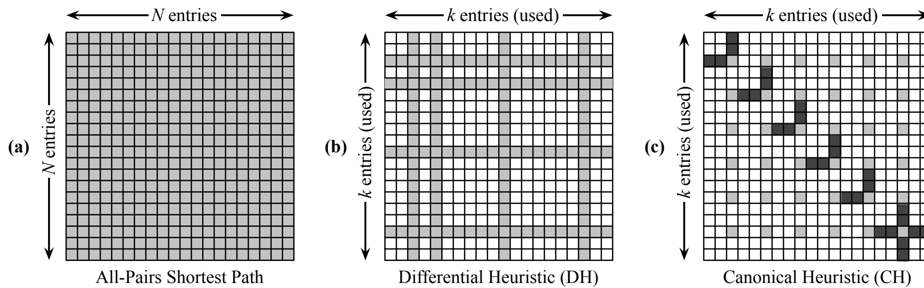

Figure 2: Types of heuristics.

tics will have an error difference of at most $[log_b(f)]$ in the distance estimates. This implies that PDB's are particularly effective on exponential domains.

Next, consider polynomial domains. Special cases are *quadratic* domains (such as maps) where the number of nodes up to depth d is d^2. Assume that we have a homomorphic abstraction that is a fraction $1/f$ of the full state space size. If we build a PDB for this abstract graph (that is, store the distances from all abstract states to the abstract goal), the abstract graph will have maximum depth of $w^2 = 1/f \cdot d^2$. Thus, $w = d/\sqrt{f}$. In this case, any heuristic which is built from an abstraction will not differ from the exact value by a constant amount, as in exponential domains, but will be some fraction of the true distance. In such domains the resulting heuristic is not likely to be strong.

Note that as the size of the search space grows to d^3 or higher dimensions, a heuristic built in this way will be more and more accurate. For a general k-dimensional problem, $w = d/(f^{1/k})$. Note also that in n-dimensional problems, a $(n-1)$-dimension version of the problem might provide an accurate heuristic for the n-dimensional problem, but whether this works well is domain dependent.

We give simple experimental evidence here to validate these claims. The 15 puzzle has 10^{13} different states and a maximum depth of 80. One possible (non-additive) 6-tile PDB needs only 6MB of storage but has a maximum heuristic value of 55. This is a 10^6 fold reduction in memory of the abstract state space over the size of the full state space while the heuristic error is approximately 1/3 of the true distance.

Now consider the map shown in Figure 1. The original map has 11,614 reachable nodes in the largest connected component. After applying one level of abstraction, there are 3,455 nodes, a 3.36 fold reduction. But, the maximum radius of the original problem is estimated to be 157 moves, while the maximum radius in the abstract state space is just 80 moves, a reduction of a factor of 2. The results explains why the original results of [Holte *et al.*, 1996b] were not impressive – the abstractions used were unable to provide very accurate heuristics on all domains.

4 True Distance Heuristics (TDHs)

This section presents the different versions of TDHs. The first two versions were also introduced in [Sturtevant *et al.*, 2009]. The *border heuristics* variant is new to this work, although similar heuristics have been proposed [Björnsson and Halldórsson, 2006].

Let N denote the number of vertices in a graph. If the full *all-pairs shortest-path* database is available, then the exact distance between two states, $d(x, y)$, can be retrieved and used as a perfect heuristic between x and y. This situation is illustrated in Figure 2a. Each row and column corresponds to a state in the world, and an entry in the grid is marked if the corresponding distance is stored. Computing such a database will require as much as $O(N^3)$ time which might not be feasible (even in an offline phase). Assuming that the size of the state space is $O(N)$, storing this database will require $O(N^2)$ memory—much more than is likely available.

We propose several abstraction methods that reduce the memory needs by using a subset of the *all-pairs-shortest-path* information to compute a heuristic distance between states.

