Iterative-Deepening Uniform-Cost Heuristic Search

Zhaoxing Bu, Richard E. Korf

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
{zbu, korf}@cs.ucla.edu

Abstract

Breadth-first heuristic search (BFHS) is a classic algorithm for optimally solving heuristic search and planning problems. BFHS is slower than A* but requires less memory. However, BFHS only works on unit-cost domains. We propose a new algorithm that extends BFHS to domains with different edge costs, which we call uniform-cost heuristic search (UCHS). Experimental results show that the iterative-deepening version of UCHS, IDUCHS, is slower than A* but requires less memory on a variety of planning domains.

Introduction and Overview

A* (Hart, Nilsson, and Raphael 1968) is a classic heuristic search algorithm used to solve search and planning problems. Many state-of-the-art optimal planners use A* as their search algorithm (Katz et al. 2018; Franco et al. 2017, 2018; Martinez et al. 2018). A* stores all generated nodes in either the Open or Closed list, and never expands a state more than once with consistent heuristics. Its main drawback is its exponential space requirement, and A* can run out of memory in minutes on common search and planning problems.

To solve problems that A* cannot solve due to memory limitations, various algorithms have been proposed. These can be divided into two categories: depth-first algorithms and frontier search. For example, Iterative-Deepening-A* (IDA*, Korf 1985) is a depth-first algorithm that only stores the nodes on the current search path. Thus, its space requirement is only linear in the maximum search depth. IDA* cannot detect when the same state is arrived at via different paths, and hence will generate duplicate nodes on problems with multiple paths between the same pair of nodes. To address this issue, IDA* variants such as IDA* with a transposition table (IDA*+TT, Sen and Bagchi 1989; Reinefeld and Marsland 1994) and A*+IDA* (Bu and Korf 2019) store nodes up to some memory bound, and check for duplicates among those stored nodes. However, these algorithms still generate many duplicate nodes when the problem is large enough that only a small fraction of the nodes can be stored in memory (Zhou and Hansen 2004; Bu and Korf 2019).

The second category is frontier search (Korf et al. 2005), which is a family of algorithms that only stores the search frontier in memory and removes expanded nodes whenever possible. Unlike IDA*, frontier search algorithms detect duplicate nodes and are guaranteed not to expand a state more than once on undirected graphs. One example of frontier search is breadth-first heuristic search (BFHS, Zhou and Hansen 2004), which stores fewer nodes than A* on a variety of planning domains (Bu and Korf 2021). However, BFHS only works on problems with unit-cost edges.

This paper introduces a new algorithm called uniform-cost heuristic search (UCHS), which extends BFHS to domains with arbitrary non-negative edge costs. First, we review BFHS and related algorithms. Second, we describe UCHS, and prove that it never expands a state more than once on undirected graphs. Third, we discuss combining UCHS with iterative-deepening (IDUCHS). Fourth, we compare A* and IDUCHS on International Planning Competition (IPC) domains with non-unit edge costs. Experimental results show that IDUCHS is slower than A*, but requires less memory than A*, sometimes by more than an order-of-magnitude, on a variety of planning domains.

Previous Work

Divide-and-Conquer Frontier-A* (DCFA*, Korf and Zhang 2000) is a frontier search algorithm based on A*. The difference between A* and DCFA* is that DCFA* stores the entire Open list and only a few layers of the Closed list, which can be done by deleting a parent node after expanding all its child nodes. Compared to A*, DCFA* saves a large amount of memory when the Closed list is significantly larger than the Open list. On the other hand, little memory is saved on problems where the Open list is much larger than the Closed list, which is often the case for exponential problems.

Breadth-First Frontier Search (BFFS, Korf 2004) is the breadth-first version of frontier search, and applies to unit-cost domains. BFFS deletes all closed nodes at depth d after expanding all nodes at depth $d+1$. For each stored node n, we define its delete g-value, $d_g(n)$, as the minimum g-value such that once all nodes with g-value of $d_g(n)$ or less have been expanded, then node n can be safely deleted from memory. For BFFS, $d_g(n)$ is simply one level greater than the depth of n.

$$d_g(n) = g(n) + 1 \quad (1)$$

For example, if a node n is at depth 6, or $g(n)=6$, then n is deleted after BFFS expands all nodes at depth 7.

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

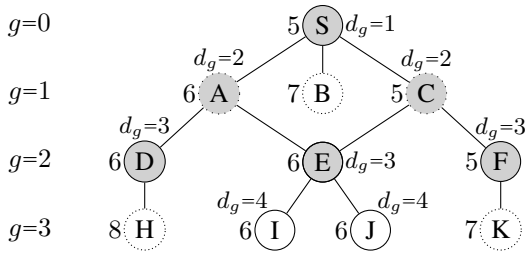

Figure 1: An example of BFHS with a cost bound of 6. f-values are at the left of nodes. Closed nodes are gray. Deleted nodes are dotted. d_g is the delete g-value.

Breadth-first heuristic search (BFHS, Zhou and Hansen 2004) is another frontier search algorithm that works on unit-cost domains, and uses less memory than A*. It is BFFS with heuristics to prune nodes. BFHS requires a user-specified cost bound U, and prunes every generated node whose f-value is greater than U.

We present an example of BFHS with $U = 6$ on an undirected graph in Figure 1, where all edge costs are 1, the numbers next to the nodes are their f-values, Closed nodes are gray, and deleted nodes are dotted. The start node S generates nodes A, B, and C. $f(\text{B}) = 7 > U = 6$ so B is immediately deleted, while A and C are stored at depth 1. Next, BFHS expands the Open nodes at depth 1. A generates S, D, and E. S is a duplicate node, while D and E are stored at depth 2. C generates S, E, and F. S and E are duplicate nodes, while F is stored at depth 2. Although $d_g(\text{S}) = 1$, S is not deleted since it is the start node. Then BFHS expands the Open nodes at depth 2. D generates A and H. A is a duplicate node, while H is immediately deleted since $f(\text{H}) > U$. E generates A, C, I, and J. A and C are duplicate nodes, while I and J are stored at depth 3. F generates C and K. C is a duplicate node, while K is immediately deleted since $f(\text{K}) > U$. After all Open nodes at depth 2 are expanded, BFHS deletes the nodes at depth 1, A and C, which have d_g-values of 2, since they cannot be generated again. BFHS then expands the Open nodes at depth 3 and continues until it finds a goal node, or proves that no solution exists within U.

