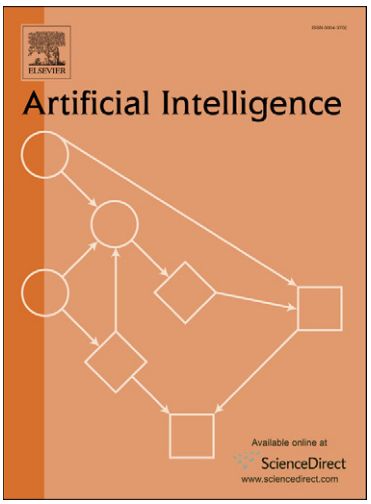


# Inconsistent heuristics in theory and practice

Ariel Felner, Uzi Zahavi, Robert Holte, Jonathan Schaeffer, Nathan Sturtevant, Zhifu Zhang

PII: S0004-3702(11)00022-1
DOI: 10.1016/j.artint.2011.02.001
Reference: ARTINT 2581

To appear in: *Artificial Intelligence*

Received date: 26 July 2010
Revised date: 10 February 2011
Accepted date: 10 February 2011

Please cite this article in press as: A. Felner et al., Inconsistent heuristics in theory and practice, Artificial Intelligence (2011), doi:10.1016/j.artint.2011.02.001

# Inconsistent Heuristics in Theory and Practice

**Ariel Felner** FELNER@BGU.AC.IL
*Department of Information Systems Engineering,*
*Ben-Gurion University of the Negev*
*Beer-Sheva, Israel, 85104*
**Uzi Zahavi** ZAHAVIU@BIU.AC.IL
*Department of Computer Science*
*Bar-Ilan University*
*Ramat-Gan, Israel, 52900*
**Robert Holte**
**Jonathan Schaeffer**
**Nathan Sturtevant**
**Zhifu Zhang** {HOLTE,JONATHAN,NATHANST,ZHANG}@CS.UALBERTA.CA
*Department of Computing Science*
*University of Alberta*
*Edmonton, Alberta, Canada T6G 2E8*

## Abstract

In the field of heuristic search it is usually assumed that admissible heuristics are consistent, implying that consistency is a desirable attribute. The term "inconsistent heuristic" has, at times, been portrayed negatively, as something to be avoided. Part of this is historical: early research discovered that inconsistency can lead to poor performance for A* (nodes might be re-expanded many times). However, the issue has never been fully investigated, and was not re-considered after the invention of IDA*.

This paper shows that many of the preconceived notions about inconsistent heuristics are outdated. The worst-case exponential time of inconsistent heuristics is shown to only occur on contrived graphs with edge weights that are exponential in the size of the graph. Furthermore, the paper shows that rather than being something to be avoided, inconsistent heuristics often add a diversity of heuristic values into a search which can lead to a reduction in the number of node expansions. Inconsistent heuristics are easy to create, contrary to the common perception in the AI literature. To demonstrate this, a number of methods for achieving effective inconsistent heuristics are presented.

Pathmax is a way of propagating inconsistent heuristic values in the search from parent to children. This technique is generalized into bidirectional pathmax (BPMX) which propagates values from a parent to a child node, and vice versa. BPMX can be integrated into IDA* and A*. When inconsistent heuristics are used with BPMX, experimental results show a large reduction in the search effort required by IDA*. Positive results are also presented for A* searches.

**Keywords:** Heuristic search, admissible heuristics, inconsistent heuristics, A*, IDA*

## 1. Introduction and overview

Heuristic search algorithms such as A* [15] and IDA* [22] are guided by the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the current path from the start node to node $n$ and $h(n)$ is a

heuristic function estimating the cost from $n$ to a goal node. If $h(n)$ is *admissible* (i.e., is always a lower bound) these algorithms are guaranteed to find optimal paths.

The A* algorithm is guaranteed to return an optimal solution only if an admissible heuristic is used. There is no requirement that the heuristic be *consistent*.[1] It is usually assumed that admissible heuristics are consistent. In their popular AI textbook *Artificial Intelligence: A Modern Approach*, Russell and Norvig write that "one has to work quite hard to concoct heuristics that are admissible but not consistent" [38]. Many researchers work under the assumption that "almost all admissible heuristics are consistent" [25]. Some algorithms require that the heuristic be consistent (such as Frontier A* [30], which searches without the closed list).[2] The term "inconsistent heuristic" has, at times, been portrayed negatively, as something that should be avoided. Part of this is historical: early research discovered that inconsistency can lead to poor performance for A*. However, the issue of inconsistent heuristics has never been fully investigated or re-considered after the invention of IDA*. This paper argues that these perceptions about inconsistent heuristics are wrong. We show that inconsistent heuristics have many benefits. Further, they can be used in practice for many search domains. We observe that many recently developed heuristics are inconsistent.

A known problem with inconsistent heuristics is that they may cause algorithms like A* to find shorter paths to nodes that were previously expanded and inserted into the closed list. If this happens, then these nodes must be moved back to the open list, where they might be chosen for expansion again. This phenomenon is known as *node re-expansion*. A* with an inconsistent heuristic may perform an exponential number of node re-expansions [32]. We present insights into this phenomenon, showing that the exponential time behavior only appears in contrived graphs where edge weights and heuristic values grow exponentially with the graph size. For IDA*, it is important to note that node re-expansion is inevitable due to the algorithm's depth-first search. The use of an inconsistent heuristic does not exacerbate this. Because no history of previous searches is maintained, each separate path to the node will be examined by IDA* whether the heuristic is consistent or not.

Inconsistent heuristics often add a diversity of heuristic values into a search. We show that these values can be used to escape heuristic depressions (regions of the search space with low heuristic values), and can lead to a large reduction in the search effort. Part of this is achieved by our generalization of *pathmax* into *bidirectional pathmax*. The idea of *pathmax* was introduced by Mero [34] as a method for propagating inconsistent values in the search from a parent node to its children. Pathmax causes the $f$-values of nodes to be *monotonic non-decreasing* along any path in the search tree. The pathmax idea for undirected state spaces is generalized into *bidirectional pathmax* (BPMX). BPMX propagates values in a similar manner to pathmax, but does this in both directions (parent to child, and child to parent). BPMX turns out to be more effective than pathmax in practice. It can easily be integrated into IDA* and, with slightly more effort, into A*. Using BPMX, the propagation of inconsistent values allows a search to escape from heuristic depressions more quickly.

Trivially, one can create an inconsistent heuristics by taking a consistent heuristic and degrading some of its values. The resulting heuristic will be less informed. Contrary to the perception in the

1. A heuristic is *consistent* if for every two states $x$ and $y$, $h(x) \leq c(x, y) + h(y)$ where $c(x, y)$ is the cost of the shortest path between $x$ and $y$. Derivations and definitions of *consistent* and *inconsistent* heuristics are provided in Section 3.
2. The breadth-first heuristic search algorithm [49], a competitor to Frontier A*, does not have this requirement and works with inconsistent heuristics too.

literature, informed inconsistent heuristics are easy to create. General guidelines as well as a number of simple methods for creating effective inconsistent heuristics are provided. The characteristics of inconsistent heuristics are analyzed to provide insights into how to effectively use them to further reduce the search effort.

Finally, experimental results show that using inconsistent heuristics with BPMX yields a significant reduction in the search effort required for many IDA*- and A*-based search applications. The application domains used are the sliding-tile puzzle, Pancake problem, Rubik's cube, TopSpin and pathfinding in maps.

The paper is organized as follows. In Section 2 we provide background material. Section 3 defines consistent and inconsistent heuristics. Section 4 presents a study of the behavior of A* with inconsistent heuristics. BPMX is introduced in Section 5 and its attributes when used with inconsistent heuristics are studied. Methods for creating inconsistent heuristics are discussed in Section 6. Extensive experimental results for IDA* and for A* are provided in Sections 7 and 8, respectively. Finally we provide our conclusions in Section 9.

Portions of this work have been previously published [14, 21, 44, 45, 46, 47]. This paper summarizes this line of work and ties together all the results. In addition new experimental results are provided.

## 2. Terminology and background

This section presents terminology and background material used for this research.

### 2.1 Terminology

Throughout the paper the following terminology is used. A *state space* is a graph whose vertices are called *states*. The execution of a search algorithm (e.g., A* and IDA*) from an initial state creates a *search graph*. A *search tree* spans that graph according to the progress of the search algorithm. The term *node* is used throughout this paper to refer to the nodes of the search tree. Each node in the search tree corresponds to some state in the state space. The search tree may contain nodes that correspond to the same state (via different paths). These are called *duplicates*.

The fundamental operation in a search algorithm is to *expand a node* (*i.e.,* to compute or *generate* the node's successors in the search tree). We assume that each node expansion takes the same amount of time. This allows us to measure the time complexity of the algorithms in terms of the total number of node expansions performed by the algorithm in solving a given problem.[3] The space complexity of a search algorithm is measured in terms of the number of nodes that need to be stored simultaneously.

A second measure of interest is the number of unique states that are expanded at least once during the search. The phrase *number of distinct expanded states* refers to this measure and is denoted by $N$.

The term $c(x, y)$ is used to denote the cost of a shortest path from $x$ to $y$. In addition, $h(x)$ denotes an admissible heuristic from $x$ to a goal while $h^*(x)$ denotes the cost of the shortest path from $x$ to a goal ( $= c(x, goal)$).

3. In experiments with IDA*, it is common to report the number of generated nodes instead of the number of node expansions. We follow this practice in our IDA* experiments.

### 2.2 Search algorithms

The A* algorithm is a best-first search algorithm [15]. It keeps an *open list* of nodes (denoted hereafter as *OPEN*), usually implemented as a priority queue, which is initialized with the start state node. At each expansion step of the algorithm, a node of minimal cost is extracted from OPEN and its children are generated and added to OPEN. The expanded node is inserted into the *closed list* (denoted hereafter as *CLOSED*). The algorithm halts when a goal node is chosen for expansion.

A* employs a *duplicate detection* mechanism and stores at most one node for any given state. Before a node is added to OPEN it is first matched against both OPEN and CLOSED. If a duplicate node (node with the same state) is found in OPEN then only the node with the smaller $g$-value is kept in OPEN. If the duplicate node is found in CLOSED with a smaller or equal $g$-value, the newly generated node is ignored. If the node is found in CLOSED with a larger $g$-value, the copy in CLOSED is removed and the copy with the smaller $g$-value is added to OPEN.

A* uses the cost function $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of reaching node $n$ from the start node (via the best *known* path) and $h(n)$ is an estimate of the remaining distance from $n$ to the goal. If $h(n)$ is *admissible* (*i.e.,* its estimate is always a lower bound on the actual distance) then A* is guaranteed to return a shortest path solution if one exists [6]. Furthermore, with a consistent heuristic, A* has been proven to be admissible, complete, and optimally effective [6]. With an inconsistent heuristic, A* is optimal with respect to the number of distinct states expanded, $N$, but may re-expand nodes many times. A* requires memory linear in the number of distinct states expanded.

IDA* is an iterative-deepening version of A* [22]. It performs a series of depth-first searches, each to an increasing solution-cost threshold $T$. $T$ is initially set to $h(s)$, where $s$ is the start node. If the goal is found within the threshold, the search ends successfully. Otherwise, IDA* proceeds to the next iteration by increasing $T$ to the minimum $f$-value that exceeded $T$ in the previous iteration. The worst-case time complexity of IDA*, even when the given heuristic is consistent, is $O(N^2)$ on trees, $O(2^{2N})$ on directed acyclic graphs [31], and $\Omega(N!)$ on cyclic or undirected graphs. The space complexity of IDA* is $O(bd)$ where $b$ is the maximum branching factor and $d$ is the maximum depth of the search (number of edges traversed from the root to the goal). Despite these worst-case time bounds, in practice, IDA* is effectively used to solve many combinatorial problems, especially ones whose state spaces do not have many small cycles. Due to its modest space complexity, IDA* can solve problems for which A* exhausts available memory before arriving at a solution.

### 2.3 Applications

We now provide an overview of the application domains used in this paper.

#### 2.3.1 Rubik's cube

Rubik's cube was invented in 1974 by Ernő Rubik of Hungary. The standard version consists of a $3 \times 3 \times 3$ cube (Figure 1), with different colored stickers on each of the exposed squares of the sub-cubes, or *cubies*. There are 20 movable cubies and six 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. There are about $4 \times 10^{19}$ different reachable states. In the goal state, all the squares on each side of the cube are the same color. Pruning redundant moves results

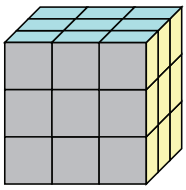

Figure 1: $3 \times 3 \times 3$ Rubik's cube

in a search tree with an asymptotic branching factor of about 13.34847 [24].[4] Pattern databases (PDBs, see below) are an effective and commonly-used heuristic this domain.

#### 2.3.2 TopSpin puzzle

The ($n$,$r$)-TopSpin puzzle has $n$ tokens arranged in a ring (see Figure 2). The ring of tokens can be shifted cyclically clockwise or counterclockwise. The tokens pass through the *reverse circle* which is fixed in the top of the ring. At any given time $r$ tokens are located inside the reverse circle. These tokens can be reversed (rotated 180 degrees). The task is to rearrange the puzzle such that the tokens are sorted in increasing order. The (20,4) version of the puzzle is shown in Figure 2 in its goal position where tokens 19, 20, 1 and 2 are in the reverse circle and can be reversed. We used the classic encoding of this puzzle which has $N$ operators, one for each clockwise circular shift of length $0 \ldots N-1$ of the entire ring followed by a reversal/rotation for the tokens in the reverse circle [4]. Each operator has a cost of one. Note that there are $n!$ different ways to permute the tokens. However, since the puzzle is cyclic, only the relative location of the different tokens matters, and thus there are only $(n-1)!$ unique states. PDBs are an effective heuristic for this puzzle.

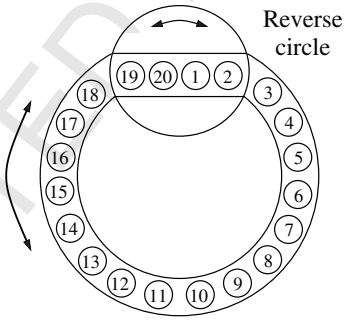


Figure 2: TopSpin (20,4) puzzle

#### 2.3.3 The Sliding-tile puzzles

One of the classic examples of a single-agent path-finding problem in the AI literature is the sliding-tile puzzle. Three common versions of this puzzle are the $3 \times 3$ 8-puzzle, the $4 \times 4$ 15-puzzle and the $5 \times 5$ 24-puzzle. They consist of a square frame containing a set of numbered square tiles,

4. We adopt the same setting first used by Korf [24] where both 90-degree and 180-degree rotation of a face count as a legal move.

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's position. The objective is to rearrange the tiles from some random initial solvable 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 that finding optimal solutions to the sliding-tile puzzle is NP-complete [37]. The 8-puzzle contains $9!/2$ (181,440) reachable states, 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 Figure 3.

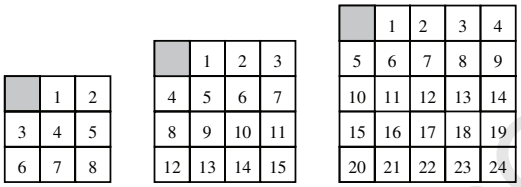


Figure 3: The 8-, 15- and 24-puzzle goal states

The classic admissible 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. PDBs provide the best existing admissible heuristics for this problem.

#### 2.3.4 THE PANCAKE PUZZLE

The pancake puzzle is inspired by a waiter navigating a busy restaurant with a stack of $n$ pancakes [8]. The waiter wants to sort the pancakes ordered by size, to deliver the pancakes in a pleasing visual presentation. 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 $0...(n-1)$. A state has $n-1$ successors, with the $k^{th}$ successor formed by reversing the order of the first $k+1$ elements of the permutation ($0 < k \leq n-1$). For example, if $n = 5$ the successors of the goal state $< 0, 1, 2, 3, 4 >$ are $< 1, 0, 2, 3, 4 >$, $< 2, 1, 0, 3, 4 >$, $< 3, 2, 1, 0, 4 >$ and $< 4, 3, 2, 1, 0 >$, as shown in Figure 4. 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 it has a constant branching factor of $n-1$. PDBs are a well-informed and commonly-used heuristic for this domain.[5]

#### 2.3.5 PATHFINDING

A map is an $m \times n$ grid of passable areas and obstacles. There are eight possible movements from a position—four cardinal moves and four diagonal moves—subject to obstacles and boundary conditions. Cardinal moves have cost 1, and diagonal moves have cost $\sqrt{2}$. Figure 5 shows one of the maps used in our experiments (a $512 \times 512$ grid). The goal is, for instance, to move from point $A$ to point $B$ in the fewest number of moves, traversing only the light area. In general, for this application the best heuristic depends on properties of the domain.

5. The best heuristic known for this puzzle is called the *gap heuristic*[16] and uses domain-dependent attributes.

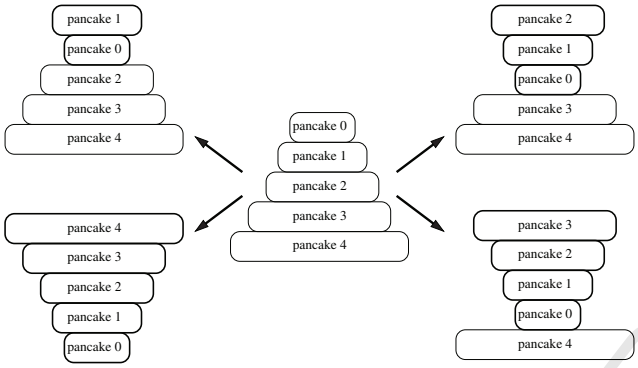


Figure 4: The 5-pancake puzzle

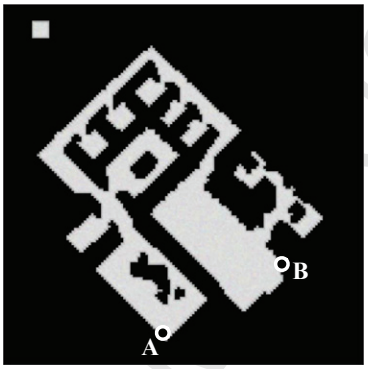


Figure 5: Sample game map

### 2.4 Pattern database heuristics

The efficiency of a single-agent search algorithm is largely dictated by the quality of the heuristic used. An effective and commonly-used heuristic for most of the application domains used in this paper are memory- or table-based heuristics. The largest body of work on these heuristics is on *pattern databases* [5] (PDBs). PDBs are therefore used in most of our experimental studies, and the purpose of this section is to give the background details. However, it is important to note that none of this paper's key ideas (inconsistency, BPMX, etc.) depend on the heuristic being a PDB; these ideas apply to heuristics of all forms. PDB heuristics allow us to achieve state-of-the-art performance for some of our application domains.

