

Using Inconsistent Heuristics on A* Search

Nathan R. Sturtevant and Zhifu Zhang and Robert Holte and Jonathan Schaeffer

University of Alberta, Department of Computing Science
Edmonton, AB, Canada T6G 2E8
{nathanst, zfzhang, holte, jonathan}@cs.ualberta.ca

Abstract

Early research in heuristic search discovered that using inconsistent heuristics in an A* search could result in an exponential blow-up in the number of nodes expanded. As a result, the use of inconsistent heuristics has largely disappeared from practice. However, recent research has shown that they can yield dramatic performance improvements to an IDA* search using the BPMX enhancement. This paper revisits inconsistency in A*-like algorithms. The DELAY algorithm is introduced which improves the worst-case complexity from $O(N^2)$ to $O(N^{1.5})$. Additional issues surrounding BPMX and A* are addressed, and a variety of performance metrics are shown across two domains.

Introduction

It is widely believed that A* is optimal; given the same heuristic and tie-breaking mechanism there is no search algorithm which will always have fewer distinct node expansions. While this is true, it hides an important fact about inconsistent heuristics. Early research on search algorithms discovered that if A* uses an inconsistent heuristic, it may perform an exponential number of re-expansions (Martelli 1977). From this perspective, A* is inferior to alternative algorithms, which have a worst-case number of expansions and re-expansions of $O(N^2)$. This result discouraged research on inconsistent heuristics for many years, in particular since most ‘natural’ heuristics seemed to be consistent. Recent research has shown that inconsistent heuristics are easy to create and can be used to great advantage in an IDA* search (Zahavi *et al.* 2007; Felner *et al.* 2005) by using the BPMX enhancement. Inconsistency hasn’t been a problem in IDA* because IDA* already performs many node re-expansions and is only used in domains where the cost of re-expansions is fully amortized over the cost of the search.

The case for using inconsistent heuristics with IDA* has been made; it is time to revisit the perceptions regarding their use with A*. This paper makes several contributions:

- A new search algorithm, DELAY, improves the worst-case complexity of A* search with inconsistent heuristics from $O(N^2)$ to $O(N^{1.5})$. This result provides insight into when an inconsistent heuristic will dominate a consistent heuristic;

Copyright © 2008, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

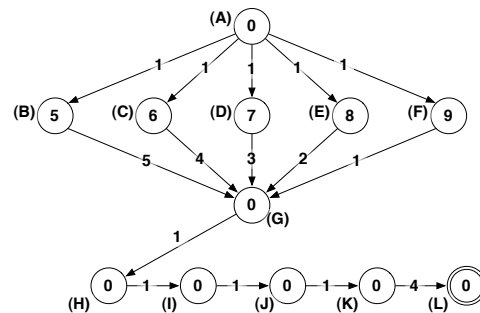


Figure 1: An example where A* will expand $O(N^2)$ nodes.

- Bounds on the number of nodes expanded, based on search graph properties, showing that in most graphs of interest A*’s worst-case will not occur;
- Several important corrections to the published literature;
- An analysis of BPMX (Felner *et al.* 2005) when applied to A* and similar algorithms; and
- Two experimental domains that evaluate the benefits of inconsistent heuristics.

This research revises the heuristic search literature. A* search with inconsistent heuristics is now practical and, for many domains, may yield the best performance.

Background

The A* search algorithm assigns each search node a f -cost, $f = g + h$, where the g -cost is the cost of the path from the start node to the current node, and the h -cost is an estimate of the cost from the current node to the goal. A heuristic is *admissible* if it never over-estimates the cost to the goal. A heuristic is *consistent* if for every pair of nodes F and G , $h(F) \leq d(F, G) + h(G)$, where $d(F, G)$ is the optimal path cost between F and G . Another definition that is often used, $|h(F) - h(G)| \leq d(F, G)$, holds only for undirected graphs. An inconsistent heuristic can be seen in Figure 1 between nodes F and G . The value inside each node is the heuristic value and edges are labeled with their costs. The heuristic in this example is not consistent because $h(F) = 9 \not\leq 1 + 0 = d(F, G) + h(G)$.

A* maintains an open list of eligible nodes to expand next, and a closed list of nodes which have already been expanded.

	A*		DELAY		
	Open	Closed	Open	Closed	Delay
1	A		A		
2	B-F	A	B-F	A	
3	C-G	A, B	C-G	A, B	
4	C-F, H	A, B, G	C-F, H	A, B, G	
5	C-F, I	A, B, G, H	C-F, I	A, B, G, H	
6	D-G, I	A-C, H	D-F, I	A-C, H	G
7	D-F, H, I	A-C, G	D-F, I	A-C, G	H
8	D-F, I	A-C, G, H	E, F, I	A-D	G, H
9	D-F, J	A-C, G-I	E, F, I	A-D, G	H
10	D-F, K	A-C, G-J	F, I	A-E	G, H
...