BFHS searches for solutions within a given cost bound. If the bound is smaller than the optimal solution cost C^*, it may never generate a goal node. If the bound is too large, it may generate many nodes n for which $f(n) > C^*$. C^* is rarely known in advance. Zhou and Hansen (2004) proposed BFIDA*, which runs a series of iterations of BFHS with increasing cost bounds. The first cost bound is the heuristic value of the start node, and the last bound is C^*.

When BFIDA* finds a goal node, the optimal cost is known, but not the solution path, since most Closed nodes have been deleted. To reconstruct the solution path, BFIDA* saves a middle layer of nodes during the search and never delete this layer. For example, BFIDA* can use the nodes stored at depth $U/2$ as the middle layer and every node below this depth will have a pointer to its ancestor in the middle layer. When BFIDA* generates a goal node, it knows which node in the middle layer led to this goal node. Then BFHS can be called to reconstruct the optimal path from the start node to the middle node, and from the middle node to the goal node (Zhou and Hansen 2004).

BFIDA* can generate more nodes than A* since it generates and expands almost all nodes n for which $f(n) = C^*$, due to its breadth-first node ordering. A*+BFHS (Bu and Korf 2021) overcomes this issue by first running A* up to a user-specified storage threshold, then runs multiple iterations of BFHS on the Open nodes of A*. Each call to BFHS in A*+BFHS starts with the Open nodes at one or more depths instead of a single node. In general, A*+BFHS is faster than BFIDA* and uses less memory, but A*+BFHS is also limited to unit-cost domains.

Dijkstra's single-source shortest path algorithm (Dijkstra 1959) generalizes breadth-first search (BFS) to the case of non-uniform edge costs. The version that terminates when a goal node is chosen for expansion is usually called Uniform-Cost Search (UCS), despite the fact that it deals with non-uniform edge costs. This terminology comes from the fact that the nodes on Open tend to have uniform g-values.

The graph search version of Iterative Budgeted Exponential Search (IBES, Helmert et al. 2019) utilizes UCS to minimize the worst-case number of node expansions on domains with various edge costs and inconsistent heuristics. However, IBES does not remove Closed nodes from memory so it is not directly comparable to our new algorithm.

Uniform-Cost Heuristic Search

We first describe Uniform-Cost Frontier Search (UCFS), a brute-force algorithm that extends BFFS to the case of non-unit edge costs.

Uniform-Cost Frontier Search

The difference between UCFS and UCS is that UCFS deletes a node after all its child nodes have been expanded, similar to the difference between BFFS and BFS. This prevents a parent node from being re-expanded. Suppose p is a parent node, and $\Phi = \{n_i | i = 0, 1, 2...\}$ are its children. In UCFS, the delete g-value of node p, $d_g(p)$, is the maximum of the g-values of its children.

$$d_g(p) = \max_{n_i \in \Phi} g(n_i) \tag{2}$$

For example, if p has two children with g-values of 5 and 6, then p can be deleted after expanding all nodes with $g = 6$.

UCFS works as follows. It first expands the start node and puts all child nodes on Open. The Open list is kept sorted by the g-values of the nodes. After all nodes on Open whose g-value is 0 are expanded, in the case of zero-cost edges, assume the next smallest g-value on Open is 3. UCFS deletes the expanded nodes whose d_g-value is less than 3. Next, it expands all nodes on Open whose g-value is 3. It continues expanding and deleting nodes until it finds a goal node on an optimal path. On unit-cost domains, BFFS can terminate when a goal node is generated. On domains with non-unit edge costs, however a node may be generated with a g-value greater than its minimal g^*-value. Therefore, UCFS only terminates when a goal node is chosen for expansion.

Figure 2 presents an example of UCFS on an undirected graph, where numbers in parentheses are edge costs. First

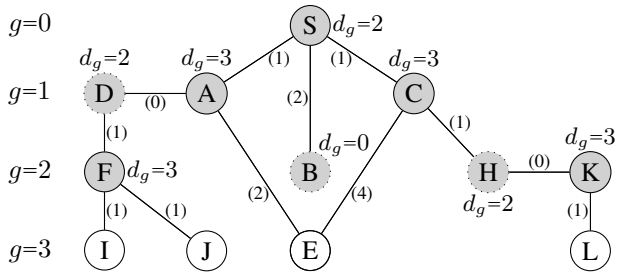

Figure 2: An example of UCFS. Numbers in parentheses are edge costs. Closed nodes are gray. Deleted nodes are dotted. d_g is the delete g-value.

the start node S is expanded, generating nodes A, B, and C. $g(\text{A}) = g(\text{C}) = 1$ and $g(\text{B}) = 2$. $d_g(\text{S}) = 2$, the maximum g-value of A, B, and C. Next, A generates nodes S, D, and E. S is a duplicate node. D and E are stored in memory with $g(\text{D}) = 1$ and $g(\text{E}) = 3$, thus $d_g(\text{A}) = 3$. D generates nodes A and F. A is a duplicate node while $g(\text{F}) = 2$, hence $d_g(\text{D}) = 2$. Then C generates nodes S, E, and H. S and E are duplicate nodes. Since $g(\text{E}) = 3$ and $g(\text{H}) = 2$, $d_g(\text{C}) = 3$. At this point, all Open nodes with $g = 1$ have been expanded. Since no node has a $d_g \leq 1$, UCFS expands nodes at $g = 2$. B only generates one node which is S, thus $d_g(\text{B}) = 0$, and B can be deleted immediately. F generates D, I, and J. D is a duplicate node while $g(\text{I}) = g(\text{J}) = 3$, thus $d_g(\text{F}) = 3$. Then H generates C and K. C is a duplicate node and $g(\text{K}) = 2$, hence $d_g(\text{H}) = 2$. K generates H and L. H is a duplicate node, $g(\text{L}) = 3$, and $d_g(\text{K}) = 3$. After expanding all Open nodes with $g = 2$, nodes D and H are deleted, as they have d_g-values of 2. S cannot be deleted since it is the start node.

A key difference between BFFS and UCFS is the order of deleting nodes. In BFFS, nodes at depth d are always deleted before nodes at depth $d+1$. In UCFS, however, a node m with a larger g-value may be deleted before a node n with a smaller g-value, if any of n's children have a higher g-value. A child node may even be deleted before its parent node. For example, in Figure 2, H is removed at $d_g = 2$, while H's parent node C will be removed later at $d_g = 3$.