PDBs are built as follows.[6] The *state space* of a permutation problem represents all the different ways of placing a given set of objects into a given set of locations (i.e., all the possible states). A *subproblem* is an abstraction of the original problem defined by only considering some of these objects while treating the others as irrelevant ("don't care"). A *pattern* (abstract state) is a specific

6. We give a definition of PDBs which is specific to permutation state spaces, since these are used in this paper. However, PDBs can be built for a much wider set of state spaces and abstractions (e.g., planning domains [9] or other combinatorial problems [10, 33]).

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.

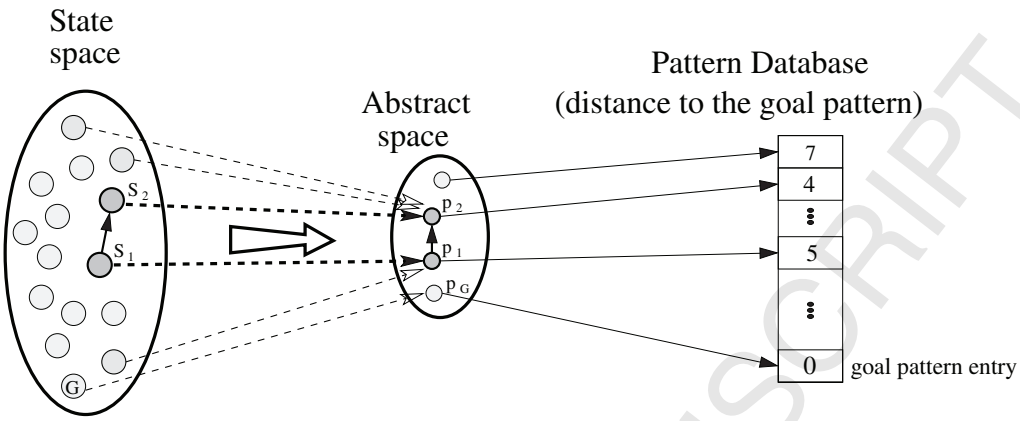


Figure 6: States in the state space are mapped to patterns in the abstract space

Each state in the original state space is *abstracted* to a state in the pattern space by only considering the pattern objects, and ignoring the others. The *goal pattern* is the abstraction of the goal state. As illustrated in Figure 6, 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 in 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[7] backwards from the goal pattern until the entire pattern space is spanned. A state $s$ in the original space is mapped to a pattern $p$ by ignoring all details in the state description that are not preserved in the pattern. The value stored in the PDB for $p$ 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 to be a powerful technique for for finding effective lower bounds for numerous combinatorial puzzle domains [24, 5, 26, 10, 11]. Furthermore, they have also proved to be useful for other search problems (e.g., multiple sequence alignment [33, 48] and planning [9]).

### 2.4.1 Pattern database example

PDBs can be built for the sliding-tile puzzles as illustrated in Figure 7. Assume that the subproblem is defined to only include tiles 2, 3, 6 and 7; all the tiles are ignored except for 2, 3, 6 and 7. 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 Figure 7(d)). Figure 7(b,d) depicts the PDB lookup for estimating a distance from a given state S (Figure 7(a)) to the goal (Figure 7(c)). State S is mapped to a 2-3-6-7 pattern by ignoring all the tiles other than 2, 3, 6 and 7 (Figure 7(b)). Then, this pattern's distance

7. This description assumes all operators have the same cost. Uniform cost search should be used in cases where operators have different costs.

to the goal pattern (Figure 7(d)) 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, then the lookup for state S is *PDB*[8][12][13][14] (tile 2 is in location 8, tile 3 in location 12, etc.).

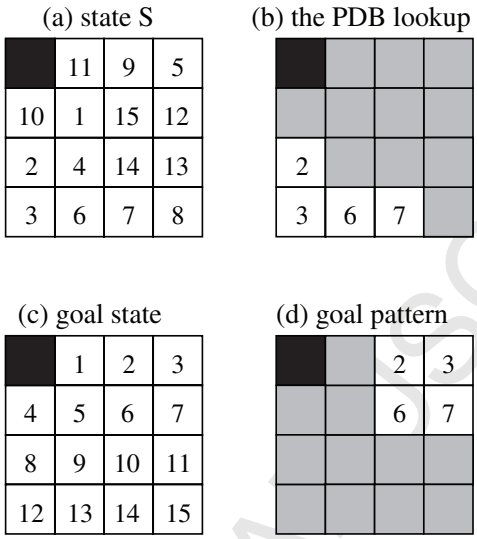


Figure 7: Example of regular PDB lookups

As another example, consider only the eight cubies of the yellow face in Rubik's cube. A "yellow face" PDB will store the distances for all configurations of the "yellow" cubies to their goal location. These distances are admissible heuristics for the complete set of cubies.

### 2.4.2 Additive PDBs

The best existing method for optimally solving the sliding-tile puzzles uses *disjoint additive pattern databases* [10, 26]. The tiles are partitioned into disjoint sets, and a PDB is built for each set. An $x - y - z$ partitioning is a partition of the tiles into disjoint sets with cardinalities $x$, $y$ and $z$. We build a PDB for each set which stores the cost of moving the tiles in the pattern set from any given arrangement to their goal positions. For each such PDB, moves of tiles in the other sets are not counted. The important attribute is that each move of the puzzle changes the location of *one* tile only. Since for each set of pattern tiles we only count moves of the pattern tiles, and each move only moves one tile, values from different disjoint PDBs can be added together and the results are still admissible. Figure 8 presents the two $7-8$ partitionings for the 15-puzzle and the two $6-6-6-6$ partitionings for the 24-puzzle that were first used in the context of additive PDBs [10, 26].

## 3. Consistent and inconsistent heuristics

Admissibility is a desirable property for a heuristic since it guarantees that the solution returned by A* and IDA* will be optimal. Another attribute for a heuristic is that it can be *consistent*. An admissible heuristic $h$ is consistent if, for every two states $x$ and $y$, if there is a path from $x$ to $y$,

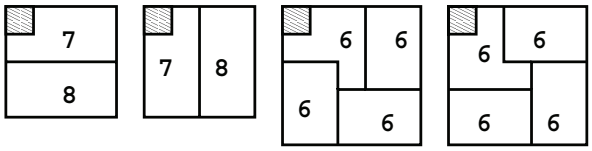


Figure 8: Partitionings and reflections of the tile puzzles

then

$$h(x) \leq c(x, y) + h(y) \tag{1}$$

where $c(x, y)$ is the cost of the least-cost path from $x$ to $y$ [15]. This is a kind of triangle inequality: the estimated distance to the goal from $x$ cannot be reduced by moving to a different state $y$ and adding the estimate of the distance to the goal from $y$ to the cost of reaching $y$ from $x$. Pearl [36] showed that restricting $y$ to be a neighbor of $x$ produces an equivalent definition with an intuitive interpretation: in moving from a state to its neighbor, $h$ must not decrease more than the cost of the edge that connects them. This means that the *cost function* $f(n) = g(n) + h(n)$ is always *non-decreasing* along any given path in the search graph. We call this the *monotonicity*[8] of the cost function $f$, which is guaranteed when $h$ is consistent. Note that consistency is a property of the heuristic $h$ while monotonicity is a property of the cost function $f(n) = g(n) + h(n)$. In Section 5 we will show different methods for enforcing monotonicity and consistency.

If the graph is undirected then the cost of going from $x$ to $y$ is the same as from $y$ to $x$. Since the heuristic is consistent we also get that

$$h(y) \leq c(y, x) + h(x). \tag{2}$$

Merging equations 1 and 2 yields an alternative definition for consistent heuristics for undirected state spaces:

$$|h(x) - h(y)| \leq c(x, y). \tag{3}$$

This inequality means that when moving from a parent to a child in a search tree, the heuristic $h$ cannot increase or decrease more than the change in $g$.

### 3.1 Inconsistent heuristics

An admissible heuristic $h$ is *inconsistent* if for at least one pair of states $x$ and $y$,

$$h(x) > c(x, y) + h(y). \tag{4}$$

If $y$ is a successor of $x$, the $f$-value will decrease when moving from $x$ to $y$. The cost function $f$ in this case is referred to as a *non-monotonic* cost function.

Similar to the reasoning above, for undirected graphs a heuristic is *inconsistent* if for at least one pair of states $x$ and $y$

$$|h(x) - h(y)| > c(x, y). \tag{5}$$

This means that the difference between the heuristic values of $x$ and $y$ is larger than the actual cost of going from $x$ to $y$.

8. Pearl [36] used the term *monotonicity* in a different sense.

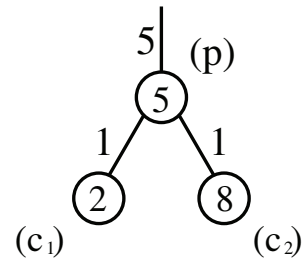


Figure 9: Inconsistent heuristic

According to this definition there are two types of inconsistencies in undirected search graphs; both are shown in Figure 9. As in all figures in the paper, the number inside a node is its admissible $h$-value. An edge is generally labeled with its cost.

- *Type 1: $h$ decreases from parent to child.* The parent node $p$ has $f(p) = g(p) + h(p) = 5 + 5 = 10$. Since the heuristic is admissible, any path from the start node to the goal node that passes through $p$ has a cost of at least 10. Since the edge from $p$ to $c_1$ has a cost of 1, $f(c_1) = g(c_1) + h(c_1) = 6 + 2 = 8$. This is a lower bound on the total cost of reaching the goal through $c_1$. This is weaker than the lower bound from the parent (which is valid for all its children). Thus the information provided by evaluating $c_1$ is "inconsistent" (in the sense that they do not agree) with the information from its parent $p$. In this case $f$ is *non-monotonic* when moving from $p$ to $c_1$.

- *Type 2: $h$ increases from parent to child.* Node $c_2$ presents another possible case for inconsistency, although this case is only inconsistent because the graph is undirected. Here the heuristic increased from 5 to 8 while the cost of the edge was only 1. The cost function $f$ is still monotonic increasing from $p$ ($f = 10$) to $c_2$ ($f = 14$). However the increase of the $h$-value is larger than the increase of the $g$-value. Note that since the graph is undirected there is also an edge from $c_2$ to $p$. Hence, logically $p$ is also one of the children of $c_2$. In this second occurrence of $p$, the $f$-value will decrease from 14 to 12 and is non-monotonic. Thus, the historical claim (e.g., of Pearl [36]) that consistency is equivalent to monotonicity is technically correct.[9]

The difference between the two types of inconsistency is important because later we will show that the pathmax propagation deals with Type 1 and corrects heuristics to be monotonic while our new bidirectional pathmax (BPMX) described below also deals with Type 2 and can cause the heuristic to be fully consistent. Note that the "good" behavior of consistent heuristics (e.g., that they do not re-expand nodes) usually comes from the cost function $f$ being monotonic.

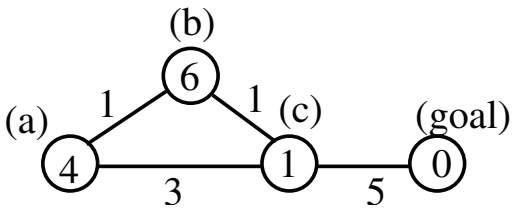


Figure 10: Re-expanding of nodes by A*

### 3.2 Inconsistent heuristics in A* and in IDA*

Assume that a state can be reached from the start state by multiple paths, each with a possibly different cost. Whenever a node is generated by A*, it is first matched against OPEN and CLOSED and if a duplicate is found, the copy with the larger $g$-value is ignored. If a consistent heuristic is used then $f$ is monotonic and all ancestors of a node $n$ have $f$-values *less than or equal to* $f(n)$. Therefore, the first time a node $n$ is expanded by A* (e.g., with $f(n) = K$) it always has the optimal $g$-value among all possible paths from the start to $n$. Otherwise, one of the ancestors of $n$ along the optimal path to $n$ must be in OPEN, but its $f$-value must be smaller than $K$ so it must have been expanded prior to $n$. As a consequence, when a node is expanded and moved to CLOSED it will never be chosen for expansion again. By contrast, with inconsistent heuristics where the $f$-function is non-monotonic, A* may re-expand nodes when they are reached via a lower cost path. A simple example of this is shown in Figure 10. Nodes $b$ and $c$ will be generated when the start node $a$ is expanded with $f(b) = 1+6 = 7$ and $f(c) = 3+1 = 4$. Next, node $c$ will be expanded, and the goal is discovered with an $f$-value of 8. Since $b$ has a lower $f$-value, it will be expanded next, resulting in a lower cost path to $c$. This operation is referred to as the *re-opening* of nodes ($c$ in our case), since nodes from CLOSED are re-opened and moved back to OPEN. Now, $c$ will be *re-expanded* with a lower $g$-cost, and a lower cost path of length 7 to the goal will be found. So, with A* the use of inconsistent heuristics comes with a real risk of many more node expansions than with a consistent heuristic. As the next section shows, this risk is not nearly as great as was previously thought. In a later section our experiments show that inconsistent heuristics can actually speed up an A* search.

IDA*, as a depth-first search (DFS) algorithm, does not perform duplicate detection.[10] Using IDA* on the state space in Figure 10, node $c$ will also be expanded twice (once for each of the paths) whether the heuristic is consistent or not. Thus, the problem of re-expanding nodes already exists in IDA*, and using inconsistent heuristics will not result in any additional performance degradation.

9. In practical applications it is a common practice (known as *parent pruning*) not to list the parent of a node as one of its children. In such cases the heuristic can be inconsistent according to equation 5 but the corresponding $f$-function is still monotonic. In practice, a full search tree where inconsistencies are only due to this second case (and the cost function $f$ is always monotonic) is probably rather rare. Therefore, in the reminder of this paper we will generally assume that all inconsistent heuristics produce a cost function $f$ that is non-monotonic.
10. In an advanced implementation of IDA* (as in any DFS) one can detect whether the current node already appeared as one of the ancestors in the current branch of the tree. However, only a small portion of the possible duplicates can be detected with this method when compared to algorithms that keep OPEN and CLOSED lists.

## 4. Worst-case behavior of A* with inconsistent heuristics

This section presents an analysis of the worst-case time complexity of A* when inconsistent heuristics are used.

If the heuristic is admissible and consistent, A* is "optimal" in terms of the number of node expansions ( [36], p. 85). However, as just explained, if the heuristic is admissible but not consistent, nodes can be re-opened and A* can do as many as $O(2^N)$ node expansions, where $N$ is the number of distinct expanded states. This was proven by Martelli [32].

### 4.1 The $G_i$ family of state spaces

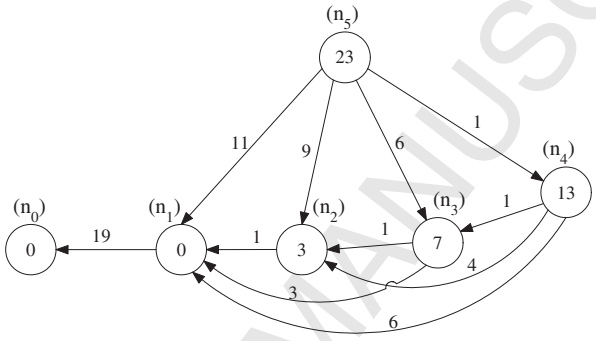


Figure 11: $G_5$ in Martelli's family

Martelli defined a family of state spaces $\{G_i\}$, for all $i \geq 3$, such that $G_i$ contains $i + 1$ states and requires A* to do $O(2^i)$ node expansions to find the solution [32]. $G_5$ from Martelli's family is shown in Figure 11; the number inside a state is its heuristic value and the number beside an edge is its cost. There are many inconsistencies in this graph. For example, $c(n_4, n_3) = 1$ but $h(n_4) - h(n_3) = 6$. The unique optimal path from the start ($n_5$) to the goal ($n_0$) has the states in decreasing order of their index $(n_5, n_4, ..., n_0)$, but $n_4$ has a large enough heuristic value ($f(n_4) = 14$) that it will not be expanded by A* until all possible paths to the goal (with $f < 14$) involving all the other states have been fully explored. Thus, when $n_4$ is expanded, nodes $n_3$, $n_2$ and $n_1$ are re-opened and then expanded again. The sequence of node expansions until reaching the goal, with the $f$-values shown inside the parentheses, is as follows: $n_5$(23), $n_1$(11), $n_2$(12), $n_1$(10), $n_3$(13), $n_1$(9), $n_2$(10), $n_1$(8), $n_4$(14), $n_1$ (7), $n_2$(8), $n_1$(6), $n_3$(9), $n_1$(5), $n_2$(6), $n_1$(4). Note that after $n_4$ is expanded the entire sequence of expansions that occurred prior to the expansion of $n_4$ is repeated but this time all these nodes are examined via paths through $n_4$. Thus, the existence of $n_4$ in $G_5$ essentially doubles the search effort required for $G_4$. This property holds for each $n_i$ so the total amount of work is $O(2^i)$. As we will show below, this worst-case behavior hinges on the state space having the properties that the edge weights and heuristic values grow exponentially with the number of states (as is clearly seen in the definition of Martelli's state spaces).

### 4.2 Variants of A*

Martelli devised a variant of A*, called B, that improves upon A*'s worst-case time complexity while maintaining admissibility [32]. Algorithm B maintains a global variable $F$ that keeps track of the maximum $f$-value of the nodes expanded thus far in the search. When choosing the next node to expand, if $f_m$, the minimum $f$-value in OPEN, satisfies $f_m \geq F$, then $f_m$ is chosen as in A*, otherwise the node with minimum $g$-value among those with $f < F$ is chosen. Because the value of $F$ can only change (increase) when a node is expanded for the first time, and no node will be expanded more than once for a given value of $F$, the worst-case time complexity of Algorithm B is $O(N^2)$ node expansions. However, even with this improvement the worst-case scenario is still poor, further reinforcing the impression that inconsistency is undesirable.

