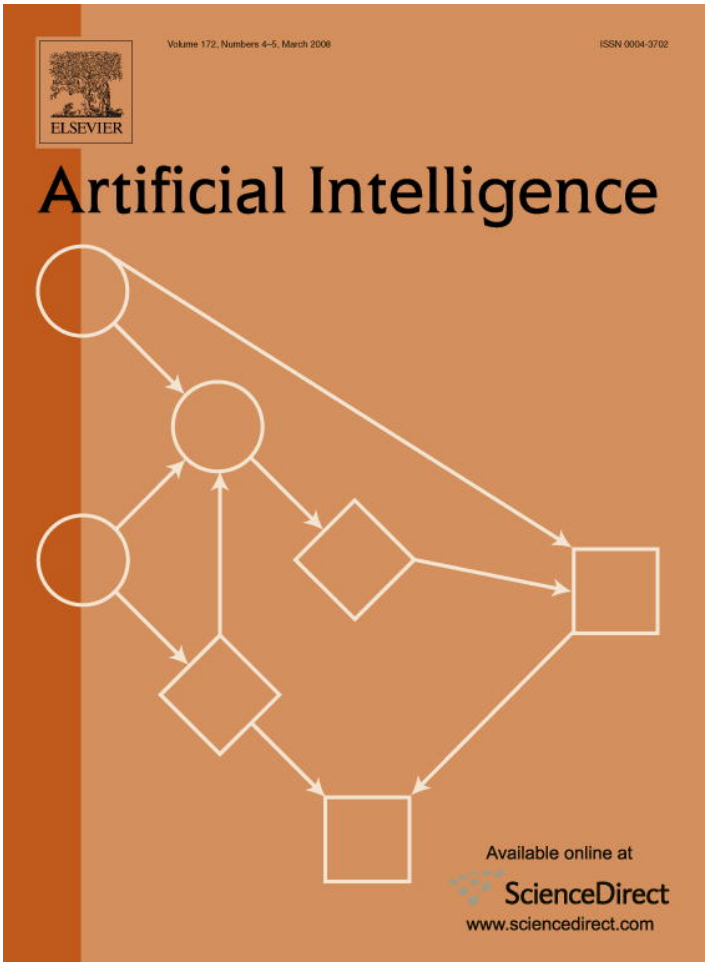


Available online at www.sciencedirect.com
ScienceDirect
Artificial Intelligence 172 (2008) 514–540
Artificial Intelligence
www.elsevier.com/locate/artint

Duality in permutation state spaces and the dual search algorithm

Uzi Zahavi [a], Ariel Felner [b,*], Robert C. Holte [c], Jonathan Schaeffer [c]

[a] *Computer Science Department, Bar-Ilan University, Ramat-Gan, Israel*
[b] *Department of Information Systems Engineering, Ben-Gurion University, Israel*
[c] *Computing Science Department, University of Alberta, Edmonton, Alberta, Canada*

Received 2 August 2006; received in revised form 27 June 2007; accepted 22 October 2007

Available online 6 November 2007

Abstract

Geometrical symmetries are commonly exploited to improve the efficiency of search algorithms. A new type of symmetry in permutation state spaces, *duality*, is introduced. Each state has a *dual* state. Both states share important attributes such as their distance to the goal. Given a state S, it is shown that an admissible heuristic of the dual state of S is an admissible heuristic for S. This provides opportunities for additional heuristic evaluations. An exact definition of the class of problems where duality exists is provided. A new search algorithm, *dual search*, is presented which switches between the original state and the dual state when it seems likely that the switch will improve the chance of reaching the goal faster. 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, for using the dual state's heuristic evaluation and/or dual search.

Keywords: Heuristics; Search; Admissibility; Duality

1. Introduction and overview

The states of many combinatorial problems (e.g., Rubik's cube, 15-puzzle) are defined as placements of a set of m *objects* into a set of n locations (where $n \geqslant m$). All the different ways to put the objects into the locations with at most one object per location defines a state space which is called a *permutation state space* in this paper.[1] Given two states in a permutation state space, *start* and *goal*, and a set of operators that transform one state into another, search algorithms such as A* [8] and IDA* [12] can be used to find the shortest sequence of operators that transform *start* into *goal*. These algorithms use a cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach state n from *start* and $h(n)$ is an *admissible* (i.e. is always a lower bound) heuristic function estimating the cost from n to *goal*.

* Corresponding author.
E-mail addresses: zahaviu@cs.biu.ac.il (U. Zahavi), felner@bgu.ac.il (A. Felner), holte@cs.ualberta.ca (R.C. Holte), jonathan@cs.ualberta.ca (J. Schaeffer).

[1] Strictly speaking, a permutation would require n, the number of locations, to be exactly the same as m, the number of objects. We have relaxed this requirement and only demand that $n \geqslant m$. We use the term *strict permutation state space* to refer to state spaces in which the states are permutations in the strict sense ($m = n$).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2007.10.019

The effectiveness of the search is greatly influenced by the accuracy of $h(n)$. When $h(n)$ is more accurate, the number of nodes generated in a search decreases and the goal state is reached sooner [16]. There is a tradeoff for this reduction, however. More accurate heuristics usually consume a larger time overhead per node generated and therefore the percentage reduction in the actual time needed to solve a problem is smaller in practice than the percentage reduction in the total number of generated nodes. Usually, the reduction in the number of generated nodes dominates the constant time per node and therefore a time reduction is seen as well [4,17].

In this paper a new type of symmetry is discussed—*duality*. It is based on the observation that in *strict*[2] permutation state spaces, i.e., when $m = n$, the roles played by objects and locations are interchangeable. By reversing these roles, a state, S, can be mapped to its dual representation, S^d. Given an admissible heuristic, h, the value $h(S^d)$ is a lower bound on the distance from S to the goal. Taking the maximum of $h(S)$ and $h(S^d)$ can result in a better heuristic value for S and, hence, less search. Further, if $h(S^d) > h(S)$, this can be exploited by using a search algorithm that *switches representations when it appears likely to be beneficial. The* dual search algorithm searches in the original or dual search space, switching representations to whichever has a higher heuristic value.

The contributions of this paper are as follows:

- A formal definition of *duality* 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. Therefore any admissible heuristic for S^d can be used as an admissible heuristic for S.
- A new type of search algorithm, *dual search*, is introduced. It has the unusual feature that it does not necessarily visit all the states on the solution path that 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.
- Using the heuristic evaluation of the dual state ($h(S^d)$) in the search shows a significant performance improvement for a number of domains. Adding the dual search algorithm further improves the results. For all the domains studied, the results represent the best in the published literature.

The idea of *duality* is also used in the constraint satisfaction problems (CSP) literature, where flipping the roles of variables and constraints produces a dual version of the problem. Independent of our work, Hnich et al. discuss methods to use duality in CSP applications [9]. For example, they exploit duality by choosing to solve the variation of the problem that appears to be faster to solve. By contrast, in this paper we introduce duality ideas in the context of heuristic state-space search.

The paper is organized as follows. Sections 2 and 3 present background material. In Section 4, the notion of simple duality is defined. Simple duality is a special case of duality that only applies to *strict permutation states spaces*. Section 5 discusses the properties of the dual heuristic. Section 6 presents a new search algorithm based on duality, DIDA* (Dual IDA*). Section 7 provides experimental evidence for the benefits of using the heuristic evaluation of the dual state and for the dual search algorithm. Section 8 provides generalization of the duality notion to a wider variety of permutation state spaces that are not necessarily *strict*. Experimental results for the general case are then provided in Section 9. A summary and suggestions for future work are provided in Section 10. Preliminary versions of this paper appeared in [7,20].

2. Problem domains and permutation state spaces

This section introduces the three application domains used in this paper and gives a formal definition of permutation state spaces. Pattern databases, used as the heuristic evaluation function for our application domains, are described.

2.1. The sliding-tile puzzles

One of the classic examples in the AI literature of a single-agent path-finding problem is the sliding-tile puzzle. Three versions of this puzzle are the 3×3 8-puzzle, the 4×4 15-puzzle and the 5×5 24-puzzle. They consist of a

[2] We also provide generalization of this idea to permutation state spaces that are not necessarily *strict*.

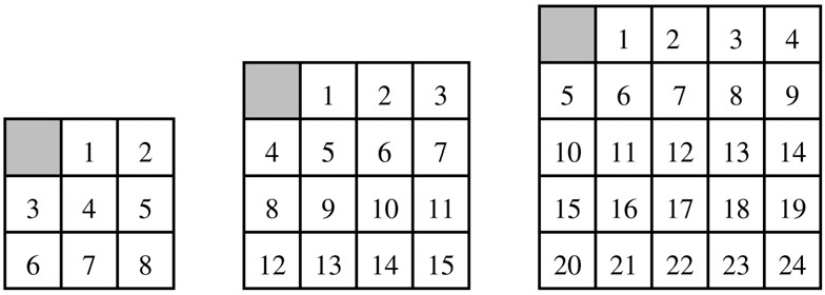


Fig. 1. The 8-, 15- and 24-puzzle goal states.

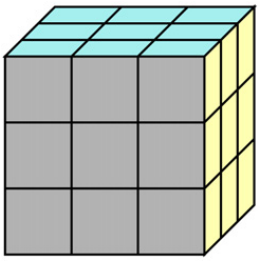


Fig. 2. $3 \times 3 \times 3$ Rubik's cube.

square frame containing a set of numbered square tiles, and an empty position called the blank. The legal operators are to slide any tile that is horizontally or vertically adjacent to the blank into the blank position. The problem is to rearrange the tiles from some random initial configuration into a particular desired goal configuration. The state space grows exponentially in size as the number of tiles increases, and it has been shown [19] that finding optimal solutions to the sliding tile problem is NP-complete. The 8-puzzle contains 9!/2 (181,440) reachable state, the 15-puzzle contains about 10^{13} reachable states, and the 24-puzzle contains almost 10^{25} states. The goal states of these puzzles are shown in Fig. 1.

The classic heuristic function for the sliding-tile puzzles is called Manhattan distance. It is computed by counting the number of grid units that each tile is displaced from its goal position, and summing these values over all tiles, excluding the blank. Since each tile must move at least its Manhattan distance to its goal position, and a legal move only moves one tile, the Manhattan distance is a lower bound on the minimum number of moves needed to solve a problem instance.