Uniform-Cost Heuristic Search

We now extend UCFS by adding heuristics and a cost bound U, allowing us to prune nodes whose f-values exceed U. We call the resulting algorithm uniform-cost heuristic search (UCHS). UCHS generalizes BFHS to handle non-unit edge costs. UCHS is to UCFS as BFHS is to BFFS, while UCFS is to UCS as BFFS is to BFS.

We present UCHS in Algorithm 1, where Open_k and Closed_k are Open and Closed nodes whose g-value is k. UCHS first expands the start node and adds to Open its child nodes n for which $f(n) \leq U$. If a node n is generated with $f(n) > U$, it is immediately deleted. The Open list is kept sorted by the g-values of the nodes. After all nodes on Open whose g-value is d are expanded, assume the next smallest g-value on Open is d'. UCHS deletes the expanded nodes for which $d_g < d'$. It then expands all nodes on Open whose g-value is d'. It continues until it finds a goal node, and verifies

Algorithm 1: Uniform-Cost Heuristic Search

```
1:  g(start) ← 0
2:  Open_0 ← {start}
3:  d ← 0
4:  while d ≤ U do
5:    for p ∈ Open_d do
6:      if p is goal then
7:        return d
8:      d_g(p) ← 0
9:      for n ∈ children(p) do
10:       /* Check duplicates against all stored nodes. */
11:       if n ∈ Open_k or n ∈ Closed_k then
12:         /* New path to n is not cheaper than old path. */
13:         if g(n) ≥ k then
14:           d_g(p) ← max(k, d_g(p))
15:           continue
16:         else
17:           Open_k ← Open_k \ n
18:       d_g(p) ← max(g(n), d_g(p))
19:       if f(n) ≤ U then
20:         Open_g(n) ← Open_g(n) ∪ n
21:   /* All nodes whose g-value is d are expanded. */
22:   Closed_d ← Open_d
23:   Open_d ← ∅
24:   /* g-value of the next node to expand. */
25:   d' ← min{k | Open_k ≠ ∅}
26:   for n ∈ Closed do
27:     if d_g(n) < d' then
28:       /* Delete Closed node n from memory. */
29:       Closed ← Closed \ n
30:   d ← d'
31: return ∞
```

that no cheaper path to a goal exists.

To show how UCHS works, we run it on the same undirected graph from Figure 2. We present the result in Figure 3, where the numbers next to the nodes are their f-values. Assume UCHS is called with $U = 8.5$. S generates nodes A, B, and C. $f(\text{A}) = f(\text{C}) < U$, so they are stored. $g(\text{B}) = 2$ and $f(\text{B}) = 9 > U$, B is deleted, but we still consider B when calculating $d_g(\text{S})$, which is 2. Next, A generates S, D, and E. S is a duplicate node, D and E are stored with $g(\text{D}) = 1$, $g(\text{E}) = 3$, and $d_g(\text{A}) = 3$. D generates A and F. A is a duplicate node, F is deleted since $f(\text{F}) > U$, and $d_g(\text{D}) = 2$. C generates S, E, and H. S and E are duplicate nodes, while H is stored, and $d_g(\text{C}) = 3$. At this point, all Open nodes with $g = 1$ have been expanded. UCHS continues expanding nodes as there are no nodes to be deleted. H generates C and K. C is a duplicate node, but K is stored with $g(\text{K}) = 2$, and $d_g(\text{H}) = 2$. K then generates H and L. H is a duplicate node, $g(\text{L}) = 3$, and $d_g(\text{K}) = 3$. After expanding all Open nodes with $g = 2$, nodes D and H are deleted, as they have d_g-values of 2. S cannot be deleted since it is the start node.

Similar to UCFS, in UCHS, a node stored at a larger g-value may be deleted before a node stored at a smaller g-value, or a child node may be deleted before its parent.

The definition of d_g in Equation 2 is simple and intu-

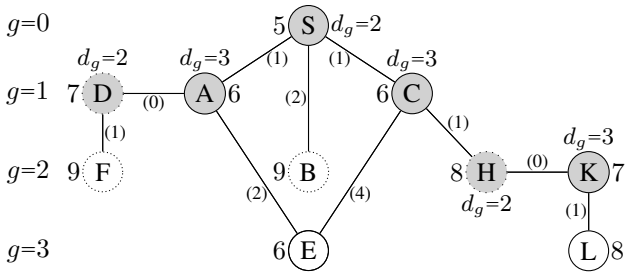

Figure 3: An example of UCHS with a cost bound of 8.5. Numbers in parentheses are edge costs and numbers next to nodes are f-values. Closed nodes are gray. Deleted nodes are dotted. d_g is the delete g-value.

itive, but on undirected graphs, it results in UCHS storing nodes for longer than it needs to. For example, D is only removed after expanding all nodes with $g = 2$ since $g(\text{F}) = 2$, even though F is not stored. This suggests that when calculating $d_g(p)$, we just ignore any of p's child nodes n for which $f(n) > U$. This choice looks tempting but it can lead to a node being re-expanded. If we used this strategy, then $d_g(\text{D}) = 1$. In our example, D is expanded before C. Suppose the problem in Figure 3 changes such that C can also generate F with an edge cost of 0.5, thus $g(\text{F}) = 1.5$, $f(\text{F}) = 1.5 + 7 = 8.5 = U$, and F will be stored. UCHS would have deleted D prior to expand F, since $d_g(\text{D}) = 1$. F would then generate D again with $g(\text{D}) = 2.5$ and $f(\text{D}) = 2.5+6 = 8.5 = U$. As a result, D will be treated as a new node and will be stored and expanded again.

This shows that we have to consider all child nodes while calculating the d_g-values, but we can actually delete D earlier. For D to be generated by F in UCHS, F must be stored and expanded. Given $U = 8.5$, the largest $g(\text{F})$ that would allow F to be stored is $U - h(\text{F}) = 8.5 - 7 = 1.5$. Therefore, if we remove D after expanding all nodes up to $g = 1.5$, then we are guaranteed that D will not be generated again by F, as F cannot be in Open when $g(\text{F}) > 1.5$.