4.1 Differential Heuristics (DHs)

The first method is shown Figure 2b. In this case, lengths of shortest paths are only stored for k of the N states ($k \ll N$). We denote these k states as *canonical states*. Because the database is symmetric around the main diagonal (the graph is undirected), this is equivalent to retaining only k rows (or columns) out of the full all-pairs database. If s is one of the canonical states then $d(x, s)$ is available for any state x. A *differential heuristic* (DH) between arbitrary states a and b is:

$$h(a, b) = |d(a, s) - d(b, s)|$$

If we use $k > 1$ canonical states, we can take the maximum from each of the independent heuristics. A DH can be built using k complete single-source searches. The time will be $O(kN)$ and the memory used is also $O(kN)$. Placement strategies are discussed in [Sturtevant *et al.*, 2009]. The best approach is to place them as far from each other as possible.

4.2 Canonical Heuristics (CHs)

Our second method uses canonical states in a different way, illustrated in Figure 2c. Again, we first select k canonical states. Here, the shortest path between *all* pairs of these k states is stored in the database (primary data). Additionally, for each of the N states in the world we store which canonical state is closest as well as the distance to this canonical state (secondary data). The shortest-path data (primary data) is marked in a light-gray in Figure 2 while the secondary data is slightly darker. Note that in a domain with regular structure (such as the sliding-tile puzzle), it might be possible to avoid storing the secondary data, instead computing it on demand. We call this a *Canonical Heuristic* (CH). Define $C(x)$ as the closest canonical state to x. Then:

$$h(a, b) = d(C(a), C(b)) - d(a, C(a)) - d(b, C(b))$$

This can be less than 0, but in practice we always take the max of the CH and an existing heuristic (e.g. air distance or Manhattan distance). Let all states which share the same closest canonical state $C(x)$ be called a *canonical neighborhood*. If two states are in the same canonical neighborhood then the DH heuristic rule can be used for them instead.

Canonical states perform best if they are uniformly distributed [Sturtevant *et al.*, 2009]. Once k canonical states are chosen, we need to perform k complete single-source searches. The time complexity is again $O(kN)$. The memory needed is $O(k^2)$ for the primary data.

When necessary (e.g., in non-regular domains) the secondary data is calculated as follows. We perform a breadth-first search simultaneously from all the canonical states until the entire state space is spanned. When a state s is first generated by the breadth-first search from the canonical state P, P and $d(s, P)$ are stored in the secondary data. The time needed for this is $O(N)$ and additional $2N$ memory might be needed for the secondary data.

4.3 Border Heuristics

An alternate form of TDHs not described in [Sturtevant *et al.*, 2009] is motivated by considering how the abstract heuristics built by Hierarchical A* might be pre-computed. Suppose we build an abstract state space that is small enough such that we can store *all-pairs-shortest-path* data between each of the abstract nodes in the state space. But, instead of storing abstract distances, as Hierarchical A* would, we store actual distances in the state space. As we need the heuristic to be a lower bound on the cost between any two states, the stored cost between any two abstract states A' and B' is the minimum distance between all states a and b where a abstracts into A' and b abstracts into B'. This is equivalent to computing the minimum distance between the borders of two canonical neighborhoods. This is the main idea behind the new TDH version which we call *border heuristic* (BH).

Define a *border state* as a state which has at least one neighbor in another neighborhood. The border heuristic (primary data) between any two states a and b, would be $h(a, b) = d(C(a), C(b))$ where $d(C(a), C(b))$ is the minimal distance between border states of the two neighborhoods.

This estimate can be improved if we know the distance from any state to the border of its canonical neighborhood. In that case we can add that distance to the heuristic estimate (secondary data). If $D_B(x)$ is the distance from state x to the border of its canonical neighborhood then the heuristic estimate can be improved to

$$h(a, b) = d(C(a), C(b)) + D_B(a) + D_B(b)$$

The primary data for BHs is calculated by performing a breadth-first search seeded with border states of a given canonical neighborhood until all other canonical neighborhoods have been reached. The final data will require $O(k^2)$ memory and will take $O(kN)$ time to compute. When necessary (e.g., in non-regular domains) the secondary data is calculated using the same multi-seed breadth-first search. When a state s (inside the neighborhood) is first generated at depth x we set $D_B(s) = x$. Again, additional $2N$ memory might be needed for the secondary data.