2.2. *Rubik's cube*

Rubik's cube was invented in 1975 by Erno Rubik of Hungary. It is one of the most famous combinatorial puzzle of our time. The standard version consists of a $3 \times 3 \times 3$ cube (Fig. 2), with different colored stickers on each of the exposed squares of the sub-cubes, or cubies. Any $3 \times 3 \times 1$ edge plane of the cube can be rotated 90, 180, or 270 degrees relative to the rest of the cube. In the goal state, all the squares on each side of the cube are the same color. The puzzle is scrambled by making a number of random moves, and the task is to restore the cube to its original goal state. There are about 4×10^{19} different reachable states. There are 20 movable cubies and 6 stable cubies in the center of each face. The movable cubies can be divided into eight corner cubies, with three faces each, and twelve edge cubies, with two faces each. Corner cubies can only move among corner positions, and edge cubies can only move among edge positions.

2.3. *The pancake puzzle*

The *pancake puzzle* is analogous to a waiter navigating a busy restaurant with a stack of n pancakes [2]. To avoid disaster, the waiter wants to sort the pancakes ordered by size. Having only one free hand, the only available operation is to lift a top portion of the stack and reverse it. In this domain, a state is a permutation of the values $0 \ldots (N-1)$. A state has $N-1$ successors, with the kth successor formed by reversing the order of the first $k+1$ elements of the

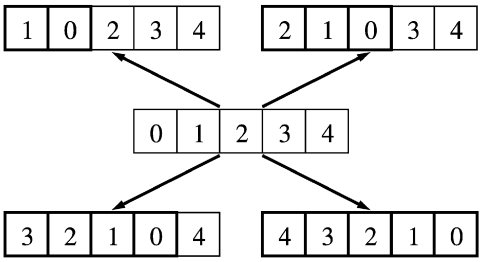


Fig. 3. The 5-pancake puzzle.

permutation ($1 \leqslant k < N$). For example, if $N = 5$ the successors of state $\langle 0, 1, 2, 3, 4\rangle$ are $\langle 1, 0, 2, 3, 4\rangle$, $\langle 2, 1, 0, 3, 4\rangle$, $\langle 3, 2, 1, 0, 4\rangle$ and $\langle 4, 3, 2, 1, 0\rangle$, as shown in Fig. 3. 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 $N - 1$.

2.4. Permutation state spaces

A *state space* is a set of states, and a set of operators that map states to states. A specific problem instance is a state space together with a particular initial state and goal state. The task is to find an optimal path from the initial state to the goal state.

Permutation state spaces are a special type of combinatorial problems, consisting of a set of *m objects* and *n locations* ($n \geqslant m$). The states in such problems are all the different ways of placing the objects in the locations with at most one object per location.

Strict permutation state spaces are a special case of permutation state spaces where the number of objects is exactly the same as the number of locations, i.e., $n = m$.

An *operator* in permutation state spaces changes the locations of some of the objects. This is a very rich and interesting class of problems, including, for example, all finite mathematical groups. The puzzles defined above are all permutation state spaces, as are many of the classic benchmark problems for planning, such as the Blocks World.

Note that general permutation problems even if solved by heuristic search do not necessarily span a *permutation state space*. For example in the Travailing Salesman Problem (TSP) the task is to find the permutation of cities with the optimal cost. However, this problem does not span a permutation state space as defined here. In TSP, there is no predefined goal state and there is no meaning of finding a path from a given initial state to the goal state via other permutation states.

3. Heuristics

The efficiency of a single-agent search algorithm is usually dominated by the quality of the heuristic used. The best known heuristics for the application domains in this paper all take the form of a pattern database (defined below). Pattern databases are therefore used in all the experimental studies, and the purpose of this section is to give the background details on pattern databases. However, it is important to note that none of this paper's key ideas (duality, dual heuristic evaluations, and dual search) depend on the heuristic being a pattern database, these ideas apply to heuristics of all forms.

3.1. Pattern databases

A powerful approach for obtaining admissible heuristics is the use of pattern databases (PDBs) [1]. The state space of a permutation state space problem is all the different ways to placing the given set of objects into the locations. A *subproblem* is an abstraction of the original problem defined by only considering some of these objects while treating the others as "don't care". A *pattern* (abstract state) is a specific assignment of locations to the objects of the subproblem. The *pattern space* or *abstract space* is the set of all the different reachable patterns of a given abstract problem.

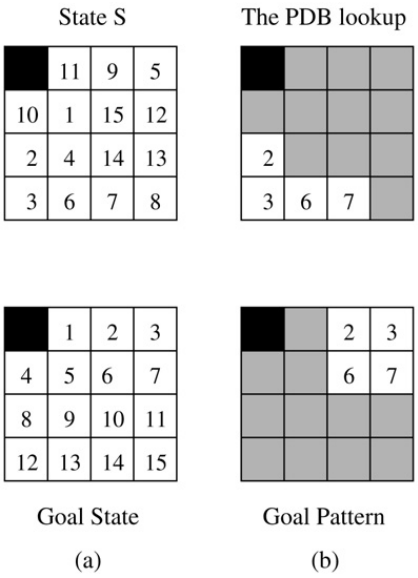


Fig. 4. Example of regular lookups.

Each state in the original state space is *abstracted* to a pattern of the pattern space by only considering the pattern objects, and ignoring the others. The *goal pattern* is the abstraction of the goal state.

There is an *edge* between two different patterns p_1 and p_2 in the pattern space if there exist two states s_1 and s_2 of the original problem, such that p_1 is the abstraction of s_1, p_2 is the abstraction of s_2, and there is an operator of the original problem space that connects s_1 to s_2.

A *pattern database* (PDB) is a lookup table that stores the distance of each pattern to the goal pattern in the pattern space. A PDB is built by running a breadth-first search[3] backwards from the goal pattern until the whole pattern space is spanned. A state S in the original space is mapped to a pattern S' by ignoring details in the state description that are not preserved in the subproblem. The value stored in the PDB for S' is a lower bound (and thus serves as an admissible heuristic) on the distance of S to the goal state in the original space since the pattern space is an abstraction of the original space.

Pattern databases have proven very useful for finding lower bounds for combinatorial puzzles [1,5,6,14,15]. Furthermore, they have proved useful for other search problems (e.g., multiple sequence alignment [18,22] and planning [3]).

3.1.1. Pattern databases example

PDBs can be built for the sliding-tile puzzles, as illustrated in Figs. 4(a) and (b). Assume that the subproblem only includes tiles 2, 3, 6 and 7. 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* has an entry for each pattern containing the distance from that pattern to the goal pattern (shown in the lower part of Fig. 4(b)). Fig. 4(b) depicts the PDB lookup in this PDB for estimating a distance from a given state S to the goal (Fig. 4(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 lookup for state S is *PDB*[8][12][13][14] (tile 2 is in location 8, tile 3 is in location 12, etc.). The value retrieved by a 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. In this paper, accessing the PDB for a state S will be referred to as a *regular* lookup, and the heuristic value will be referred to as a *regular* heuristic.

3.1.2. Additive pattern databases

Additive pattern databases provide the current best admissible heuristic for the sliding-tile puzzles [5,15]. The tiles are partitioned into disjoint sets (patterns) of tiles and a PDB is built for each set. The PDB stores the cost of moving

[3] This description assumes all operators have the same cost. The techniques easily extend to the case when operators have different costs.

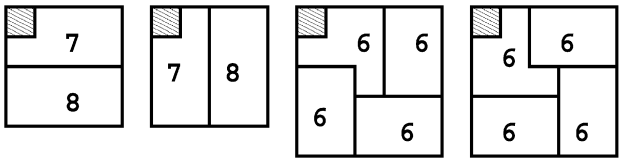


Fig. 5. Partitionings and reflections of the tile puzzles.

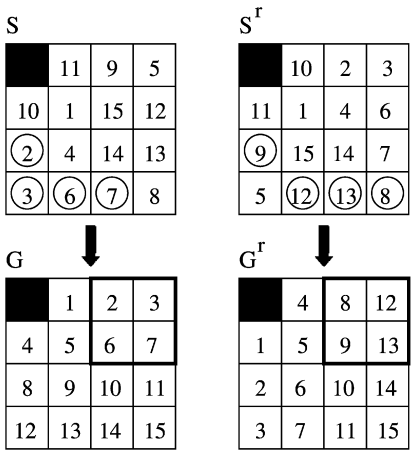


Fig. 6. Both S and G are reflected about the main diagonal to get another possible PDB lookup.

the tiles in the given subproblem from any given arrangement to their goal positions. If for each set of tiles only the moves of tiles from the given set are counted, then values from different disjoint PDBs can be *added* and the result is still admissible. An x–y–z partitioning is a partition of the tiles into disjoint sets with cardinalities of x, y and z. Fig. 5 presents the two 7–8 partitionings for the 15-puzzle and the two 6–6–6–6 partitionings for the 24-puzzle which were first used in [5,15].

3.2. Geometric symmetries

It is common practice to exploit special properties of a state space to enable additional lookups to be done in a PDB. In [1] several alternative lookups that can be made in the same PDB based on the physical symmetries of the 15-puzzle are described. For example, because of the symmetry about the main diagonal, the PDB built for the goal pattern in Fig. 4(b) can also be used to estimate the number of moves required to get tiles 8, 9, 12 and 13 from their current positions in state S to their goal locations. As shown in Fig. 6, both S and G are reflected about the main diagonal yielding S^r and G^r. The 2–3–6–7 PDB can be used to get a lower bound on the number of moves required to get tiles 8, 9, 12, and 13 from their current positions in state S^r to their locations in G^r. This is identical to the number of moves required to move them from their current positions in state S to their goal locations in G.

This idea of reflecting the domain about the main diagonal for having another set of PDBs was also used to solve the 15-puzzle and 24-puzzle with additive PDBs [5,15]. It is easy to see that the two 7–8 partitionings for the 15-puzzle (and similarly those of the 24 puzzle) in Fig. 5 are reflections of each other about the main diagonal and only one PDB is needed in practice. For another example of geometric symmetry, consider Rubik's cube and assume there is a PDB for the blue face which gives values for all cubies with blue colors. Reflecting and rotating this puzzle will enable similar lookups for any other face with a different color (e.g., yellow, red, etc.) since any two faces are symmetric.

Because all valid, alternative PDB lookups provide lower bounds on the distance from state S to G, 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 [1]. 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 [10].

4. Simple duality

In this paper, two types of duality are presented. This section introduces simple duality, which applies to strict permutation state spaces (the number of objects and locations is the same, as in Rubik's cube) in which the operators have no preconditions (every operator is applicable to every state). Section 8 generalizes the notion of duality to state spaces in which there may be more locations than objects and operators may have preconditions.

