A* with Lookahead Re-evaluated

Zhaoxing Bu
Department of Computing Science
University of Alberta
Edmonton, AB Canada T6G 2E8
zhaoxing@ualberta.ca

Roni Stern and **Ariel Felner**
Information Systems Engineering
Ben Gurion University
Beer Sheva, Israel 85104
roni.stern@gmail.com, felner@bgu.ac.il

Robert C. Holte
Department of Computing Science
University of Alberta
Edmonton, AB Canada T6G 2E8
rholte@ualberta.ca

Abstract

A with lookahead* (AL*) is a variant of A* that performs a cost-bounded DFS lookahead from a node when it is generated. We show that the original version of AL* (AL_0^*) can, in some circumstances, fail to return an optimal solution because of the move pruning it does. We present two new versions, AL_1^* and ELH, that we prove to always be correct and give conditions in which AL_0^* is guaranteed to be correct. In our experiments with unit costs, AL_0^* was usually the fastest AL* version, but its advantage was usually small. In our experiments with non-unit costs, AL_0^* substantially outperforms both A* and IDA*. We also evaluate the idea of immediately expanding a generated node if it has the same f-value as its parent. We find that doing so causes AL* to require more memory and sometimes slows AL* down.

Introduction

The main drawback of A* is that its memory requirements usually grow exponentially with depth. Consequently, for large state spaces A* usually exhausts the available memory before reaching a goal. By contrast, the memory needed by depth-first search (DFS) is only linear in the search depth. However, DFS algorithms can be very inefficient in state spaces containing many transpositions (redundant paths) since they do not perform duplicate pruning.

Stern et al. (2010) introduced a hybrid of A* and DFS called *A* with lookahead* (AL*). AL* is exactly the same as A* except that it performs a cost-bounded DFS lookahead from a node when it is generated. The main advantage of AL* over A* is that lookahead always reduces A*'s memory requirements, with deeper lookaheads producing larger reductions. Stern et al. showed that AL* can also reduce the time needed to solve a problem, but this is not guaranteed.

The main contributions of this paper are as follows.

Move pruning. In order to eliminate many of the transpositions in the state spaces they studied, Stern et al. provided AL* with a small table saying which operators may be applied immediately after a given operator. For example, in Figure 1 operators a and b commute, so there are two paths from S to U, ab and ba. Such a table might indicate that b may be applied immediately after a but that

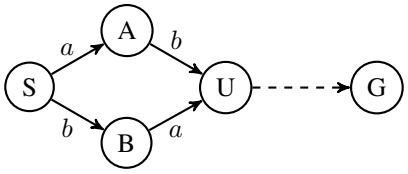

Figure 1: Sequences ab and ba are redundant.

a may not be applied immediately after b, thereby eliminating one of the redundant sequences. This table is a special case of "move pruning" (Burch and Holte 2011; 2012; Holte and Burch 2014) for operator sequences of length 2.

In this paper we examine three versions of AL*, AL_0^*, AL_1^* and ELH.[1] We show that, because of its move pruning, Stern et al.'s version (AL_0^*) can sometimes fail to return optimal solutions, but we also prove that it behaves correctly except in rather unusual circumstances. We also present two new AL* variations, AL_1^* and ELH, which we prove always return optimal solutions. We experimentally compare these three algorithms and find that their memory requirements are virtually identical and that their solving times are almost always within a factor of two of one another.

Immediate Expansion. In A*, "immediate expansion" (IE) means that if a generated node has the same f-value as its parent it is "immediately expanded" without going into OPEN (Sun et al. 2009). Stern et al. used IE in all their AL* experiments.[2] In this paper we give the first evaluation of AL* without IE. We show that IE requires more memory, produces limited speedup, and sometimes slows AL* down.

A* with Lookahead (AL*)

Figure 2 gives pseudocode capturing the essence of Stern et al.'s (2010) implementation of AL*. We refer to this pseudocode and any implementation of it as AL_0^*. AL_0^*'s expansion cycle is shown in the upper part of the figure, its lookahead algorithm in the lower part. This figure does not show the code for immediate expansion nor the many efficiency-improving details in Stern et al.'s (2010) implementation or our re-implementation of AL_0^*.

[1] We use AL* to refer to the general family of *A* with Lookahead* algorithms and names in italics, e.g. AL_0^*, to refer to specific instances of the AL* family.