Bagchi and Mahanti proposed algorithm C, a variant of B, by changing the condition for the special case from $f_m < F$ to $f_m \leq F$ and altering the tie-breaking rule to prefer smaller $g$-values [1]. C's worst-case time complexity is the same as B's, $O(N^2)$.

### 4.3 New analysis

Although Martelli proved that the number of node expansions A* performs may be exponential in the number of distinct expanded states, this behavior has never been reported in real-world applications of A*. His family of worst-case state spaces have solution costs and heuristic values that grow exponentially with the number of states. We now present a new result that such exponential growth in solution costs and heuristic values are necessary conditions for A*'s worst-case behavior to occur.

We assume all edge weights are non-negative integers (edge weights of zero are permitted). The key quantity in our analysis is $\Delta$, defined to be the greatest common divisor of all the non-zero edge weights. The cost of every path from the start node to node $n$ is a multiple of $\Delta$, and so too is the difference in the costs of any two paths from the start node to $n$. Therefore, if during a search we re-open $n$ because a new path to it is found with a smaller cost than our current $g(n)$ value, we know that $g(n)$ will be reduced by at least $\Delta$.

**Theorem 1** *If A* performs $M > N$ node expansions then there must be a node with heuristic value of at least $LB = \Delta * \lceil (M - N)/N \rceil$.*

**Proof.** If A* does $M$ node expansions and there are only $N$ distinct expanded states, then the number of re-expansions is $M - N$. By the pigeon-hole principle there must be a node, say $K$, with at least $\lceil (M - N)/N \rceil$ re-expansions. Each re-expansion must decrease $g(K)$ by at least $\Delta$, so after this process the $g$-value of $K$ is reduced by at least $LB = \Delta * \lceil (M - N)/N \rceil$. ∎

In Figure 12, $S$ is the start node, $K$ is any node that is re-expanded at least $\lceil (M - N)/N \rceil$ times (as we have just seen, at least one such node must exist), $L$ is the path that resulted from the first expansion of $K$, and the upper path to $K$ (via $B$) is the path that resulted from the last expansion of $K$. Denote the $f$- and $g$-values along path $L$ as $f_L$ and $g_L$, and the $f$- and $g$-values along the upper path as $f_{last}$ and $g_{last}$, respectively.

Node $B$ is any node on the upper path, excluding $S$, with the maximum $f_{last}$ value (that is, the maximum $f_{last}$ value among any other node on the upper path). Nodes distinct from $S$ and $K$ must exist along this path because if there were a direct edge from $S$ to $K$, then $K$ would be opened as soon as $S$ was expanded with a $g$-value smaller than $g_L(K)$. Hence $K$ would not be expanded via $L$, leading to a contradiction. Node $B$ must be one of these intermediate nodes — it cannot be $S$

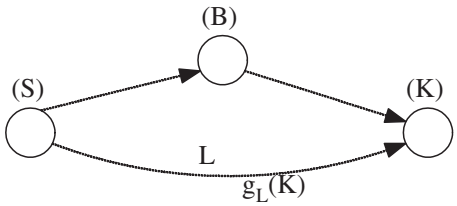


Figure 12: First and last explored path

by definition and it cannot be $K$ because if $f_{last}(K)$ was the largest $f_{last}$ value, the entire upper path would be expanded before $K$ would be expanded via $L$, again a contradiction. Hence, $B$ is an intermediate node between $S$ and $K$.

$h(B)$ must be large enough to make $f_{last}(B) \geq f_L(K)$ (because $K$ is first expanded via $L$). We will now use the following facts to show that $h(B)$ must be at least $LB$:

$$\begin{aligned} f_{last}(B) &= g_{last}(B) + h(B) && (6)\\ f_{last}(B) &\geq f_L(K) && (7)\\ f_L(K) &= g_L(K) + h(K) && (8)\\ g_{last}(B) &\leq g_{last}(K) && (9)\\ LB &\leq g_L(K) - g_{last}(K). && (10) \end{aligned}$$

So,

$$\begin{aligned} h(B) &= f_{last}(B) - g_{last}(B), \text{ by Fact 6}\\ &\geq f_L(K) - g_{last}(B), \text{ by Fact 7}\\ &= g_L(K) + h(K) - g_{last}(B), \text{ by Fact 8}\\ &\geq g_L(K) - g_{last}(K) + h(K), \text{ by Fact 9}\\ &\geq g_L(K) - g_{last}(K), \text{ since } h(K) \geq 0\\ &\geq LB, \text{ by Fact 10.} \end{aligned}$$

∎

From Theorem 1 it follows that for A* to do $2^N$ node expansions, there must be a node with a heuristic value of at least $\Delta * \lceil (2^N - N)/N \rceil$, and for A* to do $N^2$ node expansions, there must be a node with a heuristic value of at least $\Delta * (N - 1)$.

**Corollary 2** *Let $h^*(start)$ denote the optimal solution cost. If A* performs more than $N$ node expansions then $h^*(start) \geq LB$.*

**Proof.** Since in the proof of Theorem 1 A* expanded node $B$ before the goal, $h^*(start)$ must be at least $f(B)$, which is at least $LB$. ∎

**Corollary 3** *If $h^*(start) \leq \lambda$ then $M$, the number of node expansions done by A* to find a path to the goal, is less than or equal to $N + N * \lambda/\Delta$.*

**Proof.** Using Corollary 2,

$$\Delta * \lceil (M - N)/N \rceil = LB \leq h^*(start) \leq \lambda$$

which implies

$$M \leq N + N * \lambda/\Delta. \quad \blacksquare$$

**Corollary 4** *Let $m$ be a fixed constant and $G$ a graph of arbitrary size (not depending on $m$) whose edge weights are all less than or equal to $m$. Then $M$, the number of node expansions done by A* during a search in $G$, is at most $N + N * m * (N - 1)/\Delta$.*

**Proof.** Because the non-goal nodes on the solution path must each have been expanded, there are at most $N - 1$ edges in the solution path and $h^*(start)$ is therefore at most $\lambda = m * (N - 1)$. By Corollary 3,

$$M \leq N + N * \lambda/\Delta \leq N + N * m * (N - 1)/\Delta. \quad \blacksquare$$

This is just one example of conditions under which A*'s worst-time complexity is not nearly as bad as Martelli's bound suggests. The key observation arising from the analysis in this section is that there is an intimate relationship between the number of node expansions, the magnitude of the heuristic values, and the cost of an optimal path to the goal. The number of node expansions can only grow exponentially if the latter two factors do as well.

## 5. Pathmax and Bidirectional Pathmax

It is well known that the $f$-values along any path in a search tree can be forced to be *monotonic non-decreasing*. This is simply done by propagating the $f$-value of a parent to a child if it is larger. This technique is usually called *pathmax*. In this section, the idea behind pathmax is introduced and then, for undirected state spaces, generalized to a new method called *bidirectional pathmax* which provides better heuristic propagation.

### 5.1 Pathmax

Mero introduced algorithm B$'$, a variant of B, that dynamically updates heuristic values during the search while maintaining admissibility [34]. This was achieved by adding two rules (known as *pathmax rules*) for propagating heuristic values between a node and its children. Like algorithm B (described in Section 4), B$'$ has a worst-case time complexity of $O(N^2)$. While the pathmax rules were introduced in the context of algorithm B, they are applicable in A* too. The rules propagate heuristic values during the search between a parent node $p$ and its child node $c_i$ (where the edge connecting them costs $c(p, c_i)$) as follows:

**Pathmax Rule 1:** $h(c_i) \leftarrow \max(h(c_i), h(p) - c(p, c_i))$, and

**Pathmax Rule 2:** $h(p) \leftarrow \max(h(p), \min_{c_i \in Successors[p]}(h(c_i) + c(p, c_i)))$.[11]

For Rule 1, we know $h(p) \leq h^*(p)$ (where $h^*(x)$ denotes the optimal cost to the goal node) and $h(c_i) \leq h^*(c_i)$ because $h$ is admissible. We also know that $h^*(p) \leq c(p, c_i) + h^*(c_i)$ because one possible path from $p$ to the goal goes via $c_i$. By combining these facts, it can be inferred that $h^*(c_i) \geq h(p) - c(p, c_i)$. Figure 13 shows how the parent node $p$ updates the heuristic values

11. This is our version of pathmax Rule 2. The version in the original paper [34] is clearly not correct and is probably a printing error.

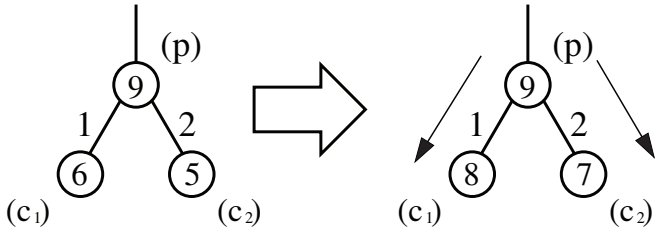


Figure 13: Pathmax Rule 1

of the child nodes $c_1$ and $c_2$ according to Rule 1. A consequence of this rule is that the child node inherits the $f$-value of the parent node if it is larger. Pathmax Rule 1 is often written as $f(c_i) := \max(f(p), f(c_i))$. This causes the $f$-value to be *monotonic non-decreasing* along any path. However, a child node can still have a heuristic value that is larger than that of the parent by more than the change in $g$ and the heuristic can still be inconsistent if the graph is undirected (as in inconsistency Type 2 presented at the end of Section 3.1). Our bidirectional pathmax method (BPMX) described below deals with such cases and corrects this type of inconsistency.

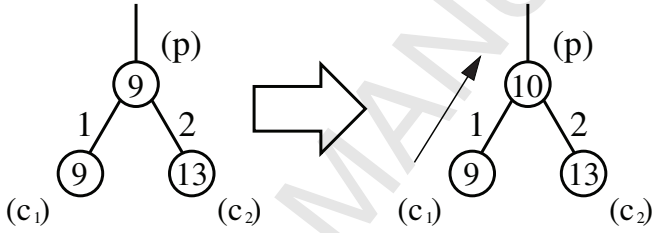


Figure 14: Pathmax Rule 2

The explanation for Rule 2 (introduced by Mero) is as follows. In directed state spaces, the optimal path from $p$ to a goal must contain one of $p$'s successors (unless $p$ is a goal), so $h^*(p)$ is at least as large as $\min_{c_i \in Successors[p]}(h(c_i) + c(p, c_i))$. Rule 2 corrects $h(p)$ to reflect this. Figure 14 shows how the child nodes $c_1$ and $c_2$ update the heuristic value of the parent node $p$ according to Rule 2. $c_1$ has the minimal $f$-value and its value is propagated to the parent. While the idea of Rule 2 is correct, its practical value is limited. First, in some state spaces (e.g., in undirected state spaces) there is an edge from state $p$ to its parent $a$ and the shortest path from $p$ to the goal might pass through state $a$. In such cases, using Rule 2 is relevant only if $a$ is actually listed as a child of $p$ in the search graph. This will be possible only if the *parent pruning* optimization is not used. We discuss more limitations of Rule 2 in Section 5.4 after we introduce our generalization of Rule 1 to bidirectional pathmax.

### 5.2 Pathmax does not make the $f$-function monotonic

It is sometimes thought that pathmax Rule 1 actually converts a non-monotonic cost function $f$ into a monotonic cost function and, as a consequence, node re-expansion will be prevented.[12] This is not

12. We do not use the term consistent because of our understanding that the cost function can be monotonic but the heuristic can still be inconsistent as in Type 2 (presented in Section 3.1) for undirected graphs.

correct. It is true that after applying pathmax the $f$-values never decrease along the path that was just traversed. However, the $f$-values can still be non-monotonic for paths that were not traversed yet. To see this, recall that with a consistent heuristic where the cost function is monotonic, closed nodes are never re-opened by A*, because when a node is removed from OPEN for the first time we are guaranteed to have found a least-cost path to it. This is the key advantage of a consistent heuristic over an inconsistent heuristic which has a non-monotonic cost function where closed nodes can be re-opened. Pathmax does not correct this deficiency of inconsistent heuristics. This was noted by Nilsson [35] (p. 153) and by Zhou and Hansen [50].

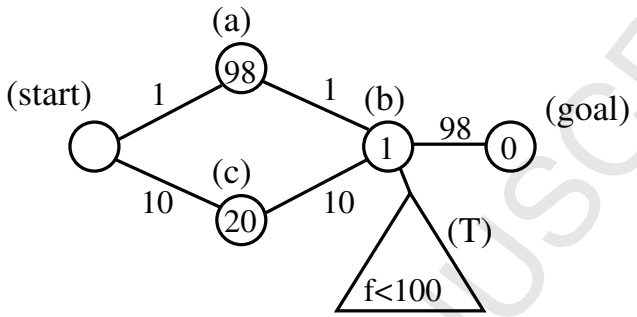


Figure 15: Example where a closed node must be re-opened with pathmax

Consider the example in Figure 15 where the heuristic is admissible but is inconsistent ($h(a) = 99$ but $h(b) = 1$), and $f$ is non-monotonic ($f(b) < f(a)$). The optimal path in this example is $start - a - b - goal$, with a cost of 100. A* will expand $start$ and then $c$ ($f$=30), at which point $a$ and $b$ will be in OPEN. $a$ will have $f = 99$ and $b$, because of pathmax, will have $f = 30$ instead of $f = 21$. $b$ will be now be expanded and closed, even though the least-cost path to $b$ (via $a$) has not been found. A* will then expand the entire set of nodes in the subtree $T$ before it expands $a$. At that point $a$ will be expanded, revealing the better path to $b$, and requiring $b$ and all of the nodes in $T$ to be expanded for a second time.

### 5.3 Bidirectional Pathmax – BPMX

Mero's Rule 1 was defined to propagate values between a parent and its child in the search tree. However, this pathmax rule can be applied from a given node $x$ to another node $y$ in any direction (not necessarily from a parent to its children in a search tree) as long as there is a path from $x$ to $y$. This can be beneficial in application domains where the search graph is undirected, e.g., when operators are invertible and costs symmetric. Assume an admissible heuristic $h$ where $h(x) > h(y) + c(x, y)$. Now, $h^*(x) \leq c(x, y) + h^*(y)$ because a possible path from $x$ to the goal passes through $y$. Therefore $h^*(y) \geq h^*(x) - c(x, y) \geq h(x) - c(x, y)$ (since $h$ is admissible) and we can apply the following general rule:

$$h(y) \leftarrow \max(h(y), h(x) - c(x, y)). \tag{11}$$

Pathmax Rule 1 used this general rule from a parent node to its children. In a search tree if there is an edge from a child $c$ to its parent $p$ then this can be achieved by introducing a new pathmax rule for children-to-parent value propagation as follows:

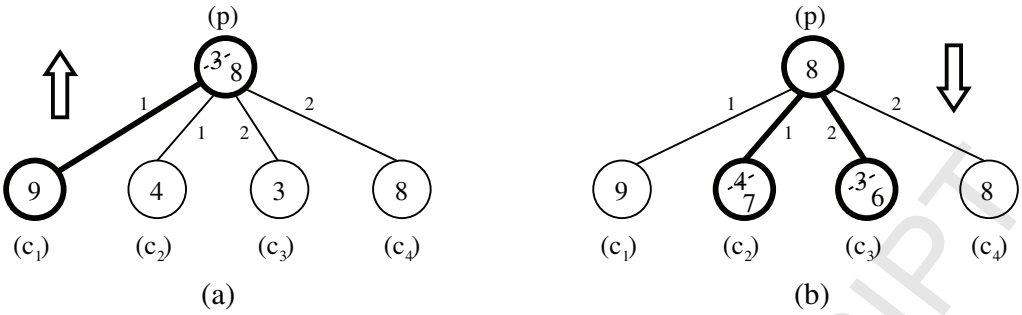


Figure 16: Example of BPMX. The arrows show direction of the propagation of heuristic values. Propagation occurs along the bold edges.

**Pathmax Rule 3:** $h(p) \leftarrow \max(h(p), h(c) - c(c, p))$.
Figure 16(a) shows how Rule 3 can be used. The heuristic of child $c_1$ is propagated to the parent $p$ and $p$'s heuristic is increased from 3 to 8.

Our new method, bidirectional pathmax (BPMX), uses Rules 1 and 3 to propagate (inconsistent) heuristic values in any direction, as described generally in Equation 11. Large heuristic values are propagated along edges but to preserve admissibility we subtract the weight of the edges along the way. Therefore, updating a node's value can have a cascading effect (to its neighbors and so on) as the propagation that started from child $c_1$ continues from the parent to the other children (as shown in Figure 16(b)). The BPMX process stops when we arrive at a node whose original heuristic value is not smaller than the propagated value. The bold edges in Figure 16(b) correspond to cases where BPMX further propagates the new heuristic value of $p$ to its children (to $c_2$ and $c_3$). By contrast, child $c_4$ cannot exploit BPMX here as its original heuristic value was 8 while BPMX would propagate a value of 6.

Note that Rule 1 only deals with inconsistencies of Type 1 (described in Section 3.1) and causes the cost function to be monotonic along the edge. BPMX further extends this to inconsistencies of Type 2 and causes the heuristic to be fully consistent.

#### 5.3.1 BPMX FOR IDA*

Before discussing BPMX for IDA* we first highlight the following observation.
**Observation:** What is important for IDA* is not the exact $f$-value of a node but whether or not the $f$-value causes a cutoff.
**Explanation:** IDA* expands a node if its $f$-value is *less than or equal to* the current threshold $T$ and backtracks if it is larger than $T$. Thus, only a cutoff reduces the work performed.

It may not be immediately obvious, but using Rule 1 with IDA* does not have any benefit.[13] This is because propagating the heuristic of the parent $p$ (with Rule 1) to the child $c$ will cause $f(c) = f(p)$. It will not increase its $f$-value above the threshold $T$ (as the $f$-value of $p$ was already less than or equal to $T$) and therefore will not result in additional pruning.