4.1. Assumptions for simple duality

Before defining the dual state, we first present four assumptions which are preconditions for simple duality:

(1) Every state is an arrangement of m objects into n locations, with $m = n$ and exactly one object per location. In other words, for simple duality we require that the state space is a strict permutation state space. For example, the most natural representation of the 8-puzzle has 9 objects, eight representing the individual tiles and one representing the blank. This assumption will be relaxed in Section 8 to allow $n \geqslant m$ and at most one object per location.
(2) A set of *operators* is given that change the locations of some of the objects. We assume that the operators' actions are *location-based permutations*, meaning that an operator re-arranges the contents of a certain set of locations without any reference to specific objects. 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 transform it into S_2, then its inverse, O^{-1}, can be applied to state S_2 and transform it into S_1 at the same cost as O.
(4) Operators have no preconditions. That is, every operator is applicable to every state. This assumption is only assumed for simple duality. In Section 8 assumption 4 is dropped, resulting in the notion of general duality where the operators are not necessarily applicable to every state but have internal preconditions.

Example domains where assumption 4 is violated are the sliding tile puzzle and the Towers of Hanoi. In the former there is a precondition which refers to the location of the blank. In the latter, there is a precondition which refers to the topmost discs on the operator's source and destination pegs.

Two definitions for the dual state follow, and a proof that they are equivalent.

4.2. Simple duality: Definition 1

For any given pair of states, S_1 and S_2, there is a unique location-based permutation, π, that transforms S_1 to S_2. For example, π in Fig. 7(a) describes how the objects move from their locations in the 4-pancake state S to their goal locations in G. The letters a, b, c and d denote the locations. π maps a to c in Fig. 7(a) because the object (3) that is in location a in S is in location c in G. The permutation π is entirely determined by the state descriptions of S and G, it does not depend on the operators that define the state space. In particular, π is defined whether or not there exists a sequence of operators that transforms S into G.

Dual state (Definition 1). 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 $\pi(G)$.

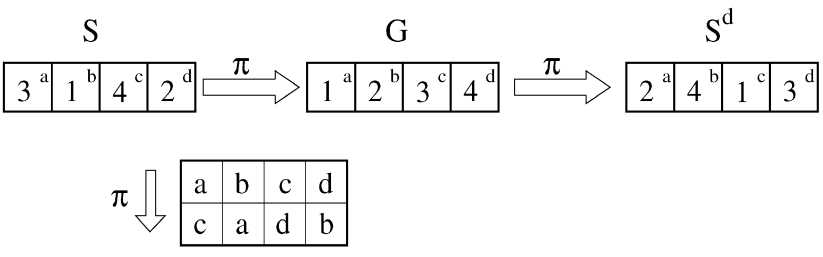


Fig. 7. Location-based permutation π that maps S to G (a) and G to S^d (b).

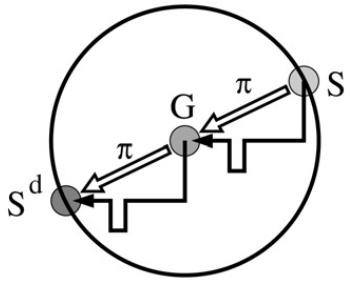


Fig. 8. Simple duality, $G = \pi(S)$ and $S^d = \pi(G)$.

Dual(State S)
1 let G be the goal state
2 for each location x
2.1 let o be the object located in x in S
2.2 let y be the location of o in G
2.3 define $\pi_x = y$
3.3 endfor
3 $S^d = \pi(G)$
4 return S^d

Algorithm 1. *Dual*-Calculating the dual state according to Definition 1.

This definition is illustrated in Fig. 8. As will be shown below, with the assumptions, the cost of reaching G from S and from S^d is the same, and therefore $\max(h(S), h(S^d))$ is an admissible heuristic for S for any admissible heuristic h. For example, if PDBs are being used and $h(S) = PDB[S]$, for some given PDB for G, then $h(S^d) = PDB[S^d]$ is also admissible for S.

In practice S^d is calculated by constructing π from the descriptions of S and G and then applying π to G as shown in Algorithm 1. Therefore, S^d can be calculated from the description of S without the need to know any actual path from S to G as illustrated in Fig. 7(b).

4.3. Simple duality: Definition 2

The advantage of our first definition is that it properly motivates the location of the dual state in the search space. We now provide an alternative definition for the dual state analogous to the dual concept in the constraint satisfaction field (e.g., [9]). We then prove that the two definitions are equivalent.

In strict permutation state spaces, the roles played by objects and locations in representing a state are interchangeable. Usually in the vector containing the state description, the *locations* are the *variables* and the *objects* are the *values*. This is called the *regular representation* of the state. Flipping the roles of *objects* and *locations* in the vector that describes S yields the *dual representation*. Here, *objects* are the *variables* and *locations* are the *values*. Given a vector of size K that represents a state, the regular representation treats it as the objects that occupy locations $\{1 \ldots K\}$. The dual representation will treat them as the locations that are occupied by objects $\{1 \ldots K\}$. For example, if the representative vector is $\langle 3, 1, 4, 2\rangle$ then the regular representation refers to a state $S = \langle 3, 1, 4, 2\rangle$ (object 3 in location 1, object 1 in location 2, etc.). The dual representation of this vector corresponds to the dual state S^d where object 1 is in location 3, object 2 is in location 1, etc. The dual state in its regular representation is $S^d = \langle 2, 4, 1, 3\rangle$.

In this section, we assume a canonical definition of the goal state G. That is, given an enumeration of both the objects and the locations, then in G object i is located in location i. For the goal state, the regular and dual representation are identical.

Dual state (Definition 2). Given a vector representation V of a state S, the dual state, S^d, is defined to be the state which is described by V in the dual representation.

Algorithm 2 is based on Definition 2 and calculates the regular representation of the dual state S^d from the regular representation of S.

Dual(State S)
1. For each object $x \in S$
1.1. Let y be the location of object x in S
1.2. In S^d place object y in location x
2. Return S^d

Algorithm 2. *Dual*-Calculating the dual state based on Definition 2.

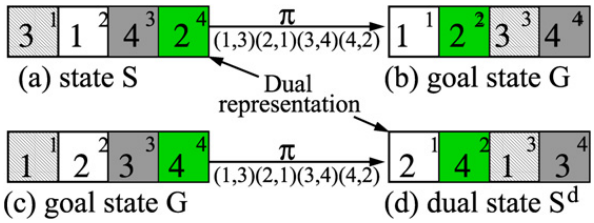


Fig. 9. The relation between a state S and its dual state S^d. Given a representative vector $\langle 3, 1, 4, 2\rangle$, then its regular representation corresponds to S while its dual representation corresponds to S^d.

4.4. Analysis

A number of theorems concerning duality are now given. The following terminology is used. $Loc_S(x) = y$ indicates that object y is located at location x in state S. Therefore, in the goal state G, $\forall i \;\; Loc_G(i) = i$. $C_{opt}(X, Y)$ denotes the cost of an optimal path from state X to state Y.

Theorem 4.1. *Definitions* 1 *and* 2 *are equivalent.*

Proof. Let G be the canonical goal state, S be a state, π be the permutation such that $\pi(S) = G$, and S^d be the dual of state S obtained by Definition 1. It needs to be shown that if in S an arbitrary object j is located in location i, then in S^d object i will be located in location j. Assume that π moves the content of location i to location j. Applying π for the first time (on S) will move object j from location i to location j (its home location in G). Applying π for the second time (on G) will move object i from its home location to location j. □

Fig. 9 shows the relation between state vector S (Fig. 9(a)) and its dual S^d (Fig. 9(d)) according to both definitions. Fig. 9(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 object 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, $\langle 3, 1, 4, 2\rangle$, means that location 1 is occupied by object 3, 2 by 1, etc. In the *dual representation* (where *objects* are the *variables* and *locations* are the *values*) this vector means that object 1 is in location 3, object 2 is in location 1, etc. The state that corresponds to the dual representation is S^d in Fig. 9(d).

Theorem 4.2. $(S^d)^d = S$.

Proof. Show that these states have the same objects in the same location. We will show that $\forall$*Location* y: $(Loc_S(y) = x) \Longrightarrow (Loc_{(S^d)^d}(y) = x)$.
And indeed,

$$\forall \textit{Location } y\text{: } (Loc_S(y) = x) \overset{\textit{def 2}}{\Longrightarrow} (Loc_{S^d}(x) = y) \overset{\textit{def 2}}{\Longrightarrow} (Loc_{(S^d)^d}(y) = x). \qquad \square$$

Theorem 4.3. *If* $O = \{o_1, o_2, \ldots, o_n\}$ *is a legal path from* S *to* G *then* $O^{-1} = \{o_n^{-1}, \ldots, o_2^{-1}, o_1^{-1}\}$ *is a legal path from* S^d *to* G *and has the same cost as* O.

Proof. If $O = \{o_1, o_2, \ldots, o_n\}$ is a legal path from S to G then O is also a legal path from G to S^d (Definition 1). Because all operators can be reversed (Assumption 3) the sequence of operators $O^{-1} = \{o_n^{-1}, \ldots, o_2^{-1}, o_1^{-1}\}$ is a legal

path from S^d to G. Since operators and their inverses cost the same (Assumption 3) the cost of O and O^{-1} is the same. □

Theorem 4.4. $C_{opt}(S, G) = C_{opt}(S^d, G)$.

Proof. There are two cases to consider. If there does not exist an operator sequence transforming S into G, then there cannot exist an operator sequence transforming S^d into G and therefore $C_{opt}(S, G) = C_{opt}(S^d, G) = \infty$. Alternatively, if there does exist an operator sequence transforming S into G, let O be a minimum-cost sequence of n operators that transforms S into G. According to Theorem 4.3, O^{-1} is a legal path from S^d to G of the same cost and therefore $C_{opt}(S, G) \geqslant C_{opt}(S^d, G)$. Applying the same reasoning to any minimum-cost path from S^d to G implies $C_{opt}(S^d, G) \geqslant C_{opt}((S^d)^d, G) \overset{\text{Theorem 4.2}}{\Longrightarrow} C_{opt}(S^d, G) \geqslant C_{opt}(S, G)$. These two inequalities together imply $C_{opt}(S, G) = C_{opt}(S^d, G)$. □

As a result, any admissible heuristic for state S^d is also admissible for state S, and vice versa. Therefore, given a heuristic h for each state S, its *dual heuristic* $h_d(S)$ can also be calculated (the regular heuristic of the dual state, $h(S^d)$). If PDBs are being used, then a PDB lookup for the dual state S^d is used as a heuristic bound for S. Such a PDB lookup for S^d, is called the *dual PDB lookup* for S.[4]

5. Attributes of the dual heuristic

In this section different attributes of the dual heuristic are discussed.