Figure 2: Node expansions for Figure 1 with A* and DELAY.

With a consistent heuristic nodes are only moved from the open list to the closed list. With an inconsistent heuristic, however, closed nodes may be opened again (and return to the open list). We illustrate this with Figure 1. Each node is marked with its h -cost. The nodes on the open and closed lists after each step are shown in Figure 2. The node about to be expanded is shown in bold.

The first four steps of the algorithm are straightforward, as A* expands nodes A, B, G, and H. At this point H’s only child, I, will have a f -cost of 8 ($g = 8, h = 0$). So, node C, with f -cost 7, will be expanded next. Node G will subsequently be re-expanded through a lower-cost path. Each node C-F provides a shorter path to G, but the shorter path is not discovered before expanding an increasingly larger subset of nodes G-K. Our new algorithm, DELAY, works by reducing the number of re-expansions in nodes G-K.

Formally, a search problem is a 4-tuple $(\mathcal{G}, h, start, goal)$, where $\mathcal{G} = (V, E)$ is a search graph and h is a heuristic function. The size of the search space explored is often measured as b^d where b is the branching factor and d is the depth of search. A different metric is used here. Let N be the set of all nodes which have the following two properties: 1) f -cost less than or equal to h_{opt} , the optimal distance from $start$ to the $goal$, and 2) are reachable from $start$ along a path for which all nodes also have f -cost less than or equal to h_{opt} . This is the set of nodes, modulo tie-breaking, which will be expanded by A*. We designate the size of N as $|N|$, although we occasionally just write N when the meaning is obvious.

Given an inconsistent heuristic, A* may perform up to $O(2^N)$ total expansions (including re-openings) (Martelli 1977). Two algorithms have been suggested that improve this worst case. The B algorithm (Martelli 1977) ignores h -costs when they are provably too low, which can only happen if they are inconsistent. This simple method reduces the worst-case number of node expansions to $O(N^2)$. A slight improvement of this algorithm, with the same worst-case performance, was proposed in (Bagchi & Mahanti 1983). The B’ algorithm (Mero 1984) introduced pathmax rules which propagate and update h -costs through the search space. B’ also has a worst-case bound of $O(N^2)$ but outperforms algorithm B on certain special cases ($O(N)$ compared to $O(N^2)$).

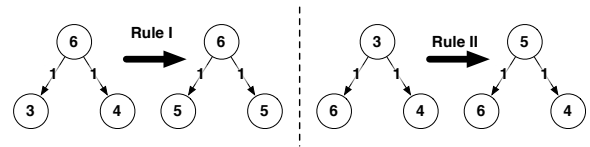


Figure 3: An example of the pathmax rules.

It is important to note that different heuristics will have different values of N . Thus, a consistent heuristic h_1 will result in at most $O(N_{h_1})$ expansions while an inconsistent heuristic h_2 will result in at most $O((N_{h_2})^2)$ expansions. Thus, if $N_{h_1} > (N_{h_2})^2$ it would always be preferable to use the inconsistent heuristic, because even in the worst case it would still lead to fewer node expansions.

Corrections to Existing Literature

Mero (Mero 1984) introduced an example graph, similar to Figure 1, where B, B’ and A* all perform $O(N^2)$ expansions. This conclusion was based on analysis, not an implementation. Our implementation of these algorithms disprove the claim. Figure 1 illustrates the fix needed to repair their example—setting the h -cost of nodes H-L to 0. Additionally, their example uses a much broader range of edge costs to achieve the same effect seen here. Figure 1 should be seen as a single example of a class of graphs which can be constructed for any size of N .

The pathmax rules were also originally introduced in (Mero 1984). Pathmax is illustrated in Figure 3. The first rule propagates the heuristic from a parent p to a child c_i by the rule $h(c_i) \leftarrow \max(h(c_i), h(p) - d(p, c_i))$. The second pathmax rule was originally stated as: “Let c_i be the son of p for which $h(p) + d(p, c_i) = h_c(p)$ is minimal. If $h_c(p) > h(p)$ then set $h(p) \leftarrow h_c(p)$.”¹ If applied to the first example in Figure 3, it may result in an inadmissible heuristic. The intention of this rule is to propagate values from the successor of a node to its parent and should read $h(p) \leftarrow \min_i (h(c_i) + d(p, c_i))$. This rule is illustrated in the right-hand-side of Figure 3.

DELAY Algorithm

In this section the DELAY algorithm is introduced, which improves the best-known worst-case performance from $O(N^2)$ to $O(N^{1.5})$. Analysis of Figure 1 reveals that $O(N^2)$ nodes are expanded when nodes G-K (total $O(N)$) are re-expanded $O(N)$ times for $O(N^2)$ total expansions. If the number of re-expansions in nodes G-K can be limited, then the worst case can be reduced.

