Dual Search in Permutation State Spaces

Uzi Zahavi
Computer Science
Bar-Ilan University
Ramat-Gan, Israel 92500
zahaviu@cs.biu.ac.il

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

Robert Holte and Jonathan Schaeffer
Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{holte,jonathan}@cs.ualberta.ca

Abstract

Geometrical symmetries are commonly exploited to improve the efficiency of search algorithms. We introduce a new logical symmetry in permutation state spaces which we call *duality*. We show that each state has a *dual* state. Both states share important attributes and these properties can be used to improve search efficiency. We also present a new search algorithm, *dual search*, which switches between the original state and the dual state when it seems likely that the switch will improve the chances of a cutoff. The decision of when to switch is very important and several policies for doing this are investigated. Experimental results show significant improvements for a number of applications.

Introduction

The states of many combinatorial problems (e.g., Rubik's cube, 15-puzzle) are defined as permutations of a set of constants (or *objects*) over a set of state variables (or *locations*). These problems can be solved optimally using search algorithms such as IDA* in conjunction with an admissible heuristic, $h(S)$. The effectiveness of the search is greatly influenced by the accuracy of $h(S)$. *Pattern databases* (PDBs) have proven to be effective for generating accurate, admissible, consistent heuristics for combinatorial puzzles and other problems (Culberson & Schaeffer 1998; Korf & Felner 2002; Edelkamp 2001; Korf 1997; Zhou & Hansen 2004). PDBs are lookup tables that focus on a given subset of objects (replacing the other objects by "don't care" symbols). Each possible configuration of these objects (pattern) has its minimal distance to the goal state computed and saved in the PDB. A *regular PDB lookup* for a state S (denoted as $PDB[S]$) is done by mapping S into the PDB and retrieving the heuristic value from the appropriate entry.

In many application domains, geometric symmetries enable additional lookups to be done in the PDB for a given state (Culberson & Schaeffer 1998; Korf & Felner 2002; Felner *et al.* 2004). The maximum over all such lookups can be a better heuristic value. For example, given a state of the 15-puzzle, S, reflecting the locations of the tiles about the main diagonal produces mirror patterns, and the maximum between the different lookups can be used as $h(S)$.

In (Felner *et al.* 2005) we introduced a new principle for making an additional PDB lookup, called the *dual PDB lookup*. In permutation state spaces, the roles played by the objects and locations are interchangeable and when these roles are flipped, we get *dual patterns* which are used for the dual PDB lookups. In this paper this principle is generalized. Specifically, our paper's contributions are as follows:

1: A general formal definition of *duality* (which applies not only to patterns but also to states) is given, along with precise conditions for it to be applicable. The dual of a state, S, is another state, S^d, that is easily computed from S and shares key search-related properties with S, such as being the same distance from the goal. The dual PDB lookup in (Felner *et al.* 2005) is precisely $PDB[S^d]$, but we show that much more can be done with S^d than a PDB lookup.

2: A new type of search algorithm, *dual search*, is introduced. It is a novel bidirectional search algorithm, with the surprising property that it does not have to maintain a (large) search frontier data structure. Further, it has the unusual feature that it does not necessarily visit all the states on the solution path it returns. Instead, it constructs its solution path from path segments that it finds in disparate regions of the state space. The jumping from region to region is effected by choosing to expand S^d instead of S whenever doing so improves the chances of achieving a cutoff in the search.

3: Experiments with Rubik's cube, the pancake puzzle, and the 24-puzzle show that dual search can reduce the number of nodes IDA* generates by up to an order of magnitude.

Simple Duality

This section defines simple duality, which applies to permutation state spaces (e.g., Rubik's cube) in which the operators have no preconditions (every operator is applicable to every state). A later section gives a general definition, applicable to state spaces in which operators have preconditions. Both definitions make three assumptions:

1: Every state is a permutation of a fixed set of *constants*. For example, the most natural representation of the 8-puzzle has 9 constants, eight representing the individual tiles and one representing the blank.