5.1. When the regular and dual heuristics provide different values

Admissible heuristics for permutation problems can be divided into two types. Let π be the permutation that transforms S to G. The first type of heuristic calculates its value by considering the effect of π (the current location) for *all* the objects of the state. For example, Manhattan distance provides a heuristic bound for moving each of the tiles. The second type of heuristic considers the effect of π for only a *subset* of the objects. PDBs, for example, provide full solutions to the relaxed problem that contains only a subset of the objects.

For the first type of heuristic (e.g., Manhattan distance), the dual and regular heuristics are equal. This is because both states are reached from the goal state by applying the same permutation π (in reverse directions) and the heuristic considers a similar effect of π for *all* the objects of the state for both S and S^d. Consider state S and S^d as provided in Fig. 10. For each tile in S a unique tile can be found in S^d with the same Manhattan distance (the same effect of π). For example, in S, tile 1 (located in location 5) has to move one up—the Manhattan distance of this tile is 1. Similarly, in S^d tile 5 (located in location 1) has to move one step down. Note that this is only true when all the objects in the state are considered.

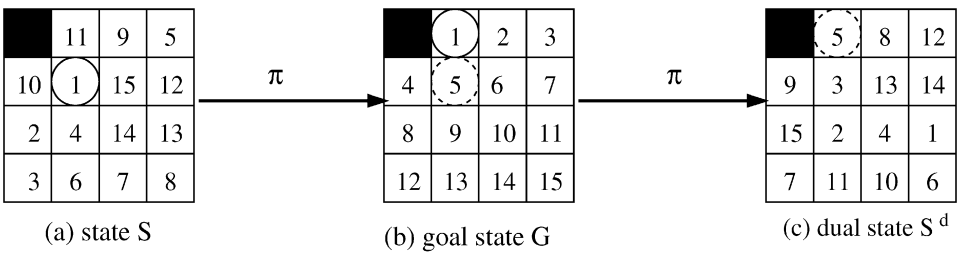


Fig. 10. 15-puzzle duality and Manhattan distance.

[4] In [7], the same idea of flipping the roles of objects and locations is used to produce *dual patterns* and *dual PDB lookups*. The dual representation presented in this paper generalizes this principle to the entire state and allows any heuristic of the dual state S^d (not just PDBs) to be used for S. The definition of dual PDB lookups of [7] is different but is equivalent to the regular lookup of the dual state (i.e., to $PDB[S^d]$) which is defined here.

For the second type, the regular and dual heuristics are not necessarily equal. Using only a subset of objects might cause a different heuristic result. For example, assume that only tile 1 is being considered by Manhattan distance. In Fig. 10 tile 1's Manhattan distance heuristic in S is 1 while in S^d it is 4. For PDBs, only a subset of the objects is considered and therefore the heuristic of the dual state can be different. For example, assume that in Fig. 10, the PDB based on tiles $\{2, 3, 6, 7\}$ is used. The PDB lookup for S has these tiles in locations $\{8, 12, 13, 14\}$. The PDB lookup for S^d has these tiles in locations $\{9, 5, 15, 12\}$. These tiles now have to travel different paths to get to the goal state and therefore S and S^d have completely different entries in the PDB with completely different values.

5.2. Inconsistency of the dual heuristic

A heuristic h is *consistent* if for any two states, x and y, $|h(x) - h(y)| \leqslant dist(x, y)$ where $dist(x, y)$ is the optimal distance between them. In other words, the difference between the heuristic values of two states is never greater then the cost of a path between them. When moving from a parent node to a child node, using the heuristic of the dual state might produce *inconsistent* [21] values even if the heuristic itself (in its regular form) is consistent. 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. This is a consequence of the following corollary.

Corollary 5.1. *Let P and S be two states and let c be the actual distance between them. The distance between P^d and S^d is not necessarily c. In particular it might be larger.*

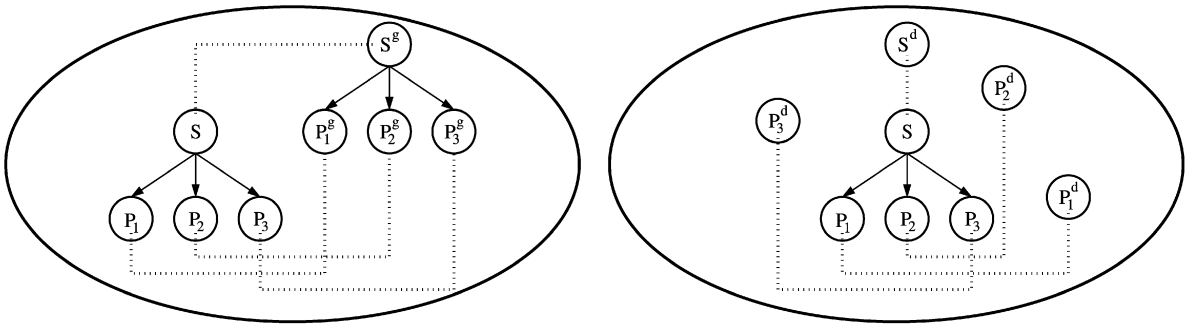
Proof. An example is sufficient. Consider the 9-pancake puzzle states shown in Fig. 11. State G is the goal state of this puzzle. State S_1 is the neighbor of G obtained by reversing the tokens at locations 1–3 (shown in the bold frame), and state S_2 obtained by further reversing the tokens in locations 1–6. States G^d, S_1^d and S_2^d are the dual states of G, S_1 and S_2 respectively. Observe that while states S_1 and S_2 are neighboring states, S_1^d and S_2^d (their duals) are not neighbors. Reversing any consecutive k first tokens of state S_1^d will not arrive at node S_2^d.[5]

A consistent heuristic might return values for S_1^d and S_2^d which differ by more than 1. Using these values for S_1 and S_2 would be inconsistent since they are neighbors. This can be shown by the following PDB example. Suppose patterns for the 9-pancake puzzle are defined by only considering tokens 4–6 while ignoring the rest of the tokens. The

	State	Cost	The corresponding pattern	h
G	1 2 3 4 5 6 7 8 9	0	* * * 4 5 6 * * *	0
S_1	3 2 1 4 5 6 7 8 9	1	* * * 4 5 6 * * *	0
S_2	6 5 4 1 2 3 7 8 9	2	6 5 4 * * * * * *	1
G^d	1 2 3 4 5 6 7 8 9	0	* * * 4 5 6 * * *	0
S_1^d	3 2 1 4 5 6 7 8 9	1	* * * 4 5 6 * * *	0
S_2^d	4 5 6 3 2 1 7 8 9	2	4 5 6 * * * * * *	2

Fig. 11. 9-pancake states.

[5] Note that in this particular example S_1 and S_1^d are identical. In this domain applying a single operator twice in a row will reach the same state and state S_1 is a single move away from the goal.



(a) Geometrical symmetry (b) Dual symmetry

Fig. 12. Geometrical and dual symmetries.

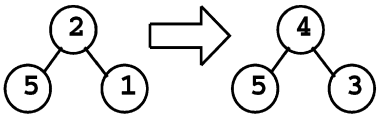


Fig. 13. Propagation of values with inconsistent heuristics.

resulting PDB provides distances to the goal pattern from all reachable patterns. The right column of Fig. 11 shows the corresponding pattern for each state obtained by using the $*$ symbol to represent a "don't care".

Regular PDB lookups produce consistent heuristic values during search [11]. Indeed, since states S_1 and S_2 are neighbors, their PDB heuristic values differ by at most 1. In state S_1, tokens 4–6 are in their goal locations and therefore $h(S_1) = 0$. In state S_2 tokens 4–6 are not in their goal locations and we need to apply one operator to reach the goal pattern and thus $h(S_2) = 1$. Dual PDB lookups are admissible, but not necessarily consistent. The dual PDB lookup for state S_1, i.e., the PDB lookup for state S_1^d returns 0 since tokens 4–6 are in their goal location for state S_1^d. However, the pattern projected from state S_2^d is two moves away from the goal pattern. Thus, performing the dual lookup for states S_1 and S_2 (i.e., PDB lookups for states S_1^d and S_2^d) will produce heuristics that are inconsistent (0 and 2). Thus when moving from S_1 to S_2 (or vice versa), even though g was changed by 1, h was changed by 2. □

5.3. Geometrical symmetries versus dual symmetries

There is a major difference between states obtained by a dual symmetry and states obtained by a geometrical symmetry. This difference is illustrated in Fig. 12. As derived from Corollary 5.1 above, given a state S and its dual S^d the neighbors of S^d are not necessarily the dual states of the neighbors of S. This is shown in Fig. 12b. For geometrically reflected states (such as S^g of Fig. 12a), however, exactly the opposite of Corollary 5.1 is true. That is if the distance between states P and S is c then the distance between P^g and S^g (the geometrical reflected state of P and S) is exactly c. Geometrical symmetries only transform the domain without changing its internal structure. As a result, neighbors of a reflected state S^g are also reflections of the neighbors of S. Therefore, using heuristics of the reflected states (that is, using the PDB lookup towards the reflection of the goal as described above in Section 3.2) also produce consistent heuristics.

5.4. Bidirectional pathmax

In [7,21], the *bidirectional pathmax* (BPMX) method for propagating inconsistent heuristic values during search was introduced, and experiments showed that it can be effective in pruning subtrees that would otherwise be explored.

The bidirectional pathmax method is illustrated in Fig. 13. 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 increase the parent's h value to 4, resulting in a cutoff without even generating the right child. BPMX should be regarded as an integral part of any search algorithm when the heuristic is inconsistent and the operators are invertible, and it is used in all the experiments reported in this paper.

6. 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. *Dual search* has the remarkable property of not necessarily visiting all the states on the solution path. Instead, it constructs its solution path from path segments that it finds in disparate regions of the state space. In this paper, the focus is on DIDA*, the dual version of IDA*. Dual versions for other algorithms can be similarly constructed.

6.1. 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 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). This is referred to as 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.

Fig. 14 illustrates the difference between IDA* and DIDA*. In Fig. 14(a), IDA* finds a path from S_0 to G. In Fig. 14(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 Fig. 14(c), the DIDA* search starts out the same as in Fig. 14(b) but at state S_2^d a jump is made back to the regular side and DIDA* continues from S_2 to G.