13. We discuss Rule 2 in the context of IDA* in Section 5.4.

Using Rule 3 with IDA* has great potential as it may prune many nodes that would otherwise be generated (and even fully expanded). For example, suppose that the current IDA* threshold $T$ for Figure 17 is 2. Without the propagation of $h$ from the left child, both the parent node ($f(p) = g(p) + h(p) = 0 + 2 = 2$) and the right child ($f(c_1) = g(c_1) + h(c_1) = 1 + 1 = 2$) would be expanded. When using BPMX propagation, the following will occur. The left child will have $f(c_2) = 1 + 5 = 6$, and with a $T = 2$ IDA* will backtrack. However, BPMX will update the parent's $h$-value to $h(p) = 4$ and its overall cost to $f(p) = 0 + 4 = 4$. This results in a cutoff, and the search will backtrack from the root node without even generating the right child (whose heuristic value can be modified to 3, e.g., in A* as discussed below).

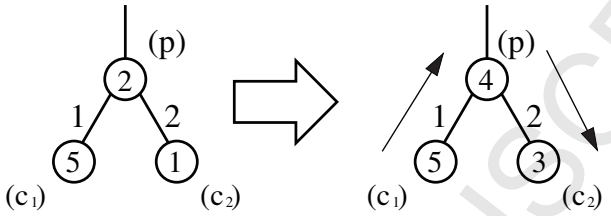


Figure 17: BPMX. In IDA*, the right branch is not even generated.

An efficient implementation of BPMX for IDA* is provided in Algorithm 1. In this implementation, Rule 3 is applied "for free" when backtracking from a child. First, the heuristic of the parent $p$ is updated by Rule 3 (line 20). Then, if the $f$-value of the parent becomes larger than the threshold, the subtree below it is immediately pruned (line 21) and control is passed back to the parent of $p$. In this case, the other children of $p$ are not generated.

An alternative exhaustive implementation will not stop at line 21 but will continue to generate all children of $p$ and calculate their heuristics. This may result in propagating higher heuristic values by Rule 3 to the parent $p$ and increase the chance of further pruning ancestors of $p$. The drawback of this implementation is that parent $p$ is fully expanded.[14] We have experimented with this variant in most of the domains studied in this paper. However, no gains were provided and the "lazy" approach of stopping as soon as a cutoff occurred consistently outperformed the exhaustive variant. Therefore, we only report experimental results with the lazy variant.

A reminiscent idea of BPMX for propagating heuristic values between nodes was introduced in the context of learning heuristics in DFS searches [3]. The difference is that unlike BPMX the "learning algorithm" of that work requires a transposition table.

#### 5.3.2 BPMX FOR A*

Due to its depth-first nature, BPMX propagation is easily implemented in IDA*, and values are propagated naturally between children and their parents. By contrast, in A* BPMX updates are more difficult as nodes that might be updated may need to be retrieved from OPEN or CLOSED.

BPMX can be parameterized with the maximum depth that a heuristic value will be propagated. BPMX(1) is at one extreme, propagating $h$ updates only among a node and its children. BPMX($\infty$) is at the other extreme, propagating $h$ updates as far as possible.

14. This process can be extended and we can perform a $k$-lookahead search to find large heuristic values.

**Algorithm 1** IDA* with BPMX ("::" adds an element to a list)

```
01: function IDA*_bpmx(initial_node s)                          ▷ Returns the optimal solution
02:     threshold ⟵ h(s)
03:     repeat
04:         GoalFound ⟵ DFS_bpmx(s, NULL, 0, Path, threshold, h_s)
05:         threshold ⟵ next_threshold()
06:     until GoalFound
07:     return Path
08: end function

09: boolean function DFS_bpmx(node p, previous_move pm, depth g, List Path,
                               integer threshold, heuristic_value &h_p)
10:     h_p ⟵ h(p)
11:     if (h_p + g) > threshold then return false
12:     if p = goal_node then return true
13:     for each legal_move m_i do
14:         if m_i = pm^-1 then continue                        ▷ Parent pruning
15:         generate child c_i by applying m_i to p
16:         if DFS_bpmx(c_i, m_i, g + c(p, c_i), Path, threshold, h_ci) = true then
17:             Path ⟵ m_i :: Path
18:             return true
19:         else
20:             h_p ⟵ max(h_p, h_ci − c(c_i, p))                ▷ Rule 3
21:             if (h_p + g) > threshold then return false      ▷ Backtrack ASAP
22:         end if
23:     end for
24:     return false
25: end function
```

BPMX(1) can be implemented efficiently if the BPMX computation happens after all children of a node have been generated (and checked for duplicates in OPEN and CLOSED) but before they are added/moved back to OPEN and/or CLOSED. Assume that a node $p$ is expanded and that its $k$ children $c_1, c_2, \ldots, c_k$ are generated. References to these nodes can be saved for faster manipulation in the following steps. Let $c_{max}$ be the node with the maximum heuristic value among all the children and let $h_{max} = h(c_{max})$. In addition, assume that each edge has unit cost and is undirected. $h_{max}$ can be propagated to the parent node by decreasing it by one (using Rule 3) and then to the other children by decreasing it by one again (using Rule 1). Thus, each of the other children $c_i$ can have a heuristic of

$$h_{BPMX}(c_i) = \max(h(c_i), h(p) - 1, h_{max} - 2).$$

After all these nodes have their value updated, then the parent node is inserted in CLOSED (with its new $f$-value) and all the children are inserted (or changed) in OPEN (with their new $f$-values).

Pseudocode for an efficient implementation of A* with BPMX(1) is shown in Algorithm 2. There is a single data structure for OPEN and CLOSED which is implicit in calls for looking up nodes. In our actual implementation most lookups are cached to reduce overhead.

BPMX($d$) with $d > 1$ starts at a new node that was just generated and continues to propagate $h$-values to its generated neighborhood (nodes on OPEN and CLOSED) as long as the $h$-values of nodes are being increased. There are a number of possible implementations and they all require finding and retrieving nodes from OPEN and CLOSED. Obviously, this will incur some (even all) of the following possible overheads associated with BPMX($d$) (with $d > 1$) within the context of A*:

(a) performing lookups in OPEN and/or CLOSED (when looking for neighbors),

(b) ordering OPEN nodes based on their new $f$-value (when these values change), and

(c) computational overhead of comparing heuristic values and assigning a new value based on the propagations.

These costs are the same as the costs incurred when performing A* node expansions. In BPMX($d$) the propagating of heuristic values can result in the equivalent of multiple expansions (re-openings). The propagation (and re-openings) must follow all children of a node until the depth $d$ parameter is satisfied. As such, we regard BPMX with $d > 1$ as an independent search process rather than a small optimization on top of the main search. Our experimental results show that node expansions that occur during the BPMX process have the same cost as A* node expansions.[15] Therefore, in the remainder of the paper we will not distinguish between A* and BPMX($d > 1$) expansions. When $d = 1$ the overhead is not included in the count of node expansions, only in time measurements.

A natural question is how to determine which value for parameter $d$ is best. It turns out that no fixed $d$ is optimal in the number of node expansions for all graphs. While a particular $d$ can produce a large reduction in the number of node expansions for a given state space, for a different state space it can result in an $O(N^2)$ increase in the number of node expansions.

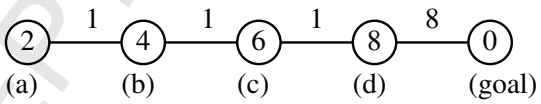


Figure 18: Worst-case example for BPMX($\infty$)

Figure 18 gives an example of the worst-case behavior of BPMX($\infty$). The heuristic values gradually increase from nodes $a$ to $d$. When node $b$ is reached, the heuristic can be propagated back to node $a$, increasing the heuristic value by 1. When node $c$ is reached, the heuristic update can again be propagated back to nodes $b$ and $a$. In general, when the $i^{th}$ node in the chain is generated a BPMX update can be propagated to all previously expanded nodes. Overall this will result in $1 + 2 + 3 + \cdots + N - 1 = O(N^2)$ propagation steps with no savings in node expansions. This

15. This is true for PDB heuristics (inexpensive to compute). However, this might not be true in cases where the heuristic calculation requires a large amount of time.

**Algorithm 2** A* with BPMX(1) (assumes symmetric edge costs)

```
01: function A*_{bpmx(1)}(start, goal)
02:   push(start)
03:   while (queue is not empty)
04:     current ⟵ pop best node from queue
05:     if current is goal return extractPath(start, goal)
06:     neighbors ⟵ generateSuccessors(current)
07:     BestH ⟵ 0                                   ▷ stores parent h-cost (from pathmax)
08:     for each neighbor 1...i in neighbors        ▷ cache these lookups for later use
09:       BestH ⟵ max(BestH, lookupH(neighbor) − c(current, neighbor))
10:     end for
11:     storeH(current, max(lookupH(current), BestH))
12:     for each neighbor 1... i in neighbors
13:       EdgeCost = c(current, neighbor)
14:       switch (getLocation(neighbor))
15:         case ClosedList:
16:           if (lookupH(neighbor) < BestH − EdgeCost)        ▷ BPMX or PMX update
17:             storeH(neighbor, BestH − EdgeCost)
18:           end if
19:           if (lookupG(current) + EdgeCost < lookupG(neighbor))   ▷ Found shorter path
20:             setParent(neighbor, current)
21:             storeG(neighbor, lookupG(current) + EdgeCost)
22:             reopen(neighbor)
23:           end if
24:         case OpenList:
25:           if (lookupG(current)+EdgeCost < LookupG(neighbor))     ▷ Found shorter path
26:             setParent(neighbor, current)
27:             storeG(neighbor, lookupG(current)+EdgeCost)
28:             updateKey(neighbor)                                  ▷ Re-sort OPEN
29:           end if
30:           if (BestH − EdgeCost > lookupH(neighbor))        ▷ BPMX or PMX update
31:             storeH(neighbor, BestH − EdgeCost)
32:             updateKey(neighbor)
33:           end if
34:         case NotFound:                          ▷ also applies BPMX or PMX update
35:           addOpenNode(neighbor, lookupG(current)+EdgeCost,
36:                       max(h(neighbor, goal), BestH − EdgeCost))
37:       end switch
38:     end for
39:   end while
40:   return nil
41: end function
```

provides a general worst-case bound. At most, the entire set of previously expanded nodes can be re-expanded during BPMX propagations, which is what happens here.

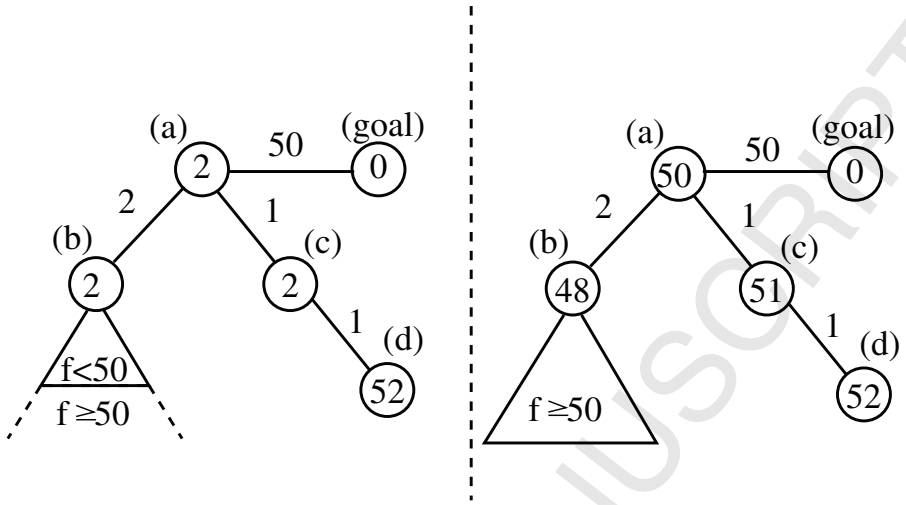


Figure 19: Best-case example for BPMX($\infty$)

By contrast, Figure 19 gives an example of how BPMX($\infty$) propagation can be effective. Assume node $a$ is the start node. It is expanded and its three children ($b$, $c$ and $goal$) are generated with $f$-values $f(b) = 4$, $f(c) = 3$ and $f(goal) = 50$. Next $c$ is expanded and $d$ is generated. If BPMX is not activated (left side), then all nodes in the subtree under $b$ with $f < 50$ will be expanded; only then $goal$ is expanded and the search terminates. Now, consider the case where BPMX($\infty$) is activated (right side). While generating node $d$ its heuristic value is propagated with BPMX to $c$, then to $a$ and then to $b$ raising the $f$-value of $b$ to 50. Note that we can infer that the entire subtree below $b$ will have $f \geq 50$. In this case $f(b) = f(goal) = 50$ and, assuming ties are broken in favor of low $h$-values, $goal$ is expanded and the search halts after expanding only three nodes.

### 5.4 Pathmax Rule 2

We have just seen the usefulness of Pathmax Rules 1 and 3. Mero also created Rule 2 for children-to-parent value propagation [34].

**Rule 2:** $h(p) \leftarrow \max(h(p), \min_{c_i \in Sucessors[p]}(h(c_i) + c(p, c_i)))$.
We now discuss the properties of Rule 2.

#### 5.4.1 Rule 2 when IDA* is used

Similar to Rule 1, there is no benefit for using Rule 2 on top of IDA* in undirected state spaces as no pruning will be caused by it. Assume that node $p$ has children $c_1$, $c_2 \ldots c_k$ and that the parent of $p$ is $a$ (as shown in Figure 20). Assume also that that $c_m$ produced the minimal $f$-value among all the children. We show that neither $p$ nor $a$ can benefit from Rule 2 when applied to $p$ when Rules 1 and 3 are used.

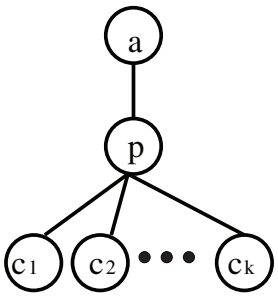

Figure 20: Example for Rule 2

- $p$ **cannot benefit from Rule 2:** Assume that $p$ is not causing a cutoff in the search. In this case, the search proceeds to the children. Now, if the minimum child ($c_m$) causes a cutoff then all the other children must also cause a cutoff. When using Rule 2, then all the children are generated in order to find the one with minimum cost. Either way, using Rule 2 or not, *all* children are generated and Rule 2 will have no added value for $p$.

- $a$ **cannot benefit from Rule 2:** Assume that Rule 2 was activated and that we set $f_{new}(p) = f(c_m)$. Now, due to the activation of Rule 1 (ordinary pathmax) the $f$-value is monotonically increasing along any path of the search tree. Thus, $f_{new}(p) \geq f(p)$. If $f_{new}(p) = f(p)$ then there is no change in the course of the search by applying Rule 2. Now, consider the case where $f_{new}(p) > f(p)$. Recall that for Rule 2 to work we must also list $a$ as one of the children of $p$. There are now two cases. The first case ($c_m = a$) is that $a$ produced the minimal $f$-value among the children. Now, if we apply Rule 3 we get that $h_{new}(a) = h(p) - c(a,p) = h(c_m) + c(p,c_m) - c(a,p) = h(c_m) + c(p,a) - c(a,p) = h(a)$. Thus, there is no change to the $h$-value of $a$. The second case ($c_m \neq a$) is that another child was chosen as the minimum, meaning that $h(a) + c(a,p) \geq h(c_m) + c(c_m,p)$. Now, if we apply Rule 3 we get that $h_{new}(a) = h(p) - c(a,p) = h(c_m) + c(p,c_m) - c(a,p) \leq h(a) + c(a,p) - c(a,p) = h(a)$. Here applying Rule 3 can only decrease the $h$-value of $a$ and it is again unchanged.

  Thus, $a$ cannot benefit from applying Rule 2 either and Rules 1 and 3 are sufficient to obtain all the potential benefits.

#### 5.4.2 Rule 2 when A* is used

Assume that we are running A* and that node $p$ is now expanded. Its children are added to OPEN while $p$ goes to CLOSED. If after applying Rule 2, its $f$-value increases then it will go to CLOSED but with a higher $f$-value because its new $h$-value is larger than its original $h$-value. This might affect duplicate pruning in the future if node $p$ is reached via a different path.

Furthermore, Rule 2 is just a special case ($k = 1$) of a $k$-lookahead search where values from the frontier are backed up to the root of the subtree. In fact, similar propagation is used for heuristic learning in LRTA* [23] when repeated search trials take place. This is also applicable for strict consistent heuristics.

Based on all the above, we did not implement Rule 2 in our experiments and focus on Rules 1 and 3 which are the core aspects of BPMX value propagation with inconsistent heuristics.

## 6. Creating inconsistent heuristics

As illustrated in the quote from *Artificial Intelligence: A Modern Approach* [38] given earlier, there is a perception that inconsistent admissible heuristics are hard to create. However, it turns out that this is not true. The following examples use PDB-based heuristics (used in many of the applications in this paper) to create inconsistent heuristics. However similar ideas can be applied to other heuristics. We show examples of inconsistent heuristics for pathfinding in explicit graphs in Section 8.

It is important to note that we can trivially make any heuristic inconsistent. For example, with a table-based heuristic (such as a PDB) one can randomly set table entries to 0. Of course, introducing this inconsistency results in a strictly less informed heuristic. In this section we give examples of inconsistent heuristics which provide more informed values that can benefit the search.

### 6.1 Random selection of heuristics

Many domains have a number of heuristics available. When using only one heuristic the search may enter a region with "bad" (low) estimation values (a heuristic depression). With a single fixed heuristic, the search is forced to traverse a (possibly large) portion of that region before being able to escape from it.

A well-known solution to this problem is to consult a number of heuristics and take their maximum value [5, 10, 18, 19, 24, 26]. When the search is in a region of low values for one heuristic, it may be in a region of high values for another. There is a tradeoff for doing this, as each heuristic calculation increases the time it takes to compute $h(n)$. Additional heuristic consultations provide diminishing returns in terms of the reduction in the number of node expansions, so it is not always recommended to use them all.

Given a number of heuristics one could select which heuristic to use *randomly*. Only a single heuristic will be consulted at each node, and no additional time overhead is needed over a fixed heuristic. Random selection between heuristics introduces more diversity to the values obtained in a search than using a single fixed selection. The random selection of a heuristic will produce inconsistent values if there is no or little correlation between the heuristics. Furthermore, a random selection of heuristics might produce inconsistent $h$-values even if all the heuristics are themselves consistent.