2: The operators' actions are *location-based permutations*, meaning that an operator permutes the contents of a given set of locations without any reference to specific domain constants. For example, an operator could swap the contents of locations A and B.

3: The operators are invertible, and an operator and its inverse cost the same. Consequently, if operator sequence

O can be applied to state S_1 and transforms it into S_2, then its inverse, O^{-1}, can be applied to state S_2 and transforms it into S_1 at the same cost as O.

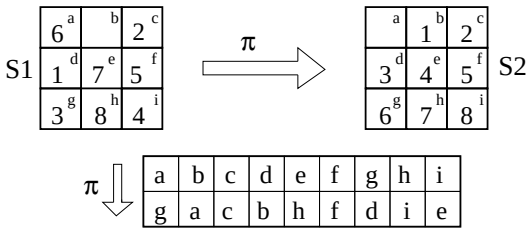


Figure 1: Location-based permutation π that maps S_1 to S_2

For any given pair of states, S_1 and S_2, there is a unique location-based permutation, π, that describes the net effect of any legal sequence of operators that transforms S_1 to S_2. For example, π in Figure 1 describes how the constants move from their locations in the 8-puzzle state S_1 to their locations in S_2. The letters a, b, etc. denote the locations. π maps a to g in Figure 1 because the constant 6 that is in location a in S_1 is in location g in S_2. Note that π can be determined by comparing the two state descriptions, without knowing an operator sequence that transforms S_1 to S_2 or even knowing if such a sequence exists.

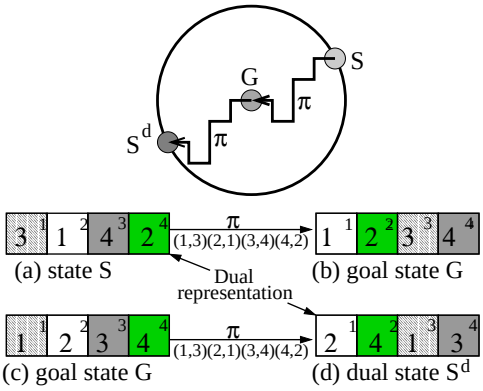


Figure 2: Simple duality, $S^d = \pi(G)$

Duality (simple definition): For state S and goal state G, let π be the location-based permutation such that $\pi(S) = G$. Then S^d, the simple *dual* of S for goal G, is defined to be $\pi(G)$. π is applicable to G because we assume, for this definition, that operators have no preconditions. In practice S^d is calculated by constructing π from the descriptions of S and G and then applying π to G.

This definition is illustrated in the circle in Figure 2. With our assumptions, the cost of reaching G from S and S^d is the same, and therefore $max(h(S), h(S^d))$ is an admissible heuristic for S for any admissible heuristic h.

If we enumerate both the objects and the locations and assume that in the goal state G object i is located in location i then the following interesting observation is true.

Observation: If in S an arbitrary object j is located in location i then in S^d object i will be located in location j.

Proof: π moves the content of location i to location j. Applying π for the first time (to S) will move object j from location i to location j (its home location in G). Applying π for the second time (to G) will move object i from its home location to location j.

In permutation state spaces, the roles played in representing a state by the objects and the locations are interchangeable. Usually, in a state description the *locations* are the *variables* and the *objects* are the *values*. If we flip the roles of *objects* and *locations* in the vector that describes S we get a *dual representation* where *objects* are the *variables* and *locations* are the *values*. Based on the above observation, this representation corresponds to S^d, the dual state[1] of S.

The lower part of Figure 2 shows the relation between state vector S (Figure 2(a)) and its dual, S^d (Figure 2(d)). Figure 2(a,b) shows S being mapped to G by the permutation π, with the definition of π written beneath the arrow ($(1,3)$ means that the constant in location 1 in S is mapped to location 3 in G, etc.). In the lower part of the figure π is applied to G to produce S^d. The vector that describes S, $< 3, 1, 4, 2 >$, means that location 1 is occupied by object 3, 2 by 1, etc. In the *dual representation* this vector means that object 1 is in location 3, 2 is in location 1, etc. The state that corresponds to the dual representation is S^d in Figure 2(d).