[2] They called it "trivial lookahead".

```
   input: n, node chosen for expansion
   input/output: UB, search upper bound
 1 foreach operator op do
 2     c ← generateNode(op, n)
 3     if goalTest(c) = True then
 4         UB = min(UB, g(c))
 5     if g(c) ≥ g_stored(c) or f_s(c) ≥ UB then
 6         Continue
 7     LHB ← min(UB, f_s(n) + k)
 8     if f_s(c) ≤ LHB then
 9         fmin ← ∞
10         Lookahead(c, op, LHB, UB, fmin)
11         h_u(c) ← fmin − g(c)
12     else
13         h_u(c) ← h_s(c)
14     Save h_u(c), update g_stored(c), and update
       OPEN and CLOSED as appropriate.
15 end
```

Algorithm 1: Expansion cycle of AL_0^*.

```
   input: v, the root node of the lookahead
   input: H, move sequence history
   input: LHB, lookahead bound
   input/output: fmin, the minimum cost of
                 lookahead frontier nodes
   input/output: UB, search upper bound
 1 foreach operator op do
 2     if operatorAllowed(op, H) = False then
 3         Continue
 4     c ← generateNode(op, v)
 5     if goalTest(c) = True then
 6         UB = min(UB, g(c))
 7         fmin = min(fmin, g(c))
 8     else
 9         if f_s(c) ≤ LHB and f_s(c) < UB then
10             Lookahead(c,H≪op,LHB,UB,fmin)
11         else
12             fmin = min(fmin, f_s(c))
13 end
```

Algorithm 2: AL_0^*'s Lookahead.

Figure 2: Algorithms for AL_0^*. $H \ll op$ appends op to H.

The main variables in AL* are as follows. UB, the cost of the best solution found so far, is a global variable that is initially infinite. LHB is the cost used to bound f-values in the lookahead search, defined by a user-provided constant k. $g(c)$ is the cost of the current path to node c, and $g_{stored}(c)$ is the cost of the cheapest path to c previously found. Two heuristic values are computed for each node c. $h_s(c)$ is the normal heuristic value. The subscript s stands for "static", since $h_s(c)$ does not change over time. The other heuristic value, $h_u(c)$, is initially equal to $h_s(c)$, but is dynamically updated by Algorithm 1 as the search proceeds. $f_s(c)$ is $g(c) + h_s(c)$, and $f_u(c)$ is $g(c) + h_u(c)$.

In each iteration of the main A* loop (not shown), the node n with the smallest f_u-value is moved from OPEN to CLOSED and expanded by calling Algorithm 1. In Algorithm 1, the children of n are generated (lines 1 and 2), and UB is updated if a child is a goal (lines 3 and 4). If the current path to child c is not strictly better than the best path so far or c cannot possibly lead to a better solution, c is ignored (lines 5 and 6). Otherwise LHB is set to the smaller of UB and $f_s(n)+k$ (line 7) and lookahead is performed from c (line 10) unless $f_s(c)$ exceeds LHB (line 8). The lookahead returns $fmin$, the minimum f_s-value among the frontier nodes of the lookahead search, which is used to compute $h_u(c)$ (line 11). Lookahead will also update UB if it finds a cheaper path to the goal (lines 5 and 6 in Algorithm 2). Line 14 of Algorithm 1 adds c to OPEN (or updates it) and saves the new values of $g(c)$ and $h_u(c)$ for subsequent use.

Note that the goal is never added to OPEN and, if an optimal path to the goal is found during a lookahead, every node p on an optimal path with $f_s(p) = UB$ generated thereafter will also not be added to OPEN (line 5, Algorithm 1). This version of A* terminates with failure if OPEN becomes empty and UB is still infinite, and terminates with success if either (1) OPEN becomes empty and UB is finite, or (2) if the smallest f-value on OPEN equals or exceeds UB.

Algorithm 2 is a straightforward cost-bounded depth-first search that updates UB when it can and returns, in $fmin$, the minimum f_s-value at the frontier of its search.

Move Pruning in AL*

In AL*, move pruning is done in lines 2 and 3 of Algorithm 2. H is the sequence of recent moves leading to v. In a preprocessing phase, move pruning analysis (Holte and Burch 2014) checks each operator sequence of length L or less to determine if it can be pruned because it is redundant (see Definition 1 below) with another sequence of length L or less. This creates a table much like the one Stern et al. constructed by hand. For each operator sequence of length $L-1$ or less, this table says which operators may be applied immediately following that sequence. In Algorithm 2, this table is consulted (line 2) to determine if op may be applied after H (considering only the last $L-1$ moves in H).

