Robust Bidirectional Search via Heuristic Improvement

Christopher Wilt and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
{wilt, ruml} at cs.unh.edu

Abstract

Although the heuristic search algorithm A* is well-known to be optimally efficient, this result explicitly assumes forward search. Bidirectional search has long held promise for surpassing A*'s efficiency, and many varieties have been proposed, but it has proven difficult to achieve robust performance across multiple domains in practice. We introduce a simple bidirectional search technique called Incremental KKAdd that judiciously performs backward search to improve the accuracy of the forward heuristic function for any search algorithm. We integrate this technique with A*, assess its theoretical properties, and empirically evaluate its performance across seven benchmark domains. In the best case, it yields a factor of six reduction in node expansions and CPU time compared to A*, and in the worst case, its overhead is provably bounded by a user-supplied parameter, such as 1%. Viewing performance across all domains, it also surpasses previously proposed bidirectional search algorithms. These results indicate that Incremental KKAdd is a robust way to leverage bidirectional search in practice.

Introduction

It is well known that the A* algorithm is optimally efficient for computing shortest paths in domains with consistent heuristics (Dechter and Pearl 1985), but only among algorithms that expand nodes solely in the forwards direction. As a result, one way to outperform A* is by searching in both the forwards and the backwards directions. The general problem with this approach is that once the search frontiers meet, proving the optimality of the solution often requires more expansions than A* would have needed (Kaindl and Kainz 1997).

Despite this difficulty, there are bidirectional search algorithms that can outperform A*. The "Add method" of Kaindl and Kainz (1997), which here we will call KKAdd, uses backwards search to establish a heuristic correction, as we explain in more detail below. Kaindl and Kainz used the KKAdd method in conjunction with A*, and the size of the perimeter was a runtime parameter, set ahead of time. For reporting results, they simply selected one of the more effective parameter settings. The central problem in making the KKAdd method practical for general purpose use is determining how big the perimeter should be without pilot experiments, because in some domains, large values are effective, and in other domains, large values only serve to introduce extra overhead without any benefit.

We introduce a technique called Incremental KKAdd that overcomes this problem by dynamically balancing the work done in the forwards search and the backwards search. This has two benefits. First, incrementally growing the set of backwards expanded nodes allows the algorithm to adjust the backwards expansions as needed, without manual tuning, and without any a priori knowledge of how difficult the problem is.

Second, it also has the benefit of keeping the number of backwards expansions within a constant factor of the number of forwards expansions. We prove below (Theorem 2) that the only nodes expanded by A* using Incremental KKAdd that are not expanded by A* are the nodes expanded backwards from the goal. By keeping the number of backwards expansions at a fixed proportion of all expanded nodes, Incremental KKAdd bounds the overhead of the search in case the backwards search fails to provide any benefit.

A* with Incremental KKAdd provides an optimal solution, with a substantial speedup over A* when the heuristic correction is large. Compared to other bidirectional searches, A* with Incremental KKAdd is applicable in a wider variety of heuristic search settings. In addition, A* with Incremental KKAdd never takes substantially more time than A*, as other bidirectional algorithms sometimes do. Viewed across all domains, its performance surpasses previously proposed bidirectional search algorithms. These results indicate that Incremental KKAdd is a robust way to leverage bidirectional search in practice.

Previous Work

The A* algorithm (Hart, Nilsson, and Raphael 1968) performs a forwards search by expanding nodes in order of increasing $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the best known path from the start state s to n, and $h(n)$ is a lower bound on the cost of the best path from n to a goal.

In domains with a consistent heuristic, the KKAdd method (Kaindl and Kainz 1997) learns a correction to the original heuristic by expanding nodes backwards from the

goal, establishing the true $h(n)$ values (notated $h^*(n)$) for nodes that have been expanded backwards. The simplest approach is to expand the nodes in g_{rev} order, where g_{rev} is the cost of getting to the node starting at a goal using backwards expansions ($g_{rev} = h^*(n)$), but any order that expands nodes with optimal g_{rev} values may be used. Nodes that have been generated but not yet expanded in the backwards direction form a perimeter around the goal region.

The original forward $h(n)$ value for each of the nodes in the perimeter can be compared to the corresponding $g_{rev}(n)$ values yielding that node's heuristic error $\epsilon(n) = g_{rev}(n) - h(n)$. The heuristic error present for all nodes that have not been reverse expanded, h_{err}, is defined as $min_{n \in perimeter}(\epsilon(n))$. All solutions starting at a node that was not yet reverse expanded go through this perimeter, therefore all outside nodes must have at least as much heuristic error as the minimum error node in the perimeter. An admissible heuristic in the forwards search is constructed by first looking up the node in the collection of reverse expanded nodes. If the node has been reverse expanded, its $h^*(n)$ value is used. Otherwise the node is assigned $h_{KKAdd}(n) = h(n) + h_{err}$ as its admissible heuristic estimate.