DELAY imposes a limit on the maximum number of nodes that can be sequentially re-expanded. The algorithm takes a parameter k and requires that for every k re-expansions, there must be at least one new expansion. Pseudo-code for DELAY is found in Figure 4, and is very similar to A*; the differences are marked with a ‘*’. A* normally has an open list (a priority queue sorted by f -cost) and a list for closed

¹The variable names have been changed for consistency.

nodes. The DELAY algorithm adds one additional list, the delay list, which is sorted by minimal g -cost. If a node is ever taken off the closed list, only possible if the heuristic is inconsistent, then it is placed on the delay list for special processing. DELAY uses the pathmax rules and a few other minor details that are not described here.

The DELAY algorithm with $k = 1$ is compared to A^* in Figures 1 and 2. The right-hand-side of Figure 2 shows the order of node expansions for DELAY. This is the same as A^* until node C is expanded, when A^* puts G back on the open list while DELAY puts it on the delay list. G will be expanded off the delay list, but because $k = 1$, H cannot be subsequently expanded until a new node is expanded—so H is left on the delay list. This delay in expanding H forces D to be expanded in step 8, and E to be expanded in step 10. Continuing this example will demonstrate that DELAY is much more efficient than A^* for this example. In Figures 1 $k = 1$ is optimal and will result in only $O(N)$ total expansions. Unfortunately, Figure 1 is not a worst-case example for DELAY.

We assume that the cost to the goal is not known *a priori*. If it was, an algorithm like Breadth-First Heuristic Search (BFHS) (Zhou & Hansen 2004) could be used to find the goal with only $O(N)$ expansions. If the goal cost is not known, an iterative deepening version of BFHS could be used, which is equivalent to the B algorithm. If the search space is a graph $\mathcal{G} = (V, E)$, and $|V| = O(N)$, Dijkstra’s algorithm (Dijkstra 1959) would also expand $O(N)$ nodes (in Figure 1, for instance). We additionally assume that $N \ll |V|$. Algorithms B and B’ are guaranteed not to expand any nodes outside of the set N , and this is where the DELAY algorithm makes its gains. DELAY may expand nodes outside of the set N , but the number of such nodes is tightly bounded.

Worst-Case Performance of DELAY

To prove the worst-case performance of DELAY we use the following procedure. The nodes expanded by DELAY are divided into two sets. The first set is N , as defined previously. The second set, A , contains those nodes that DELAY expands outside of N . Note that A begins as an empty set, and grows during the execution of DELAY. The DELAY algorithm has a parameter k which controls the number of nodes that can be re-expanded after each new expansion. We show that the total expansions resulting from new expansions in N is limited to $O(|N|k)$. We then show that if we choose $k = \sqrt{2N}$ that A is limited to size $O(\sqrt{N})$. Additionally, once $O(\sqrt{N})$ nodes are expanded in A we are guaranteed that we have the shortest path to every node inside N .

We do not have *a priori* knowledge of which nodes are in N and which are in A , but these sets are well defined. We assume that N is finite. With a consistent heuristic, nodes which are closed will never be opened or expanded again. But, with an inconsistent heuristic, this isn’t the case. Thus, it is useful to make a distinction between nodes which are closed, but may subsequently be re-opened, and those which are closed and cannot be reopened. When a node is closed for the last time we consider it to be *finished*.

```

// DELAY is sorted by g-cost
// OPEN is sorted by f-cost
Delay(start, goal, k)
1  push start onto OPEN
2  while (true)
3*   while ((OPEN.top() = goal) AND
         (DELAY.top().gcost < OPEN.top().gcost))
4*     Reexpand(1)
5     next ← open.pop()
6     if (next == goal)
7       return path from start to goal
8     else
9       Expand(next)
10*    Reexpand(k)
11  end while

```

```

Expand(node)
1  foreach child  $c_i$  of node
2    if  $c_i$  on OPEN, update cost( $c_i$ )
3*   else if  $c_i$  on DELAY, update cost( $c_i$ )
4     else if  $c_i$  on CLOSED and (can update cost)
5       update cost ( $c_i$ )
6*      add ( $c_i$ ) to DELAY
7       remove ( $c_i$ ) from CLOSED
8     else add ( $c_i$ ) to OPEN
9     add node to CLOSED

```

```

* Re-expand(k)
1  for  $i = 1 \dots k$ 
2    if (delay.empty()) return
3    Expand(delay.pop())

```

Figure 4: The DELAY algorithm.

The entire proof is quite lengthy. Due to space constraints we highlight a few aspects of the proof here, but we are unable to present the proof in its entirety. The proof that DELAY finds an optimal solution is almost identical to the proof for A^* , so we omit that portion of the proof as well.

First, we bound the work due to new expansions inside N and new expansions inside A .