When using PDBs, multiple heuristics often arise from exploiting domain specific geometric symmetries. In particular, additional PDB lookups can be performed given a single PDB. For example, consider Rubik's cube and suppose we had the "yellow face" PDB described previously in Section 2.4.1. Reflecting and rotating this puzzle will enable similar lookups for any other face with a different color (e.g., green, red, etc.) since any two faces are symmetrical. Different (but admissible) heuristic values can be obtained for each of these lookups in the same PDB. As another example, consider the main diagonal of the sliding-tile puzzle. Any configuration of tiles can be reflected about the main diagonal and the reflected configuration shares the same attributes as the original one. Such reflections are usually used when using PDBs for the sliding-tile puzzle [5, 10, 11, 26] and can be looked up from the same PDB.

In recent work, a learning algorithm was used to decide when to switch between two (or more) heuristics [7]. A classifier was used to map a state to a heuristic, considering the likely quality of the heuristic estimate and the time needed to compute the value. The resulting search has inconsistencies in the heuristic values used.

### 6.2 Compressed pattern databases

There is a tradeoff between the size of a table-based heuristic (such as a PDB) and the search performance. Larger tables presumably contain more detailed information, enabling more accurate heuristic values to be produced.

Researchers have explored building very large PDBs (possibly even on disk) and compressing them into smaller PDBs [11, 12, 27, 39, 2]. A common compression idea is to replace multiple PDB entries by a single entry (often exploiting a locality property, so that the values of the entries are highly correlated), thereby reducing the size of the PDB. To preserve admissibility, the compressed entry must store the minimum value among all the entries that it is replacing. This is called *lossy compression* because some state lookups will end up with a less effective heuristic value. It has been shown that if the values in PDBs are locally correlated, then most of the heuristic accuracy will be preserved [11]. Thus, large PDBs can be built and then compressed into a smaller size with little loss in performance. Such compressed PDBs are more informed than uncompressed PDBs which use the same amount of memory [11].

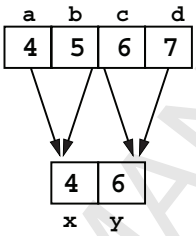


Figure 21: Inconsistency of a compressed pattern database

The compression process may introduce inconsistency into the heuristic, since there is no guarantee that the heuristic value of adjacent states in the search space will lose the same amount of information during compression. For example, consider the PDB in Figure 21 and assume that it is consistent. Assume that $b$ and $c$ are connected by an edge with cost of 1. During compression, $b$ might be mapped to $x$ in the abstract space, and $c$ to $y$. To preserve admissibility, $x$ and $y$ must contain the minimum value of the states mapping to those locations. Now states $b$ and $c$ are inconsistent in the abstract space (the difference between their heuristics (=2) is bigger than the actual distance between them (=1)).

### 6.3 Dual heuristic

The concept of *duality* and *dual heuristics* in permutation state spaces was introduced by Zahavi *et al.* [14, 44, 45]. Such heuristics may produce inconsistent heuristic values. The papers provide a detailed discussion of these concepts. Here we provide sufficient details for our purposes.

In permutation state spaces, states are different permutations of objects. Similarly, any given operator sequence is also a permutation (i.e., transfers one permutation into another permutation). For each state $s$, a dual state $s^d$ can be computed. The basic definition is as follows. Let $\pi$ be the permutation that transforms state $s$ into the goal. The *dual* state of $s$ (labeled as $s^d$) is defined as the state that is constructed by applying $\pi$ to the goal. Alternatively, if $O$ is the set of operators that transfer $s$ to the goal, then applying $O$ to the goal will reach $s^d$.

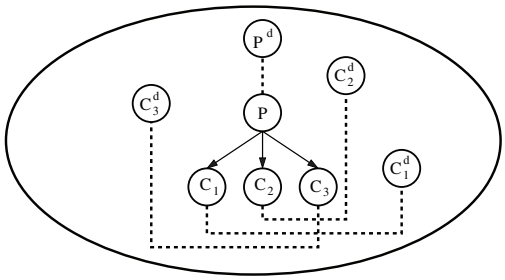


Figure 22: Dual states of the parent and its children

This dual state $s^d$ has the important property that it is the same distance from the goal as $s$. The reason is that any sequence of operators that maps $s$ to the $goal$ also maps the $goal$ to $s^d$. Since operators are reversible in permutation state spaces, the sequence can be inverted to map $s^d$ to the goal. Efficient methods have been suggested for deriving the dual state $s^d$ given a description of state $s$ [45]. Since the distance to the goal of both states is identical, any admissible heuristic applied to $s^d$ is also admissible for $s$ and can be used as a heuristic for it. For a state $s$, the term *dual lookup* is used when looking up $s^d$ in the PDB. When moving from a parent state to a child state, performing a dual lookup may produce an *inconsistent* value even if the heuristic itself (in its regular form) is consistent. The explanation for this is as follows. In a standard search, a parent state $p$, and any of its children $c_i$, are neighbors by definition. Thus, a consistent heuristic must return consistent values when applied to $p$ and $c_i$. However, the heuristic values obtained for $s^d$ and $c_i^d$ may not be consistent because $s^d$ and $c_i^d$ are not necessarily neighbors (as illustrated in Figure 22).[16]

In general, there are two easy ways to generate inconsistency for a given domain: 1) the use of multiple different heuristics and 2) using a heuristic that has some values missing or degraded. We provided some examples in this section. This list is not exhaustive and the above examples are by no means the only ways of creating inconsistent heuristics.

### 6.4 Inconsistent versus consistent heuristics

Besides the potential of $h$-value propagation, inconsistent heuristics have other attributes which might reduce the number of node expansions in a search when compared to consistent heuristics. This section addresses these attributes.

Most of the previous work on admissible heuristics mainly concentrated on improving the quality of the heuristic assessment. A heuristic $h_1$ is considered to be more informed (better quality) than $h_2$ if it typically returns a higher value for an arbitrary state [38]. A *de facto* standard usually used by researchers is to compare the average values of a given heuristic over the entire domain space or over a large sample of states of the domain (e.g., [10, 11, 24, 26]). Korf, Reid and Edelkamp (denoted as KRE) introduced the notion of the *overall distribution* of heuristic values [28, 29]. Define $p(v)$ to be the probability that a random state of the state space will have a heuristic value of $v$. Likewise, define $P(v)$ to be the probability that a random state will have a heuristic value *less than*

16. This phenomenon is explained in detail in the original papers [14, 44, 45]. An example is provided in the Appendix of this paper.

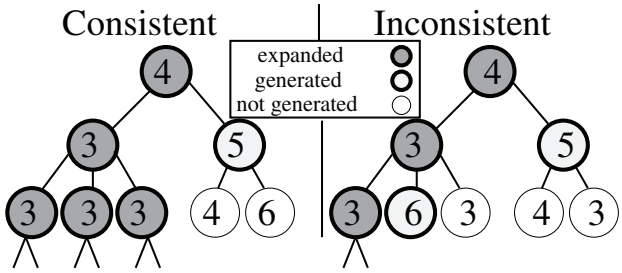


Figure 23: Consistent versus inconsistent heuristics. Nodes are marked with their $h$-value.

*or equal to* $v$. KRE suggested using the distribution of values from a heuristic function to measure the "informedness" of the function. Doing this for admissible heuristics will typically show that if a heuristic is more informed then the distribution of values will be higher, as will be the average value. We show in Section 7.2.1 that, when inconsistent heuristics are used, this distribution is not enough and there are more attributes to consider.

KRE also introduced a formula to predict the number of node expansions by IDA* on a single iteration when using a consistent admissible heuristic [28, 29]:

$$N(b, d, P) = \sum_{i=0}^{d} b^i P(d - i),$$

where $b$ is the brute-force branching factor, $d$ is the depth of the search (the IDA* threshold), and $P$ is the heuristic distribution. KRE showed that if $P(x)$ is defined in a particular way (the "equilibrium" distribution) then the number of nodes $n$ such that $f(n) \leq d$ is equal to $N(b, d, P)$ in the limit of large $d$. We call these nodes the *nodes with potential to be expanded* or *potential* nodes in short. KRE then proved that with consistent heuristics all potential nodes will eventually be expanded by IDA*. Assume that $n$ is a potential node. Since the heuristic is consistent, any ancestor $a$ of $n$ must also have $f(a) \leq d$ and is also a potential node. Then, by induction they showed that the entire branch from the root to $n$ will be expanded since all the nodes of the branch are potential nodes.

For inconsistent heuristics the behavior is different. For a potential node $n$, there may exist an ancestor $a$ with $f(a) > d$. Once IDA* visits this node, the entire subtree below it is pruned and $n$ will not even be generated. A potential node will actually be expanded only if *all* its ancestors are also potential nodes. This is guaranteed for consistent heuristics but not for inconsistent heuristics. Thus, for an inconsistent heuristic the number of potential nodes approximated by $N(b, d, P)$ is only an upper bound on the number of node expansions.[17]

Assume a given PDB and compare, for example, the dual (or random) lookup of this PDB to the regular consistent lookup of the same PDB. Since exactly the same PDB is used, all these heuristics (which perform a single lookup) will have the same overall distribution of values and the same number of potential nodes. However, as our experimental results below show, fewer nodes are expanded in practice. The explanation for this is shown in Figure 23. Observe that in both cases there is the same $h$-value distribution for each level of the tree. In particular, at depth two there are

17. Zahavi *et al.* developed an alternative formula to predict the number of node expansions [42, 43]. One of its benefits is that it provides accurate predictions for inconsistent heuristics too (as opposed to an upper bound).

three nodes with $h$-value of 3, and two single nodes with $h$-values of 4 and 6, respectively. In the case of a consistent heuristic (left side of the figure), if the current IDA* threshold is 5, all three nodes at depth two with $h$-value of 3 have $f = g + h = 2 + 3 = 5$ and will be expanded. They are all potential nodes, and since the heuristic is consistent and all their ancestors are also potential nodes, they are all expanded. The right subtree of the root is pruned because the $f$-value at level 1 is $f = g + h = 1 + 5 = 6 > 5$.

In the case of an inconsistent heuristic (right side of the figure), only one node at depth two will be expanded (the leftmost node). The node with $h$-value of 6 will be generated but not expanded because its $f$-value is 8 and that exceeds the threshold. Due to BPMX, its value will be propagated to its parent by Rule 3 and the parent's $h$-value will be changed to 5. The $f$-value of the parent will be changed to 6 and the search will backtrack without even generating the rightmost child (a potential node with $h = 3$).

## 7. Experiments with IDA*

This section presents results from different domains that illustrate the benefits of inconsistent heuristics and BPMX when used with IDA*. All experiments were performed on an Intel P4 3.4 GHz with 1 GB of RAM.

### 7.1 TopSpin

We experimented with the (17,4)-TopSpin puzzle which has $17! = 3.56 \times 10^{14}$ states. A PDB of the leftmost 9 tokens was built, representing a pattern space of $17 \times 16 \ldots \times 9 = 8.82 \times 10^9$.[18]

Given a PDB, 17 different symmetric (geometrical) lookups can be derived. For example, a PDB of 9 consecutive tokens ($[1 \ldots 9]$) can also be used as a PDB of $[2 \ldots 10]$, $[3 \ldots 11]$, etc., with an appropriate mapping of tokens. Since all the values in this PDB were smaller than 16, each entry is encoded in 4 bits. Hence the PDB only requires 247 MB of space.

Some pairs of operators are commutative, leading to the same state. When the search is done using IDA*, many duplicate nodes can be avoided by forcing two commutative operators to be applied successively in only one order. For example, the operator that reverses locations $(1, 2, 3, 4)$ is not related to the operator that reverses locations $(11, 12, 13, 14)$. By forcing the first one to always be tried before the second eliminates unnecessary duplication in the search tree. This operator ordering decreases the number of generated nodes by an order of magnitude and is applied across all our experiments.

Table 1 presents the average number of generated nodes and average time in seconds needed for IDA* to solve 1,000 random instances with different PDB lookup strategies. The first column (Lookups) shows the number of PDB lookups ($n \geq 1$) that were performed (and maximized). The following PDB-based heuristics were used:

- Regular: A fixed set of $n$ PDB lookups is chosen and used at every node in the search. Since all nodes maximize over the same lookups, the heuristic is consistent.

18. Since this puzzle is cyclic and the data is stored in a linear array, we can assume that token number 1 is always in the leftmost position. Thus, for the implementation, both numbers above can be divided by 17.

- Random: Of the 17 possible lookups, $n$ are randomly chosen (and maximized) at each node in the search. Since consecutive nodes have an $h$ value that is computed differently (possibly $n$ different lookups), the resulting heuristic is inconsistent.
- Random + BPMX: The heuristics obtained by combining (inconsistent) random lookups are updated with BPMX.

When multiple heuristics exist and IDA* is used then the following implementation enhancement can save a considerable number of potential heuristic lookups. For a node $n$ the heuristic lookup should determine whether $f(n) \leq T$ (does not exceed the threshold, meaning that $n$ should be expanded) or whether $f(n) > T$ (meaning that $n$ is pruned). When maximizing over multiple heuristics, instead of evaluating all the heuristics (*exhaustive* evaluation) the computation can stop when one of the heuristics exceeds $T$ (*lazy* evaluation); further lookups are not needed.[19] Lazy evaluation is not relevant for A* as the maximum needs to be calculated and stored. In addition, for IDA* too, for nodes that are expanded, *all* heuristics are looked up.

When BPMX is used, there is a benefit to always using exhaustive evaluations (similar to the variant described in Section 5.3.1). Exhaustive evaluations will often yield higher values than lazy evaluations, perhaps leading to additional BPMX cutoffs (higher values being propagated). Experiments on the performance of lazy and exhaustive evaluations have been done for many of the domains used in this research. In general for nodes where a lazy evaluation occurs, the time per node can drop significantly, as much as by a factor of three in some of our experiments. By contrast, exhaustive evaluations reduced the number of generated nodes, but this reduction was rather small and never more than 20%. All the results reported in this paper used lazy evaluations.

The first row in Table 1 corresponds to the benchmark case where only one lookup is allowed. The number of generated nodes is 40,019,429. Randomly selecting a single lookup reduces this number by a factor 25.5 to 1,567,769 nodes. Adding BPMX to this further reduces the number of generated nodes to 564,469—an improvement factor of 70.9 over the benchmark. This improvement was achieved with a single PDB lookup. The regular fixed selection method needs more than four different lookups (from the same PDB) to produce a heuristic of similar quality as the one using the random selection with BPMX (see Row 4). This is achieved with potentially three additional lookups, increasing the computational cost per node.

Adding more lookups provides diminishing returns. Using many lookups provide a diversity of heuristic values and the improvement factor of an additional lookup (regular or random) decreases. All the selection methods converge to the case of using all 17 lookups. The random selection of lookups converges faster. For a fixed number of potential lookups, the random selection strategy always outperforms the fixed strategy. When more lookups are possible, the relative advantage of the random selection decreases because the fixed selection also has a diversity of values. When one is interested in a time speedup, then many variants of both the regular (e.g., consulting all of them) and random lookups will provide the best time results of nearly 0.3 seconds. In practice, of course, all 17 symmetric lookups are possible and there is no reason not to use them all.

### 7.2 Rubik's cube

Rubik's cube has 20 movable cubes (cubies); 8 are corners and 12 are edges. The heuristic for Rubik's cube is usually obtained by taking the maximum over three PDBs (one for the eight corners

19. One can try to order the heuristics to increase the chance of getting a cutoff earlier [18].

| | Regular (Consistent) | | Random (Inconsistent) | | Random + BPMX (Inconsistent) | |
|---|---|---|---|---|---|---|
| Lookups | Nodes | Time | Nodes | Time | Nodes | Time |
| 1 | 40,019,429 | 53.129 | 1,567,769 | 2.857 | 564,469 | 1.032 |
| 2 | 6,981,027 | 10.686 | 404,779 | 0.865 | 279,880 | 0.622 |
| 3 | 1,787,456 | 3.213 | 224,404 | 0.555 | 187,797 | 0.480 |
| 4 | 651,080 | 1.394 | 157,710 | 0.443 | 143,051 | 0.411 |
| 5 | 332,642 | 0.835 | 123,882 | 0.388 | 116,779 | 0.373 |
| 6 | 208,062 | 0.601 | 103,377 | 0.356 | 99,653 | 0.349 |
| 7 | 148,003 | 0.484 | 89,698 | 0.337 | 87,596 | 0.332 |
| 8 | 116,208 | 0.422 | 79,911 | 0.324 | 78,609 | 0.321 |
| 9 | 95,863 | 0.382 | 72,504 | 0.317 | 71,709 | 0.315 |
| 10 | 81,749 | 0.354 | 66,690 | 0.311 | 66,184 | 0.310 |
| 11 | 71,451 | 0.335 | 62,020 | 0.306 | 61,682 | 0.306 |
| 12 | 64,227 | 0.322 | 58,119 | 0.304 | 57,947 | 0.304 |
| 13 | 58,455 | 0.312 | 54,906 | 0.302 | 54,773 | 0.303 |
| 14 | 53,926 | 0.307 | 52,145 | 0.301 | 52,079 | 0.302 |
| 15 | 50,376 | 0.303 | 49,760 | 0.302 | 49,736 | 0.302 |
| 16 | 47,784 | 0.303 | 47,688 | 0.301 | 47,663 | 0.303 |
| 17 | 45,849 | 0.304 | 45,848 | 0.303 | 45,849 | 0.304 |

Table 1: Consistent and inconsistent heuristics for TopSpin (17,4) (IDA*)

and two covering six edges [24]). The 8-corner PDB cannot be used in an inconsistent manner (all the corners are always examined; hence there are no symmetries and the dual lookup is identical to the regular lookup). This section reports results using a 7-edge PDB. There are 24 lines of geometrical symmetries which arise from different ways of rotating and reflecting the cube. For the 7-edge PDB, each of these symmetries considers a different set of edges, resulting in a different PDB lookup. Similar tendencies were observed in other experiments (based on PDBs built from a mix of edge and corner cubies).

Table 2 shows the average number of generated nodes and the average running time (in seconds) over the set of 100 Rubik's cube instances with goal distance of 14 used by Felner *et al.* [14]. The Lookups column gives the number of PDB lookups that were used to compute the heuristic value for a state. Lazy evaluation was used whenever possible.

The following PDB-based heuristics were used:

- Regular: The regular PDB lookup. This heuristic is consistent because the same set of cubies is used for the PDB lookup of both parent and child nodes.

