Dual Lookups in Pattern Databases

Ariel Felner
Department of Information
Systems Engineering
Ben-Gurion University
Beer-Sheva, Israel 85104
Email: felner@bgu.ac.il

Uzi Zahavi
Computer Science Dept.
Bar-Ilan University
Ramat-Gan, Israel 92500
Email:zahavi@cs.biu.ac.il

Jonathan Schaeffer and **Robert C. Holte**
Computing Science Department
University of Alberta
Edmonton, T6G 2E8, Canada
Email:jonathan,@cs.ualberta.ca
Email:holte,@cs.ualberta.ca

Abstract

A pattern database (PDB) is a heuristic function stored as a lookup table. Symmetries of a state space are often used to enable multiple values to be looked up in a PDB for a given state. This paper introduces an additional PDB lookup, called the dual PDB lookup. A dual PDB lookup is always admissible but can return inconsistent values. The paper also presents an extension of the well-known pathmax method so that inconsistencies in heuristic values are propagated in both directions (child-to-parent, and parent-to-child) in the search tree. Experiments show that the addition of dual lookups and bidirectional pathmax propagation can reduce the number of nodes generated by IDA* by over one order of magnitude in the TopSpin puzzle and Rubik's Cube, and by about a factor of two for the sliding tile puzzles.

1 Introduction

Heuristic search algorithms such as A* and IDA* are guided by the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the actual distance from the initial state to state n and $h(n)$ is a heuristic function estimating the cost from n to a goal state. If $h(s)$ is "admissible" (i.e. is always a lower bound) these algorithms are guaranteed to find optimal paths.

Pattern databases are heuristics in the form of lookup tables. They have proven very useful for defining heuristics for combinatorial puzzles and other problems [Culberson and Schaeffer, 1994; Edelkamp, 2001; Korf and Felner, 2002].

The *domain* of a search space is the set of constants used in representing states. A *subproblem* is an abstraction of the original problem defined by replacing some of these constants by a "don't care" symbol. A *pattern* is a state of the subproblem. The *pattern space* for a given subproblem is a state space containing all the different patterns connected to one another using the same operators that connect states in the original problem. A *pattern database* (PDB) stores the distance of each pattern to the goal pattern. Typically, a PDB is built by searching backwards, breadth-first, from the goal pattern until the whole pattern space is spanned. Given a state S in the original space, a heuristic value for S, $h(S)$, is computed using a PDB in two steps. First, S is mapped to a pattern. Then, this pattern is looked up in the PDB and the corresponding distance is returned as the value for $h(S)$.

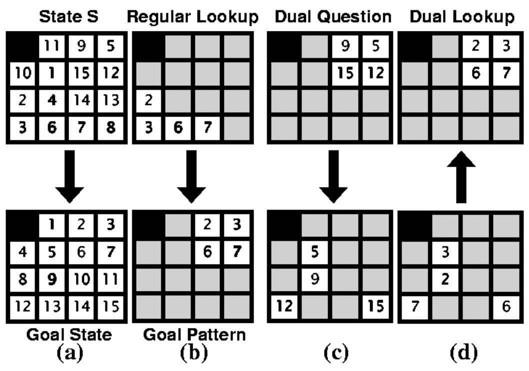

Figure 1: Example of regular and dual lookups

This paper is concerned with the first step, the mapping of a state S to a pattern that can be looked up in a given PDB. The standard mapping, called the regular PDB lookup in this paper, is illustrated in Figure 1(a) and (b) for the 15-puzzle. Patterns are created by ignoring all the tiles except for 2, 3, 6 and 7. Each pattern contains tiles 2, 3, 6 and 7 in a unique combination of positions. The resulting $\{$*2-3-6-7*$\}$*-PDB* contains a unique entry for each pattern with the distance from that pattern to the goal pattern (shown in the lower part of Figure 1(b)). Figure 1(b) depicts the regular lookup in this PDB for estimating a distance from a given state S to the goal (Figure 1(a)). State S is mapped to a 2-3-6-7 pattern by ignoring all the tiles other than 2, 3, 6 and 7. Then this pattern's distance to the goal pattern is looked up in the PDB. To be specific, if the PDB is represented as a 4-dimensional array, PDB, with the array indexes being the locations of tiles 2, 3, 6, and 7 respectively, the regular lookup for state S is $PDB[8][12][13][14]$, because tile 2 is in location 8, tile 3 is in location 12, etc. The value retrieved by a regular PDB lookup for state S is a lower bound (and thus serves as an admissible heuristic) for the distance from S to the goal state in the original space.