Lemma 1 At most $O(|N|k)$ expansions (including re-expansions) will result from new node expansions in N .

Proof. After each expansion in N , DELAY is only allowed to re-expand k nodes. There are exactly $|N|$ nodes in N , so in the process of expanding these nodes there cannot be more than $|N|k$ re-expansions. This results in $O(|N|k)$ total expansions. \square

Lemma 2 If the size of A is bounded to be no larger than A_{max} , at most $O(kA_{max})$ expansions (including re-expansions) will result from new node expansions in A .

Proof. After each expansion in A , DELAY is only allowed to re-expand k nodes. Each node in A can only be expanded as a new node once. Since A is bounded, at most $O(kA_{max})$ expansions result from new node expansions in A . \square

These lemmas tell us the total size of our search once we choose an appropriate value of k .

Lemma 3 During search, all nodes outside N (in A) will have f -cost $> h_{opt}$.

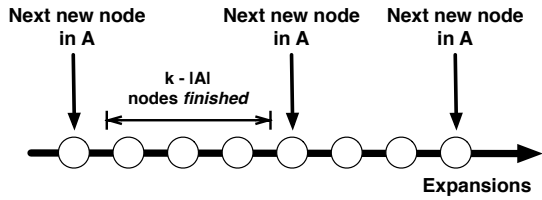


Figure 5: Overview of proof argument.

Proof. All nodes bordering N in A must have f -cost $> h_{opt}$ by definition. Using the first pathmax rule we are guaranteed that the f -cost along a path will be non-decreasing. Thus, during search, all nodes in A will have f -cost $> h_{opt}$. \square

This lemma shows that if we have an optimal path to a node in N on the open list, it will always be expanded before a node in A .

If all nodes in the open list have suboptimal g -cost, then there must be a node in the delay list with optimal g -cost but blocked by the requirement for a new expansion per k re-expansions. The top node in the delay list has optimal g -cost as well, since no other node in this list can provide it with a smaller g -cost.

Lemma 4 Between each new expansion in A , at least $k - |A|$ nodes in N are finished.

This lemma is illustrated in Figure 5, but the proof is too long to present here. The size of A is growing, but only by 1 for every new expansion in A . So, as more nodes are expanded in A , progressively fewer nodes will be finished in N . Once $|A| \geq k$ it is possible that all progress in N will end. This can happen after k new expansions in A . Thus, for any value of k , the total number of nodes guaranteed to be finished in N is:

$$k + (k - 1) + (k - 2) + \dots + 2 + 1 = k^2/2 + k/2 \approx k^2/2$$

If we let $k = \sqrt{2|N|}$, then $k^2/2 = |N|$. This choice of k is sufficiently large so that all nodes in N are finished before $|A|$ grows larger than k . Otherwise we could spend all our time re-expanding nodes in A instead of finishing nodes in N .

Theorem 1 The total number of expansions (including re-expansions) by DELAY when $k = \sqrt{2|N|}$ is bounded by $O(|N|k) = O(|N|^{1.5})$.

Proof. By lemma 1, each expansion in N will be followed by at most $\sqrt{2N}$ re-expansions, for $O(|N|^{1.5})$ expansions. If $k = \sqrt{2N}$, the total number of expansions resulting from new nodes in A is $O(N)$, by lemma 2. Once we expand $k = \sqrt{2N}$ nodes in A all nodes in N are guaranteed to be finished. So, DELAY is limited to no more than $O(|N|^{1.5})$ total expansions for $k = \sqrt{2|N|}$. \square

Theorem 2 In DELAY, $k = O(\sqrt{|N|})$ is optimal.

While we do not show it here, it is possible to construct an example for which $k \leq \sqrt{|N|}$ results in an unbounded number of expansions in A but $k \geq \sqrt{2|N|}$ does not. This

implies that, given no other assumptions about edge costs, $k = O(\sqrt{|N|})$ is optimal.

The one caveat to this analysis is that we assume N is known *a priori*, when in fact this is not true. But, there has been a lot of work on predicting the size of search trees (Korf, Reid, & Edelkamp 2001; Kilby *et al.* 2006), so this might be used to estimate N . Another, simpler approach is to estimate N as the number of unique nodes expanded during search, in which case the estimate will grow with the search. This is acceptable, because the worst-case occurs when all nodes inside N are expanded once, and then need to be re-expanded again. In this case we will have a very accurate estimate of N .

If we choose $k = \sqrt{2N}$, the asymptotic running time is $O(N^{1.5})$ in the worst case. This result suggests we should always use an inconsistent heuristic h_i instead of consistent heuristic h_c if we expect that h_i will reduce the size of N_{hc} to $N_{hi} = N_{hc}^{2/3}$. Otherwise, some measure of inconsistency is needed to determine whether or not the inconsistency in a heuristic is likely to produce the worst-case performance. In the next section we discuss how bounding the range of edge values in a graph can bound the worst-case performance for an inconsistent heuristic.