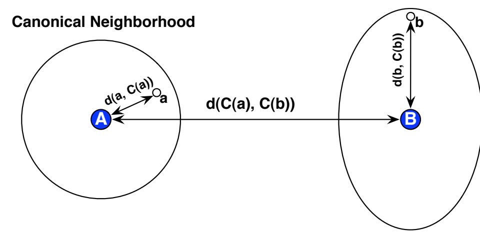

Figure 3: True distance heuristics.

Technique	Storage	Time to Build
All-Pairs Shortest Path	$O(N^2)$	$O(N^3)$
DH	$O(kN)$	$O(kN)$
CH(k, 1)	$O(k^2) + 2N$	$O(kN)$
BH(k)	$O(k^2) + 2N$	$O(kN)$
CH(k, d)	$O(k^2) + 2dN$	$O(kN)$

Table 1: Memory and time complexity.

4.4 Comparison between CH and BH

We compare the behavior of border and canonical heuristics with the help of Figure 3. There are two neighborhoods in this figure with canonical states A and B. There are two other states a (where $C(a) = A$) and b (where $(C(b) = B)$. For the sake if the illustration we assume that all true distances are identical to the straight lines. First, consider what will happen if both neighborhoods are symmetric circles with radius r (as is the case for A in the figure). In this case $d(a, C(a)) = r - D_B(a)$ (similarly for b). Assume that shortest distances between the borders is d_b. Thus $d(A, B) = d_b + 2r$. Now, $BH(a, b) = d_b + D_B(a) + D_B(b)$. The CH will be identical as $CH(a, b) = d(A, B) - d(a, A) - d(b, B) = d_b + 2r - (r - D_B(a)) - (r - D_B(b)) = d_b + D_B(a) + D_B(b)$

However, when the neighborhoods are not symmetric circles, as is the case with neighborhood B in the figure, there can either be a gain or loss. In the case of our figure, state b will have a better heuristic value (between a and b) with the border heuristic than with the canonical heuristic. However, if the neighborhood around B is rotated 90°, b will have a better heuristic with the canonical heuristic.

4.5 Unified View

Table 1 summarizes the time and memory requirements for DHs, CHs and BHs. Building any of the databases requires k single-source searches of the entire state space. However, for the same amount of memory (e.g. $10N$), DHs will have smaller k, so they can be built more quickly.

Differential and canonical heuristics can be viewed as opposite extremes of a general framework. Suppose the available memory is fixed at $10N$. Memory can be filled in one of two ways. First, 10 differential heuristics can be built, each of which takes N memory. Alternately, $k = \sqrt{8N}$ canonical states can be selected for a canonical heuristic which will use $k^2 = 8N$ memory. With the additional $2N$ memory for storing the secondary data, this will also require $10N$ memory. Similarly for border heuristics.

Consider that instead of keeping the distance to the closest canonical state (and its identity) in the secondary data, we keep the distance to the d closest canonical states among

Closest States Stored	Num Canonical States	Total Memory ($2dN + k^2$)
d < k		
d = 1	k = $\sqrt{8N}$	10N
d = 2	k = $\sqrt{6N}$	10N
d = 3	k = $\sqrt{4N}$	10N
d = 4	k = $\sqrt{2N}$	10N
d = k		
d = 5	k = 5	10N
d = 10 (optimized)	k = 10	10N

Table 2: Transition between DH and CH.

the k canonical states available ($d < k$). We denote this as $CH(d, k)$. The memory required is $2dN + k^2$. Our introductory discussion to CH implicitly used $d = 1$. When $d = k$ then every state maintains the exact distance to all k canonical states— which is actually a differential heuristic.

The possible heuristics using $10N$ memory are shown in Table 2. When $d < k$, both the optimal distance to the closest canonical states and the identity of these canonical states must be stored in the secondary data. When $d = k$ (logically a *differential heuristic*) the distance to all canonical states is stored. Thus, the identity of the canonical state is not needed, allowing twice as many canonical states to be used. Additionally, when $d = k$ the primary data of all-pairs-shortest-path distance between the k canonical states is redundant here, as it is already stored in the secondary data. This allows us to reuse the space by doubling d ('optimized' in Figure 2).