The problem with KKAdd is that it requires a runtime parameter, the amount of backwards search to do. If this parameter is set too small, the algorithm is no different from A*, but if the parameter is set too large, the algorithm expands a large number of irrelevant nodes in the backwards direction, and fails to reap a sufficiently large benefit when the search is run in the forward direction. Kaindl and Kainz (1997) report results for A* with the KKAdd heuristic, but they do not say how the size of the backwards search was set, only mentioning how many nodes each variant used in the backwards direction. It is, therefore, an open question how to make the technique practical for general use.

A related algorithm is perimeter search (Dillenburg and Nelson 1994), which uses multiple heuristic evaluations to improve the h value of nodes. First, a perimeter is established by expanding a predefined number of nodes backwards from the goal. Note that when expanding nodes forwards, as with the KKAdd method, any solution must pass through one of the perimeter nodes. Thus, it is possible to estimate the cost of going through any node in the perimeter by estimating the distance to that perimeter node and then adding the known true distance from the perimeter node to a goal node to that estimate. This algorithm requires a heuristic function that can estimate the distance between two arbitrary states which we call a point-to-point heuristic. Since any solution must pass through a perimeter node, an admissible estimate is the minimum of the estimates over all perimeter nodes.

This algorithm poses two challenges for general purpose use. First, it requires an efficient heuristic between any two states, which not all domains have, and even when it is possible to compute a point-to-point heuristic, doing so is often more computationally intensive. The other problem is that it requires a large number of heuristic evaluations. Each forwards expansion requires one heuristic evaluation for every perimeter node, thus as the perimeter grows, the cost of computing a node's heuristic value also grows. These problems can be ameliorated by using a pattern database initialized with the perimeter, but this is not always practical, especially if the goal state is not the same for all instances. Manzini (1995) discusses an algorithm for reducing the number of heuristic evaluations when the heuristic is used for pruning but not sorting, as is the case for IDA* (Korf 1985).

Naturally, a central issue surrounding perimeter search is how to size the perimeter. López and Junghanns (2002) discuss how to optimize the size of the perimeter in unit cost domains, but this analysis does not apply to KKAdd.

More recently, Barker and Korf (2012) describe bidirectional breadth-first iterative deepening A*, which is an algorithm that proceeds as breadth-first iterative deepening A* would, except that it does so in both the forwards and the backwards directions. The first search iteration is in the forwards direction, but subsequent directions are chosen by comparing the total number of nodes expanded in the most recent forwards search and the most recent backwards search, and selecting whichever direction last had fewer nodes expanded. When a node is generated that has already been expanded in the opposite direction, this creates an incumbent solution. When the cost of the incumbent is the same as the current f-bound, the search can terminate immediately, because all solutions always have cost greater than or equal to the current f-bound. If the f-bound is incremented to the cost of the incumbent solution, the algorithm can terminate, without searching the last f layer. This can represent a massive reduction in computation effort, as f layers tend to grow exponentially.

One limitation of bidirectional breadth-first iterative deepening A* is the fact that the algorithm requires a heuristic function that can estimate the distance from any node to both the start state and the goal state, which is sometimes not straightforward. For example, domains where a pattern database is the heuristic would require both a forwards and a backwards pattern database if the start and goal configurations were unrelated. This can be difficult because it would require a new pattern database for each instance.

A more fundamental limitation of this approach is that it will not cut off more than the last f layer. The key advantage of KKAdd heuristic improvement is that it can be used to eliminate multiple f layers in the forward search. If the heuristic correction h_{err} is large, the f value of the head of the open list will reach that of an incumbent solution in a much lower f layer, because of the heuristic correction. For example, if there is an incumbent solution that costs 100, and a heuristic correction of 25, A* with the KKAdd heuristic can terminate in the $f = 75$ layer, because the f value of the head of the open list will be 100 with the correction. A* with the original heuristic must complete all f layers less than 100.

Felner et al. (2010) describe single-frontier bidirectional search, which is an algorithm that searches both forwards and backwards, but only maintains a single open list. At every node, the algorithm considers searching forwards and backwards, and proceeds to search in whichever direction appears to be more promising. Like perimeter search, this

algorithm requires a heuristic that is capable of evaluating the distance between any two states. Another major drawback of this algorithm is that it is brittle. In many of the experiments reported below, it does not find solutions at all due to running out of memory.