Using the heuristic of the dual state might produce *inconsistent* values even if the heuristic itself is consistent (i.e., for any two states, x and y, $|h(x) - h(y)| \leq cost(x, y)$). In a standard search, a parent state, P, and any of its children, S, are neighbors by definition. Thus a consistent heuristic must return consistent values when applied to P and S. However, the heuristic values obtained for P^d and S^d might not be consistent because P^d and S^d are not necessarily neighbors. In (Felner *et al.* 2005) we introduced the *bidirectional pathmax* (BPMX) method for propagating inconsistent heuristic values during search, and showed that it can be very effective in pruning subtrees that would otherwise be explored. We regard BPMX as an integral part of any search algorithm and used it in all the experiments reported in this paper.

Dual search

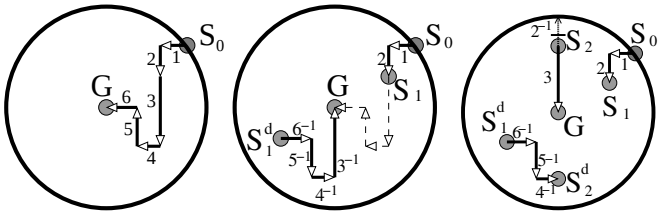
Traditionally, heuristic search algorithms find optimal solutions by starting at the initial state and traversing the state space until the goal state is found. The various traditional search algorithms differ in their decision as to which state to expand next, but in all of them a solution path is found only after all the states on the path have been traversed. We introduce the *dual search* algorithm, which has the remarkable property of not necessarily visiting all the states on the solution path. Instead, it constructs its solution path from solution path segments that it finds in disparate regions of the state space. In this paper we demonstrate this idea with DIDA*, the dual version of IDA*. Dual versions for other algorithms can be similarly constructed.

Dual IDA* (DIDA*)

Recall that the distance to the goal G from both S and S^d is identical and therefore the inverse, O^{-1}, of any optimal

[1] In (Felner *et al.* 2005) we used this idea of flipping the roles of objects and locations to produce *dual patterns*. Our current observation generalizes this principle to the entire state and allows any heuristic of the dual state (not just PDBs) to be used. The dual PDB lookups as defined in (Felner *et al.* 2005) are equivalent to the regular lookup of the dual state (i.e., to $PDB[S^d]$).

path, O, from S^d to G is an optimal path from S to G. This fact presents a choice, which DIDA* exploits, for how to continue searching from S. For each state S, DIDA* computes $h(S)$ and $h(S^d)$. Suppose that $max(h(S), h(S^d))$ does not exceed the current threshold. DIDA* can either continue from this point using S, as IDA* does, or it can switch and continue its search from S^d. Switching from S to S^d is called *jumping*. A simple policy for making this decision is to jump if S^d has a larger heuristic value than S – larger heuristic values suggest that the dual side has a better chance of achieving a cutoff sooner (due to the locality of the heuristic values). We call this the *jump if larger* (JIL) policy. Deciding when to jump is an important part of the algorithm, and alternatives to JIL are discussed later. Of course, later on in the search, DIDA* might decide to jump back to the regular side (e.g., when that heuristic value is better). Once the goal state is reached an optimal solution path can be reconstructed, as described below, from the sequence of dual and regular path segments that led to the goal from the start.



(a) No jumps (b) One jump (c) Two jumps

Figure 3: Dual IDA* Search (*DIDA**)

Figure 3 illustrates the difference between IDA* and DIDA*. In Figure 3(a), IDA* finds a path from S_0 to G. In Figure 3(b), the DIDA* search starts the same: starting at regular state S_0 moves 1 and 2 are made, leading to state S_1. Then, because of its jumping policy, DIDA* switches to the dual state S_1^d. No further switches occur, and DIDA* continues on the dual side until the goal G is reached. In 3(c), the DIDA* search starts out the same as in Figure 3(b) but at state S_2^d a jump is made back to the regular side and DIDA* continues from S_2 to G.