In this unified scheme, there are many possible heuristic lookups. For any two states a and b we need to choose two out of d different canonical states as reference points for the canonical heuristic; a total of d^2 possible lookups. As well, there could be as many as d valid differential lookups. Clearly there is a tradeoff; the maximum over multiple heuristic values yields a better heuristic but at the cost of increased execution time. In addition, larger d means fewer canonical states. The border heuristics can be similarly generalized using negative distances, but we just consider $d = 1$ here.

The advantage of using $d > 1$ is shown in Figure 4. The search is between the start (S) and the goal (G), both of which are canonical states. While the canonical heuristic will store the exact distance between these states (16), it will give no guidance to an A* search as to which nodes are on the optimal path to G. Consider states a and b. They are equally far from S (say, 5) so their heuristic value from G will be the same ($16 - 5 = 11$), they will have the same f-cost (16), and will both be expanded. But, we can use canonical state C to improve the heuristic estimate for b. In particular, $h(b, G) = d(C, G) - d(b, C) - d(G, G) = 32 - 11 = 21$. With a g-cost of 5, b will have an f-cost of 26 and will not be expanded ($26 > 16$). The second lookup can be seen as triangulating the position of state b to improve its heuristic value.

5 Potential of CH's in Polynomial Domains

We would like to estimate the error of a CH. Assume that there are k canonical states which are uniformly distributed. Thus, each has N/k different states in its neighborhood.

Exponential domains: Assume that the state space grows exponentially in depth with branching factor b. If r is the radius of a neighborhood we get $b^r = N/k$. Solving for r gives $r = \log_b N/k$. If $N = b^d$ where d is the radius of the search space we get $r = \log_b N/k = \log_b b^d - \log_b k = d - \log_b k$. We chose values such as $k = \sqrt{CN}$ where C is a small constant so that the CH would fit in memory. In this case $r = d - \log_b(\sqrt{CN}) = d - 0.5(\log_b Cb^d) = 0.5(d - \log_b C)$. As most states will be at the borders of the canonical neighborhoods, the heuristic estimate between two arbitrary states is likely to be no better than $d - 2r = \log_b C$. As C is a small constant, this suggests that CH's will provide no gain for most states in an exponential domain.

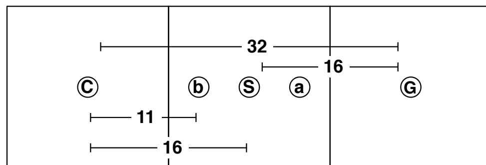

Figure 4: Two heuristic lookups are better than one.

Quadratic domains: Assume that the state space grows quadratically with the depth. If we assume that r is the radius of a neighborhood we get $r^2 = N/k$ and $r = \sqrt{N/k}$. But, now $N = d^2$, so $r = d/\sqrt{k}$. Again, let $k = \sqrt{CN}$, in which case $r = \sqrt[4]{N/C}$. The heuristic between two states given that the true distance is d_t can be as low as $d_t - 2r = d_t - 2\sqrt[4]{N/C}$. This implies that the heuristic will be accurate for states which are far apart, but less for states which are closer together. In general polynomial domains, we have $r^i = N/k$ and $N = d^i$. Thus $r = \sqrt[i]{N/k} = \sqrt[2i]{N/C}$.

The outcome from our analysis is that while PDBs seem to work best for exponential domains (as explained above), CHs are better in polynomial domains. This analysis is specifically for CHs, but a similar analysis exists for BHs. A detailed analysis of DHs is an area of future research.

6 Experimental Results

We now provide experimental results on a number of domains. In all experiments we use the max of an existing heuristic (e.g. Manhattan distance, air distance etc.) with the TDH. All times are reported in seconds. The results for pathfinding overlap with [Sturtevant *et al.*, 2009].

6.1 Pathfinding: Differential Heuristics

Pathfinding is an example of a domain where the entire state space is usually kept in memory. The real-time nature of this domain requires finding a path as quickly as possible.

Two types of maps were used which are illustrated in Figure 5: mazes (left) and rooms (right). In a simple maze there is only one path between any two points, however we use corridors width two, which increases the average branching factor from two to five. The octile-distance heuristic, which is similar to Manhattan distance except that it allows for diagonal moves, can be very inaccurate on mazes. Room maps are composed of small (16×16) rooms with randomly opened doors between rooms. Octile-distance is more accurate on these maps. All maps used here are publicly available.