Tighter Bounds for A* Expansions

Previous research (Martelli 1977; Mero 1984) demonstrated cases in which A* performs an exponential number of expansions, but B and B' only perform a quadratic number of expansions. If the heuristic values, solution cost, or edge costs in a domain have certain properties, A*'s worst-case performance cannot occur.

We define δ as the minimum possible amount by which the g -value of a node can change. For instance, if all edges have cost either 2 or 3 then $\delta = 1$. If edge costs are 1.0, 1.6, 1.8, then $\delta = 0.2$.

First, we show that the value of δ implies a lower bound on the maximum heuristic value of a node expanded during a search.

Theorem 3 If A* performs $\phi(N)$ node expansions, then there must be a node with h -cost at least $O(\delta \cdot \phi(N)/N)$, and the path to the goal must have cost at least $O(\delta \cdot \phi(N)/N)$.

Proof. If there are $\phi(N)$ total expansions in a graph of size $|N|$, by the pigeon-hole principle there must be a node, v_1 , with at least $(\phi(N) - N)/N$ re-expansions. Each re-expansion must decrease the g -cost of v_1 by at least δ , so after this process the g -cost of v_1 has been reduced by at least $\delta(\phi(N) - N)/N$.

We will now show, using Figure 6, that this implies that there must exist a node (B in Figure 6) with h -cost $\geq \delta(\phi(N) - N)/N$. Suppose the optimal path from A to v_1 is via the path $A \rightarrow B \rightarrow v_1$, with first expansion of v_1 occurring along the path L. If L is not an optimal path, exploring L before B requires that $f(B) > f_L(v_1)$, where $f_L(v_1)$ is the f -cost of v_1 when explored through the path L. Thus:

$$\begin{aligned}
f(B) &\geq f_L(v_1) \\
h(B) &\geq g_L(v_1) + h(v_1) - g(B) \\
h(B) &\geq g_L(v_1) + h(v_1) - g(v_1) \quad [g(B) < g(v_1)] \\
h(B) &\geq g_L(v_1) - g(v_1) \\
h(B) &\geq \delta \cdot (\phi(N) - N)/N
\end{aligned}$$

Therefore $h(B)$ is at least $O(\delta \cdot (\phi(N)/N))$. If we expand a node with h -cost $O(\delta(\phi(N)/N))$, then the cost of the optimal path must also be at least $O(\delta(\phi(N)/N))$. \square

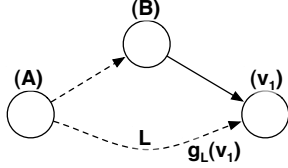


Figure 6: Two paths from A to v_1 .

For A^* to expand $\phi(N) = 2^N$ nodes, there must be a node with h -cost at least $O((\delta \cdot 2^N)/N)$. For A^* to expand $\phi(N) = N^2$ nodes, there must be a node with h -cost at least $O(\delta \cdot N)$. In many search problems edge costs are constant and we consider the depth of the solution to be d and the nodes expanded, $N = b^d$. In this case, $d = \lg(N)/\lg(b)$, which is the maximum heuristic value during search, so the worst-case complexity is expected to be $O(N \lg(N)/\lg(b))$. These results imply that A^* 's worst-case performance will not occur on many graphs of interest. They also suggest that we may be able to use a lower value of k for the DELAY algorithm in some domains.

Next, we use this result to show that if there is an upper bound on the edge weights in a problem, there is also an upper bound on the worst case runtime complexity. This bound is for A^* , not for DELAY, which has a better worst-case bound.

Theorem 4 If the edge weights are constant-bounded (do not grow with problem size), then the total number of node expansions by A^* is at most $O(N^2/\delta)$.

Proof. Since the edge weights are constant-bounded, the cost of any path from any node in N to goal is at most $O(N)$. Thus, this is an upper bound on the admissible h -cost of any node in N . Using the result of the previous lemma,

$$\begin{aligned}
O(\delta * \phi(N)/N) &\leq O(N) \\
\phi(N) &\leq O((N^2)/\delta)
\end{aligned}$$

\square

These results have implications for DELAY. Primarily, they serve to limit the number of re-expansions that can occur in A , and can reduce the worst-case bound. For instance, in a domain where the optimal solution cost is $O(\lg(N))$ and δ is constant-bounded, no node can be re-expanded more than $O(\lg(N))$ times. As a result, DELAY can use $k = \lg(N)$ instead of $k = \sqrt{N}$, and reduce its overhead.

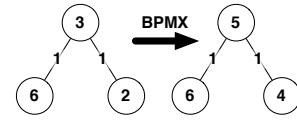


Figure 7: BPMX propagation.