DIDA* has similarities to bidirectional search (e.g. (Kaindl & Kainz 1997)) because it searches for the optimal path in both directions. DIDA* does not depend on the actual states of the path – only the operator sequence of the optimal path matters. While bidirectional search suffers from the need to maintain a (large) search frontier to detect when the forward and backward search meet, DIDA* has no additional storage needs.

Constructing the solution path

IDA* constructs its solution path by backtracking from the goal state to the start state, recovering the path in reverse order. This will not work in DIDA* since some of the moves are on the regular side while some are on the dual side. The solution is to maintain an additional bit per state during the search, the *side* bit, indicating whether the search at that point is on the regular or the dual side. At the start of the search, the side bit is set to $REGULAR$. A child inherits the bit value of its parent, but if a jump occurs, the value of the side bit is flipped. To construct the solution path, DIDA* backtracks up the search tree to recover the moves made to reach the goal. If the side bit for the current move, o, has the value $REGULAR$, then o is added to the $front$ of the partially built path as usual. However, if the side bit indicates that o is on the dual side, then its inverse, o^{-1}, is added at the end of the partially built path.

In Figure 3(a), when IDA* backtracks, the solution path is reconstructed by adding the moves to the front of the partially built path, resulting in the path being built in the order $\{6\}, \{5,6\}, \ldots, \{1,2,3,4,5,6\}$. Figure 3(b) illustrates how this works in DIDA*. Backtracking from G will lead to the following pairs of values (corresponding to the move and the side bit) in the order $\{(3^{-1}, D), (4^{-1}, D), (5^{-1}, D), (6^{-1}, D), (2, R), (1, R)\}$. Since the side bit of the first four moves indicates that they belong to the dual side, the inverses of those moves are added to the end of the partially built path, yielding the partially paths of $\{3\}, \{3,4\}, \{3,4,5\}, \{3,4,5,6\}$. Now the side bit indicates that the search occurred in the regular side. Hence the next two moves are inserted at the front of the path, obtaining $\{2,3,4,5,6\}$ and $\{1,2,3,4,5,6\}$. The dashed line in Figure 3(b) shows how to concatenate the solution path from S_1^d to G in its correct place.

Algorithm 1 DIDA*. "::" adds an element to a list.

```
1   DIDA*(initial_state S) (returns an optimal solution)
2     let threshold = max(h(S), h(S^d))
3     let Path = NULL
4     repeat{
4.1   GoalFound = DDFS(S,NULL,NULL,0,REGULAR,Path)
4.2    threshold = next threshold
4.3   } until GoalFound
5     return Path

1   boolean DDFS(state S, previous_move pm_r,
      previous_dual_move pm_d, depth g, bool side_bit, list Path)
2     let h = max(h(S), h(S^d))
3     if (h + g) > threshold return false
4     if S = goal_state return true
5     if should_jump(S, S^d){
5.1       S = S^d
5.2       swap(pm_r , pm_d)
5.3       side_bit = ¬side_bit
5.4   } endif
6     for each legal_move m {
6.1       if m = pm_r^-1 continue    /*operator pruning*/
6.2       generate child C by applying m to S
6.3       if DDFS(C, m, pm_d, g + 1, side_bit, Path) = true{
6.3.1         if (side_bit = REGULAR) then Path = m :: Path
6.3.2         else Path = Path :: m^-1
6.3.3         return true
6.3.4      } endif
6.4    } endfor
7     return false
```

Algorithm 1 presents the pseudo code of DIDA*. DIDA* mirrors IDA* by iteratively increasing a solution cost threshold until a solution is found. Each iteration calls DDFS (dual depth-first search), which recurses until a solution is found

or the cost threshold is exceeded. DIDA* differs from a standard IDA* search in several respects. First, each call to DDFS includes extra parameters: a side_bit (indicating if the search is currently on the REGULAR or DUAL side) and the last move made on the regular and dual sides (explained later). Second, a jump decision is included, possibly resulting in a jump (lines $5-5.4$). Finally, when the goal has been found, the reconstruction of the solution path distinguishes between the regular and dual sides (lines $6.3.1-6.3.2$).