Tuning the KKAdd Heuristic

When deploying the KKAdd heuristic, the first step is to figure out exactly how much backwards search should be done. This is typically done by generating sample instances and running A* with KKAdd using a variety of backwards search sizes. The most effective parameter setting is then chosen and is used for all future instances of the domain.

This approach has two problems. First, it requires the user to spend time figuring out how much backwards search to do for each domain. Second, using a single value to determine how much backwards search to do for any given domain requires making compromises on some instances, because instances vary in how difficult they are. For example, for the Towers of Hanoi problems discussed below, the best overall average solving time results with 100,000 nodes, but this metric is deceiving. For the less difficult instances, KKAdd with 100,000 nodes in the backwards search is slower than A*, whereas KKAdd with 10,000 nodes outperforms A*.

Table 1 shows the result of using KKAdd with a variety of fixed backwards search sizes. Looking at this table, it is clear that even when considering only the average solving time, for all domains, it is still critical to select the proper amount of backwards search to do, otherwise the performance is unacceptably poor, possibly taking orders of magnitude more time and space than A*. Our objective is to come up with an algorithm that manages the number of backwards expansions on its own.

A* with Incremental KKAdd

We introduce Incremental KKAdd, a method for correcting the heuristic by incrementally increasing the backwards perimeter's size. The approach relies upon the key observation that the backwards searches can be done at any time in the search, not just at the beginning, meaning backwards search can be interleaved with forwards searches. This means that one way to effectively manage the number of expansions in the backwards search is to maintain a bounded proportion relative to the number of expansions done in the forwards search. This is the approach taken in A* with Incremental KKAdd (see Algorithm 1).

Incremental KKAdd requires a single parameter, which is the desired proportion of backwards expansion. As we shall see in the empirical results, unlike many other algorithms with parameters we consider, the parameter used by A* with Incremental KKAdd is extremely forgiving, producing reasonable results across three orders of magnitude. A* with Incremental KKAdd initially proceeds in the backwards direction (line 10), expanding a small number of nodes backwards (we used 10). Next, the algorithm expands nodes in the forwards direction (line 12) such that when finished, the ratio is equal to the proportion of expansions done in the backwards direction. Last, the algorithm doubles the num-

Algorithm 1 A* with Incremental KKAdd

```
 1: fOpen = {Initial} sorted on base f, breaking ties on base h (not corrected)
 2: bOpen = {All Goal Nodes} sorted on ε_n
 3: closed = {Initial}
 4: h_err = 0
 5: incumbent = nil
 6: function INCR KKADD A*(ratio, initial, goal)
 7:     expLimit = 10
 8:     while !solutionFound() do
 9:         searchBackwards(expLimit)
10:         if !solutionFound() then
11:             searchForwards(expLimit · (1−ratio)/ratio)
12:         expLimit = expLimit * 2
13: function SEARCHFORWARDS(count)
14:     repeat
15:         next = fOpen.pop()
16:         for all child : next.expand() do
17:             inc, direction = closed.get(child)
18:             if inc ≠ nil and direction = REV then
19:                 makeIncumbent(child, inc)
20:                 continue
21:             else if inc == nil or inc.g < child.g then
22:                 fOpen.remove(inc)
23:                 closed.remove(inc)
24:             else if inc.g ≥ child.g then
25:                 continue
26:             fOpen.add(child)
27:             closed.add(child, FWD)
28:     until solutionFound() or at most count times
29: function SEARCHBACKWARDS(count)
30:     repeat
31:         next = bOpen.pop()
32:         inc, direction = closed.get(next)
33:         if inc ≠ nil and direction = REV then
34:             continue
35:         else
36:             fOpen.remove(inc)
37:             makeIncumbent(inc, next)
38:         closed.add(next, REV)
39:         for all child : next.reverseExpand() do
40:             closedInc, dir = closed.get(child)
41:             if closedInc ≠ nil and dir = REV then
42:                 continue
43:             else if closedInc ≠ nil and dir = FWD then
44:                 makeIncumbent(closedInc, child)
45:             revInc = bOpen.get(child)
46:             if revInc = nil then
47:                 bOpen.add(child)
48:                 goalFrontier.add(child)
49:             else if revInc.g > child.g then
50:                 bOpen.replace(revInc, child)
51:     until solutionFound() or at most count times
52:     h_err = min(ε(n) ∈ goalFrontier)
```

ber of nodes that will be expanded in the backwards direction, and repeats the process.

fOpen is the forwards open list, and contains all nodes that have been generated but not yet expanded in the forwards direction, and have not been expanded backwards. bOpen is the open list for the backwards search. Closed contains all nodes that have been expanded or generated in the forwards direction, and all nodes that have been generated in the backwards direction.