BPMX and A^*

Bidirectional pathmax (BPMX) (Felner *et al.* 2005) is an extension of pathmax to graphs with undirected edges. We illustrate BPMX in Figure 7. The heuristic value of the left child is used to update the heuristic values of the parent and the right child.

In IDA^* the heuristic updates from BPMX are not stored, but just propagated during search. This isn't a problem because IDA^* can easily perform the propagation step at no cost as it searches the tree. However, in a best-first search this isn't always the case. After an inconsistent heuristic value is found, it can be propagated one or more steps, with variable cost. This raises the issue of how far to propagate a heuristic update when one is found. We demonstrate here that there is no single optimal policy. We give one example where propagating values as far as possible will result in $O(N^2)$ BPMX propagation steps, and another example where propagating values will result in arbitrarily large savings in node expansions.

In the example at the top of Figure 8, the heuristic values gradually increase from nodes A to G. When node B is reached, the heuristic can be propagated back to node A, increasing the heuristic value to 1. When node C is reached, the heuristic update can again be propagated back to node (A), increasing the heuristic value to 2. With each new expansion, $O(N)$ propagations will occur, for $O(N^2)$ total propagation steps with no savings in node expansions.

In the bottom portion of Figure 8, the opposite happens. The start node is (A). The search proceeds to node (B) where a heuristic value of 100 is found. If the BPMX propagation is limited, B will remain on the open list with f -cost 102 while all the children of C are expanded. The possible BPMX updates are shown in the dark ovals. Here, BPMX can update C's h -cost to 97 and f -cost to 100. After this update the direct edge from A to the goal will be expanded and the search will terminate before expanding any of C's children.

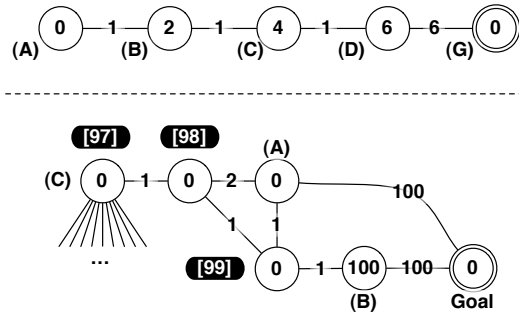


Figure 8: Bad and good examples for BPMX propagation.

These examples demonstrate that there cannot be a fixed propagation policy which is optimal for all graphs. In the experimental results we show that the best propagation policy is domain dependent.

Experimental Results

We present experimental results in two domains, the topspin puzzle and pathfinding.

Top Spin

Top Spin is a combinatorial puzzle in which the goal is to arrange the tokens in increasing order. The (T, K) -TopSpin puzzle has T tokens arranged in a ring. An action in this space corresponds to flipping (reversing) any set of K consecutive tokens.

This domain is tricky to implement for algorithms such as IDA*, as there are a large number of transpositions in the state space. For IDA* to solve this puzzle efficiently, as many of these transpositions as possible must be detected and removed. However, using A*, we can detect the transpositions to avoid re-expanding states.

The state-of-the-art heuristic for this domain is a pattern database (PDB) with dual lookups (Felner *et al.* 2005). The dual of a state will have a different heuristic value from the regular state, and is often much better than a regular heuristic lookup. Even if a PDB is consistent with respect to regular lookups, it will not be consistent when used for dual lookups, because the dual of two adjacent states may not necessarily be dual.

We tested a variety of algorithms in TopSpin over a variety of problem sizes. We generated 1000 random problems of the (T, K) -TopSpin puzzle for $K = 4$ and $T = 9 \dots 14$. For each puzzle, we built a PDB of size $\lfloor T/2 \rfloor$. We report comprehensive results in Table 1 for (14,4)-TopSpin, although the average over all problems is nearly identical. When BPMX is used, it is parameterized by the depth of the BPMX propagation. DELAY is parameterized by k .

All algorithms are using inconsistent heuristics except A* in the last line, which is using the normal consistent lookup

Algorithm	Nodes Expanded
Delay(log)+BPMX(∞)	24952
Delay(2)+BPMX(∞)	24956
Delay(2)+BPMX(1)	25132
Delay(log)+BPMX(1)	25147
A* +BPMX(2)	30190
A* +BPMX(∞)	30212
A* +BPMX(1)	30213
A* +BPMX(3)	30215
B'	40528
A*	40559
B	40634
Delay(log)	40800
Delay(2)	40804
A* (normal lookup)	71999

Table 1: Average nodes expanded in (14,4)-TopSpin.