To begin, we compare the search effort required with the full *all-pairs-shortest-path* data to the differential heuris-

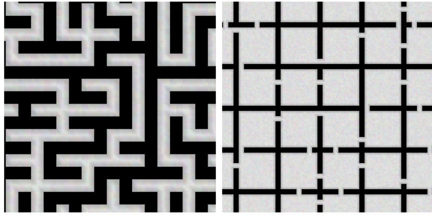

Figure 5: Example mazes/rooms (left/right).

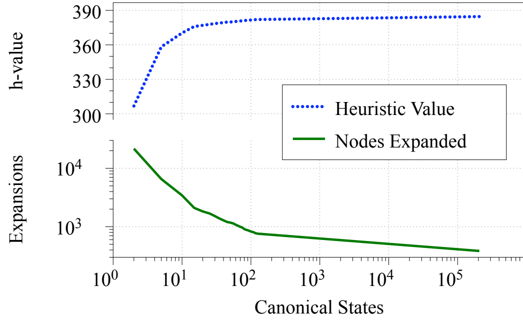

Figure 6: Comparison as memory usage grows.

tics. The results in Figure 6 use a 512×512 room map with 206,720 states. The number of canonical states varied from 0 to 125 by intervals of 5. Because we cannot plot '0' canonical states on a log-plot, the first point denotes the results with the default octile heuristic. The results are averaged over 640 problems with solution lengths between 256 and 512. The all-pairs data was estimated by assuming that with a perfect heuristic only the optimal path would be explored. We drew a line from the 125 data point to the 206,720 data point (all pairs) to approximate data in between.

The top of the graph shows how the average h-value grows as the number of canonical states is increased. Note that the x-axis is logarithmic. The optimal heuristic value is 384.59 and would require the full all-pairs data and 21 billion heuristic entries. With 125 canonical states (26 million entries) we get an average heuristic value of 382.04. With just 10 canonical states (2 million entries) the heuristic value is 370.34. In contrast, the average octile-distance heuristic is 306.83.

The number of node expansions are shown at the bottom of Figure 6. This is a log-log graph, so the slope of the line looks much shallower than it actually is. With octile-distance, 21,686 nodes are expanded on average. 10 canonical states reduces this to 3,440 nodes. 125 canonical states reduces this further to 760 nodes (29× reduction). The absolute minimum, assuming a perfect heuristic would be 384.6 nodes. A 29× reduction in nodes expanded can be achieved with only $1/1000$ of the total memory needed for the full all-pairs information (which will only achieve an additional reduction of a factor of two.)

6.2 Pathfinding: Canonical Heuristics

Next, we look at the transition between canonical heuristic parameters. We begin with the default heuristic, octile distance. Then, fixing the total memory at $10N$ we build a CH(d, k) (where $d = \{1 \dots 5\}$ and $k = \sqrt{(10N - 2Nd)}$) and a DH$(k = 10)$. The canonical and differential heuristics were built twice: once with random and once with advanced placement of canonical states.

h		Mazes			Rooms		
d	k	nodes	h-val	time	nodes	h-val	time
Octile		7792	151	0.068	21354	309	0.296
1	1448	2377	611	0.026	8698	372	0.123
2	1254	1845	626	0.022	6011	375	0.091
3	1042	1729	627	0.021	5472	376	0.083
4	724	1776	619	0.023	5646	373	0.092
5	5	1793	610	0.026	14473	337	0.246
10	10	707	636	0.010	3479	370	0.054

Table 3: Results on maze and room maps. Memory = 10N.

We present the average number of nodes expanded by A*, starting h-cost, and average time for the search. The results are averaged over 640 problem instances on each of 5 maze and room maps (3,200 total instances for each map type). All maps are 512 × 512. Paths were evenly distributed between lengths 256 and 512 on the room maps and between lengths 512 and 768 on the maze maps. Paths are longer on maze maps, but fewer nodes are expanded because the search is more restricted in the maze corridors.