A key feature of AL_0^* is that the move sequence history H used for move pruning during lookahead is initialized to be the operator op that generated the root node of the lookahead search (op in Algorithm 1 is passed as the second parameter when Algorithm 2 is called in line 10). This, however, can cause the updated heuristic value for child c, $h_u(c)$, to be inadmissible. For example, consider Figure 1. When operator b is applied to S, child B is generated, and a lookahead from B is initiated with H initialized to b, the operator that generated B. Move pruning does not permit a to be applied after b so this lookahead search will immediately terminate and $h_u(B)$ will be set to infinity, even though there is a path from B to the goal G. In this example, the inadmissible values do not cause AL_0^* to return a suboptimal solution; Figure 3 gives an example where they do.

In Figure 3 S is the start and G is the goal. Operators a and b generate the same state (T) when applied to S and have the same cost, but are not redundant so neither is pruned. Sequences ac and bc are redundant with one another and the move pruning system, based on a predetermined ordering of

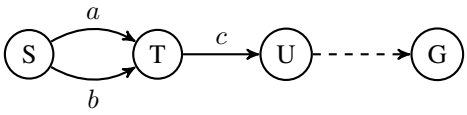

Figure 3: Situation in which AL_0^* fails to return an optimal solution (adapted from Holte (2013)).

move sequences, chooses to keep ac and prune bc (c will not be executed after b). Suppose that AL_0^* applies b first when S is expanded. Lookahead proceeds from T with H initialized to b. Because bc has been pruned, c is not applied to T given that $H = b$ so lookahead terminates with $fmin = \infty$. This makes $h_u(T) = \infty$, so T is not added to OPEN. Next a is applied to S, generating T, but because $g(T)$ via this path is not strictly better than the best previously seen $g(T)$ value ($g_{stored}(T)$, see line 5 in Algorithm 1), T reached via a will be ignored, OPEN will be empty, and AL_0^* will terminate without finding a path from S to G.

The key to this example is that the order in which the operators were applied by AL_0^* (b was given priority over a) was different than the order used by move pruning to decide whether to prune ac or bc (ac was given priority over bc). Even if the same order is used by both, AL_0^* could still fail to return an optimal solution if the order is not a "length-lexicographic order" (see Definition 3 below). If a length-lexicographic order is used for move pruning and the situation in Figure 3 does not occur then AL_0^* is guaranteed to return an optimal solution. In the next section we express these conditions formally in Definition 12 and prove AL_0^*'s correctness if the conditions in this definition are satisfied. These conditions are satisfied in the experiments done by Stern et al. (2010) and in all the experiments below.

Versions of AL* that are always correct

What always goes wrong when AL_0^* fails is that move pruning eliminates some paths in the first lookahead below n assuming that these paths will be examined by lookahead when n is reached by a different path of the same cost. However, the second lookahead is never done because AL_0^* only does a second lookahead for a node if it is reached by a strictly better path (line 5 in Algorithm 1). One way to correct this is to allow a lookahead to be done for a node every time it is reached by a path with the best known g-value. This means changing line 5 in Algorithm 1 to

if $g(c) > g_{stored}(c)$ **or** $f_s(c) \geq UB$ **then**

This change means that multiple lookaheads can now be done from the same node c, and some of these might have different initializations of H. Hence, the move pruning during these lookaheads might be different, which could result in several different h_u-values being computed for c for a given g-value. It is important that the smallest of these be kept as the value of $h_u(c)$ because all the others might be inadmissible. Therefore, line 11 in Algorithm 1 must be changed to

$h_u(c) \leftarrow \min(h_u(c), fmin - g(c))$

where the $h_u(c)$ inside the "min" refers to the smallest previously calculated $h_u(c)$ value for this $g(c)$. AL_1^* is the algorithm produced by making these two changes to AL_0^*. These changes will usually cause AL_1^* to be slower than AL_0^*.

On the example in Figure 3, AL_1^* would perform a lookahead when T is generated via operator a, even though it had previously generated T by a path (b) of the same cost. This lookahead (with H initialized to a) would apply operator c to T and return an admissible h_u-value, and T would be added to OPEN with $f_u(T)$ computed using this h_u-value. A proof of AL_1^*'s correctness is given in the next section.

Another way to change AL_0^* so it is safe to use under any conditions is to change the call to Algorithm 2 so that the empty sequence (ε) is passed as the initial move sequence history for move pruning instead of op, i.e. to change line 10 of Algorithm 1 to

Lookahead($c, \varepsilon, LHB, UB, fmin$)

We call this method ELH (for "Empty Lookahead History"). On the example in Figure 3, when T is generated for the first time, using operator b, ELH would perform a lookahead with H initialized to ε, not b. Therefore, the lookahead would apply operator c to T and return an admissible h_u-value, and T would be added to OPEN with $f_u(T)$ computed using this h_u-value.