It is common practice to exploit special properties of a state space to enable additional lookups to be done in a PDB. [Cullberson and Schaeffer, 1998] describe several alternative lookups that can be made in the same PDB based on the physical symmetries of the 15-puzzle. For example, because of the symmetry about the main diagonal, the

PDB built for the goal pattern in Figure 1(b) can also be used to estimate the number of moves required to get tiles 8, 12, 9 and 13 from their current positions in state S to their goal locations. We simply reflect the tiles and their positions about the main diagonal and use the $\{$*2-3-6-7*$\}$-*PDB* for retrieving the symmetric values. The idea of reflecting the domain about the main diagonal for having another set of PDBs was also used by [Korf and Felner, 2002; Felner *et al.*, 2004] when solving the 15 and 24 tile puzzles with additive PDBs (see Figure 4 below).

Because all valid, alternative PDB lookups provide lower bounds on the distance from state S to goal, their maximum can be taken as the value for $h(S)$. Of course, there is a tradeoff for doing this—each PDB lookup increases the time it takes to compute $h(S)$. Because additional lookups provide diminishing returns in terms of the reduction in the number of nodes generated, it is not always best to use all possible PDB lookups [Cullberson and Schaeffer, 1998]. A number of methods exist for reducing the time needed to compute $h(S)$ by making inferences about some of the values without actually looking them up in a PDB [Holte *et al.*, 2004].

The main contribution of this paper is a new, alternative PDB lookup that is based on properties other than physical symmetries. We call it the dual PDB lookup. The dual lookup is always admissible but, unlike previously considered PDB lookups, it can return values in the search that are inconsistent. Our second contribution is a simple but useful extension of the well-known pathmax method so that inconsistencies in heuristic values are propagated in both directions (child-to-parent, and parent-to-child) in the search tree. Our final contribution is that dual lookups, with bidirectional pathmax propagation, produce state-of-the-art performance for three standard test applications.

2 Dual Lookups in Pattern Databases

The states of many problems, such as the sliding tile puzzles, Rubik's cube, Towers of Hanoi, etc., are defined by assigning objects (e.g. tiles, cubies, disks) to locations.

To explain the dual PDB lookup, consider again the $\{$*2-3-6-7*$\}$-*PDB* for the 15-puzzle defined above. The regular PDB lookup asks the question of what is the cost of getting tiles 2, 3, 6 and 7 from their current locations to their goal locations? In general, the regular lookup focuses on a fixed set of objects (the ones that define the patterns), and bases its lookup on their current locations, which vary from state to state.

In the dual PDB lookup the roles of locations and objects are switched. The dual PDB lookup focuses on a fixed set of locations – the goal locations of the objects in the pattern – and bases its lookup on the objects that occupy those positions, which vary from state to state. In particular, the dual PDB lookup of Figure 1 asks this question: what is the cost of moving the tiles that are currently in the goal location of tiles 2, 3, 6 and 7 to their home locations? For the state S in Figure 1(a), for example, the dual PDB lookup asks, what is the cost of getting tiles 9, 5, 15 and 12 from their current locations (the goal locations of tiles 2, 3, 6 and 7) to their goal locations? See Figure 1(c).

The dual question cannot be answered directly because a PDB based on tiles 5, 9, 12 and 15 is not available. However, because costs in the 15-puzzle are symmetric and independent of the exact tiles involved, the dual question can be answered using the $\{$*2-3-6-7*$\}$-*PDB*. As shown in Figure 1(d), if we replaces the names of tiles 5, 9, 12 and 15 with the name of the tile whose goal location they each currently occupy, the 5-9-12-15 goal pattern at the bottom of Figure 1(c) turns into a 2-3-6-7 pattern that can be looked up in the 2-3-6-7 PDB. In particular, the dual lookup would retrieve the value $PDB[9][5][15][12]$, because in state S tile 9 is in tile 2's goal location, tile 5 is in tile 3's goal location, etc.

Another example for dual looukups is the (N,K)-TopSpin puzzle which has N tokens arranged in a ring. Any set of K consecutive tokens can be reversed (rotated 180 degrees in the physical puzzle). Our encoding of this puzzle has N operators for each possible reversal/rotation. Figure 2(a) shows the goal state of the (9,4)-TopSpin puzzle. Figure 2(b) shows the result of reversing the tokens at locations 6-9, and Figure 2(c) shows the result of reversing the tokens in locations 4-7 of Figure 2(b).