The forwards search is similar to A*, with a few differences. First, the definition of a goal is broader, as now any state that was reverse expanded can be treated as a goal, because its $h^*(n)$ and path to a goal are known. In addition, goal tests are done at generation, and complete paths are either stored as a new incumbent solution or deleted (if worse than the current incumbent), and in either case, are not put on the open list (line 20). The solutionFound() function (line 28, 51) tells if an acceptable solution has been found, checking quality of the incumbent against the head of the open list. If the head of the forward search's open list ever has a corrected f value greater than or equal to that of the current incumbent, the algorithm terminates, returning the incumbent, which has now been proven to be optimal. Note that fOpen is ordered on uncorrected f values, obviating the need to resort as h_{err} rises.

The backwards search is also able to generate incumbent solutions by generating states that have already been discovered by the forwards search (line 37 and 44). Whenever the backwards search generates an incumbent solution, the quality of the incumbent is compared to the head of the forward search's open list, and if the incumbent's quality is less than or equal to the f value of the head of the forward search's open list, the algorithm immediately terminates.

The backwards search expands $count$ nodes in the backwards direction in order of increasing $g_{rev}(n)-h(n)$, where $g_{rev}(n)$ is the g value of a node in the reverse direction. The goal of the backwards search is to provide a large heuristic correction, and expanding nodes in order of heuristic error will raise the heuristic correction most quickly. This requires that the g_{rev} values be optimal, which we now prove.

Theoretical Properties

Lemma 1. (Lemma 5.2 of Kaindl and Kainz (1997)) *In any domain with a consistent heuristic, heuristic error is non-increasing along any optimal path to a goal.*

Theorem 1. *In any domain with a consistent heuristic, expanding nodes backwards from the goal in $g_{rev}(n) - h(n)$ order results in each node being expanded with its optimal g_{rev} value, and therefore the g_{rev} value is the same as the h^* value.*

Proof. First, let us suppose that a node $n1$ is the first node with the smallest value of $g_{rev}(n1) - h(n1) = minerr$ that does not have $g_{rev}(n1) = g^*_{rev}(n1)$. This means that there exists some positive δ such that $g_{rev}(n1) - \delta = g^*_{rev}(n1)$.

In order to expand this $n1$ with its optimal g_{rev} value, there must exist some other node $n2$ that should instead be expanded, which will eventually lead to $n1$ via the optimal reverse path. Since we assumed that $n1$ had the minimum value of $g_{rev}(n1) - h(n1)$, we know:

$$g_{rev}(n1) - h(n1) \leq g^*_{rev}(n2) - h(n2) \quad (1)$$

Since the optimal path backwards from the goal to $n1$ goes through $n2$, by Lemma 1 we also know:

$$g^*_{rev}(n2) - h(n2) \leq g^*_{rev}(n1) - h(n1) \quad (2)$$

Adding δ to both sides of Equation 2, and making the substitution $g^*_{rev}(n1) + \delta = g_{rev}(n1)$ on the right side yields:

$$g^*_{rev}(n2) - h(n2) + \delta \leq g_{rev}(n1) - h(n1) \quad (3)$$

Equations 1 and 3 yield:

$$g^*_{rev}(n2) - h(n2) + \delta \leq g^*_{rev}(n2) - h(n2) \quad (4)$$

which is a contradiction. □

Thus, the strategy of ordering the backwards search by error preserves the correctness of the heuristic correction.

A* with Incremental KKAdd relies upon the heuristic correction to reduce the number of expansions compared to A*. Although an improved heuristic is generally a good thing, it is known that, even when one heuristic strictly dominates another, it is still possible for A* with the weaker heuristic to terminate sooner than A* with the stronger heuristic (Holte 2010), due to tie breaking in the final f layer. An important question to ask is if this can happen with the Incremental KKAdd heuristic. Theorem 2 tells us this will not happen.

Theorem 2. *The forwards search of A* with Incremental KKAdd will never expand a node that A* would not expand, if both algorithms are tie breaking in favor of low h, and any remaining ties are broken identically.*

Proof. A* with Incremental KKAdd sorts nodes on their f value, breaking ties in favor of low h, just as A*. Thus, A* with Incremental KKAdd will have the same nodes on the open list as A* with one exception: since nodes that have been reverse expanded have the optimal path to the goal known, these nodes are not expanded, so these nodes, and all of their children, will never be on the open list of A* with Incremental KKAdd. This pruning may cause other nodes currently on the open list of A* with Incremental KKAdd to have g values that are too large (if the optimal g value is achieved via a path going through a pruned node), delaying the node's expansion in A* with Incremental KKAdd. Since there is already a path to the goal as good as the path going through these nodes, A* gains nothing by expanding these nodes earlier, as A* with Incremental KKAdd already has an incumbent solution as good as any solution going through these nodes. □