Another case to consider is when to delete K. $d_g(\text{K}) = 3$ due to $g(\text{L}) = 3$. Since c(K,L), the cost of the edge between K and L, is 1, when L generates K, we will have $g(\text{K}) = g(\text{L})+1 = 3+1 = 4$ and $f(\text{K}) = g(\text{K})+h(\text{K}) = 4+5 = 9 > U = 8.5$, which means this new copy of K will be deleted. This suggests that instead of deleting K after expanding all nodes with $g = 3$, we can delete K before expanding any node with $g = 3$. To calculate $d_g(\text{K})$, in addition to the value of $g(\text{L})$, we also consider $U - c(\text{K,L}) - h(\text{K}) = 8.5 - 1 - 5 = 2.5$. This means that if L is stored with $g(\text{L}) \leq 2.5$, then during the expansion of L, K will be generated with $f(\text{K}) = g(\text{L}) + c(\text{K,L}) + h(\text{K}) \leq 2.5+1+5 = 8.5 = U$. In other words, if L is expanded with $g(\text{L}) > 2.5$, then the new copy of K generated by L will have $f(\text{K}) > U$, and will be deleted. Therefore, UCHS can use $d_g(\text{K}) = 2.5$, and K will be deleted before UCHS expands any nodes with $g = 3$.

Taking the above examples into consideration, we propose a new definition of d_g. Suppose U is the cost bound, p is a parent node, $\Phi = \{n_i | i = 0, 1, 2...\}$ are the children of p, and c_i is the cost of the edge between p and n_i. Then $d_g(p)$ can be defined as follows:

$$d_g(p) = \max_{n_i \in \Phi} \{ \min\{g(n_i), U - h(n_i), U - c_i - h(p)\}\} \quad (3)$$

Theorem 1. *UCHS using Equation 3 is guaranteed not to expand a state more than once on undirected graphs.*

Proof. We consider $f(n_i)$ of a newly generated node n_i.

Case 1: $f(n_i) \leq U$. n_i is stored and $g(n_i) \leq U - h(n_i)$, therefore we only need to compare $g(n_i)$ with $U - c_i - h(p)$.

(a): $g(n_i) \leq U - c_i - h(p)$. When UCHS expands n_i, p is still in memory, so the new copy of p will be deleted.

(b): $g(n_i) > U - c_i - h(p)$. UCHS may delete p before expanding n_i. However, p will only be deleted after expanding all nodes at $g = U - c_i - h(p)$. Thus, even if n_i is expanded at $g(n_i) > U - c_i - h(p)$, we still have $f(p) = g(n_i) + c_i + h(p) > U - c_i - h(p) + c_i + h(p) = U$. Therefore, this new copy of p will be immediately deleted. It is possible that after the expansion of p, n_i is generated again with a smaller g-value, but the above analysis still holds.

Case 2: $f(n_i) > U$. Since $g(n_i) > U - h(n_i)$, we only need to consider $U - h(n_i)$ and $U - c_i - h(p)$.

(a): $U - h(n_i) \leq U - c_i - h(p)$. p will not be deleted before expanding all nodes up to $g = U - h(n_i)$, the largest value of $g(n_i)$ such that n_i is stored and expanded. Therefore, if n_i is ever expanded, for example via a different and cheaper path, it must be expanded before p is removed.

(b): $U - h(n_i) > U - c_i - h(p)$. The analysis is similar to Case 1 (b) and is omitted here. □

On directed graphs, the above analysis and the guarantee that no state will be expanded more than once no longer holds, and p may be expanded multiple times. The reason is that a node we have not seen yet can regenerate node p. However, this is the price we have to pay as frontier search algorithms in general cannot prevent expanded nodes from being re-generated and re-expanded on directed graphs (Korf et al. 2005; Zhou and Hansen 2004).

Iterative-Deepening UCHS (IDUCHS)

UCHS requires a user-specified cost bound, but the optimal solution cost C^* is rarely known in advance. Therefore, we add iterative-deepening to UCHS, which we call IDUCHS. IDUCHS generalizes BFIDA* to domains with non-unit edge costs. On unit-cost domains, BFIDA* usually increases the cost bound by one with each successive iteration. This strategy does not work well on domains with non-unit edge costs, however, as there may be too many iterations with very few new nodes expanded in each iteration.

On domains with non-unit edge costs, IDA* (Korf 1985) has the same problem, as it uses the smallest f-value among the nodes generated but not expanded on the previous iteration as the cost bound of the next iteration. IDA*_CR (Sarkar et al. 1991) addresses this issue by using a larger cost bound increment between iterations. During each iteration, it counts the number of nodes generated at each f-value greater than the current cost bound. After each iteration, it decides how many new nodes to expand in the next iteration, say m. It then determines an f-value that will result in

at least m new nodes being expanded in the next iteration, and uses that as the cost bound for the next iteration.

The same idea can also be applied to UCHS on domains with non-unit edge costs. We call the resulting algorithm UCHS_CR, but it did not work well, for several reasons. First, UCHS_CR uses duplicate detection while IDA*_CR does not. As a result, UCHS_CR is not guaranteed to expand m new nodes in the next iteration. Second, on planning domains, doubling the number of expanded nodes does not guarantee that the number of generated nodes will also be doubled. Third, sometimes the number of nodes generated but not expanded is lower than the desired m-value, making it difficult to choose the next cost bound.

Korf, Reid, and Edelkamp (2001) showed that compared to brute-force search, the effect of heuristic functions in IDA* is to reduce the search depth instead of reducing the branching factor. They accurately predicted the performance of IDA* on sliding-tile puzzles and Rubik's Cube. Inspired by their work, we propose a new way to calculate the cost bounds for IDUCHS on domains with non-unit edge costs. Let f_k be the cost bound of iteration k and N_k be the number of nodes generated in iteration k. We define a pseudo branching factor b by the equation $b^{f_k - f_{k-1}} = N_k / N_{k-1}$. Note that b is not the brute-force branching factor of the problem domain, but tends to remain constant between iterations. Suppose we want $r = N_{k+1}/N_k$, where r is a user-specified ratio of the generated nodes between iterations. Then we have $b^{f_{k+1} - f_k} = r$. Taking the logarithm of both sides, we get $(f_k - f_{k-1}) \ln b = \ln (N_k / N_{k-1})$ and $(f_{k+1} - f_k) \ln b = \ln r$. Combining both equations, we get

$$\frac{\ln (N_k / N_{k-1})}{(f_k - f_{k-1})} = \ln b = \frac{\ln r}{f_{k+1} - f_k}$$

$$(\ln N_k - \ln N_{k-1})(f_{k+1} - f_k) = (f_k - f_{k-1}) \ln r$$

and finally

$$f_{k+1} = \frac{(f_k - f_{k-1}) \ln r}{\ln N_k - \ln N_{k-1}} + f_k \tag{4}$$

Thus, given the cost bounds and the numbers of generated nodes of the previous two iterations, and the desired ratio of generated nodes between iterations, we can calculate an estimated cost bound for the next iteration. This equation does not precisely model the generated nodes in IDUCHS, but is an efficient way to calculate successive cost bounds.