The Benefit of Jumping

The regular and dual states are different and, hence, there can be large differences in the (admissible) heuristic values between states S and S^d. By using the side that has the highest heuristic value (for the current context), one is increasing the chances of moving into a region of the search space with values high enough to create a cutoff. Of course, the decision to switch sides is a heuristic and not guaranteed to improve the search every time a jump is made.

The Penalty for Jumping

Usually, depth-first search algorithms avoid generating duplicate nodes by disallowing operators that can be shown to be irrelevant based on the previous sequence of operators. The simplest example of this is disallowing the inverse of the previous operator. More sophisticated techniques enforce an ordering on the operators, disallowing redundant sequences. We call such mechanisms *operator pruning*. Operator pruning can significantly reduce the branching factor. For example, the branching factor of Rubik's cube at the root node is 18, but the average branching factor below the root can be reduced to 13.35 (Korf 1997).

There can be no operator pruning at the start state, because there is no search history. Let its branching factor be b. Subsequent nodes in a normal search have a smaller branching factor, at most $b-1$, because of operator pruning. By contrast, bidirectional search, in addition to expanding the start state and some of its descendants, will begin its backwards search by expanding the goal state. Since search in this direction has no history, the goal will have a branching factor of b. Hence bidirectional search pays a penalty; it has two states with a branching factor of b, not just one.

DIDA* pays a penalty for the same reason as bidirectional search. As before, the start state has a branching factor of b, and subsequent nodes on the regular side have a lower branching factor. However, on every branch of the search tree, when a jump is made to the dual side for the first time *only*, the dual state has no search history and will have a branching factor of b. On subsequent jumps we can use the history on that side to do operator pruning. In Algorithm 1, the previous moves from the regular and dual sides are passed as parameters, allowing DIDA* to prune the inverse of the previously applied operator on a given side.

To illustrate this, consider Figure 3(c). DIDA* has to consider all operators at the start state, S_0. Moves 1 and 2 are made on the regular side, reaching S_1. Here DIDA* decides to jump to S_1^d; a completely new state with no history. Thus, operator pruning is not possible here and all the operators must be considered. DIDA* makes moves 6^{-1}, 5^{-1} and 4^{-1} on the dual side until state S_2^d is reached. DIDA* then jumps to the dual state of S_2^d, S_2, back on the regular side. Because it is returning to the regular side, a history of the previous moves is known and operator pruning can be used in expanding S_2. For example, the previous operator on this side is operator 2, so its inverse, 2^{-1}, can be ignored. To understand why operator pruning can be applied, even though S_1 bears no apparent relation to S_2, recall how DIDA* constructs its final solution path. If a path is found leading from S_2 to the goal, the first operator on this path will be placed immediately after the operator that leads to S_1 in the final solution path. Since this path is optimal, it cannot possibly contain an operator followed immediately by its inverse. The same reasoning justifies the use of more sophisticated operator pruning techniques as well.

To avoid the penalty of jumping, a degenerate jumping policy, which only allows a jump at the root node, can be used (JOR). If $h(root) > h(root^d)$ then the search is conducted on the regular side, otherwise it is conducted on the dual side. No further jumps are allowed for JOR.

Experimental results: Rubik's Cube

Table 1 shows the average results for 100 "easy" instances of Rubik's cube (the start states are 14 random moves from the goal state, and have an average solution length of 10.66). The heuristic used was a PDB based on a pattern with seven edge cubies. The table columns are as follows:
H: Heuristic (PDB lookups: r for the regular state and d for its dual state).
O: Operator pruning (yes or no).
S: Search algorithm (IDA* or DIDA*).
P: Jumping rule used by DIDA*.
Nodes and **Time:** Average number of generated nodes and the average number of seconds needed to solve the problem.
Jumps: Average number of times that DIDA* jumped between the regular and dual sides.