6.2. Constructing the solution path

The correctness of DIDA* is best seen by considering how the path segments it finds are joined together to create a path from start to goal. 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 (i.e., the forward search) while some are on the dual side (i.e., the backward search). The solution is to maintain an

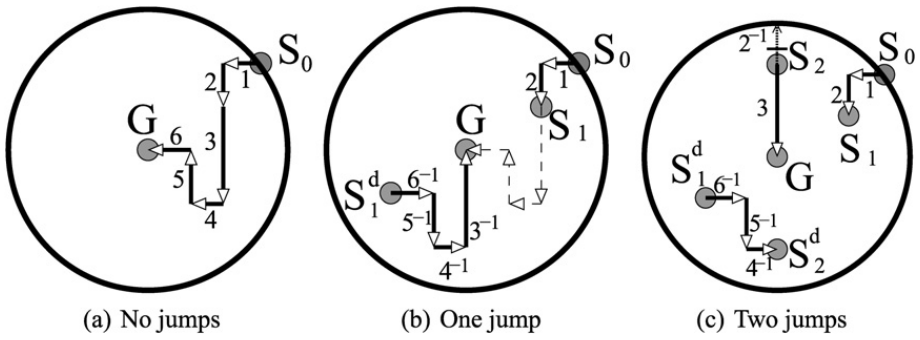


Fig. 14. Dual IDA* search (*DIDA**).

```
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,threshold)
4.2     threshold = next threshold
4.3   } until GoalFound
5     return Path
```

Algorithm 3. DIDA*.

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.

It is important to note that (based on Theorem 4.3) only the operators are taken from the dual side to the regular side. The exact states visited by the path in the dual side are not necessarily the duals of the states of the actual solution path.

In Fig. 14(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 $\{6\}, \{5, 6\}, \ldots, \{1, 2, 3, 4, 5, 6\}$. Fig. 14(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 built 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 Fig. 14(b) shows how to concatenate the solution path from S_1^d to G in its correct place.

Algorithm 3 presents the pseudocode for 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 is presented in Algorithm 4. DDFS 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 (used for operator pruning, as explained in Section 6.4). Second, a jump decision is included in DDFS, 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).

6.3. 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.[6] 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.

Consulting the heuristic of the dual state introduces diversity into the heuristic values obtained during the search; information is obtained from a different area of the search space. DIDA* introduces a stronger diversity since it is not only peeking but is physically jumping into that area.

[6] For example, experiments on the 17-pancake problem with a heuristic which has a maximum value of 14, the difference observed was up to 8.

6.4. 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. Such mechanisms are referred to as *operator pruning* in this paper. 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 by operator pruning to 13.34 [14].