The above equation cannot be used to calculate the second iteration's cost bound. In the first iteration, we calculate the average g-value ($avg(g)$) and depth ($avg(d)$) of all generated nodes, and increase the cost bound of the first iteration by $\frac{avg(g)}{avg(d)}$ to get the second cost bound.

IDUCHS works on all domains with non-negative edge costs, including unit-cost domains, and hence subsumes BFIDA*. IDUCHS is admissible and complete on both directed and undirected graphs when using admissible heuristics. Given a cost bound $U \geq C^*$, where C^* is the optimal solution cost, there always exists a node n on Open such that n is on an optimal path, and hence reached via its minimal g-value. Furthermore, on undirected graphs, a single iteration of IDUCHS is guaranteed to never expand a state more than once when using the d_g-value defined in Equation 3.

Solution Reconstruction

Similar to BFIDA*, when IDUCHS stops, it only has the solution cost, and additional searches are needed to reconstruct the actual solution path. Instead of calling BFHS recursively, we (2021) used two A* searches to compute the solution path from the start node to the middle node, and from the middle node to the goal node in BFIDA*. This is because heuristic functions such as the iPDB heuristic (Culberson and Schaeffer 1998; Haslum et al. 2007) are pre-computed before the search starts, and only return estimates of the cost to the original goal node. Thus, we do not have a heuristic to estimate the cost to a middle node. We also use two A* searches to reconstruct the solution path in IDUCHS.

To compute the solution path from the middle node to the original goal node, we use A* with the original heuristic function. Computing the solution path from the start node to the middle node is more complex, as we do not have a heuristic estimate to the middle node. For this task, in BFIDA*, we (2021) used A* with the heuristic to the original goal node, but we modified A* so that any node whose g-value exceeds that of the middle node, or whose f-value exceeds the original problem's C^*-value, is pruned. For IDUCHS, we similarly pruned nodes n for which $f(n) > C^*$ or $g(n) > g(middle)$. To accommodate zero-cost edges, a node n for which $g(n) = g(middle)$ is not pruned.

In BFIDA*, we (2021) saved the nodes at depth $U/4$ as the middle layer when U is the current iteration cost bound. This is because the layer at depth $U/2$ is usually larger than that of $U/4$. For IDUCHS, we save as the middle layer the Open nodes that exist at the point when all nodes up to $g = U/4$ have been expanded. Therefore the middle layer contains nodes with various g-values.

Experimental Results and Analysis

We implemented IDUCHS in the planner Fast Downward 20.06 (Helmert 2006). We used the original code for node expansions and heuristic functions. The original hash map in Fast Downward does not support removing a stored node, so we modified the hash map to allow this. Whenever a node was removed from the hash map, we stored the next new node into the resulting vacant slot. Therefore, we used a single hash map to store all nodes. The nodes in the middle layer used for solution reconstruction were not deleted.

We have solved about 300 problem instances from 19 International Planning Competition (IPC) domains with non-unit edge costs. These domains fall into three categories. First, binary domains, where the edge costs are either 0 or 1. Second, positive domains, where the edge costs are positive integers. Third, integer domains, where the edge costs are positive integer or 0. We are most interested in solving hard problems, so we present the results of the 20 hardest problem instances we solved from 11 domains. The remaining 280 problem instances were easily solved, usually within a few seconds. All experiments were run on a machine with a 3.33 GHz Xeon X5680 CPU and 236 GB of RAM.

We did not test IDUCHS on unit-cost domains, where IDUCHS will perform similarly to BFIDA*. A*+BFHS is the state-of-the-art algorithm for unit-cost domains when A*

Instance	C^*	Last Bound	Peak Stored Nodes & RAM in GB A*	Peak Stored Nodes & RAM in GB IDUCHS	Total Nodes A*	Total Nodes UCHS (C^*)	Total Nodes IDUCHS	Time (s) A*	Time (s) IDUCHS
elevators08 16*	87	88	53,943,052 (3.1)	3,379,185 (0.2)	151,476,407	177,867,354	410,304,955	131	337
transport11 09*	313	319	56,090,444 (3.1)	5,656,701 (0.4)	177,076,746	187,555,930	551,348,016	166	496
transport11 15*	1,079	1,086	81,297,837 (3.5)	8,577,209 (0.7)	237,858,963	244,074,950	587,047,357	238	593
data-network18 06	107	110	54,856,222 (3.2)	22,677,495 (1.4)	451,733,882	1,491,433,606	3,486,933,470	242	3,646
transport14 08*	1,173	1,338	62,784,307 (2.9)	13,200,288 (0.9)	396,833,140	397,501,043	1,325,938,181	248	1,375
sokoban08 28	43	46	42,280,857 (1.8)	19,696,951 (1.4)	104,786,508	287,762,405	938,253,805	275	2,566
spider18 18	27	27	32,956,806 (3.5)	14,725,684 (1.8)	35,247,917	197,343,317	245,024,146	313	1,859
ged14 8-9	6	5	138,961,033 (6.4)	51,194,955 (3.6)	155,044,171	333,194,875	176,316,713	332	231
transport11 08*	322	322	128,705,840 (6.0)	8,051,402 (0.4)	387,834,091	412,873,224	884,718,548	346	801
spider18 13	39	39	22,425,296 (3.0)	11,237,994 (1.7)	24,212,999	90,945,206	136,020,346	386	1,998
woodworking08 27	350	357	147,870,490 (7.4)	11,904,397 (0.7)	339,097,806	861,630,216	2,069,219,635	459	3,832
barman11 02-005	121	132	120,520,775 (6.1)	68,296,332 (5.5)	521,644,317	2,608,774,309	7,918,031,472	544	15,723
transport14 09*	966	996	127,472,989 (5.9)	37,110,556 (2.6)	741,922,587	747,591,228	2,308,571,006	549	2,327
agricola18 02	878	935	118,891,472 (6.7)	47,721,893 (2.9)	494,412,749	512,747,517	1,288,107,521	617	3,087
agricola18 03	788	793	152,528,391 (7.5)	112,549,403 (7.5)	706,797,822	823,748,774	2,287,807,416	790	5,617
agricola18 01	786	811	165,044,134 (7.9)	80,711,867 (5.5)	748,674,870	862,715,079	1,870,507,373	801	4,170
elevators08 29*	101	102	510,125,709 (2 4)	39,385,742 (2.6)	1,585,443,774	1,978,650,724	4,915,391,696	1,404	4,477
agricola18 05	979	980	262,635,603 (1 3)	79,632,741 (5.7)	1,184,932,854	1,398,771,125	3,238,751,246	1,533	8,788
elevators08 19*	112	115	765,075,247 (3 0)	114,952,004 (6.7)	2,159,252,861	2,571,563,855	8,409,534,183	2,140	7,957
floortile11 05-009	76	76	55,090,801 (2.9)	5,458,652 (0.3)	117,955,924	186,575,430	370,354,510	5,332	47,932