H	O	S	P	Nodes	Time	Jumps
max(r,d)	-	IDA*	-	29,583,452	30.27	0
max(r,d)	-	DIDA*	JIL	19,022,292	20.44	3,627,504
max(r,d)	+	IDA*	-	2,997,539	3.43	0
max(r,d)	+	DIDA*	JIL	2,697,087	3.16	15,013
max(r,d)	+	DIDA*	JOR	2,464,685	2.82	0.23

Table 1: Rubik's cube (7-edges PDB) results

The first two rows compare IDA* and DIDA* with operator pruning disabled. Results show that DIDA* with JIL reduced the number of generated nodes by one third. In lines $3-5$ operator pruning was enabled. Line 3 presents the results from (Felner *et al.* 2005). Line 4 shows that DIDA* with the JIL jumping policy yields a modest improvement over line 3. The results show that operator pruning is important and, in this domain, the penalty of DIDA* almost offsets the benefits. Applying the JOR policy (line 5) improves the results by a modest amount. The *Jump* value reveals that in 23 of the cases the dual heuristic at the start state was better and the search was performed in the dual side; the other 77 cases had ties or a better regular heuristic.

Experimental results: Pancake Problem

For the pancake puzzle (Dweighter 1975), DIDA* with the JIL policy produces significant performance improvements.

In this domain, a state is a permutation of the values $0...(N-1)$. A state has $N-1$ successors, with the k^{th} successor formed by reversing the order of the first $k+1$ elements of the permutation ($1 \leq k < N$). From any state it is possible to reach any other permutation, so the size of the state space is $N!$. In this domain, every operator is applicable to every state. Hence its branching factor is $N-1$.

In this domain there are no obvious redundant operator sequences, so only the trivial pruning of the parent is possible, making the branching factor below the root $N-2$. When performing the first jump to the dual side, on any particular branch, the branching factor increases by only one, from $N-2$ to $N-1$.

H	O	S	P	Nodes	Time	Jumps
max(r,d)	+	IDA*	-	2,205,610,700	3176	0
max(r,d)	+	DIDA*	JIL	223,305,375	344	903,892

Table 2: 17-pancake puzzle results

Table 2 presents results averaged over 30 random instances of the 17-pancake problem (search space is 3.55×10^{14} states). The heuristic used was a PDB based on tokens $10, 11, \ldots, 16$ (which gives slightly better average heuristic values than a PDB based on tokens $0, 1, \ldots, 6$). DIDA* gives an impressive 9.88-fold improvement over IDA* – from 53 minutes to less then 6 minutes.

General Duality

The simple definition of duality used so far assumes that any operator sequence that can be applied to any given state S can also be applied to the goal G. This only applies to search spaces where operators have no preconditions. In the sliding tile puzzles, for example, operators have preconditions (the blank must be adjacent to the tile that moves) and an operator sequence that applies to S will not be applicable to G if the blank is in different locations in S and G. A more general definition of duality, allowing preconditions on operators, will now be given. Assumptions 1–3 are still needed. Nevertheless, with this general definition, dual heuristic evaluations and dual search are possible for a much wider range of state spaces, including the tile puzzles.

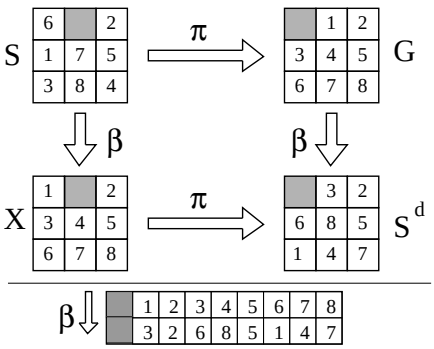


Figure 4: General duality $S^d = \pi(\beta(S)) = \beta(\pi(S))$

Duality (general definition): The dual of a given state, S, for goal state G, can be defined with respect to any state X such that any sequence of operators that can be applied to state S can also be applied to state X and vice versa. If π is the location-based permutation such that $\pi(S) = G$, then S^d, the dual of S with respect to X, is defined to be $\pi(X)$. As a special case, if $X = G$ (this is possible if any operator sequence applicable to S is also applicable to G) then this definition becomes the simple definition given earlier. The 8-puzzle state S and the goal state G of Figure 4 do not have the same applicable operators. For example, the operator "*move the tile in the upper left corner to the right*" is applicable to S but not to G. We want to find a state X such that all operator sequences applicable to S will be applicable to X. This is done with the mapping β, which renames the tiles to transform S into X. For the given S this X could be any state having the blank in the same position as S. S^d can be derived in two ways, either by applying π to X or by renaming the tiles in G according to β. π (shown in Figure 1), for example, maps the tile in the upper left location in S, or in X, to the lower left location in G, or S^d, respectively. β, by contrast, renames the constant 6 in S, or in G, to the constant 1 in X, or S^d, respectively.