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, DIDA* sometimes pays a branching-factor penalty for jumping to a dual state. 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 on a branch, the history on that side can be used to do operator pruning. In Algorithm 4, 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 Fig. 14(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. In IDA*, operator pruning can be used at all nodes except the root. In DIDA*, the first time a jump is made, on any given branch, no history is available and operator pruning is unavailable. For example, in Rubik's cube, when performing the first jump, on any branch, DIDA* has 18 children to consider, as opposed to the average of 13.34 that would be seen by IDA*.

```
1   boolean DDFS(state S, previous_move pm_r,
      previous_dual_move pm_d, depth g, bool side_bit, List Path,int threshold)
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, threshold) = 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 4. DDFS "::" adds an element to a list.

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.

7. Experimental results for simple duality

This section provides experimental results that show the benefit of performing dual lookups and using the dual search algorithm. Pattern databases are used in all the experimental domains because they represent the state-of-the-art heuristics.

7.1. Rubik's cube

Korf first solved the $3 \times 3 \times 3$ Rubik's cube with PDBs [14]. As discussed above, the cubies of Rubik's cube can be divided to corner cubies and edge cubies. As a first experiment to test the duality ideas, a 7-edge-cubies PDB was built, the largest that can be stored in 1 GB of memory. There are 510,935,040 possible permutations of the seven edge cubies. At four bits per entry, 255 MB are needed for this PDB. The heuristics used in this set of experiments were based on this 7-edges PDB.

Table 1 presents results for this set of experiments. The experiments above were on 100 instances with length $\leqslant 14$. The table columns are as follows:

Heuristic: Which heuristic was used for the PDB lookups: r stands for the regular state and d for its dual state. Similarly, $4r$ ($4d$) means that we took the maximum of 4 regular (dual) heuristics.

OP—operator pruning: "+" means that the operator leading to a node's parent is pruned; "−" means no operators are pruned.

Search: Search algorithm (IDA* or DIDA*).

Policy: The jumping policy used by DIDA*.

Nodes and *Time*: Average number of generated nodes and the average time needed to solve a problem with 3.0 GHz Pentium 4 machine with 2 GB of memory.

Jumps: Average number of times that DIDA* jumped between the regular and dual sides.

The first line of this table shows the results of IDA* using a regular PDB lookup. These searches generated an average of 90 million nodes. Intuitively, one might think that performing only a dual lookup should produce the same results since the exact same PDB is being queried. Surprisingly, however, line 2 shows that when using the dual heuristic the number of generated nodes decreases to only 8 million nodes, an improvement factor of 11. The reason for this dramatic improvement is as follows. While values in a PDB are locally correlated, the dual lookup frequently looks in ("jumps" to) different areas of the PDB. Thus, the general flow of the search benefits from a large diversity of areas in a PDB and a "bad" area can be quickly escaped from. Since the dual heuristic is inconsistent, BPMX was

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

#	Heuristic	OP	Search	Policy	Nodes	Time	Jumps
1	r	+	IDA*	–	90,930,662	28.18	–
2	d	+	IDA*	–	8,315,116	3.24	–
3	$\max(r, d)$	+	IDA*	–	2,997,539	1.34	–
4	$\max(r, d)$	+	DIDA*	JIL	2,697,087	1.16	15,013
5	$\max(r, d)$	+	DIDA*	JOR	2,464,685	1.02	0.23
6	$\max(r, d)$	−	IDA*	–	29,583,452	30.27	–
7	$\max(r, d)$	−	DIDA*	JIL	19,022,292	20.44	3,627,504
8	$\max(4r, 4d)$	+	IDA*	–	615,563	0.51	–
9	$\max(24r, 24d)$	+	IDA*	–	362,927	0.90	–

Table 2
Rubik's cube results

Heuristics	Nodes	Time	Memory
max(8, 6, 6)	352,656,042,894	102,362	130,757
max(8, 6, 6, 6d, 6d)	253,863,153,493	91,295	130,757
max(8, 7, 7, 7d, 7d)	54,979,821,557	44,201	299,757

also used.[7] In line 3, the maximum of both the regular and the dual heuristics is used. This further reduced the number of generated nodes to roughly 3 million, an improvement of a factor of 30 over the benchmark of line 1.

Lines 4–5 shows the results for DIDA* using different jumping policies. DIDA* with JIL (line 4) yields a modest improvement over line 3. Applying the JOR policy (line 5) further 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.

To better understand the penalty incurred by the first jump of DIDA*, operator pruning was disabled and the results for IDA* and DIDA* compared. Results are provided in lines 6–7. Here, the operator that leads to a node's parent is not pruned. In both cases, the maximum of the PDB lookups for the regular and dual states was used. Disabling operator pruning increases the search effort by a factor of 10 when compared to line 3 where operator pruning was enabled. Results show that in this setting DIDA* with JIL reduced the number of generated nodes by one third compared to IDA*. The improvement factor of DIDA* over IDA* was more significant than those reported in lines 4 and 5 since now the penalty of the first DIDA* jump was minor because the operator pruning was disabled.

Due to geometrical symmetries in this domain there are multiple possible regular and dual lookups. Many combinations of geometrical reflected regular lookups and geometrical reflected dual lookups were tried. Since DIDA* does not seem to produce significant improvements for this domain, only IDA* was used here (no jumping). The best results achieved reduced the number of nodes generated (when taking $24r + 24d$) by a factor of 250, and the time ($4r + 4d$) by a factor of 55. All this was possible with just one 7-edge-cubies PDB stored in memory.

The results obtained for this set of experiments yield the following insights. First they show that dual lookups are effective and using them reduces the search effort by an order of magnitude. Second, these results show that operator pruning is important and using it reduced the search effort by an order of magnitude. Third, it shows that in this domain, the penalty of DIDA* almost offsets the benefits and using DIDA* only improves IDA* by a modest amount.

Korf's original 1997 Rubik's cube experiments on 10 random instances were repeated [14]. Again, since DIDA* does not seem to produce significant improvement in this domain, only IDA* was used with dual PDB lookups for this set of experiments. Korf used three PDBs for this domain: one PDB for the eight corner cubies and two PDBs for two sets of six edge cubies. Since a legal move in this domain moves eight cubies, the only way to combine these three PDBs is by taking their maximum. Note that there are eight corner cubies and all eight 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.[8]

Results for the same set of 10 random instances used in [14] were obtained and are provided in Table 2. 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 six to seven 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. This can be seen in the following rates, which were measured over 10 million random instances. For the $8 + 6r + 6r + 6d + 6d$ setting, the 8-corner PDB had the maximum value for 73.5% of the instances while one of the lookups in the 6-edges cubies was the maximum for only 7.3% of the instances (the rest of the instances were a tie). These numbers changed to 40.8% and 21.3% respectively for the $8 + 7r + 7r + 7d + 7d$ setting.

[7] In fact, we observed that a significant part of the 11-fold improvement (a 2.3-fold improvement) is due to activating BPMX. See [7,21] for a deeper treatment of BPMX.

[8] Since corner cubies can switch locations only with corner cubies and the entire space corner cubies is in the database then the heuristic always returns the optimal cost. The regular state and the dual state share the same optimal distance to the goal state thus both evaluations will return the exact value (which is the real cost).

7.2. Pancake puzzle

Unlike the other puzzles discussed in this paper, the pancake puzzle does not have geometrical symmetries [2]. This is a consequence of the special structure of the problem; each location has different attributes (such as how many operators are applicable to a location and where the location can be permuted to). Therefore, the dual heuristic is important because it provides the only additional possibility for obtaining another heuristic "for free".

Table 3 presents results averaged over 10 random instances of the 17-pancake puzzle. The heuristic used was a PDB based on the rightmost tokens $10, 11, \ldots, 16$ (which gives slightly better average heuristic values than a PDB based on tokens $0, 1, \ldots, 6$). Here again, the phenomenon is observed that the dual heuristic is much better than the regular heuristic and the improvement factor is 23.8 in terms of generated nodes.[9] When taking the maximum of the regular and dual heuristic, an improvement was obtained over the simple case of the regular heuristic: a factor of 138 in nodes generated and a factor of 92 in time.

The last line of the table shows the results when DIDA* was used. DIDA* with the JIL policy produces a roughly 10-fold performance improvement over IDA* (from 2,478 million to 260 million nodes) when using the same heuristic. In this domain there are no obvious redundant operator sequences, so a depth-first search cannot prune any of the operators based on the previous operators. 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 $N - 1$.

Note that, while using the exact same PDB, the total improvement of DIDA* over the simple case is by three orders of magnitude and the time to solve a problem was reduced from more than three days to only six minutes.

Table 4 compare results averaged over 100 random instances of the pancake puzzle for sizes 11 to 15. For the last lines of the 16-pancake and the 17-pancake problems (which demand days of computations) only results over 50 and 10 instances were compared respectively. The heuristic used was a PDB based on the seven rightmost tokens. The first column indicates the size of the pancake puzzle. The second column indicates the average optimal solution cost for each set of random instances. The following columns presents the average number of generated nodes using different heuristics and different search methods. The table shows that the larger the problem space (i.e., the bigger the IDA* search needed), the larger the improvement for the various methods of using duality.

Fig. 15 shows the improvement factor of the different variations over the basic regular lookup (column 3 of Table 4). The figure shows that the improvement factor of DIDA* steadily increases with the size of the problem. For the problem of size 11, it is a factor of 59. An improvement of 1314-fold is seen for a problems of size 17. For the smaller

Table 3
17-pancake puzzle results over 10 random instances

Heuristic	Algorithm	Nodes	Time
r	IDA*	342,308,368,717	284,054
d	IDA*	14,387,002,121	12,485
$\max(r, d)$	IDA*	2,478,269,076	3086
$\max(r, d)$	DIDA* (JIL)	260,506,693	362

Table 4
Pancake puzzle results for different sizes of problems

Size	Ave.	IDA* $h = r$	IDA* $h = d$	IDA* $h = \max(r, d)$	DIDA* (JIL) $h = \max(r, d)$
11	9.83	16,407	867	404	275
12	10.50	148,380	6414	2538	1597
13	11.88	4,268,700	98,605	29,423	15,291
14	12.67	66,213,088	2,143,328	474,082	229,348
15	13.78	864,968,140	38,953,014	6,259,061	2,306,745
16	14.72	18,184,871,249	608,590,928	95,124,495	18,469,496
17	15.60	342,308,368,717	14,387,002,121	2,478,269,076	260,506,693

[9] Since the dual heuristic is inconsistent, BPMX was also used. Again, part of the 23.8-fold improvement is due to activating BPMX.

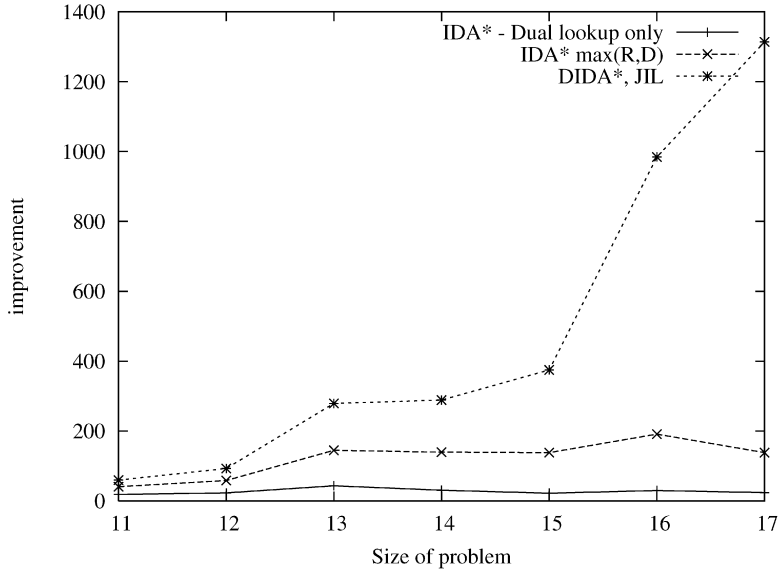


Fig. 15. Improvement on IDA* with regular lookup.

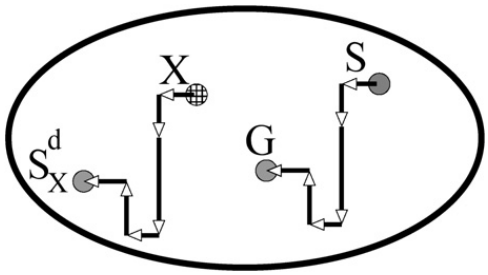


Fig. 16. General duality.

problems, the regular lookup in the 7-token PDB is more accurate and provides tighter bounds on the solution, limiting the opportunities for large performance improvements. When the problem is large the PDB is less accurate, enabling other search and heuristic methods to find large performance improvements. Note that the other variations are also always better than the simple version (regular lookup only) but the improvement of these systems does not seem to increase as dramatically as DIDA*.

8. 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 (assumption 4 in Section 4.1). 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. Assumption 4 is dropped but assumptions 1–3 are still needed, although assumption 1 is relaxed to allow n, the number of locations, to be greater than m, the number of objects. With this general definition, dual heuristic evaluations and dual search are possible for a much wider range of state spaces, including the sliding-tile puzzles, the Blocks World, and the Towers of Hanoi.