The results for mazes and rooms are in Table 3. For mazes, the average heuristic between start and goal points is 151 with octile distance, while the optimized DH (last line) has an average heuristic value of 636. The DH expands over 11× fewer nodes than the octile heuristic but is only 6.8× faster due to the overhead of the heuristic lookups.

For room maps the octile heuristic is more accurate in these maps with an average value between start and goal pairs of 309, compared to 370 with the best canonical heuristic. There is a saddle point in the canonical results, where the best results are with $d = 3$ (for mazes too). The best time performance is with the optimized DH, 5.5× faster than the octile heuristic with 6× fewer nodes.

The average DH value between the start to the goal is 370, lower than the CH$(d = 3)$ (376), but fewer nodes are expanded with the DH. To investigate this, we recorded the h-value of every node expanded over 640 problems on a single room map. A histogram of values is in Figure 7. Searching with either heuristic expands the same number of nodes with high heuristic values. But, the DH results in far fewer nodes expansions with low heuristic values. CHs are inaccurate near the borders of the canonical neighborhoods which suggests there are enough nodes along these borders to significantly increase the cost of search.

6.3 Experiments on the Sliding-Tile Puzzle

To test TDHs on exponential domains we implemented them on the 8 puzzle. This puzzle has $N = 181,440$ different reachable states. 250 states were randomly generated, and we ran an IDA* search between every pair of these states, a total of 31,125 different searches.

We performed two sets of experiments. In the first set we bounded the size of the memory of the databases and randomly chose a given number of canonical states for the differential and canonical heuristics. Table 4 shows the results where the rows are ordered according to the number of nodes generated in decreasing order. The first line used Manhattan

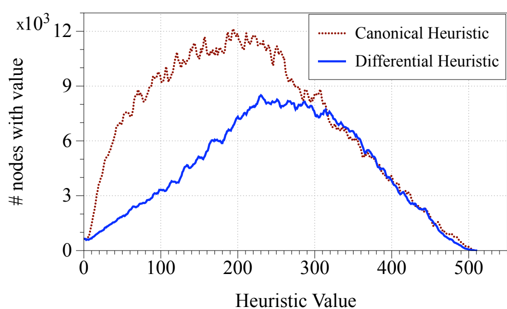

Figure 7: Nodes expanded with CH and DH.

h	d	k	h start	nodes	Mem
Manhattan	-	-	14.21	3000.26	-
DH	1	1	14.26	2941.35	1N
DH	5	5	14.34	2784.32	5N
DH	10	10	14.41	2570.33	10N
CH	1	1204	14.50	2376.23	10N
CH	3	852	14.44	2422.28	10N
DH	50	50	14.95	2184.43	50N
DH	200	200	16.11	832.71	200N
blank in the center					
CH	1	20,160	18.55	345.95	1120N
BH	-	20,160	18.92	211.31	1120N
CH	3	20,160	19.83	69.42	1120N

Table 4: Results on the 8 puzzle.

Distance (MD) as a benchmark comparison. The next lines show results with increasing size of memory and with different settings for d and k. The gains provided by DHs and CHs up to 50N are modest. Only with 200N are the gains more significant. Canonical and differential heuristics will likely work best in domains where paths cover long distances. This is possible in quadratic domains such as the pathfinding domains above. This puzzle is an exponential domain and these methods achieve only modest gains.

To check the limit of of this method, in the next set of experiments we used as much memory as possible and took advantage of the internal structure of this puzzle. Define a *corner* state as one where the blank is in the corner. *Edge* and *center* states are similarly defined. We divided the domain to neighborhoods as follows. First, all the 20,160 center states were chosen as canonical states, each had the nearest 4 edge states and 4 corner states in its neighborhood as shown in Figure 8. In order to relate a corner state to a single canonical state we ordered the operators such that *left-right* moves are performed before *up-down* moves. Bold arrows show moves within the same neighborhood while dashed arrows are moves to another neighborhood. The only way to exit the neighborhood is to have the blank in the rightmost or leftmost columns and to move it either up or down. It is easy to see that corner states are two moves away from the canonical state while edge states are one step away. In addition, all states where the blank is either in the left or the right column are border states and all other states have a border distance of 1. Thus, secondary data need not be stored and can be easily determined on the fly.