Corollary 1. *The only nodes expanded by A* with Incremental KKAdd that A* might not also expand are the nodes expanded backwards from the goal.*

Proof. Direct consequence of Theorem 2. □

The ratio of backwards expanded nodes to forwards expanded nodes is bounded after the first iteration of forwards search by three times the ratio. When the backwards search begins the perimeter's size is $ratio \cdot forwardExp$, and after the backwards search terminates, the perimeter's size is $ratio \cdot forwardExp + 2 \cdot ratio \cdot forwardExp$. Thus, the size of the perimeter is never more than three times the ratio times the number of forward expanded nodes.

A* with Adaptive Incremental KKAdd

Incremental KKAdd requires the user to decide what portion of effort should be dedicated to searching backwards. This raises two issues. First, while specifying a ratio is more robust than an absolute number, the ratio does make a difference, and the user might not know an ideal value, and second, the algorithm is unable to adjust to the domain or instance: sometimes backwards search is extremely helpful and more would provide further benefit, other times backwards search is not helpful at all and should be stopped. Incremental KKAdd commits to a specific ratio, which while it bounds the worst case, may not be the best ratio.

A* with Adaptive Incremental KKAdd eliminates the ratio parameter and uses experience during search to decide if it should expand nodes forwards or backwards. Intuitively, the algorithm must answer the question "will the overall search progress faster expanding backwards or forwards?". To answer this question, we begin with the observation that the termination condition of the search is when the incumbent solution's cost is less than or equal to the f value of the head of the open list. The f value of the head of the open list can be split into two independent components: the base f in the forwards direction (without the heuristic correction) and the heuristic correction from the backwards direction. The goal, therefore, is to increase the sum of these quantities using the fewest expansions possible. The algorithm must figure out which quantity is rising faster per expansion: the minimum f from the forward search's open list, or the h_{err} correction from the backwards search.

To do this, the algorithm begins by using a fixed ratio (we used 1:10) and doing three backwards/forwards iterations of A* with Incremental KKAdd using this ratio. After each forward search, it records the minimum f and number of forwards expansions done in the forwards search and h_{err} and the number of backwards expansions done in the backwards search. After three backwards/forwards iterations, we have three data points that can be fit using linear regression to predict how future expansions can be expected to modify h_{err} and the f value of the head of the open list. We selected the number three because the minimum number of points needed for linear regression is two, and three provides additional robustness to the estimate, while four can give too much overhead. In most domains, the number of nodes with an f value below a given threshold increases exponentially with f (This is also known as the heuristic branching factor (Korf, Reid, and Edelkamp 2001)). Thus, it is not unreasonable to assume that log(expansions) is linearly proportional to the f value of the head of the open list. Our experiments have also shown that the relationship between the log(backwards expansions) is also linearly proportional to h_{err}, as the correlation coefficients are reasonable (all are above .83).

Once A* with Adaptive Incremental KKAdd has completed three forwards/backwards iterations, it considers doubling the size of either the forwards search or the backwards search. In order to decide which is a better investment, it calculates the expected increase in f per expansion if the forward open list's size is doubled, and compares that to the expected increase in h_{err} per expansion if the backwards search's expansions are doubled. Once it figures out which direction looks more promising, it doubles the size of the search in that direction, adding an additional data point informing the next prediction.

IDA* with Incremental KKAdd

Iterative Deepening A* (Korf 1985) can also be augmented using Incremental KKAdd. For large problems where duplicate states are rare, IDA*, rather than A*, is the state of the art algorithm. In IDA* with Incremental KKAdd, the forwards IDA* search is identical to an ordinary IDA* search, except it evaluates nodes using the KKAdd heuristic, and whenever the forwards search finds a node that has been reverse expanded, that node is turned into an incumbent solution and not expanded. The backwards search is unmodified, except it no longer needs to look for nodes that have been expanded in the forwards direction, since the forwards search is no longer storing nodes. In order to meet its time complexity bound, IDA* doubles the amount of work done in each iteration. IDA* with Incremental KKAdd can take advantage of this property by doing backwards expansions between forwards IDA* iterations, expanding enough nodes to achieve the desired ratio. Note that the backwards search may change the heuristic correction, and this may interfere with the forward search's ability to double. The overall performance will still be superior to that of IDA* without the improved heuristic, because each iteration will be smaller than the corresponding uncorrected IDA* iteration.