By definition, any legal sequence of operators that produces S^d when applied to X can be legally applied to S to produce G, and vice versa. Because an operator and its inverse cost the same, duality provides an alternative way to estimate the distance from S to G: any admissible estimate of the distance from S^d to X is also an admissible estimate of the distance from S to G. If PDBs are being used, general duality suggests using a PDB, PDB_X (with X as the goal state), in addition to the usual PDB, PDB_G (with G as the goal). Given a state S, in addition to the standard heuristic value, $PDB_G[S]$, we will also get a heuristic value for the dual state by computing π for S and then looking up $PDB_X[\pi(X)]$.

It is possible to have multiple states, $\{X_i\}$, each playing the role of X in the definition. In this case, a state S will not have just one dual, it would have a dual with respect to each X_i that had the all-important property that any sequence of operators applicable to S is also applicable to X_i and vice versa. A PDB, PDB_{X_i} would be built for each X_i (with X_i as the goal). Lookups for the dual state of S could be made in PDB_{X_i} for each X_i for which a dual of S is defined.

This is precisely what we will do for the tile puzzle. X_i will be a state in which the blank is in position i, and we will build a PDB for each X_i. Then, given a state S with the blank in position i, we calculate the dual of S with respect to X_i and look up its value in PDB_{X_i}. For example, in the 8-puzzle there are 9 different locations the blank could occupy. Define 9 different states, $X_0 \ldots X_8$, with X_i having the blank in position i, and compute 9 PDBs, one for each X_i. Of course, geometric symmetries can be used to reduce the number of distinct PDBs that must actually be created and stored. For example, below only seven PDBs are needed to cover all possible blank locations in the 24-puzzle.

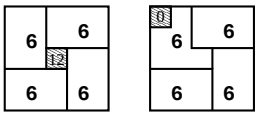
Suppose dual search is proceeding on the "regular side" (with G as the goal) and decides at state S to jump to S_i^d, the dual of S with respect to X_i. Search now proceeds with X_i playing the role of the goal in all respects. In particular: (1) if X_i is reached, the search is finished and the final solution path can be reconstructed; and (2) the permutation π is calculated using X_i instead of G. The latter point has an

important implication for the sliding tile puzzles: the dual of any state generated when the search goal is X_i will have the blank in location i.

Experimental results on the 24-puzzle

0	1	10	1	0
5	6	11	6	5
10	11	12	11	10
5	6	11	6	5
0	1	10	1	0

(a) symmetries



(b) 6-6-6-6 PDB

PDB	Average
0	19.82
1	19.09
5	19.76
6	19.60
10	19.32
11	19.89
12	20.30

(c) Quality

Figure 5: 24 puzzle heuristic

The sliding tile puzzle has two important attributes that did not arise in the previous domains, but should be taken into account in DIDA*'s jumping policy. First, the branching factor is not uniform, it varies from 2 to 4 depending on the location of the blank, and will often be different for S and S^d. Second, as explained above, we will have several different PDBs, each based on an X_i having the blank in a different position. The X_i are chosen to maximally exploit the geometrical symmetries of the puzzle, so that although there are 25 positions the blank could be in, only 7 PDBs are needed. They are numbered $0, 1, 5, 6, 10, 11, 12$, with the number reflecting the location of the blank in the X_i that defined the PDB. Figure 5(a) indicates which PDB is to be used for each possible position of the blank, and Figure 5(b) shows two different 6-6-6-6 additive PDB partitionings, PDB_{12} and PDB_0 (the PDB case used in (Korf & Felner 2002)). Figure 5(c) shows the average heuristic value for each of the PDBs. Note that a small difference in average PDB value can have a dramatic effect on the PDB's pruning power. Because S and S^d will often have the blank in a different location, and therefore draw their heuristic values from different PDBs, it is important for the jumping policy to take the average value of the PDBs into account.