- Dual: For each node, the dual state is calculated and is looked up in the PDB. This will produce inconsistent heuristic values because the dual lookup of the parent may consult different cubies than the dual lookup of the child.

| Row | Lookups | Heuristic | Nodes | Time |
|---|---|---|---|---|
| One PDB lookup | | | | |
| 1 | 1 | Regular | 90,930,662 | 28.18 |
| 2 | 1 | Dual | 19,653,386 | 7.38 |
| 3 | 1 | Dual + BPMX | 8,315,116 | 3.24 |
| 4 | 1 | Random | 9,652,138 | 3.30 |
| 5 | 1 | Random + BPMX | 3,829,138 | 1.25 |
| Maxing over multiple PDB lookups | | | | |
| 6 | 2 | 2 Regular | 13,380,154 | 4.91 |
| 7 | 2 | Regular + Dual + BPMX | 2,997,539 | 1.34 |
| 8 | 2 | 2 Random + BPMX | 1,902,730 | 0.83 |
| 9 | 4 | 4 Regular | 1,053,522 | 0.64 |
| 10 | 4 | 4 Random + BPMX | 1,042,451 | 0.64 |

Table 2: Consistent and inconsistent heuristics for Rubik's cube (IDA*)

- Random: Randomly select one of the different 24 possible symmetric PDB lookups for the given node. This is inconsistent because the set of cubies that are used for the parent are not necessarily the same as for the child.

The table shows that single random and dual lookups perform much better than a single regular lookup. In addition, BPMX further improves the results. The dual lookup is much more diverse than the regular lookup and there is much less correlation between successive lookups [14]. Therefore, the search is not stuck in a region with low heuristic values as frequently happens with regular lookups. A random lookup with BPMX is much faster than either one regular lookup (by a factor of 24) or one dual lookup with BPMX (by a factor of 2.5).

Rows 6–10 show the results of maximizing over 2 and 4 regular and random lookups. It is interesting to see that one random lookup with BPMX outperforms two regular lookups by a factor of 3.5 in the number of generated nodes and by a factor of 3.9 in time. Two random lookups are better than a regular and a dual lookup because it has a better diversity of values (see below). When four lookups are allowed, the values obtained using our four regular lookups are diverse enough that there is no advantage to taking four random lookups.

#### 7.2.1 Dynamic distribution of heuristic values

We claimed above that part of the reason for the success of an inconsistent heuristic is the diversity of values that get introduced into the search. This section attempts to give greater understanding to this claim.

It is easy to analyze a domain and produce a graph showing the distribution of values produced by a heuristic. However, the obvious question to ask is whether this static (pre-computed) distribution reflects the values that are actually seen during a search. Of interest is the *dynamic distribution* of values generated during a search. Distinguishing between static and dynamic distributions of heuristic values is not new; it has been previously used to explain why the maximum of several weak heuristics can outperform one strong heuristic [18].

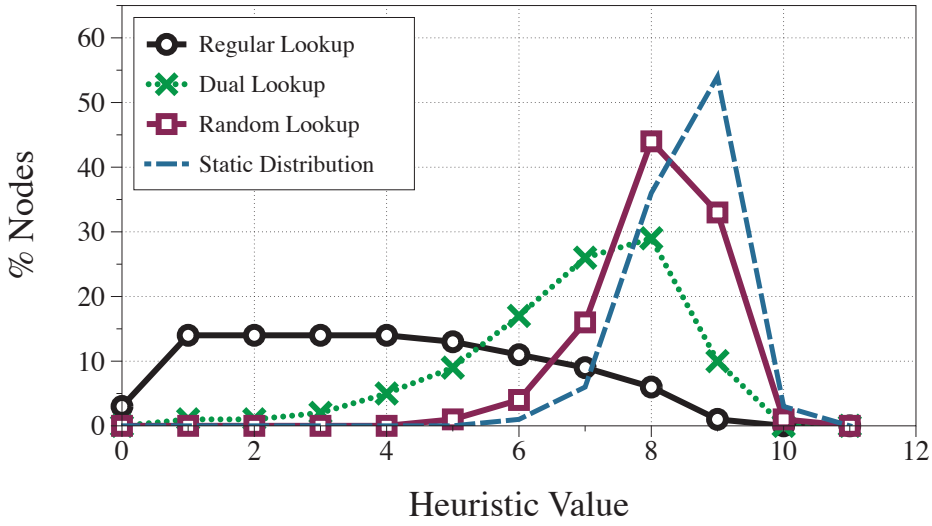


Figure 24: Rubik's cube heuristic distributions

Figure 24 shows the dynamic distribution of the heuristic values seen during the searches reported in Table 2, as well as the static distribution of values in the PDB used. The following observations can be made from these results. First, there is a dramatic difference between the static and dynamic distribution of values for the regular (consistent) heuristic. As can be seen, the dynamic distribution for the regular lookup is greatly shifted towards the smaller heuristic values, compared to their static distribution in the PDB. This phenomenon was discussed and explained by Holte *et al.* [18]. The main reason for this is that most of the generated nodes are deep in the search tree, and their values are necessarily small to be generated at all. Second, it is easy to recognize that the heuristic with the best performance also had a superior (shifted to the right) dynamic distribution of heuristic values. Note that all these versions used exactly the same PDB represented by the (overall) static distribution of values. Third, the regular heuristic had a poor dynamic distribution because it is consistent; when the heuristic value for a state is low, the children of that state must also have low values. Inconsistent heuristics do not have this problem; a node can receive any value, meaning that the distribution of values seen is closer to the static distribution of the PDB. Finally, inconsistency has the effect of improving the dynamic distribution towards that of the static distribution. The greater the degree of inconsistency, the closer the dynamic distribution approaches the static distribution of the PDB.

### 7.2.2 Dynamic branching factor and BPMX

The effectiveness of BPMX can be characterized by its effect on the branching factor during the search. The dynamic branching factor (DBF) is defined as the average number of children that are generated for each node that is expanded in the search. When the heuristic function is inconsistent and BPMX is employed, the dynamic branching factor can be smaller than the normal branching factor.

| Choice | Lookups | DBF | Nodes | BPMX Cuts |
|---|---|---|---|---|
| 1 | 1 | 13.355 | 90,930,662 | 0 |
| 2 | 1 | 9.389 | 17,098,875 | 717,151 |
| 3 | 1 | 9.388 | 14,938,502 | 623,554 |
| 4 | 1 | 9.382 | 14,455,980 | 598,681 |
| 5 | 1 | 7.152 | 5,132,396 | 457,253 |
| 8 | 1 | 7.043 | 4,466,428 | 402,560 |
| 12 | 1 | 7.036 | 3,781,716 | 337,114 |
| 16 | 1 | 6.867 | 3,822,422 | 356,327 |
| 20 | 1 | 6.852 | 3,819,699 | 357,436 |
| 24 | 1 | 6.834 | 3,829,139 | 360,067 |
| dual | 1 | 7.681 | 8,315,117 | 796,849 |

Table 3: Rubik's cube: random heuristic with BPMX

Table 3 presents DBF results for Rubik's cube obtained using the 7-edge PDB. An experiment was performed where the number of possible PDB lookups was varied, but only a single lookup was used by randomly selecting from this set. The first column gives the number of available heuristics to randomly select from. The other three columns show results averaged over the same set of instances of Table 2.

In the first row, only one PDB lookup was used. Since the same PDB lookup was performed at all nodes, this benchmark case is for a single consistent regular heuristic. The dynamic branching factor here is equal to the actual branching factor, 13.355, once redundant operators are removed (consistent with the results of Korf [24]).

As the number of possible heuristic lookups increases, the DBF decreases. This results in a significant reduction in the number of generated nodes. Note two phenomena in these results. First, the range of heuristic values in Rubik's cube is rather small, as can be seen in Figure 24. Thus, the potential for a large difference between a parent's heuristic value and its children's is small. Even in this domain inconsistency caused a dramatic performance improvement. Second, no extra overhead is needed by these heuristics as only a single PDB lookup is performed at each node. Thus, the reduction in the number of generated nodes is fully reflected in the running times.

### 7.3 The 15-puzzle

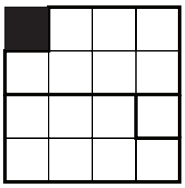

Figure 25: BPMX (a 7-7-1 partitioning into disjoint sets of the 15-puzzle)

| Compress | BPMX | Nodes | Time | Av. h | Memory |
|---|---|---|---|---|---|
| - | - | 464,978 | 0.058 | 43.59 | 536,871 |
| + | - | 565,881 | 0.069 | 43.02 | 268,435 |
| + | + | 526,681 | 0.064 | 43.02 | 268,435 |

Table 4: Results on the 7-7-1 partitioning of the 15-puzzle

Another source of inconsistency can be data compression (Section 6.2). Previous research that compressed PDBs of the 15-puzzle [11] used a 7-7-1 additive partitioning (shown in Figure 25). These experiments were repeated, however this time BPMX was used. The results (averaged over the same set of 1000 random instances first used by Korf and Felner [26]) are reported in Table 4. The first line corresponds to a regular PDB using a 7-7-1 partitioning (536 MB of memory used by PDBs that were represented with a sparse mapping [11]). In the second line the PDB is compressed to roughly half its size (268 MB). Due to the resulting loss of information, the number of nodes generated increased by 100,903 (from 464,978 to 565,881) agreeing with previous results [11]. Compressing the PDBs can produce inconsistency and this is born out by the BPMX results in the third line (a decrease in the number of nodes generated to 526,681). At first glance, this seems like a modest reduction of less than 10%. However, a different way of viewing this is that BPMX reduced the loss of information introduced by compression by 40%, from 100,903 to 61,703. This was done with no additional cost in memory or time.

### 7.4 The 24-puzzle

| Row | Lookups | Heuristic | Nodes | Time |
|---|---|---|---|---|
| One PDB lookup | | | | |
| 1 | 1 | Regular | 26,630,050,115 | 15,095 |
| 2 | 1 | Dual | 24,155,327,789 | 20,105 |
| 3 | 1 | Dual + BPMX | 18,188,651,278 | 10,761 |
| 4 | 1 | Random | 3,386,033,015 | 3,040 |
| 5 | 1 | Random + BPMX | 1,938,538,739 | 1,529 |
| Two PDB lookups | | | | |
| 6 | 2 | Regular + Regular* | 1,631,931,544 | 1,483 |
| 7 | 2 | 2 Randoms + BPMX | 908,186,066 | 1,065 |
| Three PDB lookups | | | | |
| 8 | 3 | Regular + Regular* + Dual + BPMX | 852,810,804 | 1,142 |
| 9 | 3 | 3 Randoms + BPMX | 818,601,469 | 1,022 |
| Four PDB lookups | | | | |
| 10 | 4 | Regular + Regular* + Dual + Dual* +BPMX | 751,181,974 | 1,331 |

Table 5: Results on the 24-puzzle

We now present results on the 24-puzzle using 6-6-6-6 additive PDBs [26]. Similar tendencies were observed for the 15-puzzle with the 7-8 additive PDBs [26]. The results in Table 5 are aver-

aged over the 10 instances with the smallest solution length from standard 50 random states [26]. Four heuristics are available based on 6-6-6-6 additive PDBs [26] (Figure 8): regular lookup, regular lookup reflected about the main diagonal (indicated by a * in the table), dual lookup, refection of the dual lookup. The random heuristic randomly chooses a heuristic from the set of these four heuristics. A single dual or random lookup outperforms the regular lookup. We showed in Section 7.1 that there is a diminishing return for adding more lookups for both the regular and random case. In the 24-puzzle, adding more lookups (up to the maximum of four) was beneficial. While the smallest number of nodes was achieved by using all four lookups, the best time was obtained by an inconsistent heuristic using two or three lookups.

### 7.5 The Pancake puzzle

| Row | Lookup | Nodes | Time | DBF |
|---|---|---|---|---|
| Normal Operator Order | | | | |
| 1 | Regular | 342,308,368,717 | 284,054 | 15.00 |
| 2 | Dual | 27,641,066,268 | 19,556 | 15.00 |
| 3 | Dual + BPMX | 14,387,002,121 | 12,485 | 10.11 |
| 4 | Regular + Dual + BPMX | 2,478,269,076 | 3,086 | 10.45 |
| Operators Ordered by Average Heuristic Difference | | | | |
| 5 | Regular | 113,681,386,064 | 95,665 | 15.00 |
| 6 | Dual | 13,389,133,741 | 9,572 | 15.00 |
| 7 | Dual + BPMX | 85,086,120 | 74 | 4.18 |
| 8 | Regular + Dual + BPMX | 39,563,288 | 49 | 5.93 |

Table 6: 17-pancake results

Table 6 shows results for IDA* optimally solving 10 random instances of the 17-pancake puzzle with a PDB of 7 pancakes. There are no geometrical symmetric PDB lookups in this domain; and the only way to achieve inconsistency is with the dual lookup. Rows 1–3 have a single PDB lookup. The dual heuristic reduces the number of nodes generated by more than a factor of 12. This improvement is the consequence of the larger diversity of inconsistent heuristic values encountered in the search. When BPMX is used with the dual heuristic, the number of nodes generated is further reduced by almost a factor of two, the result of the dynamic branching factor falling from 15 to 10.11. The best results (row 4) are achieved by performing two lookups (regular and dual) and using BPMX to propagate inconsistencies. This combination produces a 138-fold reduction in nodes generated over the regular lookup on its own.

#### 7.5.1 Operator ordering to increase BPMX cutoffs

Consider the following insight which can be used to further enhance performance in some domains. If a node has a child that would cause a BPMX cutoff, it should be generated as early in the set of children as possible. This would allow the cutoff to be made before subtrees under the other children are searched. If different operators tend to create inconsistency at different rates, the search could be sped up by ordering the operators accordingly. The operators on Rubik's cube and TopSpin are

symmetric and it is difficult to find a useful way to order them. This is not the case for the pancake puzzle; each operator differs in the number of pancakes moved.

| Operator | Regular | Dual |
|---|---|---|
| 2–10 | 0.370 – 0.397 | 0 |
| 11 | 0.396 | 0.613 |
| 12 | 0.397 | 0.958 |
| 13 | 0.400 | 1.165 |
| 14 | 0.401 | 1.291 |
| 15 | 0.402 | 1.358 |
| 16 | 0.411 | 1.376 |
| 17 | 0.216 | 1.321 |

Table 7: Average Heuristic Difference (AHD) of the operators of the 17-pancake puzzle

We introduce a new term, the *average heuristic difference* (AHD). The AHD($op_h$) for a given operator $op$ and heuristic $h$ is the average, over all states $s$ to which $op$ can be applied, of $|h(s) - h(op(s))|$. To estimate the AHD of an operator, a random state was chosen ($s_1$) and then the relevant operator was applied to this state (yielding state $s_2$). The difference in the heuristic value between $s_1$ and $s_2$ was measured. This was repeated for 100 million different states. Table 7 shows the AHD results for the operators of the 17-pancake puzzle. The Regular column presents the AHD for each operator when the regular lookup was performed and the Dual column presents the AHD for the dual PDB lookup.

The regular PDB lookup is consistent and therefore cannot have an AHD greater than 1. For the dual PDB lookups the results are more interesting. Operators 2–10 all have AHD values of exactly 0, an artifact of the particular PDB used for these experiments. The PDB is based on locations 11–17 and moves which did not affect any of these locations (operators 2–10) could not cause a change in the dual heuristic [14, 45]. However, for larger operators (13–17), the AHD for the dual lookup was more than 1. Note that operator 16 has a larger AHD than operator 17, even though it changes a smaller number of locations.

In Table 6, the results for rows 1–4 were obtained by using the operators in the order of most to least tokens moved. For rows 5–8, the operators were ordered in decreasing order of AHD of the dual lookup, as measured in Table 7. Even when BPMX is not used (compare rows 5 and 6 to rows 1 and 2), significant improvements are seen. When BPMX is used, AHD ordering roughly halves the DBF and dramatically reduces the number of generated nodes (compare rows 7 and 8 to rows 3 and 4). The best result (regular and dual lookup, enhanced with BPMX and AHD ordering—row 8) reduces the number of generated nodes by four orders of magnitude as compared to doing the usual single regular lookup.[20]

20. We use simple PDBs for the pancake puzzle to demonstrate the benefits of inconsistent heuristics. However, enhanced PDB methods [41, 17], as well as the domain specific *gap* heuristic [16], have been developed for this problem. Applying our techniques on top of these heuristic will likely show similar performance gains. In fact, an advantage of using BPMX and random heuristics for this application has already been demonstrated with one of these recent PDBs [17].

## 8. Pathfinding Experiments with A*

A* has different properties than IDA*. To properly assess inconsistent heuristics in the A* setting, we need an application domain for which A* is well suited. Whereas IDA* is the algorithm of choice for combinatorial puzzles, A* is preferred for pathfinding in explicit state spaces. In this section we will demonstrate that inconsistent heuristics can incur significant overhead in A* if BPMX is not used. We then demonstrate a number of cases when inconsistent approaches can outperform consistent approaches.

The application domain is a set of 75 grid maps from commercial games, all scaled to grids of size 512×512. Each location on the map is either blocked or unblocked. On each map, problems are broken into 128 buckets according to the optimal path length, with path lengths varying between 1 and 512. We randomized 1,280 problem instances from each map (different start/end locations). The agent can move horizontally, vertically or diagonally (eight possible directions). All experiments in this section were conducted on a 2.4GHz Intel Core 2 Duo with 4GB RAM. Most of our reported results are only for BPMX(1) (see Section 5.3.2) and in this section when the term BPMX is used without a parameter it refers to BPMX(1).

All running times are measured in seconds.

### 8.1 Pathfinding Heuristics

*Octile distance* is the most common heuristic in this domain. If the distances along the $x$ and $y$ coordinates between two points are $(dx, dy)$, then the octile distance between them is $\sqrt{2}*\min(dx, dy)+|dx - dy|$. This is the optimal distance between the two points if 1) there were no restrictions from obstacles or boundaries, and 2) you are allowed to go to any of your neighbors in all eight possible directions (including diagonals). The octile heuristic is consistent and does not require any memory.