One possible complication with integrating the Incremental KKAdd heuristic into IDA* is the fact that calculating the KKAdd heuristic for a node requires looking up that node in a hash table. When doing A*, this hash table lookup is already done as a part of duplicate detection, but many implementations of IDA* omit cycle detection. In many domains, looking up a state in a hash table is substantially more expensive than expanding the state, which can cut into the overall savings.

Empirical Evaluation

We compare A* with Incremental KKAdd and Adaptive Incremental KKAdd A* against A*, A* with the KKAdd heuristic, single frontier bidirectional search, and perimeter search on seven different popular heuristic search benchmarks. We used a Java implementation of all the domains and algorithms, running on a Core2 duo E8500 3.16 GHz with 8GB RAM under 64 bit Linux.

For A* with Incremental KKAdd we selected three values for the ratio: 0.1, 0.01, and 0.001. If the ratio gets larger than 0.1, in many domains, the overhead becomes exces-

sive. Values smaller than 0.001 would have resulted in only a handfull of backwards expansions. For perimeter search, we ran with a variety of perimeters ranging from 1 to 50, and only consider the best setting for each domain. For A* with the KKAdd heuristic, we considered backwards searches varying in size from 10 to 1,000,000. For single frontier bidirectional search, we implemented the enhancements described by Lippi, Ernandes, and Felner (2012) and tried both enhanced single frontier bidirectional search (eSBS) and enhanced single frontier bidirectional search lite (eSBS-l), using the always jump (A) jumping policy. Our implementation of eSBS produces comparable node and expansion counts to those reported by Lippi, Ernandes, and Felner (2012), but our times are sometimes substantially slower for the grid and tiles domains, although on pancakes, both the times and node counts roughly align. To ensure that we don't discount an algorithm for which implementation tuning would provide an important advantage, we provide data on three metrics: runtime using our implementation, the number of times either a forwards or reverse expansion was requested, and the number of times the domain's base heuristic function was called. Sometimes, one or more of these algorithms ran out of either memory (7.5GB) or time (10 minutes) for one or more instances, and when that occurred, results for that configuration were marked as DNF.

The results of this comparison are shown in Table 1. The first domain we consider is dynamic robot path planning (Likhachev, Gordon, and Thrun 2003). We used a 50x50 world, with 32 headings and 16 speeds. The heuristic is the larger of two values. The first value estimates time to goal if the robot was able to move along the shortest path at maximum speed, which the real robot cannot do due to restrictions on its turn rate and acceleration. The second value makes the free space assumption, and estimates time to goal given the robot's current speed and the need to decelerate to zero at the goal. These worlds are unusually small, which was necessary to accommodate the algorithms that required a point-to-point heuristic (all pairs shortest paths) in this domain. To compensate for the easiness associated with having a small map, 35% of the cells were blocked randomly. All nodes in the space have a certain amount of heuristic error due to the fact that neither of these heuristics is a particularly accurate measure of the true cost of bringing the robot into the goal. As a result, the backwards search is extremely effective, correcting significant error in the heuristic. A* with all varieties of Incremental KKAdd are able to handily outperform A* in this domain. Here, perimeter search is not particularly effective, being slower than A*. The lite varieties of single frontier bidirectional search did not terminate. Across all instances, eSBS(A) averaged 123 seconds, which is substantially slower than any of the other algorithms.

The next domain is city navigation, a domain which is designed to simulate a hub and spoke transportation network (Wilt and Ruml 2012). We used 1500 cities and 1500 places in each city, with each city connected to its 5 nearest neighbors, and each place connected to its 5 nearest neighbors. In this domain, the heuristic is euclidean distance, which assumes it is possible to move directly from one location to another. The backwards search is able to correct some of the heuristic error, which makes the A* with Incremental KKAdd variants faster than A*. Perimeter search, with its additional heuristic evaluations and lack of heuristic correction, was naturally slower. All varieties of eSBS did not terminate on this domain.

The third domain we consider is the Towers of Hanoi. We considered a 14 disk 4 peg problem with two disjoint pattern databases (Korf and Felner 2002) tracking the bottom 12 disks and the top two disks respectively. In this domain, the backwards search is able to identify the fact that some of the states have substantial heuristic error, due to the fact that the disjoint pattern databases do not accurately track the relationship between the top two disks and the bottom 12 disks. Thus, error correction is extremely helpful, and allows the A* with Incremental KKAdd variants to terminate much faster than basic A*. Since it is not clear how to adapt the informative pattern database heuristic to give point-to-point information, we could not run either perimeter search or eSBS on this domain (N/A in Table 1).