We then built a CH and BH databases with a table of size

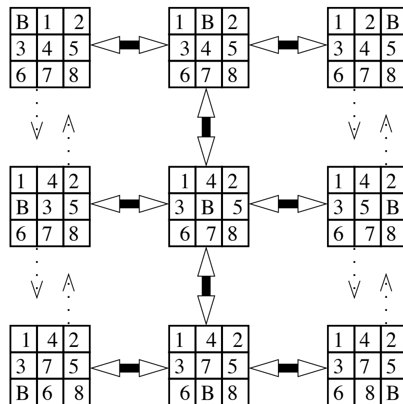

Figure 8: Neighborhoods for the 8 puzzle

$20,160 \times 20,160 = 406,425,600$. At one byte per entry the total memory used was roughly 200 megabytes, very reasonable on current machines. For both cases we ran 20,210 breadth-first searches, one for each neighborhood. For CH we seeded the queue with the canonical states while for the BH we seeded the queue with all six border states. Each time a canonical state (for the CH) or a border state (for BH) was first seen we updated the corresponding entry in the database. The process of generating these databases generated roughly 3 Billion states and a database was built in about half an hour.

Both CH and BH were rather effective and reduced the number of generated nodes by a factor of 10 compared to MD. The best performance was achieved with CH when $d = 3$. This version generated only 69 nodes which is a $43\times$ reduction over MD. It is important to note that PDBs are not directly comparable because they are built for a given goal state while our searches are between two arbitrary states.

6.4 Border Heuristics for Towers of Hanoi

The four-peg Towers of Hanoi problem (TOH4) is a common testbed for search algorithms. The task is to move n discs (ordered by their size) from their initial stack on an *initial* peg to another *goal* peg. Each action moves one disc from the top of one peg to the top of another peg with the constraint that a larger disc cannot be placed on top of a smaller disc. Although theoretical results suggest that CHs won't work well in domains that grow exponentially, TOH is unique in that the solutions also grow exponentially, which is differs from our PDB analysis. TOH is also unique in that the normal domain abstractions for TOH only abstract nearby states, so in this domain explicit homomorphic abstractions would closely resemble the implicit abstractions of PDBs.

PDBs [Felner *et al.*, 2004] and compressed PDB [Felner *et al.*, 2007] have been effectively applied to this problem. In a compressed PDB several PDB entries are merged into one entry. In order to guarantee admissibly, only the minimal value among the entries of the original PDB is stored. If merged entries are highly correlated then the loss of information is small and in many cases only a modest increase in the search effort is caused by the compression despite the fact that a large reduction in memory is achieved.

A PDB for N discs contains 4^N entries. Consider a specific configuration c of the largest $N-K$ discs. There are now

		simple compression		simple + border	
δ	Mem	Avg h	Nodes	Avg h	Nodes
0	256M	87.04	36,479,151	87.04	36,479,151
1	64M	86.48	37,964,227	86.48	37,963,596
2	16M	85.67	40,055,436	85.67	38,160,236
3	4M	84.45	44,996,743	84.82	41,854,341
4	1M	82.74	45,808,328	83.49	43,918,650
5	256K	80.85	61,132,726	82.09	51,420,682
6	64K	78.54	76,121,867	80.46	57,708,367
7	16K	74.81	97,260,058	77.63	70,090,868
8	4K	68.34	164,292,964	72.28	102,829,813
9	1K	62.71	315,930,865	68.01	174,873,646

Table 5: Solving 16 discs. The 14-disc PDB was compressed

4^K combinations of the K smallest discs that can be placed on top of c. All these have a different entries in the original PDB. In the compressed PDB the minimum among all these configurations is stored in one entry and the size of the compressed PDB is 4^{N-K} (See [Felner *et al.*, 2007]). Note that each entry of the compressed PDB has a value that was taken from a specific configuration of the K smallest discs in original PDB (the one with the minimal value).

It turns out that this PDB is identical to a BH. Define a neighborhood to contain all the states (in the pattern space of N discs) where the $N - K$ large discs are in the same location. In the compressed PDB, the minimal distance from all the states of the neighborhood was stored. This distance belongs to one of the states in the neighborhood. This state (call it X) must be on the border. That is, you can move from another neighborhood (where the largest discs are in different location) to this border state.