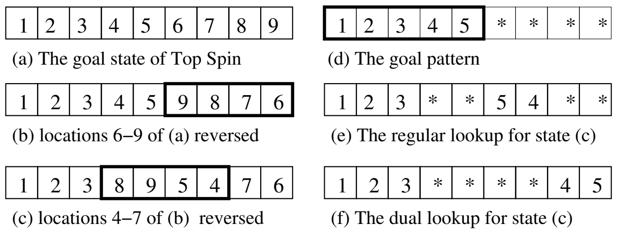
1 2 3 4 5 6 7 8 9
(a) The goal state of Top Spin
1 2 3 4 5 * * * *
(d) The goal pattern
1 2 3 4 5 9 8 7 6
(b) locations 6–9 of (a) reversed
1 2 3 * * 5 4 * *
(e) The regular lookup for state (c)
1 2 3 8 9 5 4 7 6
(c) locations 4–7 of (b) reversed
1 2 3 * * * * 4 5
(f) The dual lookup for state (c)

Figure 2: (9,4)-TopSpin states

Suppose patterns for TopSpin are defined by ignoring tokens 6-9. The resulting PDB provides distances to the goal pattern (shown in Figure 2(d)) from all reachable patterns. Consider Figure 2(c). The regular PDB lookup for this state is based on the pattern in Figure 2(e), obtained from (c) by mapping tokens 6-9 to "*". Since tokens 1-5 are in locations 1, 2, 3, 7 and 6, respectively, the regular lookup would be $PDB[1][2][3][7][6]$. For the dual lookup, we note that the goal locations of tiles 1-5 (as in state (c)) are occupied by tokens 1, 2, 3, 8, 9. The dual lookup, $PDB[1][2][3][8][9]$, gives the cost of moving these tiles to their goal locations. This lookup corresponds to the pattern shown in Figure 2(f).

Dual PDB lookups are possible where there is symmetry between objects and locations of the domain in the sense that each object is located in one location and each location occupies only one object. Example domains where this is true are the TopSpin puzzle (Section 4) and Rubik's cube (Section 5). A counter example is the towers of Hanoi where there is no symmetry between locations and objects. The 15 puzzle domain is more difficult because the blank has to be handled differently from the other tiles. Nevertheless, even in this seemingly asymmetric domain, the PDBs can be constructed in such a way as to enable the dual symmetry (see Section 6).

3 Bidirectional Pathmax ($bpmx$)

Regular PDB lookups produce consistent heuristic values during search [Holte *et al.*, 1996]. Dual lookups are admissible, but not necessarily consistent. For our TopSpin example, let h_d be the result of a dual lookup on this PDB. In Figure 2(b), tokens 1-5 are in their goal locations and therefore $h_d(b) = 0$. Figure 2(c) is obtained from (b) by a single move. However, the dual lookup for this state uses the pattern in Figure 2(f), which is two moves away from the goal pattern. Therefore, $h_d(c) = 2$, which is inconsistent with $h_d(b) = 0$ since (c) is only one move from (b).

[Mero, 1984] described two methods of propagating heuristic values between a state and its children to take advantage of inconsistencies. Let P be any state, $\{C_i\}$ the children of P, and $dist(P, C_i)$ the cost of reaching C_i from P. Mero's first propagation method, now known as pathmax, propagates heuristic values from P to its children: $h(P) - dist(P, C_i)$ is a lower bound on $dist(C_i, Goal)$ and therefore can be used instead of $h(C_i)$ if it is larger. Mero's second method propagates heuristic values upwards, from the children of P to P. The path from P to the goal must pass through a child of P [1]. Thus, $min(h(C_i) + dist(P, C_i))$ is a lower bound on $dist(P, Goal)$ and can be used instead of $h(P)$ if it is larger.

Previous work failed to notice that when operators are invertible (and costs symmetric), pathmax allows values to be propagated in both directions and is also applicable in undirected graphs. This might produce a more useful children-to-parent propagation than Mero's second method.