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_X^d, the dual of S with respect to X, is defined to be $\pi(X)$. This idea is illustrated in Fig. 16. The same path that transforms S to G also transforms X to S_X^d. As a special case, if

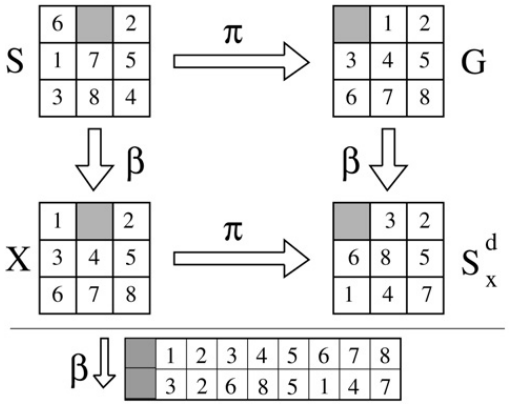


Fig. 17. General duality $S_X^d = \pi(\beta(S)) = \beta(\pi(S))$.

Dual(State S, Goal G, State X)
1 for each location p {
1.1 let o be the object located in p in S
1.2 let y be the location of o in G
1.3 define $\pi_p = y$
1.4 } endfor
2 $S_X^d = \pi(X)$
3 return (S_X^d)

Algorithm 5. Calculation of the general dual state of S with respect to X.

$X = G$ (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 Fig. 17 do not have the same applicable operators. For example, the operator "*move up the tile in the middle*" is applicable to S but not to G. A state X needs to be found 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_X^d can be derived in two ways, either by applying π to X (as shown in Algorithm 5) or by renaming the tiles in G according to β. π (shown in Fig. 17), for example, maps the tile in the upper left location in S, or in X, to the lower left location in G, or S_X^d, respectively. By contrast, β renames object 6 in S, or in G, to object 1 in X, or S_X^d, respectively.

By definition, any legal sequence of operators that produces S_X^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_X^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, $PDB_G[S]$, a heuristic value for the dual state can be used 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 could have more than one dual—it will have a dual with respect to each X_i that has 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.

For the sliding-tile puzzles, we define X_i to be a state in which the blank is in position i, and build a PDB for each X_i. Then, given a state S with the blank in position i, the dual of S with respect to X_i is calculated and its value is looked-up in PDB_{X_i}. For example, in the 8-puzzle there are nine different locations that the blank could occupy. Define nine different states, $X_0 \ldots X_8$, with X_i having the blank in position i, and compute nine 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 four 7-tile PDBs are needed to cover all possible blank locations in the 15-puzzle.

8.1. Other domains

To convey the generality of general duality, we will briefly describe how it applies to two additional domains, the Blocks World, and the Towers of Hanoi.

In the Blocks World, there are B objects (blocks), each of which may be placed on a "table" or on another block.There may be at most one block on a block, so towers are formed of various heights when blocks are stacked upon each other.[10] The size of the state space is approximately $p(B) * (B!)$, where $p(B)$ is the number of ways of partitioning the integer B. The standard Block World operators allow any block to be picked up (by a "hand" that then holds the picked-up block) if it has no block on top of it, and to put the block being held down onto the table or onto any block that has nothing on top of it. Two states can have the same set of operator sequences applied to them only if they have the same "structure", i.e. they partition B the same way (e.g. into two towers, one of height 3, the other of size $B - 3$). In order for each state to have a non-trivial dual, we need one X_i for each different structure – in other words we need $p(B)$ different X_i states. This number grows fairly quickly as B increases but it is not excessively large for the values of B typically used in experiments. For example, for $B = 12$ blocks the number of structures, $p(12)$, is 77. If practical considerations force only some of the 77 X_i PDBs to be computed, duals will exist for the states that have the same structure as one of X_i's for which a PDB was built.

The Towers of Hanoi is the same as the Blocks World except for these differences: (1) there are a limited number of locations on the table (called "pegs"), typically three or four, and each peg has an identity; and (2) each block (called a "disc") has a distinct "size", and a larger disc cannot be placed on top of a smaller one. The latter constraint causes the number of distinct structures to explode exponentially—two states can have the same set of operator sequences applied to them only if they contain exactly the same towers of discs, which means they can differ only in which pegs the various towers are on. General duality applies in this case, but it is of no benefit.

8.2. Dual search for the general case

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.

9. Experimental results for the sliding-tile puzzles (general duality)

General duality has been implemented for the 15-puzzle and 24-puzzle. In this section, results for both using dual heuristics and using the dual search algorithm are given for these domains.

9.1. 15-puzzle

For the 15-puzzle, the same 7–8 PDB partitioning from [15] was used (as shown in Fig. 5). As explained in Section 8, for each possible blank location a unique PDB has to be built to be able to perform a lookup for the dual state and calculate the dual heuristic. However, the number of unique PDBs that must be built can be reduced. 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 eight locations. One region (call it A) has 8 locations which are occupied by eight real tiles, and another region of 8 locations (B) which are occupied by seven real tiles and the blank. The 8-tile group is not affected by the blank since it has exactly 8 location with exactly eight tiles in it and therefore, the regular 8-tile PDB can be used for the dual state.

This is not the case for the 7-tile PDB which is affected by the location of the blank. However, as shown in Fig. 18 there are only four different unique blank locations for the 7–8 partitioning. A unique 7-tile PDB should be built for

[10] Note that there could be B different stacks, each with up to B objects. A location is any possible location in any of these stacks. Of course, many of the locations are empty. But, notionally, the number of distinct locations is $O(B^2)$.

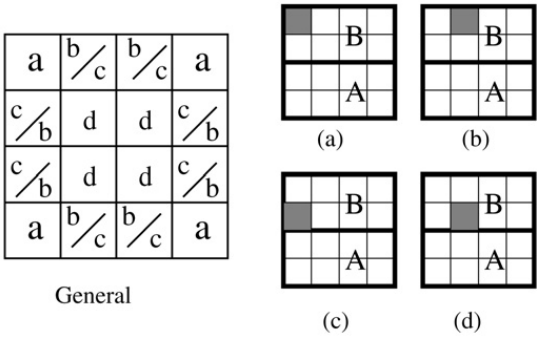


Fig. 18. Four different (dual) 7-tile pattern databases.

Table 5
Results for the 15-puzzle

#	Heuristic	Algorithm	Jump	Av. H	Nodes	Time
			One PDB lookup			
1	r	IDA*	–	44.75	136,289	0.081
2	d	IDA*	–	44.39	247,299	0.139
			Two PDB lookups			
3	$\max(r, r^*)$	IDA*	-	45.63	36,710	0.034
4	$\max(r, d)$	IDA*	-	44.40	65,349	0.069
5	$\max(r, d)$	DIDA*	JIL	44.40	51,633	0.066
6	$\max(r, d)$	DIDA*	J15	44.40	36,937	0.047
			Four PDB lookups			
7	$\max(r, r^*, d, d^*)$	IDA*	-	46.12	18,601	0.022
8	$\max(r, r^*, d, d^*)$	DIDA*	J15	46.12	13,687	0.019

each of these cases. Any state S of the 15-puzzle can be mapped into one of these cases in order to calculate the relevant PDB for its dual state S^d. The frame on the left of Fig. 18 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 519 KB for the 8-tile PDB and 57.5 KB for a 7-tile PDB. Thus the total memory needs (the 8-tile and 4 7-tile PDBs) is 749 KB. The three extra PDBs needed to handle all the dual cases correctly, represent a small increase of memory.

Table 5 presents results of the different heuristics averaged over the same 1000 instances used in [15]. The average solution for this set of instances is 52.52. The first column indicates the heuristic used, with 'r^*' and 'd^*' representing the reflected regular and dual PDB lookups.

Line 1 presents the results when only the regular PDB is used, while line 2 presents the results when only the dual heuristic is used. An interesting phenomenon is that unlike the other domains, in the 15-puzzle the pure dual PDB lookup was worse than the pure regular lookup (it generated almost twice as many nodes). The reason for this is 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 Fig. 18, the dual state might also consult one of the other 7-tile PDBs (labeled b, c and d). The current state S always aims for a region B configuration such that the blank is located in a corner (the goal state) while the dual state S^d 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 Fig. 18 is 20.91, the average values of the PDBs labeled b, c and d are 20.81, 20.31 and 20.53 respectively. Thus, the values obtained by the PDBs that correspond to b, c and d will be smaller than those obtained by the PDB of a.

Historically, the goal location of the blank is in the corner. However, if a goal state is set such that the blank is in location 4 (as in Fig. 18(c)) then the regular heuristic will always look in the weakest PDB while the dual heuristic will consult the other PDBs as well. Such experiments have been made and, indeed, the pure dual PDB lookup generated nearly 40% regular PDB.

Given one PDB lookup, one either performs a lookup on the regular or the dual PDB state. When two lookups are allowed many other combinations are possible. Line 3 of Table 5 used the maximum of the regular and reflected PDBs. Note that lines 1 and 3 are the same results obtained by [15], but on different (faster) hardware. Line 3 presents the best published results for this puzzle [15]. Line 4 uses the maximum of the regular and the dual heuristic. For the same reason as line 2, line 4 was worse than line 3 since it used all four PDBs and not just the best one.

DIDA* experiments included two jumping policies: JIL and J15. J15 works as follows. 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 two to four depending on the location of the blank, and will often be different for S and S^d. Second, as explained above, there will be 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 16 positions the blank could be in, only four PDBs are needed. The average heuristic value for each of these PDBs is different. Note that a small difference in average PDB value can have a dramatic effect on the PDB's pruning power. Because S and S_i^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.

J15, considers both these attributes. It is a three-part 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.[11] Second, if there is a tie, then the quality (average value) of the relevant PDB is considered. Preference is given to the PDB with the higher average. The average values of the four PDBs were given above. Third, if there is still a tie, then the JIL policy is used.

The results for DIDA* with JIL and with J15 are presented in lines 5 and 6. DIDA* with J15 is almost twice as efficient as IDA* using $\max(r, d)$ (line 4). However, as explained earlier the dual heuristic was inferior in this particular domain because the regular heuristic used a PDB with higher average values. Thus, $\max(r, d)$ was almost two times slower than the benchmark results from [15] (line 3). Using DIDA* with J15 can overcome this problem. DIDA* with J15 (line 6) generated roughly the same number of nodes as the benchmark results when using only two PDB lookups.

Finally, the lines 7 and 8 perform all four possible PDB lookups on this domain. This is achieved using both regular and dual lookups and their reflections about the main diagonal. Line 7 presents the maximum over the four PDB combinations. Using all four lookups reduces the number of generated nodes by more than a factor of two and eliminated one third of the execution time compared to the best results of [15] (line 3 of Table 5). The time improvement is smaller because in the new setting four PDB lookups are performed, as opposed to only two PDB lookups for the previous benchmark.

To the best of our knowledge using the four regular/dual normal/reflected PDB lookups gives the best existing heuristic for this puzzle.

It is important to note that the dual lookups for the sliding-tile puzzles are of great importance as there is only one geometrical 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 two over the previous benchmarks.

When performing all four possible lookups with DIDA* and J15, the result is a new state-of-the-art solver. The number of nodes is now reduced to only 13,687 and the average time per problem is now 0.019. 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 [12].

Note from the table that the constant time per node is not significantly increased when moving from IDA* to DIDA*. The reason is that the most time consuming stage of these algorithms is the overhead of the PDB lookups

[11] The reliance of J15 on the branching factor causes a subtle problem. Suppose the start state, S, has the blank in location 5. It will have a branching factor of four but its dual, S_5^d, calculated with respect to X_5, will have a branching factor of 2, because, it will have the blank in the same location as goal state G (the upper lefthand corner). Dual search with J15 will therefore jump to S_5^d and proceed searching from there with X_5 as the goal. The states generated during this search will have branching factors of at most 3, but their duals will all have a branching factor of 4. They will have the blank in the same location as the current search goal, X_5, but without having any history on the other side (because the jump was made at the root state). J15 will therefore never make another jump. To avoid this problem, jumps from states with the blank at interior locations that are within a few moves of the start state are not permitted.