The fourth domain is a TopSpin puzzle with 14 disks and a turnstile that swaps the orientation of 4 disks. For a heuristic, we used a pattern database that contained information about 7 disks. In this domain, the backwards searches were never able to find a correction to the heuristic, because there were a large number of nodes where the heuristic is perfect. Since the backwards searches were never able to find a heuristic correction, the investment A* with Incremental KKAdd variants make in the backwards search turns out to be unnecessary. The amount of investment in the unneeded backwards search is bounded, so the overall performance of A* with Incremental KKAdd is only slightly slower than A*.

The fifth domain is the sliding tile puzzle (15). We filtered out instances that any algorithm failed to solve. The A* with Incremental KKAdd variants are actually able to solve more instances (90) than A* (83), although not as many instances as Perimeter search (94). In the sliding tile puzzles, backwards search failed to derive a heuristic error correction. In this domain, like in TopSpin, without the benefit of a heuristic error correction the A* with Incremental KKAdd variants are not able to terminate early, but once again, we see that the overhead compared to A* is bounded.

A similar phenomenon can be observed in the 40 pancake problem. We used the gap heuristic (Helmert 2010). which is extremely accurate. None of the A* with Incremental KKAdd variants were able to get a heuristic correction, and their tie breaking was not much more effective than the tie breaking inherited from A*. Thus, without any benefit from the additional overhead, the A* with Incremental KKAdd variants performed marginally worse than A* on this problem as well. eSBSA(l) is extremely effective in this domain, and its counterpart eSBSA is competitive with A*. Perimeter search took longer than A* to find solutions.

The seventh and last domain we consider is grid path planning. The grid path planning problems we considered were 2000x1200, with 35% of the grid cells blocked. We used a 4 way motion model and the Manhattan distance heuristic. The A* with Incremental KKAdd variants were able to get a heuristic correction, and this correction allowed them to terminate the forwards search earlier than A*. Unfortu-

Domain		A*	KKAdd						A* with Incremental KKAdd				Peri-	SBS	SBS
			10	10^2	10^3	10^4	10^5	10^6	Adpt	10^{-1}	10^{-2}	10^{-3}	meter	(A)	(A)-l
	T	1.06	1.02	0.62	0.47	0.67	4.13	34.64	0.82	**0.45**	0.54	0.82	2.00	123	
Robot	E	53.1	40.2	21.3	**11.8**	17.9	101.0	683	**11.8**	12.6	18.4	31.1	67.3	25.8	DNF
	H	2,475	1,745	787	383	**340**	2,061	30e3	380	410	693	1,305	3,257	1,087	
	T	0.89	1.05	0.96	0.82	0.78	0.90	8.32	**0.70**	0.72	0.77	0.94	1.04		
CityNav	E	192	175	157	140	100	134	1,000	108	**92.5**	119	151	176	DNF	DNF
	H	1,184	1,056	947	845	604	826	6,005	656	**561**	716	908	6,345		
	T	23.2	21.3	16.9	10.6	5.3	**3.91**	18.4	4.79	4.33	6.27	10.88			
Hanoi	E	1,457	1,457	1,185	769	389	272	1,051	310	**266**	413	694	N/A	N/A	N/A
	H	5,583	5,583	4,542	2,951	1,513	1,259	6,194	1,251	**1,094**	1,599	2,665			
	T	4.88	4.88	4.90	**3.50**	3.80	6.89	42.79	5.27	5.27	5.13	4.97	3.71	7.99	
TopSpin	E	94	93	94	94	103	197	1,089	94	108	108	94	93	**21**	DNF
	H	1,319	1,315	1,316	1,329	1,452	2,718	15e3	1,329	1,516	1,330	1,317	1,313	**306**	
	T	17.47	18.45	18.45	18.58	18.63	19.56	25.03	18.45	18.40	19.65	19.01	**10.11**	36.56	31.26
Tiles	E	2,922	2,918	2,919	2,920	2,926	3,019	3,885	2,919	2,920	3,004	2,929	1,082	**98**	**98**
	H	5,790	5,786	5,786	5,789	5,813	6,112	9,026	5,786	8,424	6,066	5,816	32e3	**202**	**202**
	T	1.88	1.93	1.86	2.04	2.18			2.13	2.62	1.96	1.89	2.15	1.88	**1.19**
Pancake	E	19	19	19	20	29	DNF	DNF	21	25	20	19	19	**3**	**3**
	H	771	772	775	810	1,161			825	990	793	774	771	**132**	133
	T	**1.24**	1.27	1.28	1.26	1.49	1.49	4.25	1.28	1.30	1.27	1.34	1.31		
Grid	E	382	382	381	376	370	429	1,183	373	385	**371**	378	382	DNF	DNF
	H	638	638	636	629	628	819	2,977	638	686	**624**	632	638		

Table 1: CPU seconds (T), expansions (E), and heuristic evaluations (H) (both in thousands) required to find optimal solutions.

nately, the backwards search required to get the correction required approximately the same number of nodes as the forwards search was able to save, resulting in no net performance gain.