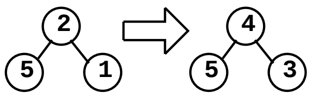

Figure 3: Propagation of values with inconsistent heuristics

The bidirectional pathmax method (*bpmx*) is illustrated in Figure 3. The left side of the figure shows the (inconsistent) heuristic values for a node and its two children. When the left child is generated, its heuristic ($h = 5$) can propagate up to the parent and then down again to the right child. To preserve admissibility, each propagation reduces h by the cost of traversing that path (1 in this example). This results in $h = 4$ for the root and $h = 3$ for the right child. When using IDA*, this bidirectional propagation can cause many nodes to be pruned that would otherwise be expanded. For example, suppose the current IDA* threshold is 2. Without the propagation of h from the left child, both the root node ($f = g + h = 0 + 2 = 2$) and the right child ($f = g + h = 1 + 1 = 2$) would be expanded. Using the propagation just described, the left child will improve the parent's h value to 4, resulting in a cutoff without even generating the right child.

4 TopSpin

We have implemented the above ideas on the (17,4)-TopSpin puzzle. This domain has $17! = 3.55 \times 10^{14}$ states. We generated a PDB of the leftmost 9 tokens, a pattern space of $17 \times 16 \ldots \times 9 = 8.82 \times 10^9$. Since this puzzle is cyclic, we can assume that token number 1 is always in the leftmost

[1]This is only true in a directed graph. In an undirected graph the shortest path from P to the goal might pass through the parent of P.

Heuristic	Nodes	Time	*bpmx*
1r+0d	40,019,429 / 1.0	67.76 / 1.0	0
0r+1d	7,618,805 / 5.3	15.72 / 4.3	0
0r+1d+c	1,397,614 / 28.6	2.93 / 23.1	194,135
2r+0d	6,981,027 / 1.0	21.90 / 1.0	0
1r+1d+c	492,686 / 14.2	1.47 / 14.9	34,725
0r+2d+c	372,414 / 18.7	1.46 / 15.0	29,302
4r+0d	651,080 / 1.0	4.05 / 1.0	0
0r+4d+c	143,177 / 4.6	1.09 / 3.7	4,638
8r+0d	116,208 / 1.0	1.48 / 1.0	0
4r+4d+c	82,606 / 1.4	0.94 / 1.6	1,155
0r+8d+c	74,610 / 1.6	1.12 / 1.3	915
17r+17d+c	27,575	1.34	29

Table 1: Solutions to the (17,4) TopSpin puzzle

position. Thus, for implementation, both numbers above can be divided by 17. Since all the values in the PDB were smaller than 16, each entry needs 4 bits and the PDB needs 259MB.

A PDB of 9 tokens has actually 17 different ways of choosing which tokens are included. A PDB of tokens $[1 \ldots 9]$ can also be used as a PDB of $[2 \ldots 10]$, $[3 \ldots 11]$, etc, with the appropriate mapping of tokens. Thus, a single PDB gives us 17 regular heuristics and 17 dual heuristics. The search was done using IDA*. Many duplicate states can be avoided by forcing two unrelated operators to be applied successively in only one order. For example, the operator that reverses locations $(1, 2, 3, 4)$ is not related to the operator that reverses locations $(11, 12, 13, 14)$. This operator ordering decreased the number of generated nodes by an order of magnitude.

Table 1 presents data for different heuristics and combinations. Each value in the table is an average over a set of 1,000 random permutations. The average solution length for this test set is 14.8. All the experiments reported in this paper were run on a 1.7GHz Pentium 4 PC with 1GB of memory.

The table columns give the number of regular ('r') and dual ('d') lookups used, the presence of *bpmx* cutoff ('c'), the number of generated nodes (nodes and ratio to the same number of lookups with only-r result), the time (seconds and ratio), and the number of times that the *bpmx* cutoff occurred.

The results show that a single dual lookup outperforms a regular lookup by a factor of 5.3 in generated nodes (4.3 in running time). This is because the dual lookup frequently "jumps" to different areas of the PDB and has a larger diversity of different heuristic values as will be further explained in Section 7. The *bpmx* cutoff further improves this to a factor of 28.6 in nodes (23.1 in time). The *bpmx* cutoff was applicable 194,135 times, pruning 6,221,191 nodes, and averaging 32 nodes per instance of cutoff. These performance gains are achieved using no additional storage, just by looking at one PDB in different ways.