Table 1: Peak stored nodes, total generated nodes, and running time of A* and IDUCHS on domains with non-unit edge costs. Instances sorted by A* running time. Numbers in parentheses are memory usage in GB. An underline means more than 8 GB of memory was needed. Undirected graphs are marked with *.

fails due to memory limitations (Bu and Korf 2021). We did not test IDA* and A*+IDA*, which were shown to be orders-of-magnitude slower than BFIDA* and A*+BFHS due to many more duplicate nodes being generated (Zhou and Hansen 2004; Bu and Korf 2021).

sokoban08 and *spider18* are binary domains. *data-network18*, *elevators08*, and *ged14* are integer domains. The rest are positive domains. *elevators08*, *transport11*, and *transport14* are undirected graphs, while the other 8 domains are directed graphs. We tested three different heuristic functions and picked the best one for each domain: the iPDB heuristic with the default configuration in Fast Downward (Haslum et al. 2007; Sievers, Ortlieb, and Helmert 2012), the merge-and-shrink heuristic (M&S) with the recommended configuration in Fast Downward (Sievers, Wehrle, and Helmert 2014, 2016; Sievers 2018), and the landmark-cut heuristic (LM-cut, Helmert and Domshlak 2009). We used LM-cut for *floortile11*, M&S for *agricola18*, *barman11*, and *woodworking08*, and iPDB for the rest.

We present the results of A* and IDUCHS in Table 1. We used Equation 3 to calculate the d_g-values and $r = 2$ in Equation 4 to calculate the cost bounds of IDUCHS. The first column is the problem instance. The second column is the optimal solution cost C^* and the third column is the cost bound of IDUCHS's last iteration. The fourth and fifth columns are the peak numbers of stored nodes in A* and IDUCHS. The numbers in parentheses are the peak memory usage in GB reported by Fast Downward. The sixth column is the total nodes generated by A*. The seventh column is the number of nodes generated in a single iteration of UCHS with the cost bound equal to C^*, excluding the nodes generated in solution reconstruction. The eighth column is the total number of nodes generated by IDUCHS, including the nodes generated in solution reconstruction. The last two columns are the running time of A* and IDUCHS, including solution reconstruction in IDUCHS but excluding the time spent on building the heuristic function. Problem instances from the three undirected graph domains are marked with *. An underline means that more than 8 GB of memory was needed.

We found that using $U/4$ for the g-cost of the middle layer was not very practical, as the A* search from the middle node to the original goal node could be relatively expensive. Instead, we dynamically set the depth of the middle layer. In each iteration of IDUCHS, we checked if the size of the middle layer was less than 1% of the peak stored nodes for that iteration. If so, we then increased the g-value threshold for the middle layer by $U/10$ in the next iteration, with $U/2$ as the upper bound. The number of nodes generated during solution reconstruction was usually around 1% of the total nodes generated in IDUCHS, and never more than 10%.

A* vs. IDUCHS

IDUCHS required less memory than A* on 19 out of these 20 problem instances, sometimes significantly less, while A* was faster than IDUCHS. IDUCHS reduced the peak number of stored nodes by up to an order of magnitude over A* on the *elevators08*, *floortile11*, *transport11*, and *woodworking08* domains, by a factor of about 4 on *transport14*, and by a factor of around 2 on the other 6 domains. On the other hand, IDUCHS always generated more nodes than A*, and sometimes significantly more. As a result, IDUCHS was slower than A* on 19 out of these 20 problem instances.

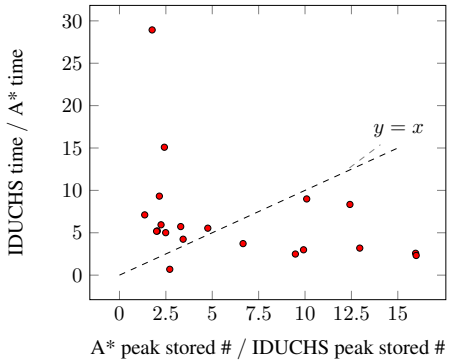

Figure 4: A* vs. IDUCHS in memory and time.

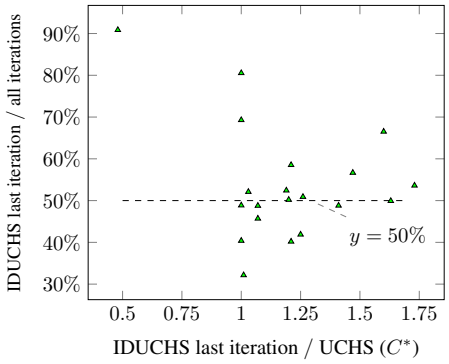

Figure 5: Comparisons of IDUCHS's last iteration.

We present the memory and time comparisons in Figure 4, where the x-axis is A*'s peak stored nodes over IDUCHS's, and the y-axis is IDUCHS's running time over A*'s. Each red circle represents one problem instance in Table 1. The points below the $y=x$ line represent problems for which the memory reduction ratio of IDUCHS over A* is higher than the increased running time ratio, while the points above the line represent the problems with the opposite property.

Comparing the memory usage in GB, the memory reduction ratio is usually less than the peak stored nodes reduction ratio for two reasons. First, the memory usage in GB includes memory used by the planner itself and the heuristic functions. On some problem instances, the heuristic function took up to 1 GB of memory. Second, our implementation was not optimized for memory, compared to the original A* implementation in Fast Downward. In our code, we used several C++ STL `vectors`, whose memory usage grows multiplicatively by a factor of 2. Optimizing memory usage of our code is a future work. Furthermore, a state's heuristic value is computed only once in A* no matter how many times the state is generated. In IDUCHS, the heuristic value of a state can be calculated more than once if that state is generated but then deleted. Therefore, the running time ratio of IDUCHS over A* is different from the total nodes ratio.