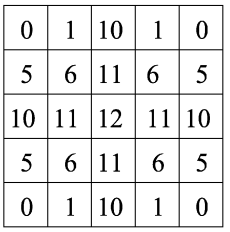
which slow down the calculations because they perform queries into main memory. This overhead depends on the number of PDB lookups and is similar in both IDA* and DIDA*. The only additional overhead in DIDA* is activating the jumping policy which is relatively very small.

9.2. 24-puzzle

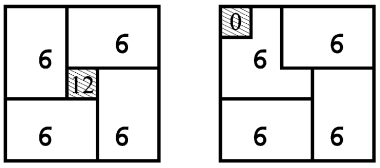
Similar experiments were performed using the 24-puzzle. The original 6–6–6–6 partitioning from [15] (Fig. 5) needed storage for only two 6-tile PDBs since all the 3×2 rectangles are symmetric. As before, additional PDBs are needed to handle the blank. Eight 6-tile PDBs are used: one for all the 3×2 rectangles and their duals, and seven 6-tile PDBs for the irregular shape in the top left corner (see Fig. 5). They are numbered 0, 1, 5, 6, 10, 11, 12 in Fig. 19, with the number reflecting the location of the blank in the X_i that defined the PDB. Fig. 19(a) indicates which PDB is to be used for each possible position of the blank, and Fig. 19(b) shows two different 6–6–6–6 additive PDB partitionings, PDB_{12} and PDB_0 (the PDB used in [15]). Fig. 19(c) shows the average heuristic value for each of the PDBs. Each 6-tile PDB needs 122 MB and the new system needs eight times as much memory. When using DIDA*, the JIL and J24 heuristics were used (J24 works exactly the same way as J15 but for the 24-puzzle).

In [15] 50 random instances were optimally solved. This data set was sorted in increasing order of their optimal solutions. Table 6 presents the average results over the first 25 random instances for all the different variations. Table A.3 in Appendix A gives further results for the entire set of 50 instance for the best variations of DIDA*. The first line presents the benchmark results from [15] 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. Line 3, is DIDA* with JIL. Finally, line 4 shows that DIDA* with J24 outperforms the benchmark results by a factor of 5.3. Detailed results for each of the 50 instances from [15] with variations on the PDB lookups is provided in Appendix A.

The last two lines (5 and 6) present the case where all possible four PDB lookups were used. IDA* with all four lookups improved the benchmark results by a factor of 3.2. DIDA* with all four lookups further improved this to a total of improvement factor of 11.0 over the benchmark. Furthermore, note that DIDA with two lookups (r, d) outperform IDA* with the entire set of four lookups by a factor of 1.65.



(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

Fig. 19. 24-puzzle heuristic.

Table 6
DIDA* results on the 24-puzzle on the first 25 random instances

#	Heuristic	Search	Policy	Nodes	Jumps
			Two PDB lookups		
1	$\max(r, r^*)$	IDA*	–	43,454,810,045	–
2	$\max(r, d)$	IDA*	–	31,103,112,895	–
3	$\max(r, d)$	DIDA*	JIL	16,302,942,680	176,075,343
4	$\max(r, d)$	DIDA*	J24	8,248,769,713	23,851,828
			Four PDB lookups		
5	$\max(r, r^*, d, d^*)$	IDA*	–	13,549,943,868	–
6	$\max(r, r^*, d, d^*)$	DIDA*	J24	3,948,614,947	13,083,286

The versions with two PDB lookups ran at around 300,000 nodes per second while the versions with 4 PDB lookups ran at around 220,000 nodes per second. Thus, as observed in the other domains the improvement in the total running time was a little smaller.

10. Conclusions and future work

Duality is a new form of symmetry between states in permutation state spaces and it allows the usage of multiple heuristics for a given state. DIDA* is a novel search algorithm which exploits this symmetry. 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. Using the dual heuristic significantly improves the heuristic value. Adding the dual search algorithm provides additional performance gains (up to an order of magnitude) in several application domains using a state-of-the-art heuristic search algorithm.

Future work can continue in the following directions:

Table A.1
DIDA* results on the 24-puzzle on the 50 random instances

#	Heuristic	Search	Policy	Nodes	Jumps
			Two PDB lookups		
1	$\max(r, r^*)$	IDA*	–	360,892,479,670	–
3	$\max(r, d)$	DIDA*	J24	75,201,250,617	147,733,547
			Four PDB lookups		
5	$\max(r, r^*, d, d^*)$	DIDA*	J24	37,674,826,649	78,134,424

Table A.2
24-puzzle first 25 instances. DIDA* uses the J24 jumping policy

No	Sol	Benchmark (r, r^*)	IDA* (r, d)	DIDA*-J24 (r, d)	DIDA*-J24 (r, r^*, d, d^*)
1 (25)	81	292,174,444	547,754,446	152,941,190	86,623,738
2 (40)	82	65,099,578	78,265,289	13,720,424	8,027,134
3 (29)	88	4,787,505,637	29,093,280,876	2,811,214,623	1,052,360,568
4 (36)	90	2,582,008,940	3,128,723,824	1,209,506,402	569,488,356
5 (20)	92	312,016,177,684	50,287,497,984	29,036,511,649	4,464,625,873
6 (30)	92	1,634,941,420	1,950,647,389	2,484,991,641	1,225,111,000
7 (47)	92	30,443,173,162	18,915,533,169	6,421,296,555	4,151,834,900
8 (44)	93	867,106,238	373,833,955	37,432,750	27,828,310
9 (1)	95	2,031,102,635	815,220,874	114,270,740	60,208,978
10 (22)	95	3,592,980,531	4,006,328,755	1,762,446,935	814,538,591
11 (2)	96	211,884,984,525	149,827,435,325	22,818,488,960	14,893,883,061
12 (16)	96	3,803,445,934	1,829,307,204	174,895,012	125,947,856
13 (38)	96	38,173,507	45,976,055	8,435,471	5,298,259
14 (3)	97	21,148,144,928	49,550,582,547	29,121,290,662	9,394,905,290
15 (32)	97	428,222,507	1,707,750,974	1,685,362,622	623,772,078
16 (4)	98	10,991,471,966	14,523,612,651	2,480,394,914	1,500,144,838
17 (28)	98	2,258,006,870	2,106,454,886	543,149,059	360,755,098
18 (35)	98	116,131,234,743	74,794,747,604	6,553,916,243	3,763,906,855
19 (27)	99	53,444,360,033	34,150,080,391	3,447,475,095	1,999,173,109
20 (31)	99	26,200,330,686	19,627,677,414	20,768,799,210	14,456,575,816
21 (5)	100	2,899,007,625	3,785,640,311	484,549,876	178,355,244
22 (37)	100	1,496,759,944	1,471,627,382	1,563,432,726	1,014,271,808
23 (46)	100	65,675,717,510	57,066,411,687	32,733,564,072	18,191,427,181
24 (49)	100	108,197,305,702	56,056,805,705	9,106,414,744	5,362,475,537
25 (6)	101	103,460,814,368	201,836,625,690	30,684,741,238	14,383,834,203
Average	**95**	**43,454,810,045**	**31,103,112,895**	**8,248,769,713**	**3,948,614,947**

- Obtaining a better understanding of the jumping policies. Given an application, how does one go about determining the best policy?
- Analysis to see if the duality concept can be generalized from permutation state spaces to encompass a wider set of application domains and perhaps other forms of permutation problems or even general search problems.
- Integrating the idea of duality into other search algorithms (e.g., A* [8], RBFS [13], breadth-first heuristic search [23]). Initial results with Dual-A* on the 15-pancake puzzle reduce search time by roughly 20%.

Acknowledgements

This research was supported by the Israel Science Foundation (ISF) under grant number 728/06 to Ariel Felner, by the Natural Sciences and Engineering Research Council of Canada (NSERC), and Alberta's Informatics Circle of Research Excellence (iCORE).

Appendix A. Experimental results of the 24-puzzle

In this section, results for the entire set 50 random instances of the 24-puzzle are given. They are compared to the results in [15]—IDA* with $\max(r, r^*)$—and are referred to as the "Benchmark" in the following tables. Table A.1 summarizes the results on the entire set of 50 instances. The best version outperforms the benchmark by an order of magnitude.

The 50 instances have been sorted by increasing order of length of the optimal solution. In Tables A.2 and A.3 the instances are given according to this order. The number in the parenthesizes is the instance number given in [15]. The *Sol* column gives the length of the optimal solution path. The next three columns provide the number of generated nodes for the four different algorithms. The *Benchmark* column corresponds to the $r + r^*$ system from [15]. The next column, IDA*(r, d), uses IDA* but takes the maximum between the regular and dual heuristic. Finally the last two columns present results obtained by DIDA* with the J24 jumping policy and using two and four PDB lookups. For all

Table A.3
24-puzzle the rest of the 50 instances

No	Sol	Benchmark ($r + r^*$)	DIDA*-J24 ($r + d$)	DIDA*-J24 ($r + r^* + d + d^*$)
26 (13)	101	1,959,833,487	2,196,890,327	1,525,086,336
27 (45)	101	79,148,491,306	16,455,892,507	8,903,606,545
28 (34)	102	481,039,271,661	59,384,485,258	33,346,319,761
29 (15)	103	173,999,717,809	104,581,763,680	48,205,749,584
30 (21)	103	724,024,589,335	47,574,279,914	23,105,133,315
31 (7)	104	106,321,592,792	112,115,069,816	41,489,057,096
32 (23)	104	171,498,441,076	25,019,468,325	15,308,110,752
33 (39)	104	161,211,472,633	34,094,740,377	21,449,225,377
34 (43)	104	55,147,320,204	19,521,199,995	12,031,249,938
35 (26)	105	12,397,787,391	4,710,801,259	2,293,128,380
36 (11)	106	1,654,042,891,186	223,800,028,896	81,918,451,417
37 (19)	106	218,284,544,233	71,328,672,853	29,200,386,532
38 (33)	106	1,062,250,612,558	697,848,426,065	410,610,357,344
39 (41)	106	26,998,190,480	1,831,465,730	866,811,661
40 (24)	107	357,290,691,483	58,794,690,620	22,991,124,359
41 (48)	107	555,085,543,507	102,741,654,326	45,159,149,715
42 (8)	108	116,202,273,788	46,087,884,506	32,266,488,302
43 (42)	108	245,852,754,920	70,605,794,609	32,982,573,378
44 (12)	109	624,413,663,951	33,355,872,842	14,264,946,735
45 (17)	109	367,150,048,758	64,989,490,579	33,540,174,776
46 (18)	110	987,725,030,433	560,055,473,298	314,071,218,585
47 (14)	111	1,283,051,362,385	531,467,600,978	220,384,669,296
48 (9)	113	1,818,005,616,606	36,575,158,063	21,812,286,743
49 (50)	113	4,156,099,168,506	285,616,821,863	180,090,018,836
50 (10)	114	1,519,052,821,943	343,089,661,403	137,210,634,028
Average	**107**	**678,330,149,297**	**142,153,731,524**	**71,401,038,351**

instances one can see the improved performance of the new methods. Table A.3 further gives results for DIDA* with J24 and two and four PDB lookups for the rest of the 50 cases.

References

[1] J.C. Culberson, J. Schaeffer, Pattern databases, Computational Intelligence 14 (3) (1998) 318–334.
[2] H. Dweighter, Problem e2569, American Mathematical Monthly 82 (1975) 1010.
[3] S. Edelkamp, Planning with pattern databases, in: Proceedings of the 6th European Conference on Planning (ECP-01), 2001, pp. 13–34.
[4] A. Felner, Solving the graph-partitioning problem with heuristic search, Annals of Mathematics and Artificial Intelligence 67 (2006) 19–39.
[5] A. Felner, R.E. Korf, S. Hanan, Additive pattern database heuristics, Journal of Artificial Intelligence Research (JAIR) 22 (2004) 279–318.
[6] A. Felner, R. Meshulam, R.C. Holte, R.E. Korf, Compressing pattern databases, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-04), July 2004, pp. 638–643.
[7] A. Felner, U. Zahavi, R.C. Holte, J. Schaeffer, Dual lookups in pattern databases, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-05), 2005, pp. 103–108.
[8] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems Science and Cybernetics SCC-4 (2) (1968) 100–107.
[9] B. Hnich, B. Smith, T. Walsh, Dual modelling of permutation and injection problems, Journal of Artificial Intelligence Research 21 (2004) 357–391.
[10] R.C. Holte, J. Newton, A. Felner, R. Meshulam, D. Furcy, Multiple pattern databases, in: ICAPS, 2004, pp. 122–131.
[11] R.C. Holte, M.B. Perez, R.M. Zimmer, A.J. MacDonald, Hierarchical A*: Searching abstraction hierarchies efficiently, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-96), 1996, pp. 530–535.
[12] R.E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, Artificial Intelligence 27 (1985) 97–109.
[13] R.E. Korf, Linear-space best-first search, Artificial Intelligence 62 (1) (1993) 41–78.
[14] R.E. Korf, Finding optimal solutions to Rubik's Cube using pattern databases, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-97), 1997, pp. 700–705.
[15] R.E. Korf, A. Felner, Disjoint pattern database heuristics, Artificial Intelligence 134 (2002) 9–22.
[16] R.E. Korf, M. Reid, S. Edelkamp, Time complexity of Iterative-Deepening-A*, Artificial Intelligence 129 (1–2) (2001) 199–218.
[17] R.E. Korf, L. Taylor, Finding optimal solutions to the twenty-four puzzle, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-96), 1996, pp. 1202–1207.
[18] M. McNaughton, P. Lu, J. Schaeffer, D. Szafron, Memory efficient A* heuristics for multiple sequence alignment, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-02), 2002, pp. 737–743.
[19] D. Ratner, M. Warmuth, Finding a shortest solution for the $n \times n$ extension of the 15-puzzle is intractable, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-86), Philadelphia, PA, 1986, pp. 168–172.
[20] U. Zahavi, A. Felner, R.C. Holte, J. Schaeffer, Dual search in permutation state spaces, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-06), 2006, pp. 1076–1081.
[21] U. Zahavi, A. Felner, J. Schaeffer, N. Sturtevant, Inconsistent heuristics, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-07), 2007, pp. 1211–1216.
[22] R. Zhou, E. Hansen, Space-efficient memory-based heuristics, in: Proceedings of the National Conference on Artificial Intelligence (AAAI-07), 2004, pp. 677–682.
[23] R. Zhou, E. Hansen, Breadth-first heuristic search, Artificial Intelligence 170 (4–5) (2006) 385–408.