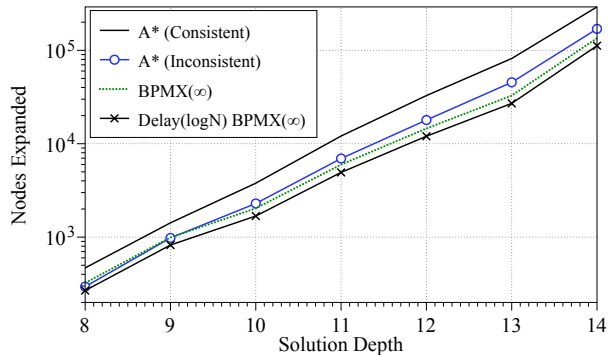


Figure 9: Nodes expanded based on solution depth.

in the PDB. Using dual lookups, there are three distinct classes of algorithms. DELAY with BPMX has the best results, averaging around 25k expansions, about three times fewer than A*. The next class of algorithms are A* with BPMX. The amount of propagation does not make a significant difference in the number of nodes expanded in this domain. The last class of algorithms are those which do not use BPMX and all have similar performance. Each of these groupings is statistically significant.

In Figure 9, we plot results over all instances solved. The x -axis is the solution depth and the y -axis is the average number of nodes expanded for all instances (regardless of the number of tokens) which had that solution depth. Note that the y -axis is a log plot, so all algorithms are growing exponentially in the solution depth. This means that in TopSpin DELAY is effective, but it is not improving the asymptotic complexity.

Pathfinding

In the pathfinding domain, an agent must find the shortest path between two points in a map. Both cardinal moves (cost 1) and diagonal moves (cost $\sqrt{2}$) are allowed. The natural heuristic in this domain is octile distance. The octile distance between two points is $\sqrt{2} \cdot \min(\Delta x, \Delta y) + |\Delta x - \Delta y|$.

While octile distance is a perfect heuristic on a blank map, it will often lead agents into dead-ends in the search space. Memory-based heuristics can be used on top of the octile-distance to speed up the search. A perfect heuristic can be obtained by solving the all-pairs-shortest-path problem, however this can be expensive. If the size of the map is $O(M)$, this will take memory $O(M^2)$ and time $O(M^3)$. In many domains this isn't feasible. Modern computer games, for instance, have tight memory budgets for pathfinding, meaning that there isn't $O(M^2)$ memory available.

Instead of storing the shortest distance from all points on the map, choose t points on a map and build a table with the single-source shortest path from each t to all other nodes. Each of these t tables can then generate an admissible heuristic between any nodes a and b by computing $h(a, b) = |d(a, t) - d(b, t)|$. The effectiveness of this heuristic will depend on the heuristic being queried. The more tables that are built, the better chance that one of the heuris-

tics will give good results. To further reduce memory costs, the tables can be compressed, eliminating a fraction of the entries.

We used this method to generate an inconsistent heuristic for pathfinding problems. For these experiments we take the max of the octile distance and one of 10 possible table-based heuristics, which results in an inconsistent heuristic. The table-based heuristic lookup is chosen based on a hash function of the current state. For our experiments, we didn't need to compress our heuristic, but if we did, we wouldn't be able to lookup every heuristic from every state. This allows us to compare the maximum possible gain from looking up and taking the max of all 10 heuristics, versus the gains from using the inconsistent heuristic.

Our experiments are performed on computers with Intel P4 3.4GHz CPUs, and 1GB of RAM. We perform experiments on a collection of 116 game maps, all of which are scaled to 512x512. There are 1200 problem instances on each map, ranging from easy (solution length ≤ 2) to hard (solution length ≥ 500).

We summarize the results of these experiments in Figures 10 and Figure 11. These graphs show the average number of nodes needed to solve one problem in the problem set. The x -axis is the path length. Results were averaged in buckets of size 4, so the last point on the curve is the average number of nodes expanded on paths with length between 508 and 512.

In Figure 10 we show the results for DELAY, B, B' and A*. A* was tested with both the consistent octile-distance heuristic and the inconsistent heuristic described above. None of these algorithms use BPMX. Although B' was proven theoretically to be no worse than B, in practice it ends up being significantly worse than B, because it isn't possible to maintain consistent tie-breaking between B and B' after B' applies the pathmax rules. DELAY with $k = 2$ has better performance than all of these algorithms, however, they all have performance worse than A* with octile distance.

This result is significant – the inconsistent heuristic used in these experiments dominates the consistent heuristic, in that it always provides the same or a better value. Yet, despite a more informed heuristic, it still leads to more node expansions as a result of the inconsistency. This means that the original concerns about inconsistency were not unwarranted, as we have shown that in practice a more informed inconsistent heuristic leads to worse performance than a less informed consistent heuristic.

This is not the complete story. The results with BPMX are shown in Figure 11 (note the change in the y -axis). The top line in this graph is the same line as the bottom line in Figure 10. Here we plot A* with BPMX propagation of 1, 2, and ∞ , DELAY with BPMX, and A* using the max of all the heuristic tables, which is a consistent heuristic. This last line is provided as a best-case reference, in that this heuristic could not always be stored in domains with tight memory constraints.