We compared IDUCHS using Equations 2 and 3 for computing the d_g-values. Using Equation 2 stored around 60% more nodes on *elevators08*, 30% more nodes on *floortile11* 05-009, 40% to 90% more nodes on *transport11*, and 50% more nodes on *woodworking08* 27. In contrast, using Equation 2 generated 60% fewer nodes than Equation 3 on *data-network18* 06 while storing 8% more nodes.

When the cost bound equals or exceeds the optimal solution cost C^*, the order in which nodes are generated can significantly impact the time of this final iteration, depending on how soon an optimal solution is found. This is called node ordering. BFIDA* in general has very poor node ordering, since its breadth-first search order usually generates almost all nodes in the final iteration. As a result, on unit-cost domains, the number of nodes generated by a single call to BFHS with a cost bound C^* is usually significantly larger than that of A*. However, the situation is different on domains with non-unit edge costs. In Table 1, we see UCHS (C^*) generated a similar number of nodes to A* on many problem instances, including all instances from the three undirected graph domains. In fact, the only domains where UCHS (C^*) generated significantly more nodes than A* due to node ordering are *ged14*, *spider18*, and *woodworking08*. This is because nodes generated on domains with non-unit edge costs usually have many more different f-values than that of unit-cost domains. For example, there are hundreds of different f-values on the two *transport* domains, which is very rare in unit-cost domains. As a result, on domains with non-unit edge costs, when the cost bound of IDUCHS is increased by one, usually only a few new nodes will be expanded. Thus, the almost-worst node ordering of IDUCHS is not a serious drawback on these problems.

On undirected graphs, frontier search can avoid leaking back into the interior of the search space, and re-expanding nodes that have already been expanded. This cannot be done in general with directed graphs, since there is nothing to prevent the generation of a node in the interior of the search. As a result, on the directed-graph domains *barman11*, *data-network18*, and *sokoban08*, UCHS (C^*) generated many more nodes than A*. To reduce the number of node re-expansions on directed graphs, we modified IDUCHS to increase the d_g-value of each node by $U/5$, where U was the cost bound. For example, if a node n originally had $d_g(n) = 50$ according to Equation 3 and $U = 100$, then we increased $d_g(n)$ to $50+100/5=70$. Compared to IDUCHS with Equation 3, IDUCHS with this new d_g formula stored around 15% more nodes on *barman11* 02-005 and *data-network18* 06, but reduced the total generated nodes by 78% and 61% respectively. On *sokoban08* 28, IDUCHS with this new d_g formula also pruned most duplicate nodes, but did not save memory compared to A* due to the last iteration being larger than A*.

Cost Bounds in IDUCHS

Table 1 shows the cost bound of the last iteration of IDUCHS. These bounds were calculated by Equation 4. The last cost bound was usually greater than C^*, with an exception on *ged14* 8-9, where the last cost bound was C^*-1. This is because IDUCHS generated a goal node during that iteration, thus avoiding an iteration with cost bound C^*.

To analyze our cost-bound equation, we present the last iteration analysis in Figure 5, where the x-axis is the number

of nodes generated in IDUCHS's last iteration divided by that of UCHS (C^*), the y-axis is the number of nodes generated in IDUCHS's last iteration divided by that of all iterations, and each green triangle corresponds to one problem instance in Table 1. With $r = 2$ for Equation 4, we expected to see all data points having $y \approx 50\%$ and $x \leq 2$.

From Figure 5, we see that the number of nodes generated in the last iteration of IDUCHS was within a factor of two of the nodes generated in UCHS (C^*) on all 20 problem instances. The number of nodes generated in the last iteration of IDUCHS was around 50% of the nodes generated in all IDUCHS iterations on most problem instances. This percentage is higher for some problem instances. For example, *ged14* 8-9 had $y = 91\%$ and *spider18* 13 and 18 had $y = 70\%$ and $y = 81\%$ respectively. This is because those instances have a small number of unique f-values, and the cost bound difference between iterations was just one, as in unit-cost domains. As a result, a generated nodes ratio of two between iterations was not possible, as one is the minimum cost bound increment. On the other hand, *transport14* 08 had a percentage of 32%. This is because compared to the penultimate iteration, not many new nodes were generated in the last iteration, despite the large cost bound. In short, Figure 5 and the above analysis show that Equation 4 was very accurate in calculating the cost bounds of IDUCHS on domains where each iteration is not much larger than its previous iteration, when using a cost bound increment of one.

A*+UCHS

Inspired by A*+BFHS (Bu and Korf 2021), we implemented A*+UCHS, a hybrid algorithm combining A* and UCHS. A*+UCHS first runs A* up to a user-specified memory threshold, then runs a series of UCHS iterations on the Open nodes of A*. A*+UCHS uses Equation 4 to calculate the cost bounds in the UCHS phase. Similar to (Bu and Korf 2021), we first generated the heuristics, then allocated $1/10$ of the remaining 8 GB of memory for the A* phase.

Figure 6 shows the comparison between IDUCHS and A*+UCHS. The x-axis is IDUCHS's peak stored nodes divided by A*+UCHS's, the y-axis is IDUCHS's time divided by A*+UCHS's, and each yellow square represents one problem instance in Table 1. In Figure 6, the data points to the right of the $x = 1$ line represent problem instances where A*+UCHS used less memory than IDUCHS, while the data points above the $y = 1$ line represent problem instances where A*+UCHS was faster than IDUCHS.

Little speedup was achieved when switching from IDUCHS to A*+UCHS on domains with non-unit edge costs. A*+UCHS was faster than IDUCHS by a factor of at least two on only three of the 20 problem instances in Table 1. The speedups of A*+BFHS over BFIDA* mainly come from A*+BFHS terminating early in the last iteration, as A*+BFHS can immediately stop when a goal node is generated. However, on domains with non-unit edge costs, cost bounds usually increase by more than one in A*+UCHS's UCHS phase. Thus, when A*+UCHS finds a goal node, it usually cannot stop but has to continue searching to verify that the solution is optimal. Therefore, early termination is usually not possible in A*+UCHS, with the exception of problem instances where the cost bounds between iterations increase by one. It is worth noting that on the easy non-unit cost problem instances which can be solved in A*+UCHS's A* phase, A*+UCHS is faster than IDUCHS but IDUCHS also quickly solves the easy problem instances. On unit-cost domains, however, A*+UCHS behaves similarly to A*+BFHS, and hence is faster than IDUCHS.