#### 8.1.1 True distance heuristics

True-distance heuristics (TDHs) are memory-based heuristics that were recently developed for pathfinding applications [40, 13]. An example of a TDH is the *differential heuristic* [40] (DH) which is built as follows: choose $K$ *canonical* states from the domain; compute and store the shortest path distance from all $K$ canonical states to all other reachable states. For each canonical state, $S$ memory is required, where $S$ is the number of states in the state space. For the $i$th canonical state, $k_i$, an admissible heuristic for any two points $a$ and $b$ can be obtained using $h_i(a, b) = \max(|c(a, k_i) - c(b, k_i)|, octile(a, b))$, where $c(x, y)$ is the shortest path from $x$ to $y$ that is stored in the database. Because $c(a, b) + c(b, k_i) \geq c(a, k_i)$ for any $a$, $b$ and $k_i$, it follows that $c(a, b) \geq c(a, k_i) - c(b, k_i)$. Hence $|c(a, k_i) - c(b, k_i)|$ is an admissible consistent heuristic for the distance between $a$ and $b$.

$|c(a, k_i) - c(b, k_i)|$ can sometimes produce a heuristic value higher than the octile heuristic. For example, this can happen when $b$ is on the optimal path from $a$ to $k_i$, and the exact distance from $a$ to $b$ is larger than the octile distance. However, DH can sometimes produce values smaller than the octile distance. Taking the maximum of the DH and the octile heuristic guarantees that the new heuristic dominates the octile heuristic.

For a given state, if one takes the maximum of all available differential heuristics, the resulting value is a consistent heuristic. However, if a random subset of the available heuristics is considered then the resulting value will be inconsistent.

### 8.1.2 INTERLEAVED DIFFERENTIAL HEURISTICS

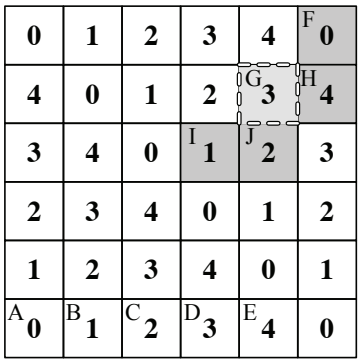


Figure 26: Interleaved differential heuristics

We introduce the *interleaved differential heuristic* (IDH), a convenient way to get most of the benefits of multiple DHs (i.e., with multiple canonical states) but with the storage of only one (similar to what was done in [5]). Consider having five DHs 0...4. Instead of storing the distances to all five canonical states at all states, only store a single distance at each state. Consider Figure 26 with an empty grid. Each cell is labeled with the canonical state whose distance is stored at that state. In this setup, $S$ memory is used to store portions of all five heuristics, but the search can benefit from all of them as follows.

A heuristic value between two states is only available if the distance to the same canonical state is stored at each state. Thus, if the current node being expanded is state $D$ at the bottom of the grid and the goal is state $G$ (dotted border), then DH can be used to directly lookup a heuristic value between these two states (both use canonical state 3). However, if state $A$ is being expanded, no differential heuristic between $A$ and $G$ can be directly computed (different canonical states). However, neighboring $G$ is state $F$ for which such a heuristic can be computed. Thus, $h(A, G) = |c(A, k_0) - c(F, k_0)| - c(F, G)$, for canonical state $k_0$. There are many lookups that can be performed using both the neighborhood of the current search node and the neighborhood of the goal. More lookups will improve the heuristic estimate but take more time. In this work only lookups from the current state to neighbors of the goal are performed.

The final heuristic is the maximum of the computed IDH heuristic and the octile heuristic. Because different heuristics are used at each state, the overall heuristic is inconsistent.

For further efficiency and improved performance, when we use IDHs we perform a small breadth-first search starting at the goal until one possible state is found for each possible DH lookup. We cache all distances and associated errors. Then, for any given node during the search we perform a single lookup in the cache to lookup a heuristic value. This cache approach is efficient because the identity of the neighbor state is unimportant; only the distance and additional error are needed. Therefore the cache is the size of the number of interleaved heuristics, not the number of neighbors of the goal.

### 8.2 Random heuristic

The first set of experiments illustrates the effect of BPMX on A* when an inconsistent heuristic is used. Three heuristics are compared:

(a) octile distance (consistent, used as the baseline);

(b) ten DHs (10 canonical states) were built and in a given state one was randomly chosen to use (inconsistent, called *random*). This was done with and without BPMX; and

(c) maximum of all ten DHs (consistent, best possible heuristic).

The memory needs for 10 DHs (for both random and maximum) is $10S$.

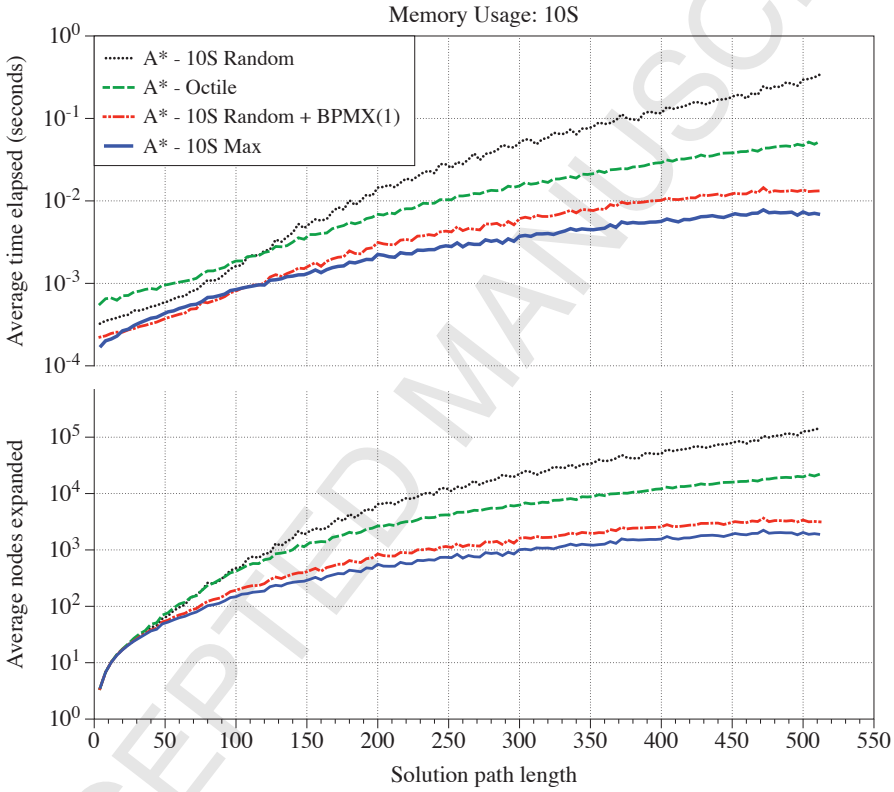


Figure 27: Nodes expanded in pathfinding with $10S$ memory

Figure 27 presents experimental results for the number of nodes (bottom) and the CPU time (top). The problem instances were partitioned into buckets based on their solution length. The $x$-axis presents the different solution lengths, with each point being the average solution length of a bucket. The $y$-axis is the number of node expansions using a logarithmic scale.

As expected, A* with the max of all possible heuristics expands the fewest number of nodes. A* with the random heuristic and no BPMX expands the most. The random heuristic can never produce

worse heuristic values than the octile heuristic. However, random without BPMX performs almost an order of magnitude more node expansions than the octile heuristic due to node re-expansions. From the slope of the lines, it appears that random without BPMX adds a slight polynomial overhead to A*. When BPMX is added to random it performs better than A* with the octile heuristic, and the performance is quite close to the consistent (max) heuristic. This shows that BPMX is effective at overcoming the node re-expansion problem. It is impressive that BPMX enhances random, with its single lookup, to achieve nearly the performance of max, with its ten lookups.

Timing results (top of the figure) show similar trends. Random (with and without BPMX) has faster lookup times than the max heuristic (fewer heuristics are consulted) and for short paths has better time performance. Unlike IDA*, where the algorithm only needs to know whether a cutoff occurs, in A* the correct maximum value is needed. Hence, lazy evaluation is not possible in A*.

#### 8.2.1 Fixed number of lookups

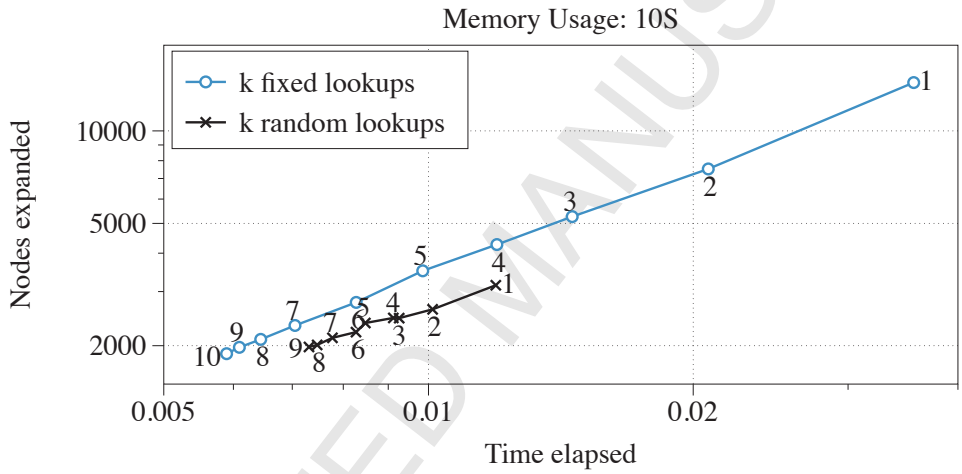


Figure 28: Comparing fixed and random lookups

Figure 28 presents a comparison where $k$ heuristics were used. These $k$ lookups were either fixed (same heuristic used at all nodes; consistent) or randomized (random selection at a given node; inconsistent) out of 10 available heuristics. BPMX was used for the inconsistent heuristics. In this experiment only the problems with the longest solutions on each map were considered. Each point represents the average of approximately 500 instances, plotting both time and node expansions on logarithmic scales. The top curve shows the performance given $k$ fixed lookups are used, while the bottom curve shows the performance using $k$ random lookups.

This experiment shows that if the number of lookups is fixed, then a random strategy is better than a fixed strategy for low values of $k$, as more diversity is added to the resulting heuristic. When $k$ increases, the significance of this effect decreases as more lookups implicitly adds more diversity of values with the fixed lookups too. In this domain, the consistent max-of-10 heuristic achieved the best time results.

The results are explained by a number of key differences between using maps and using combinatorial puzzles as application domains. These differences cause more difficulties to achieve speedup in the search with inconsistent heuristics in the pathfinding domains than in the puzzle domains.

- **Memory:** Unlike the permutation puzzles, where $k$ lookups (random or fixed) need the same amount of memory, here $k$ fixed lookups need $kS$ memory, while $k$ random lookups (out of 10) need $10S$ memory. $k$ fixed lookups uses less memory and therefore may have better memory and cache performance. This explains why when $k \geq 7$ the fixed lookups have a lower search time than the random lookups.
- **Indexing time:** In the puzzle domains the cost of determining where to find a heuristic lookup is relatively expensive. Typically a non-trivial indexing function is needed, and possibly the application of one or more symmetries and/or permutations. In the map domain, indexing is easier to compute, leading to (slightly) faster lookup times.
- **Node re-expansion:** In IDA* (as used in the puzzle domains), an inconsistent heuristic does not affect performance. In A* (as used in the maps domain), the problem of node re-expansions can cause problems (even when BPMX is used).

Thus, the maximum of 10 heuristic would be the best choice in the pathfinding domain both in nodes and in time, while in the puzzles inconsistent heuristics with fewer lookups might yield faster times (e.g., in our 24-puzzle results).

Given a new domain, these are important factors which determine whether inconsistent approaches will be successful. In Section 8.3 we show that the interleaved heuristic allows multiple lookups and improves the performance of differential heuristics using inconsistency and BPMX.

### 8.2.2 Varying the Number of Lookups Performed

This section examines the performance of random lookups as the amount of available memory (and number of heuristics) increases up to $100S$. Three approaches for performing lookups are considered:

(a) take the maximum of all available heuristics at each node,

(b) take the maximum of 10% of the available heuristics at random (and use BPMX), and

(c) take the maximum of 20% of the available heuristics at random (and use BPMX).

For example, if 10 heuristics are available ($10S$ memory), then 1 or 2 lookups are performed at each state for the 10% and 20% heuristics, respectively. Similarly, if 50 heuristics are available, 5 or 10 lookups are performed for the 10% and 20% heuristics respectively.

Figure 29 shows three curves, one for each approach. Between 10 and 100 differential heuristics were built, increasing by intervals of 10. The nodes expanded and time elapsed to solve the hardest problems (length 508–512) on each map are computed and compared for each of the three approaches. The points which correspond to having 10, 50 and 100 differential heuristics available are labeled on each curve. This is a log-log plot, making the differences easier to see.

Consider the consistent heuristic which takes the maximum over all available heuristics. With 10 differential heuristics, an average of 7 milliseconds is needed to complete a search with 1,884 node

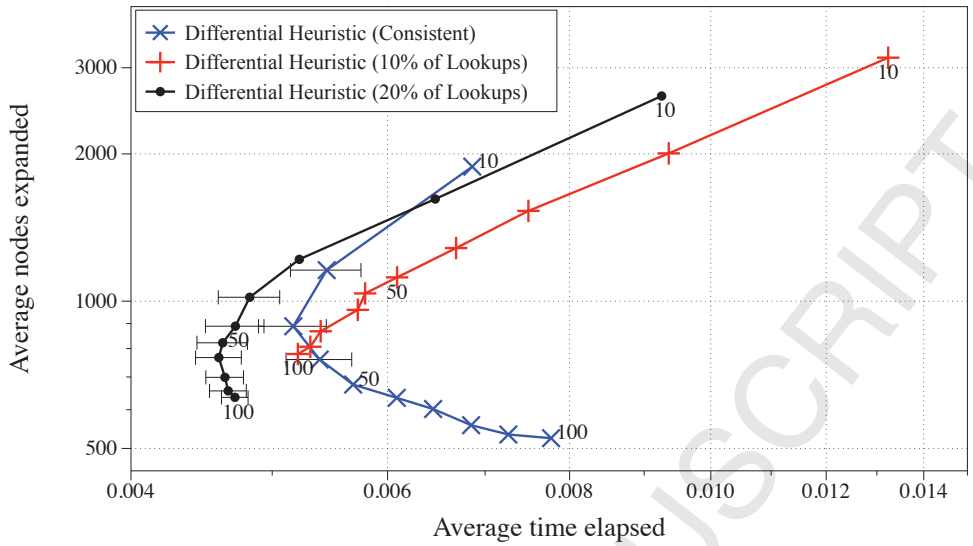


Figure 29: Time versus nodes tradeoff as more heuristics are available

expansions. As more differential heuristics are used, the number of node expansions monotonically decreases. However, execution time only decreases until 30 differential heuristics are used, at which point the cost of performing additional lookups overtakes the reduction in nodes expanded.

Randomly using only 10% of the available heuristics is inconsistent. This curve begins with the worst performance of the three in terms of both nodes and time. However, when 10 random heuristics (out of 100) are used it is able to match the best time performance of the consistent heuristic (30 lookups) and is faster than the consistent heuristic with 40 or more lookups.

Randomly using 20% of the available heuristics matches the time performance of 30 consistent lookups when performing only six random lookups (both use $30S$ memory). The fastest performance is when $60S$ memory is available (12 lookups are performed) and is significantly better than the best fixed lookup result. The error bars on this curve correspond to 95% confidence intervals, showing that this result is statistically significant, albeit by a small margin.

### 8.3 Interleaved Differential Heuristics

In this section, experimental results comparing a number of approaches that all use $1S$ memory are reported. The octile heuristic is used as the baseline (using the same results shown previously). A single consistent lookup is compared to an interleaved differential heuristic (IDH, defined in Section 8.1.2) built using 10 differential heuristics ($1S$ memory). Figure 30 presents the node expansions and the CPU time for these approaches. The results are plotted as a function of the solution length. Unlike previous figures, both axes have linear scales.

The nodes expanded and timing results results reinforce the earlier discussion. BPMX is critical to achieving good performance. Again, even though the heuristic values are no worse than that of the octile heuristic, the performance of the interleaved (inconsistent) heuristic without BPMX is

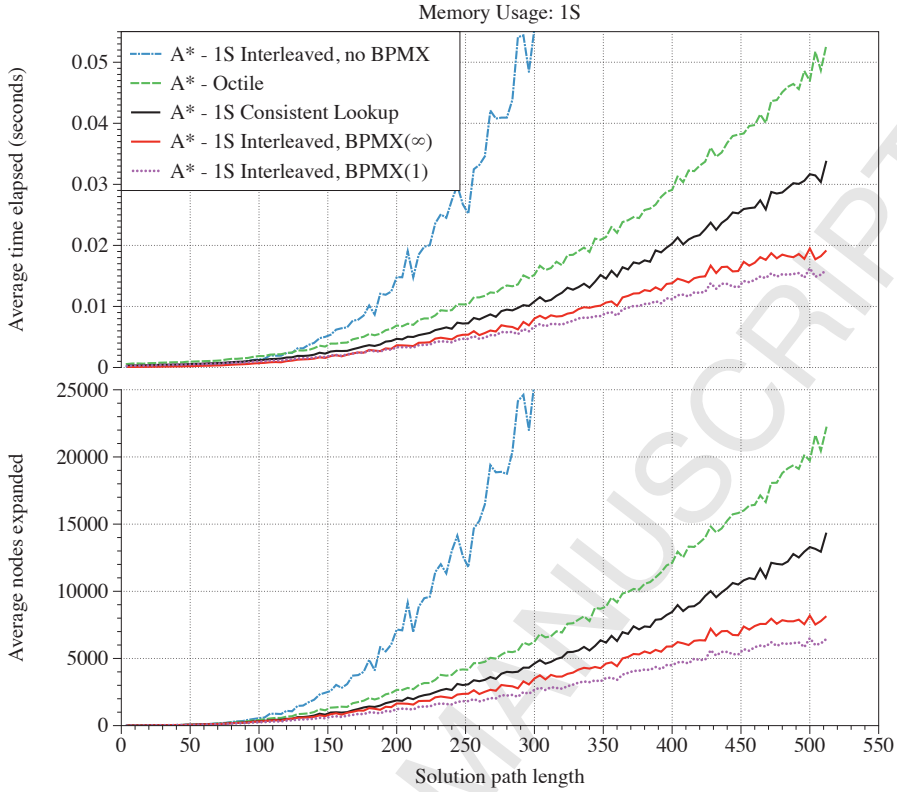


Figure 30: Nodes expanded with $1S$ memory

poor (roughly by a factor of 10 for the hardest problems). Both versions of the interleaved heuristic with BPMX outperform the consistent DH heuristic. BPMX(1) was better than BPMX($\infty$). When $1S$ memory is available for this domain, an inconsistent heuristic outperforms a consistent heuristic and produces the best results. The curves for timing results maintain their same orderings and the same relative performance, supporting that BPMX node expansions (for $d > 1$) should count the same as A* node expansions

Figure 31 shows the results for using 80 DHs interleaved into $10S$ memory. This is compared to $10S$ memory for 10 fixed lookups. The consistent heuristic with 10S is relatively informed but a slight reduction was achieved by the inconsistent heuristic.