Using BPMX, the inconsistent heuristic is now better than the octile-distance heuristic. In this domain, with this heuristic, a propagation distance of 1 is best. Given the way the heuristic is built, this makes sense because the different

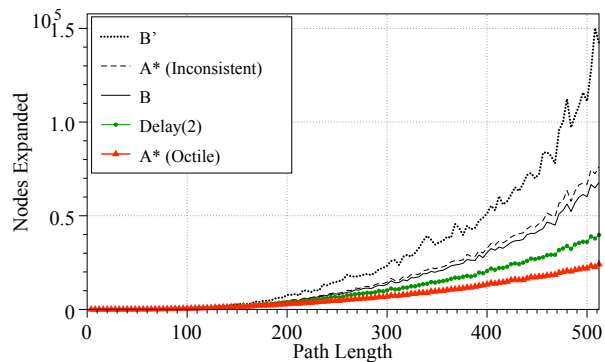


Figure 10: Search tree size for A* and DELAY.

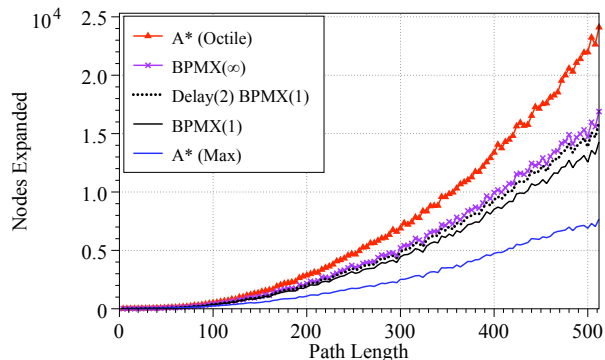


Figure 11: Search tree size for A* and DELAY with BPMX.

heuristic values are laid out in a uniform manner, meaning that good heuristic values will always be nearby. DELAY is competitive with BPMX, however it expands slightly more nodes than BPMX. When we timed the experiments, DELAY did slightly better than BPMX(1), depending on the hardware used to run the experiments.

The problem being tested here is a realistic one, using a realistic heuristic. In these experiments, like with TopSpin, BPMX has a significant effect on the number of nodes expanded. Unlike TopSpin, however, DELAY is not as effective. This is a result of the heuristic being used in each domain. In TopSpin, the dual lookup gives a larger range of values that will not be predictable from state to state. But in this pathfinding setup, after doing a BPMX propagation, the heuristic will often be consistent, or only slightly inconsistent. Thus, nodes will not be re-expanded a large number of times, and DELAY is less effective. We are able to set up problems in the pathfinding domain where there is significantly more inconsistency. In these contrived examples, DELAY is much more effective.

Conclusions

Inconsistent heuristics have had a negative connotation in the literature. In part it comes from the very name, “inconsistent”, and in part comes from the early heuristic search publications which demonstrated a pathological worst case

for A*. Recent research has shown that inconsistent heuristics are not only practical for IDA*, they often can be quite powerful. This paper provides the algorithmic insight needed to complete the resurrection of inconsistent heuristics: the problems with A* have been resolved. This research presents the first evidence that the benefits of inconsistency that have been so beneficial to IDA* search applications can also apply to A*. In effect, there is no longer any reason to assign a negative attribute to inconsistent heuristics.

However, there are still open issues to be solved. We are working to come up with precise metrics that will help predict how far BPMX propagations should be performed, as well as predicting what value of k should be used for DELAY.

Regardless, we have made progress in understanding the issues surrounding inconsistent heuristics. In addition to improving the worst-case performance when using inconsistent heuristics, we have improved the understanding of BPMX in algorithms besides IDA*, and shown the effectiveness of inconsistent heuristics in practice.

References

- Bagchi, A., and Mahanti, A. 1983. Search algorithms under different kinds of heuristics – a comparative study. *J. ACM* 30(1):1–21.
- Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.
- Felner, A.; Zahavi, U.; Schaeffer, J.; and Holte, R. C. 2005. Dual lookups in pattern databases. In *IJCAI*, 103–108.
- Kilby, P.; Slaney, J. K.; Thiébaux, S.; and Walsh, T. 2006. Estimating search tree size. In *AAAI*, 1014–1019.
- Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of Iterative-Deepening-A*. *Artificial Intelligence* 129(1-2):199–218.
- Martelli, A. 1977. On the complexity of admissible search algorithms. *Artificial Intelligence* 8:1–13.
- Mero, L. 1984. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence* 23:13–27.
- Zahavi, U.; Felner, A.; Schaeffer, J.; and Sturtevant, N. R. 2007. Inconsistent heuristics. In *AAAI*, 1211–1216.
- Zhou, R., and Hansen, E. 2004. Breadth-first heuristic search. In *Proc. ICAPS-04*, 92–100.