The table also shows the results of using two lookups and taking their maximum. Compared to two regular PDB lookups, two dual lookups (with *bpmx*) give an 18.7-fold reduction in nodes. This result is better than combining one regular and one dual lookup. When four lookups are used, again the dual-only lookup solution is better than the regular-only lookup solution. We see diminishing returns when more

Heuristic	Nodes	Time
1r+0d	90,930,662 / 1.0	28.18 / 1.0
0r+1d	19,653,386 / 4.6	7.38 / 3.8
0r+1d+c	8,315,116 / 10.9	3.24 / 8.7
2r+0d	12,649,720 / 1.0	4.68 / 1.0
1r+1d+c	2,997,539 / 4.2	1.34 / 3.5
0r+2d+c	5,290,272 / 2.4	2.32 / 2.0
4r+0d	1,053,522 / 1.0	0.64 / 1.0
2r+2d+c	1,667,320 / 0.6	0.90 / 0.7
0r+4d+c	1,053,759 / 1.0	0.67 / 0.9
4r+4d+c	615, 563	0.51
24r+24d+c	362,927	0.90

Table 2: Solutions to Rubik's cube from one 7-edges PDB

and more lookups are done. Many lookups provide a diversity of heuristic values anyway. Therefore, the improvement factor of any additional lookup (dual or regular) decreases.

Note that our fastest implementation uses 4 regular and 4 dual lookups took 0.94 seconds – 72 times faster than a single regular lookup. Using 17 regular and 17 dual lookups produces the smallest search tree of only 27,575 generated nodes – a factor of 1,451 over a single regular lookup.

We used our fastest implementation (4r+4d+c) to solve larger versions of TopSpin. TopSpin $(19, 4)$ is 18×17 times larger than the $(17, 4)$ variant. We solved 20 instances for TopSpin $(19, 4)$. The average solution length is 17.3, 73 million nodes are generated and the search takes 172 seconds. For TopSpin $(20, 4)$ (19 times larger), we tested on five problems with an average solution length of 20. These problems averaged 2.9 billion nodes and took 7,716 seconds.

5 Rubik's Cube

[Korf, 1997] solved the $3 \times 3 \times 3$ Rubik's cube, containing roughly 4×10^{19} different reachable states. There are 20 movable sub-cubes, or *cubies*. They can be divided into eight *corner cubies*, with three faces each, and twelve *edge cubies*, with two faces each. As a first experiment, we built a 7-edge-cubies PDB, the largest that can be stored in 1GB of memory. There are 510,935,040 possible permutations of the 7 edge cubies. At 4 bits per entry, 255MB are needed for this PDB. As with TopSpin, symmetries in the domain mean that there are multiple possible regular and dual lookups.

Table 2 presents results for a number of possible combinations of this setting (the table headings have the same meaning as in Table 1). The start states used were "easy"—100 different states obtained by 14 random moves from the goal configuration (average solution length of 10.66).

The results are similar to the TopSpin experience, albeit slightly lower. Again, the dual lookup and $bpmx$ cutoffs result in large reductions in the search effort. However, for this puzzle by the time you hit four lookups, diminishing returns has taken over and the advantage of the dual has dissipated. Our best implementations reduced the number of nodes generated (24r+24d+c) by a factor of 250, and the time (4r+4d+c) by a factor of 55. All this was possible with just one 7-edge-cubies PDB stored in memory.

[Korf, 1997]'s original 1997 Rubik's cube experiments were repeated, this time with dual PDB lookups. Korf used three PDBs for this domain: one PDB for the 8 corner cubies and two PDBs for two sets of 6 edge cubies. Since a legal move in this domain moves 8 cubies, the only way to combine these 3 PDBs is by taking their maximum. Note that there are 8 corner cubies and all 8 are used by the 8-corner PDB. Thus, performing a dual lookup for this particular PDB is irrelevant. Here, the entire space of corner cubies is in the database and both lookups give the same result.

Results for the same set of 10 random instances used in [Korf, 1997] were obtained. The results for Korf's set of $8 + 6 + 6$ PDBs were improved by a modest amount by adding the dual lookups for both 6-edge PDBs (from 353 billion nodes to 253 billion). Increasing the edges PDB from 6 to 7 cubies and using a $8 + 7r + 7r + 7d + 7d$ setting reduced the search to 54 billion nodes – an improvement of a factor of 6.4 over Korf's initial setting. The improvements of adding dual lookups for the 6- and 7-edges PDBs are modest since most of the time the 8-corner PDB has the maximum value; this PDB is larger and contains more cubies than the 6- and 7-edge PDBs. We measured these rates over 10 million random instances. For the $8 + 6r + 6r + 6d + 6d$ setting, the 8-corner PDB had the maximum value in 73.5% of these cases while one of the lookups in the 6-edges cubies was the maximum in only 7.3% of the cases (the rest of the cases are a tie). These numbers were changed to 40.8% and 21.3% respectively for the $8 + 7r + 7r + 7d + 7d$ setting.