### 8.4 Different degrees of BPMX

The final experiment examines the effect of increasing the BPMX propagation depth. Figure 32 shows the effect of using different BPMX propagation depths. The set of problems from Figure 27 are used, plotting the number of nodes expanded as a function of the solution length. The heuristic is a random selection from the 10 available DHs. *Node expansions* refers to each time the neighbors of a node are generated and looked up in OPEN or CLOSED. As explained above, this process is

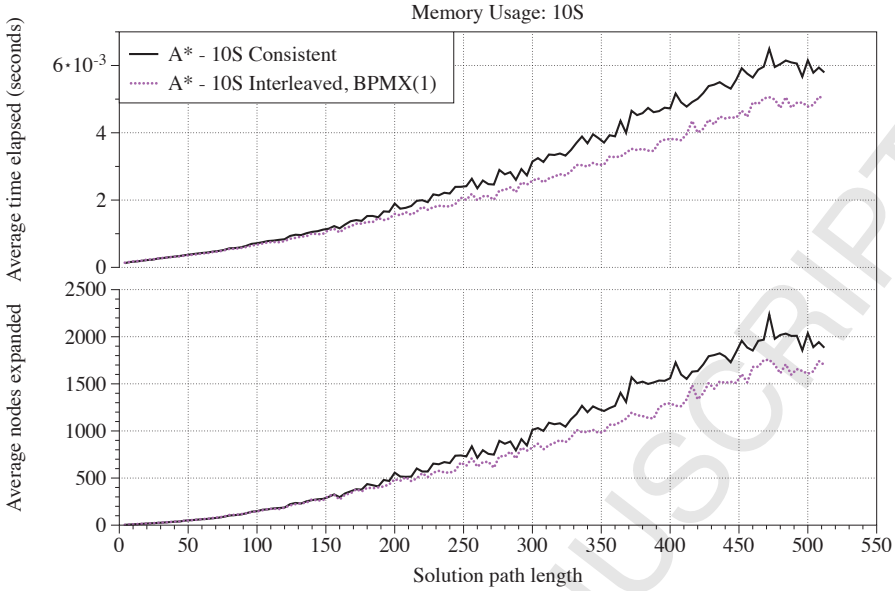


Figure 31: Average nodes expanded and time with $10S$ memory

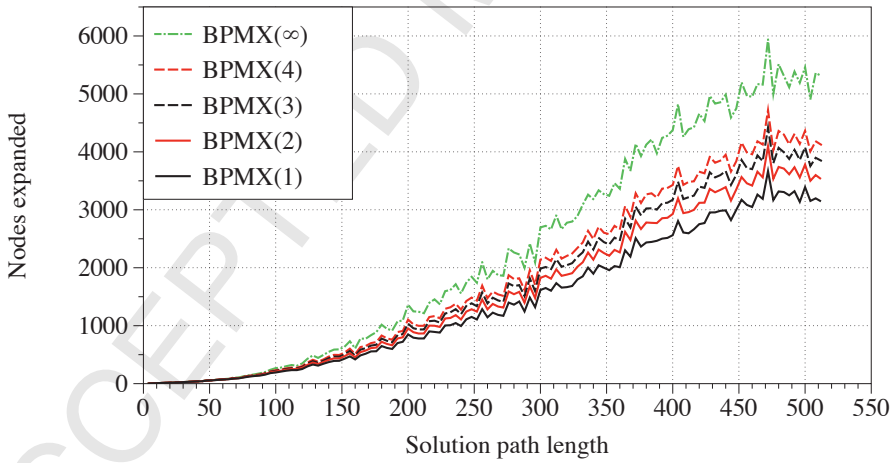


Figure 32: Average nodes expanded for different degrees of BPMX propagation

exactly the same in BPMX($k$) for $k > 1$ as in a regular A* expansion. Hence, BPMX expansions are counted the same as A* expansions. Time results are omitted as they show the same trend.

For this domain and for this heuristic, BPMX(1) was the best. Larger values of $k$ do not help on average, as sufficiently large heuristics seem to always be within one step. In a sense this is fortunate, as BPMX(1) is easy to implement and produces the best results.

## 9. Discussion and conclusions

Historically, inconsistent heuristics have been generally avoided when searching for optimal solutions because of the cost of re-expanding closed nodes with A* and the belief that inconsistent heuristics are hard to concoct. This paper has demonstrated that effective inconsistent heuristics are easy to create, can be integrated into IDA* and A*, and that the benefits of doing so often substantially reduce the search effort. This represents an important change to the conventional wisdom for heuristic search.

IDA* re-expands nodes whether the heuristic is consistent or not, so using inconsistent heuristics does not hurt its behavior. We showed that A*'s worst-case exponential behavior is only valid under unrealistic graph settings. Furthermore, we generalized the known pathmax propagation rules to bidirectional pathmax (BPMX) and showed that BPMX can lead to further performance gains.

Indeed, experimental results showed major performance gains with inconsistent heuristics for IDA* and for A*. For all domains where IDA* was used (the puzzle domains), a very large reduction (more than an order of magnitude for many cases) in the number of generated nodes (and CPU time) was obtained when a single inconsistent heuristic was used instead of a single regular consistent heuristic. This was the consequence of introducing more diversity into the heuristic values encountered in a search. A further reduction in the number of generated nodes was obtained when BPMX was implemented on top of the inconsistent heuristic, as one large heuristic value might influence the entire neighborhood of states.

In all the domains studied in this paper, more than a single heuristic is available either due to internal symmetries of the same PDB (in the puzzles) or to manually creating more heuristics (in the pathfinding domain). When multiple heuristics exist, clearly taking the maximum of all heuristics provides the best heuristic value for all states and generates the fewest nodes. This comes with an increase in the runtime overhead per node because of the cost of the additional lookups. More heuristics being considered increases the diversity of heuristic values, reducing the number of node expansions. Therefore, when multiple heuristics are available and more lookups are performed, the performance advantage of inconsistent over consistent heuristics decreases. The results presented in this paper vary, in part, because of the number of heuristics available in each of the experimental domains.

For TopSpin, the relative advantage of inconsistent heuristics over regular heuristics remains valid for a large range of the number of multiple lookups that are perfomed. In practice, one might use all 17 lookups as they perform equally to other variants of the random lookups. For Rubik's cube when four lookups are possible the advantage of inconsistent heuristics disappears.

For the 24-puzzle, the maximum number of lookups is four. Two or three random lookups were shown to outperform (in time) the maximum of all four. For the pancake puzzle, only a single lookup exists and the use of inconsistent heuristics produced spectacular gains.

When A* is used (in the pathfinding domain), the problem of node re-expansion arises. This issue can cause a single (random) inconsistent heuristic to generate more nodes than a consistent (octile) heuristic, even though the inconsistent heuristic returns a superior heuristic value for all states. When BPMX is added, the inconsistent heuristic (random) outperforms a single consistent

heuristic and is almost as good as the maximum of ten heuristics (both in node expansions and time). However, the max of 10 heuristics is still the best choice.

It is more difficult to obtain a speed up when using multiple heuristics in an inconsistent manner in the pathfinding domain than in the puzzle domains for a number of reasons. First, the number of states in the pathfinding domain grows quadratic in the depth of the search while in the puzzle domains it grows exponentially—there is more room for improvement. Second, no symmetries are possible in the pathfinding domain and more (potential) lookups need more memory. Third, the lookup time is much smaller than a PDB lookup, so performing multiple lookups is not as costly. Finally, there is the issue of node re-expansions. However, despite this complication, in this domain too the use of an inconsistent heuristic provides the best results. An inconsistent heuristic that is based on interleaving a number of heuristics was shown to outperform a consistent heuristic given the same amount of memory.

The major result of this paper is the demonstration that inconsistent heuristics can increase the diversity of values encountered in a search, leading to improved performance. Based on these results, it is our expectation that the use of inconsistent heuristics will become an accepted and powerful tool in the development of high-performance search algorithms.

A number of directions remain for future research. Identifying more ways for creating inconsistent heuristics will help make their usage more common and beneficial. As well, more research is needed on different variations of BPMX. In particular, different levels of lookahead searches for finding large heuristic values might result in better overall performance.

### 9.1 Acknowledgments

This research was supported by the Israel Science Foundation (ISF) under grants number 728/06 and 305/09 to Ariel Felner. The research funding from Alberta's Informatics Circle of Research Excellence (iCORE) and Canada's Natural Sciences and Engineering Research Council (NSERC) is greatly appreciated.

## References

[1] A. Bagchi and A. Mahanti. Search algorithms under different kinds of heuristics – A comparative study. *Journal of the ACM*, 30(1):1–21, 1983.

[2] M. Ball and R. C. Holte. The compression power of symbolic pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS-08)*, pages 2–11, 2008.

[3] B. Bonet and H. Geffner. Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to MDPs. In *International Conference on Automated Planning and Scheduling (ICAPS-06)*, pages 142–151, 2006.

[4] T. Chen and S. Skiena. Sorting with fixed-length reversals. *Discrete Applied Mathematics*, 71(1–3):269–295, 1996.

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

[6] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

[7] C. Domshlak, E. Karpas, and S. Markovitch. To max or not to max: Online learning for speeding up optimal planning. In *AAAI Conference on Artificial Intelligence (AAAI-10)*, pages 1701–1706, 2010.

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

[9] S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP-01)*, pages 13–24, 2001.

[10] A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 22:279–318, 2004.

[11] A. Felner, R. E. Korf, R. Meshulam, and R. C. Holte. Compressed pattern databases. *Journal of Artificial Intelligence Research*, 30:213–247, 2007.

[12] A. Felner, R. Meshulam, R. C. Holte, and R. E. Korf. Compressing pattern databases. In *National Conference on Artificial Intelligence (AAAI-04)*, pages 638–643, 2004.

[13] A. Felner, N. Sturtevant, and J. Schaeffer. Abstraction-based heuristics with true distance computations. In *Symposium on Abstraction, Reformulation and Approximation (SARA-09)*, 2009.

[14] A. Felner, U. Zahavi, J. Schaeffer, and R. C. Holte. Dual lookups in pattern databases. In *International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 103–108, 2005.

[15] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SCC-4(2):100–107, 1968.

[16] M. Helmert. Landmark heuristics for the pancake problem. In *Third Annual Symposium on Combinatorial Search (SOCS-10)*, pages 109–110, 2010.

[17] Malte Helmert and Gabriele Röger. Relative-order abstractions for the pancake problem. In *ECAI*, pages 745–750, 2010.

[18] R. C. Holte, A. Felner, J. Newton, R. Meshulam, and D. Furcy. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence*, 170:1123–1136, 2006.

[19] R. C. Holte, J. Newton, A. Felner, R. Meshulam, and D. Furcy. Multiple pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 122–131, 2004.

[20] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *National Conference on Artificial Intelligence (AAAI-96)*, pages 530–535, 1996.

[21] A. Junghanns and J. Schaeffer. Domain-dependent single-agent search enhancements. In *International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 570–575, 1999.

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

[23] R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(3):189–211, 1990.

[24] R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI-97)*, pages 700–705, 1997.

[25] R. E. Korf. Recent progress in the design and analysis of admissible heuristic functions. In *National Conference on Artificial Intelligence (AAAI-00)*, pages 1165–1170, 2000.

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

[27] R. E. Korf and A. Felner. Recent progress in heuristic search: A case study of the four-peg Towers of Hanoi problem. In *International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2324–2329, 2007.

[28] R. E. Korf and M. Reid. Complexity analysis of admissible heuristic search. In *National Conference on Artificial Intelligence (AAAI-98)*, pages 305–310, 1998.

[29] R. E. Korf, M. Reid, and S. Edelkamp. Time complexity of iterative-deepening-A*. *Artificial Intelligence*, 129(1–2):199–218, 2001.

[30] R. E. Korf, W. Zhang, I. Thayer, and H. Hohwald. Frontier search. *Journal of the JACM*, 52(5):715–748, September 2005.

[31] A. Mahanti, S. Ghosh, D. Nau, A. Pal, and L. Kanal. On the asymptotic performance of IDA*. *Annals of Mathematics and Artificial Intelligence*, 20(1–4):161–193, 1997.

[32] A. Martelli. On the complexity of admissible search algorithms. *Artificial Intelligence*, 8(1):1–13, 1977.

[33] M. McNaughton, P. Lu, J. Schaeffer, and D. Szafron. Memory efficient A* heuristics for multiple sequence alignment. In *National Conference on Artificial Intelligence (AAAI-02)*, pages 737–743, 2002.

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

[35] N. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.

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

[37] D. Ratner and M. K. Warmuth. Finding a shortest solution for the N $\times$ N extension of the 15-puzzle is intractable. In *National Conference on Artificial Intelligence (AAAI-86)*, pages 168–172, 1986.

[38] S. Russell and P. Norvig. *Artificial Intelligence, A Modern Approach, Third Edition*. Prentice Hall, 2010.

[39] M. Samadi, M. Siabani, A. Felner, and R. C. Holte. Compressing pattern databases using learning. In *European Conference on Artificial Intelligence (ECAI-08)*, pages 495–499, 2008.

[40] N. Sturtevant, A. Felner, M. Barer, J. Schaeffer, and N. Burch. Memory-based heuristics for explicit state spaces. In *International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 609–614, 2009.

[41] F. Yang, J. Culberson, R. C. Holte, U. Zahavi, and A. Felner. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research*, 32:631–662, 2008.

[42] U. Zahavi, A. Felner, N. Burch, and R. C. Holte. Predicting the performance of IDA* with conditional distributions. In *AAAI Conference on Artificial Intelligence (AAAI-08)*, pages 381–386, 2008.

[43] U. Zahavi, A. Felner, N. Burch, and R. C. Holte. Predicting the performance of IDA* (with BPMX) with conditional distributions. *Journal of Artificial Intelligence Research*, 37:41–83, 2010.

[44] U. Zahavi, A. Felner, R. C. Holte, and J. Schaeffer. Dual search in permutation state spaces. In *National Conference on Artificial Intelligence (AAAI-06)*, pages 1076–1081, 2006.

[45] U. Zahavi, A. Felner, R. C. Holte, and J. Schaeffer. Duality in permutation state spaces and the dual search algorithm. *Artificial Intelligence*, 172(4–5):514–540, 2008.

[46] U. Zahavi, A. Felner, J. Schaeffer, and N. R. Sturtevant. Inconsistent heuristics. In *National Conference on Artificial Intelligence (AAAI-07)*, pages 1211–1216, 2007.

[47] Z. Zhang, N. Sturtevant, J. Schaeffer, R. C. Holte, and A. Felner. A* search with inconsistent heuristics. In *International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 634–639, 2009.

[48] R. Zhou and E. Hansen. Space-efficient memory-based heuristics. In *National Conference on Artificial Intelligence (AAAI-04)*, pages 677–682, 2004.

[49] R. Zhou and E. Hansen. Breadth-first heuristic search. *Artificial Intelligence*, 170(4-5):385–408, 2006.

[50] R. Zhou and E. A. Hansen. Memory-bounded A* graph search. In *Florida Artificial Intelligence Research Society (FLAIRS-02)*, pages 203–209, 2002.

## Appendix A. Example for dual state heuristic

In this section, we provide an example that shows why the dual heuristic can provide inconsistent values. Consider the 9-pancake puzzle states shown in Figure 33. 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.

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. It is easy to see that applying the same sequence of operators (reverse locations 1-3, reverse locations 1-6) to $S_2^d$ will produce the goal state. 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

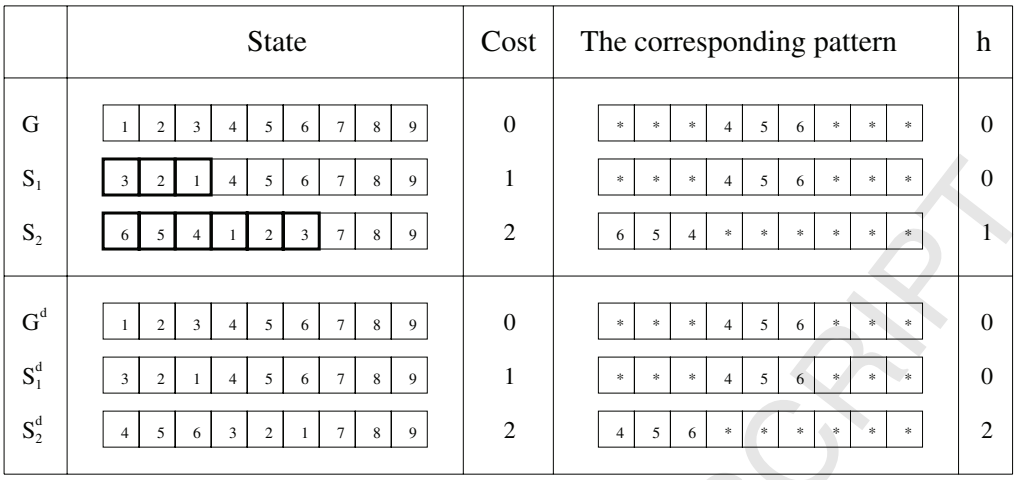


Figure 33: 9-pancake states

arrive at node $S_2^d$. Therefore, 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 resulting PDB provides distances to the goal pattern from all reachable patterns. The right column of Figure 33 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 [20]. 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). When moving from $S_1$ to $S_2$ (or vice versa), even though $g$ was changed by 1, $h$ was changed by 2.