Our jumping policy for the 24-puzzle, J24, considers both these attributes. It is a three-step decision process. First, the effective branching factor of the regular and dual states is compared. This is done by considering the blank location and the history of the previous moves, choosing to prefer the state with the smaller effective branching factor. Second, if there is a tie, then the quality (average value) of the PDB (as presented in Figure 5(c)) is considered. Preference is given to the PDB with the higher average. Third, if there is still a tie, then the JIL policy is used.

H	#	S	P	Nodes	Jumps	Imp
max(r,r*)	25	IDA*	-	43,454,810,045	-	1.00
max(r,d)	25	IDA*	-	31,103,112,894	-	1.40
max(r,d)	25	DIDA*	JIL	16,302,942,680	176,075,343	2.67
max(r,d)	25	DIDA*	J24	8,248,769,713	23,851,828	5.27
max(r,r*)	50	IDA*	-	360,892,479,671	-	1.00
max(r,r*)	50	DIDA*	J24	75,201,250,618	147,733,548	4.79

Table 3: 24-puzzle results

In (Korf & Felner 2002), 50 random instances of the 24-puzzle were solved. The first four lines of Table 3 presents the average results over the 25 problems of that set with the shortest optimal paths. The average solution length for this set is 95 moves. The first line presents the benchmark results from (Korf & Felner 2002) where the maximum between the regular PDB (r) and its reflection about the main diagonal (r^*) were taken. The second line is IDA* with regular and dual PDB lookups. The next line is DIDA* with JIL. Finally, the last line shows DIDA* with J24. The results show an increasing improvement factor. The last two lines present results for the entire set of 50 instances. DIDA* with J24 outperforms the benchmark results by a factor of 4.79. Note that all the variations in Table 3 are for programs using exactly two PDB lookups. Since the overhead of DIDA* over IDA* (e.g., the jumping decision) is significantly dominated by the PDB lookup overhead, all these versions ran at roughly 300,000 nodes per second on our machine.

Conclusions and future work

DIDA* is a novel form of a bidirectional search. By exploiting the logical symmetries in a domain, DIDA* can switch between state representations to maximize the overall quality of the heuristic values seen in the search. The algorithm has several surprising properties, including no need for a search frontier data structure and solution path construction from disparate regions of the search space. The resulting algorithm provides significant (up to an order of magnitude) performance gains in several application domains.

Future work can continue in the following directions. 1. Obtaining a better understanding of the jumping policies. Given an application, how does one go about determining the best policy? 2. Analysis to see if the duality concept can be generalized to encompass a wider set of application domains. 3. Integrating the idea of duality in other search algorithms (e.g. A*).

Acknowledgments

The financial support of the Israeli Ministry of Science Infrastructure grant No. 3-942, of NSERC and of iCORE is greatly appreciated.

References

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

Dweighter, H. 1975. Problem e2569. *American Mathematical Monthly* 82:1010.

Edelkamp, S. 2001. Planning with pattern databases. *Proc, ECP-01* 13–34.

Felner, A.; Meshulam, R.; Holte, R.; and Korf, R. 2004. Compressing pattern databases. In *Proc. AAAI-04*, 638–643.

Felner, A.; Zahavi, U.; Holte, R.; and Schaeffer, J. 2005. Dual lookups in pattern databases. In *IJCAI-05*, 103–108.

Kaindl, H., and Kainz, G. 1997. Bidirectional heuristic search reconsidered. *JAIR* 7:283–317.

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

Korf, R. E. 1997. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proc. AAAI-97*, 700–705.

Zhou, R., and Hansen, E. 2004. Space-efficient memory-based heuristics. In *Proc. AAAI-04*, 677–682.