6 Sliding-tile Puzzles

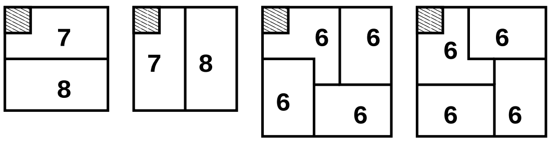

Figure 4: Partitionings and reflections of the tile puzzles

We have also implemented the new ideas for the sliding-tile puzzles. For the 15-puzzle, we used the same $7 - 8$ partitioning from [Korf and Felner, 2002] (Figure 4). These PDBs are constructed so that their heuristic values can be $added$ [Felner *et al.*, 2004] together and preserve admissibility. The PDBs can be reflected across the diagonal, obtaining another set of $7 - 8$ heuristics (also shown in Figure 4).

Dual lookups for this domain are not obvious. While there are 16 similar locations, the 16 tiles are not similar as there are 15 real tiles and one blank. Given the location of the blank, then a horizontal line (or a symmetric vertical line) across the middle of the puzzle divides it into two regions of 8 locations. One region (call it A) has 8 locations which are occupied by 8 real tiles, and another region of 8 locations (B) which are occupied by 7 real tiles and the blank. Performing a dual lookup for region A in the 8-tile PDB is identical to what was done in TopSpin and Rubik's cube.

A dual lookup in B is complicated by the blank. Figure 5 shows four possible blank locations in B for a horizontal partitioning. Other locations of the blank (as well as a vertical

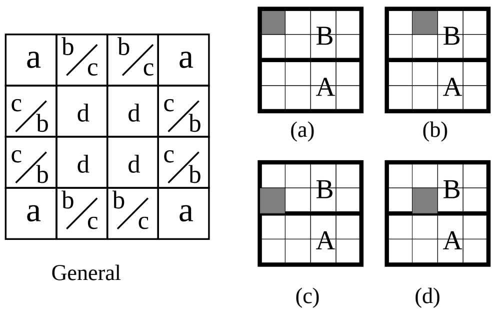

Figure 5: Four different (dual) 7-tile pattern databases

partitioning into two regions) can be easily transformed into one of these 4 configurations (e.g., by reflection). Consider the regular 7-tile PDB of tiles $(1 \dots 7)$ (which corresponds to the 7 real tiles of region B of Figure 5(a)). This PDB answers the question of how to move these real tiles from any possible configuration to their goal configuration. The dual lookup should answer the question of how many moves it takes to distribute the tiles that are currently located in locations $(1 \dots 7)$ to their goal configuration. If these tiles are all real tiles (and the blank is in the corner) then getting this data from the regular PDB is as before. However, when the blank is not in the corner, (the other cases of Figure 5) one of the tiles occupying locations $(1 \dots 7)$ is the blank. Therefore, the answer from a dual lookup of the regular 7-tile PDB will also count moves of the blank while the real tile in the corner will be ignored. This might lose admissibility.

There are two possible ways to solve this problem. The first is to artificially move the blank tile to the nearest corner. In effect, this means reducing the PDB value by one or two (to account for the extra blank moves) to preserve admissibility. Now we can use the regular 7-tile PDB for a dual lookup, at the cost of a weaker heuristic (and, hence, additional search).

A better idea is to add three more 7-tile PDBs for a total of four—one for each of the cases in Figure 5. For each blank scenario we build a regular 7-tile PDB assuming that the blank is located in the relevant location. For example, for region B of Figure 5(b) we build a PDB for the tiles $\{0, 2, 3, 4, 5, 6, 7\}$ and assume that tile 1 is the blank tile. This PDB cannot be consulted as a regular PDB since we assume that tile 1 is the blank. However, for any partitioning where region B corresponds to this case, we can perform a dual lookup in this PDB and retrieve the correct value for the tiles that are currently located in locations $\{0, 2, 3, 4, 5, 6, 7\}$ (or their reflections). Similarly for the other blank locations.