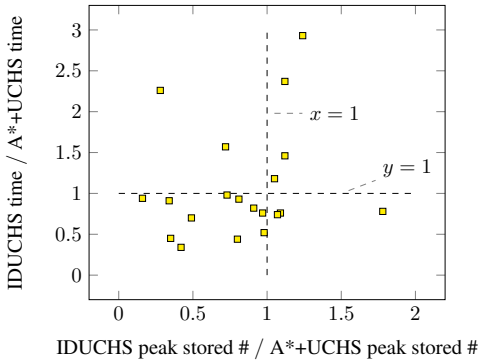

Figure 6: IDUCHS vs. A*+UCHS in memory and time.

Future Work

First, test IDUCHS on more problem instances. Second, refine Equation 4 to achieve more accurate results in calculating the cost bounds. For example, we may take the previous three or more iterations into consideration. The third is to explore adjusting the d_g-values automatically to reduce the duplicate nodes of IDUCHS on directed graphs. For example, at the beginning of IDUCHS, we can make a few calls to UCHS, each with the same cost bound but a different d_g-value adjustment to find out if the problem is a directed graph and whether the d_g-value adjustment works. Finally, we can optimize the code to reduce the memory usage.

Conclusions

We propose uniform-cost heuristic search (UCHS) and its iterative-deepening version IDUCHS, for optimally solving heuristic search and planning problems. Unlike breadth-first heuristic search, which only works on unit-cost domains, UCHS works on domains with arbitrary non-negative edge costs. UCHS is a uniform-cost frontier search that uses a heuristic evaluation function to prune nodes. UCHS is guaranteed not to expand a state more than once on undirected graphs. On directed graphs, UCHS can re-expand nodes, but we show how to adjust UCHS to avoid many such re-expansions. We also propose a new way to calculate the cost bounds of IDUCHS iterations. We have solved around 300 problem instances from 19 planning domains with non-unit edge costs, and presented the results of the hardest 20 problem instances. Experimental results show that IDUCHS is slower than A*, but requires less memory than A*, sometimes significantly, on a variety of planning domains. In practice, when faced with a new problem, A* can be run first. If A* cannot solve the problem within the available memory, then IDUCHS can be run.

References

Bu, Z.; and Korf, R. E. 2019. A*+IDA*: A Simple Hybrid Search Algorithm. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 1206–1212. ijcai.org.

Bu, Z.; and Korf, R. E. 2021. A*+BFHS: A Hybrid Heuristic Search Algorithm. *CoRR*, abs/2103.12701.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern Databases. *Comput. Intell.*, 14(3): 318–334.

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1: 269–271.

Franco, S.; Lelis, L. H.; Barley, M.; Edelkamp, S.; Martines, M.; and Moraru, I. 2018. The Complementary2 planner in the IPC 2018. *IPC-9 planner abstracts*, 28–31.

Franco, S.; Torralba, Á.; Lelis, L. H. S.; and Barley, M. 2017. On Creating Complementary Pattern Databases. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 4302–4309. ijcai.org.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Syst. Sci. Cybern.*, 4(2): 100–107.

Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-Independent Construction of Pattern Database Heuristics for Cost-Optimal Planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, July 22-26, 2007, Vancouver, British Columbia, Canada*, 1007–1012. AAAI Press.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009*, 162–169. AAAI.

Helmert, M.; Lattimore, T.; Lelis, L. H. S.; Orseau, L.; and Sturtevant, N. R. 2019. Iterative Budgeted Exponential Search. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, 1249–1257. ijcai.org.

Katz, M.; Sohrabi, S.; Samulowitz, H.; and Sievers, S. 2018. Delfi: Online planner selection for cost-optimal planning. *IPC-9 planner abstracts*, 57–64.

Korf, R. E. 1985. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artif. Intell.*, 27(1): 97–109.

Korf, R. E. 2004. Best-First Frontier Search with Delayed Duplicate Detection. In McGuinness, D. L.; and Ferguson, G., eds., *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, 650–657. AAAI Press / The MIT Press.

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening-A*. *Artif. Intell.*, 129(1-2): 199–218.

Korf, R. E.; and Zhang, W. 2000. Divide-and-Conquer Frontier Search Applied to Optimal Sequence Alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, 910–916. AAAI Press / The MIT Press.

Korf, R. E.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *J. ACM*, 52(5): 715–748.

Martinez, M.; Moraru, I.; Edelkamp, S.; and Franco, S. 2018. Planning-PDBs planner in the IPC 2018. *IPC-9 planner abstracts*, 63–66.

Reinefeld, A.; and Marsland, T. A. 1994. Enhanced Iterative-Deepening Search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(7): 701–710.

Sarkar, U. K.; Chakrabarti, P. P.; Ghose, S.; and Sarkar, S. C. D. 1991. Reducing Reexpansions in Iterative-Deepening Search by Controlling Cutoff Bounds. *Artif. Intell.*, 50(2): 207–221.

Sen, A. K.; and Bagchi, A. 1989. Fast Recursive Formulations for Best-First Search That Allow Controlled Use of Memory. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989*, 297–302. Morgan Kaufmann.

Sievers, S. 2018. Merge-and-Shrink Heuristics for Classical Planning: Efficient Implementation and Partial Abstractions. In *Proceedings of the Eleventh International Symposium on Combinatorial Search, SOCS 2018, Stockholm, Sweden, July 14-15, 2018*, 90–98. AAAI Press.

Sievers, S.; Ortlieb, M.; and Helmert, M. 2012. Efficient Implementation of Pattern Database Heuristics for Classical Planning. In *Proceedings of the Fifth Annual Symposium on Combinatorial Search, SOCS 2012, Niagara Falls, Ontario, Canada, July 19-21, 2012*, 105–111. AAAI Press.

Sievers, S.; Wehrle, M.; and Helmert, M. 2014. Generalized Label Reduction for Merge-and-Shrink Heuristics. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada*, 2358–2366. AAAI Press.

Sievers, S.; Wehrle, M.; and Helmert, M. 2016. An Analysis of Merge Strategies for Merge-and-Shrink Heuristics. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling, ICAPS 2016, London, UK, June 12-17, 2016*, 294–298. AAAI Press.

Zhou, R.; and Hansen, E. A. 2004. Breadth-First Heuristic Search. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, 92–100. AAAI.