So, assume a state Y in the neighborhood of X. In the compressed PDB we take the distance from X to the goal as the heuristic for Y. But, we from Y we first need to go to some border. Thus, based on the analysis behind BHs we can also add the distance from Y to a border state. A border state in this problem is a state where at least two pegs do not have any of the smallest k disc on them and that one of the large discs can move. We can build a small lookup table for each configuration of K discs (4^K) by running 4^K different breadth-first searches until a state was reached where any two pegs are empty and thus a large disc can move.

Experimental results for this new enhancement compared to the published results from [Felner *et al.*, 2007] are shown in Table 5. The problem had 16 discs and the original uncompressed PDB contained 14 discs. Each row corresponds to the number of discs that were compressed (the δ column). The next three columns give the memory needed for the compressed PDB, the average heuristic over a large sample of random states and the number of nodes generated for solving initial. Similar data is then presented when we added the border distances too (with the help of the border distance table). The results clearly show that adding this extra knowledge reduces the search effort by up to a factor of 2.

7 Conclusions

In the past decade, PDBs have received considerable attention in the heuristic search literature. This paper explains the limits of PDBs and introduces new memory-based heuristics that use memory in a novel way to solve more general problems with arbitrary start and goal states.

We provide analysis that shows that PDBs work best for exponential domains while CHs are better for polynomial domains. Experimental results showed a important performance gains of CH for real-time pathfinding as a quadratic domains but much more modest gains for the 8 puzzle which is an exponential domains. Considerable research remains. The best number and location of canonical states is an open problem. More insights are also needed into the nature of these heuristics to give guidance to an application developer as to which heuristic (and its parameters) to choose for a given problem.

References

[Björnsson and Halldórsson, 2006] Y. Björnsson and K. Halldórsson. Improved heuristics for optimal path-finding on game maps. In *AIIDE*, pages 9–14, 2006.

[Cullberson and Schaeffer, 1998] J. Cullberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[Felner *et al.*, 2004] A. Felner, R. E. Korf, and Sarit Hanan. Addtive pattern database heuristics. *Journal of Artificial Intelligence Research (JAIR)*, 22:279–318, 2004.

[Felner *et al.*, 2005] A. Felner, U. Zahavi, R. Holte, and J. Schaeffer. Dual lookups in pattern databases. In *IJCAI*, pages 103–108, 2005.

[Felner *et al.*, 2007] A. Felner, R. E. Korf, R. Meshulam, and R. C. Holte. Compressed pattern databases. *JAIR*, 30:213–247, 2007.

[Gasching, 1979] J. Gasching. A problem similarity approach to devising heuristics: First results. *IJCAI*, pages 301–307, 1979.

[Goldberg and Harrelson, 2005] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. In *SODA*, pages 156–165, 2005.

[Holte *et al.*, 1996a] R. C. Holte, T. Mkadmi, R. M. Zimmer, and A. J. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artif. Intell.*, 85(1-2):321–361, 1996.

[Holte *et al.*, 1996b] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. *AAAI*, pages 530–535, 1996.

[Korf, 1997] R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *AAAI*, pages 700–705, 1997.

[Pearl, 1984] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison & Wesley, 1984.

[Silver, 2005] D. Silver. Cooperative pathfinding. In *AIIDE*, pages 117–122, 2005.

[Sturtevant and Buro, 2006] N. R. Sturtevant and M. Buro. Improving collaborative pathfinding using map abstraction. In *AIIDE*, pages 80–85, 2006.

[Sturtevant and Jansen, 2007] N. R. Sturtevant and Renee Jansen. An analysis of map-based abstraction and refinement. In *SARA*, pages 344–358, 2007.

[Sturtevant *et al.*, 2009] N. Sturtevant, A. Felner, M. Barer, J. Schaeffer, and N. Burch. Memory-based heuristics for explicit state spaces. *Accepted for publication in IJCAI-09, To appear.*, 2009.

[Valtorta, 1984] M. Valtorta. A result on the computational complexity of heuristic estimates for the A* algorithm. *Information Sciences*, pages 47–59, 1984.