The frame on the left of Figure 5 indicates the relevant PDB for the dual lookup of each possible blank location. In those locations where two PDBs are given, then the right label indicates the PDB to use for a horizontal partition while the left corresponds to a vertical partition. The amount of memory needed is 519KB for the 8-tile PDB and 57.5KB for a 7-tile PDB. Thus the total memory needs (the 8-tile and 4 7-tile PDBs) is 749KB. The three extra PDBs needed to handle all the dual cases correctly represent a small increase of memory.

Table 3 presents results of the different heuristics averaged over the same 1000 instances used in [Korf and Felner, 2002].

Heuristic	Av. H	Nodes	Time
1r	44.75	136,289 / 1.0	0.081 / 1.0
1r+1r*	45.63	36,710 / 3.7	0.034 / 2.4
1d	44.39	278,820 / 0.5	0.157 / 0.5
1d+c	44.39	247,299 / 0.6	0.139 / 0.6
1d+1d*+c	44.40	65,349 / 2.1	0.076 / 1.1
1r+r*+1d+1d*+c	46.12	18,601 / 7.3	0.022 / 3.7

Table 3: Results for the 15 puzzle

PDBs	Nodes	Time	Memory
1r+1r*	16,413,254,279	32,826	244,140
1r+1r*+1d+1d*+c	6,877,105,604	22,955	976,562

Table 4: 24 puzzle results

The average solution for these instances is 52.55. The first column indicates the heuristic used, with 'r*' and 'd*' representing the reflected regular and dual PDB lookups. The first row presents the results when only the regular PDB is used, while the second row took the maximum of the regular and reflected PDBs. Note that these two rows are the same results obtained by [Korf and Felner, 2002] but on our current machine. The next three rows present results for different versions of the dual lookup. Note that for this domain the $bpmx$ cutoff yielded a reduction in nodes of only 10% for a single dual lookup. Finally, the last row presents the maximum over the four PDB combinations. Using dual lookups reduces the number of generated nodes by more than a factor of 2 and eliminated one third of the execution time compared to the best results of [Korf and Felner, 2002] (line 2 of Table 3). To our knowledge using the four regular/dual normal/reflected PDB lookups gives the best existing heuristic for this puzzle. Of historical note is that the number of generated nodes is now nearly 30,000 times smaller than when IDA* first solved the 15-puzzle using only Manhattan distance [Korf, 1985].

Similar experiments were performed using the 24-puzzle. The original 6-6-6-6 partitioning from [Korf and Felner, 2002] (Figure 4) needed storage for only two 6-tile PDBs since all the 3×2 rectangles are symmetric. As before, we need additional PDBs to handle the blank. We use 8 6-tile PDBs: one for all the 3×2 rectangles and their duals, but we need 7 6-tile PDBs for the irregular shape in the top left corner (see Figure 4). Each 6-tile PDB needs 122MB and our new system needs 8 times as much memory.

In [Korf and Felner, 2002] 50 random instances of the 24-puzzle were solved. Table 4 presents the average results over the 25 easiest problems of that set (the 25 with the fewest nodes generated). The average solution length for this set is 96.2 moves. Using both regular and dual lookups and their reflections reduces the number of generated nodes by a factor of 2.38 and the time by a factor of 1.43 when compared to the results of [Korf and Felner, 2002] (first line of Table 4).

7 Discussion

Dual PDB lookups double the number of possible PDB lookups. They are effective for all the domains studied but perform differently on the various domains. There are two phenomena that need explantion. First, the $bpmx$ cutoff was much better in TopSpin and Rubik's cube (a factor of up to 5) than in the tile puzzles (only 10%). Second, in TopSpin and Rubik's cube a single dual PDB lookup (even without the $bpmx$ cutoff) was better than a single regular PDB by a factor of up to 5.3, while in the tile puzzles it was worse.

To explain this we define the *inconsistency rate* of a heuristic as the average difference between the heuristics of an arbitrary pair of neighboring states. For a consistent heuristic, this rate must not exceed 1 (assuming a uniform edge cost of 1). We measured this rate for 10 million pairs of states for each domain. For the dual lookups the rates were 2.01 for TopSpin, 1.74 for Rubik's cube (7-edge PDB) and only 1.16 for the 15-puzzle.