Overall, we can see that in domains like dynamic robot navigation, Towers of Hanoi, and City Navigation, where there is a large heuristic error correction, the A* with Incremental KKAdd searches are all able to find optimal solutions faster than A* by a sizable margin. In domains like TopSpin, sliding tiles, and the pancake puzzle, in which the backwards search failed to derive a heuristic correction, the overall performance was comparable to A*, with only a small amount of overhead lost to the backwards search. In domains like grid path planning, the backwards search is able to correct the forwards heuristic, but the cost of the correction is approximately equal to the savings in the forward search, so there is no net gain. In every situation, the A* with Incremental KKAdd searches are competitive with A*, due to the fact that the amount of work they do above and beyond A* is, in the worst case, bounded by a constant factor, and in the best case, there is a substantial speedup over A*. Although the other bidirectional searches are sometimes able to outperform both A* and the A* with Incremental KKAdd variants, they are plagued by brittleness: on some domains, both SBS and Perimeter Search are substantially slower than A*, or are not able to find solutions at all.

We can also see that the A* with Incremental KKAdd variants do a reasonable job of capturing the potential of the KKAdd heuristic without the tedium of having to manually tune the size of the backwards search, and without the risk of sometimes getting extremely poor performance, as sometimes happens if the KKAdd heuristic is used with too much or too little backwards search.

Conclusion

We have introduced a significantly improved incremental variant of the KKAdd procedure of Kaindl and Kainz (1997) that preserves its benefits while provably bounding its overhead. A* with Incremental KKAdd is empirically an effective way to speed up optimal heuristic search in a variety of domains. We have also shown that, in the worst case, the amount of additional work done is bounded by a constant factor selected at runtime, and that numbers ranging from 0.1 to 0.001 all result in reasonable performance, producing an algorithm that is extremely effective in the best case, and only induces a minimal constant factor additional work in the worst case. We also introduce A* with Adaptive Incremental KKAdd, which manages the ratio of forwards expansions to backwards expansions on its own and reliably yields good performance.

Both A* with Incremental KKAdd and A* with Adaptive Incremental KKAdd are able to provide substantial speedup as compared to A*. Unlike other bidirectional search algorithms that are sometimes much slower than A* or unable to find solutions at all, in the worst case, the A* with Incremental KKAdd varieties always return a solution in comparable time to A*, making Incremental KKAdd a robust and practical alternative to previous state-of-the-art bidirectional searches and A*.

Acknowledgements

Our thanks to Michael Leighton, who played a significant role early in the development of this paper. We also gratefully acknowledge support from NSF (grants 0812141 and 1150068) and DARPA (grant N10AP20029).

References

Barker, J. K., and Korf, R. E. 2012. Solving peg solitaire with bidirectional BFIDA. In *AAAI*.

Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *Journal of the Association for Computing Machinery* 32(3):505–536.

Dillenburg, J. F., and Nelson, P. C. 1994. Perimeter search. *Artificial Intelligence* 65(1):165–178.

Felner, A.; Moldenhauer, C.; Sturtevant, N. R.; and Schaeffer, J. 2010. Single-frontier bidirectional search. In *AAAI*.

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

Helmert, M. 2010. Landmark heuristics for the pancake problem. In *Proceedings of the Third Symposium on Combinatorial Search*.

Holte, R. C. 2010. Common misconceptions concerning heuristic search. In *Proceedings of the Third Annual Symposium on Combinatorial Search*, 46–51.

Kaindl, H., and Kainz, G. 1997. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research* 7:283–317.

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

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening-A*. *Artificial Intelligence* 129:199–218.

Korf, R. E. 1985. Iterative-deepening-A*: An optimal admissible tree search. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, 1034–1036.

Likhachev, M.; Gordon, G.; and Thrun, S. 2003. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of the Seventeenth Annual Conference on Neural Information Processing Systems*.

Lippi, M.; Ernandes, M.; and Felner, A. 2012. Efficient single frontier bidirectional search. In *Proceedings of the Fifth Symposium on Combinatorial Search*.

López, C. L., and Junghanns, A. 2002. Perimeter search performance. In *Computers and Games*, 345–359.

Manzini, G. 1995. BIDA: An improved perimeter search algorithm. *Artificial Intelligence* 75(2):347–360.

Wilt, C., and Ruml, W. 2012. When does weighted A* fail? In *Proceedings of the Fifth Symposium on Combinatorial Search*.