Values in PDBs are locally correlated. However, in TopSpin 4 tokens change their locations at each move (8 cubies for Rubik's cube). The identity of the tokens queried by the dual lookup can dramatically change between consecutive steps. This is a dramatic "jump" to different place in the PDB. There is no locality of values with such a jump, meaning there is higher chance of getting a radically different heuristic value (better or worse). This causes IDA* to generate less nodes as mistakes of heuristics (such as low estimations) are being corrected fast. This also increases the rate of inconsistency and thus the $bpmx$ cutoff is performed frequently.

In the 15-puzzle however, every operator only moves one tile. This means that in two consecutive PDB lookups (regular or dual) most of the indices stay the same – and you get similar values. Furthermore, in most cases of the 15-puzzle, the blank and the tile that exchange their locations belong to the same set of 8 locations indicated by region B above. Thus the identity of the 7 and 8 tiles for the dual lookups remain the same in consecutive steps. Only when the blank crosses the partition line (such as moving from position 4 to position 8 in Figure 5c) then the 7-tile set and the 8-tile sets for the dual lookups are changed and there is some chance for a dramatic jump and for inconsistency. However, even in this case the inconsistency rate is low because we *add* heuristics of two different subproblems which together report values for all the 15 tiles. Thus, the identity of the tiles being reported is not changed. Therefore, the diversity and inconsistency rate is low and the $bpmx$ cutoff is not performed frequently.

We can conclude that dual heuristics and $bpmx$ cutoff are more effective in domains where each operator changes larger parts of the state and the identity of objects being reported in consecutive steps is different.

Another interesting phenomenon is the fact that unlike the other domains, in the 15 puzzle the pure dual PDB lookup was worse than the pure regular lookup and generated almost twice as many nodes. The reason for this is again the location of the blank. Note that while the regular PDB lookup always consults the 8-tile PDB and the 7-tile PDB labeled a in Figure 5, the dual PDB might also consult the other 7-tile PDBs (labeled b, c and d). This is because the regular lookup always aims for a region B configuration such that the blank is located in a corner (the goal state) while the dual lookup needs to consider other possibilities for region B. It turns out that getting the blank to the corner is a harder task and needs more moves. While the average value over all the entries of the PDB labeled a in Figure 5 is 20.91, the average values of the PDBs labeled b, c and d are 20.81, 20.31 and 20.53 respectively. Thus, we expect that the values obtained by the dual lookups will be smaller than those obtained by the regular PDB. Historically, the goal location of the blank is in the corner. However, if we set a goal state such that the blank is in location 4 (as in figure 5.c) then the regular heuristic will always look in the weakest PDB while the dual heuristic will consult the other PDBs as well. We have made such experiments and indeed, the pure dual PDB lookup generated nearly 40% less nodes than the pure regular PDB.

It is important to note that the dual lookups for the tile puzzles are of great importance. TopSpin and Rubik's cube have many symmetries and thus enable many possible regular PDB lookups. In the tile puzzles, however, there is only one symmetry available for the state-of-the-art additive heuristic—the reflection about the main diagonal. Thus, the dual idea doubles the number of possible lookups and achieved a speedup of a factor of 2 over the previous benchmarks.

8 Summary and Conclusions

We presented a new way of using pattern databases. We show that the dual lookup heuristic values are inconsistent, which allows additional opportunities for achieving cut-offs in the search. Results on TopSpin, Rubik's cube, and the sliding tile puzzles confirm the advantages of this new heuristic. To the best of our knowledge we have the best published optimal solvers for all three domains.

Inconsistent heuristics have a negative reputation; something to be avoided. We showed that there is nothing to fear with inconsistent heuristics and there can be real benefits.

References

[Culberson and Schaeffer, 1994] J. C. Culberson and J. Schaeffer. Efficiently searching the 15-puzzle. Technical report, Department of Computer Science, University of Alberta, 1994.

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

[Edelkamp, 2001] S. Edelkamp. Planning with pattern databases. *ECP-01*, 2001.

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

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

[Holte *et al.*, 2004] R. Holte, J. Newton, A. Felner, and D. Furcy. Multiple pattern databases. *ICAPS*, pages 122–131, 2004.

[Korf and Felner, 2002] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134:9–22, 2002.

[Korf, 1985] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *AIJ*, 27:97–109, 1985.

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

[Mero, 1984] L. Mero. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, 23:13–27, 1984.