ELH has the advantage over AL_0^* and AL_1^* that $h_u(c)$ is always admissible.[3] This means that h_u (or f_u) can be used in ELH in places where AL_0^* and AL_1^* must use h_s (or f_s): lines 5 and 8 in Algorithm 1 and everywhere in Algorithm 2.[4] This will make ELH faster than AL_0^* and AL_1^* because its larger heuristic values will mean there are more nodes for which lookahead is not done. On the other hand, ELH will do less move pruning during lookahead because its move sequence history is initially empty, making it slower than AL_0^* and AL_1^*. It is not obvious which of these effects will dominate in a given problem.

Proofs of Correctness for AL_0^* and AL_1^*

Our correctness proofs for AL_0^* and AL_1^* have the same general structure as the standard proof for A* (e.g. pp. 75–78 in (Pearl 1984)). For brevity we omit the proofs that AL_0^* and AL_1^* always terminate and return a solution path if one exists, and focus on the optimality of the path they return, which is Pearl's Theorem 2 (p.78). Pearl's proof rests on the fact that at any time during A*'s execution, there exists on OPEN at least one node n with these properties: (a) n is on a least-cost path from start to goal; (b) the current value of $g(n)$ is optimal; and (c) the value of $h(n)$ is admissible.

To prove that AL_0^* and AL_1^* are guaranteed to return an optimal solution (if one exists) it would therefore suffice to prove that at any time during their execution there exists on OPEN at least one node n with the required properties, with $h(n)$ in requirement (c) being understood as $h_u(n)$. However, this might not be true because if an optimal solution is found on iteration T, UB will be set to f^*, the optimal solution cost, and there may be no nodes on OPEN with the three required properties on subsequent iterations because the goal itself is never added to OPEN and new nodes with

[3] The proof is omitted due to lack of space, but is very similar to the proof of Fact 5 in the proof of AL_1^*'s correctness.

[4] In Algorithm 2, AL_0^* can use f_u when $g(c) \geq g_{stored}$, but must use f_s when $g(c) < g_{stored}(c)$.

$f_s(n) \geq UB$ are not added to OPEN (line 5 in Algorithm 1). However, the existence of such a node on OPEN after an optimal solution is found is not necessary because once an optimal solution is found it will never be replaced by a sub-optimal (or no) solution, so when AL_0^* and AL_1^* terminate they will return an optimal solution. It suffices therefore to prove that there exists a node on OPEN with the three required properties at the start of iteration t for $0 \leq t \leq T$.

Theorems 2 and 3 prove, for AL_1^* and AL_0^* respectively, that at the start of iteration t ($0 \leq t \leq T$) there is a node n on OPEN with the required properties. Both proofs use the same reasoning as the analogous proof for A*. Initially ($t = 0$), $start$ is on OPEN and has the required properties. If a node, n, with the required properties is on OPEN at the start of an iteration, there will be a node on OPEN with the required properties at the end of that iteration for one of two reasons: (1) n was not chosen for expansion, or (2) n was expanded and one of its children will have the required properties and be added to OPEN. The standard proof is straightforward because the heuristic value of a node is constant and there is no move pruning. Our proofs have to cope with a dynamically changing heuristic (h_u), the fact that some heuristic values may be inadmissible, and the fact that many paths, even optimal ones, may be eliminated by move pruning.

There are two main differences between the proofs for AL_0^* and AL_1^*. First, the proof for AL_0^* requires conditions (see Definition 12 below) that have no counterparts in the proof for AL_1^*. This difference is mainly seen in the proof of Fact 6 for the two algorithms. Second, the proof for AL_1^* shows that there is always a node from a specific least-cost path, $min(start, goal)$ (see Definition 4), that is on OPEN with the required properties. By contrast, AL_0^* might eliminate $min(start, goal)$ from consideration, so its proof must consider all least-cost paths. This difference is mainly seen in the definitions of Ω^t, Φ^t, and $L(s)$, which are otherwise very similar. Note that because Ω^t, Φ^t, and $L(s)$ have different definitions for AL_0^* and AL_1^*, the definitions based on these, e.g. of L_Ω^t, L_Φ^t, Ψ^t, etc., are also different even though they look identical.

Due to space limitations, proofs of lemmas and facts are omitted if they are straightforward.

Definition 1 *Operator sequence B is redundant with operator sequence A iff (i) $cost(A) \leq cost(B)$, and, for any state s that satisfies the preconditions of B, both of the following hold: (ii) s satisfies the preconditions of A, and (iii) applying A and B to s leads to the same state.*

ε is the empty sequence. If A is an operator sequence, $|A|$ is the number of operators in A. If A and B are operator sequences, $B \geq A$ indicates that B is redundant with A. If $\mathcal{O}$ is a total order on operator sequences, $B >_\mathcal{O} A$ indicates that B is greater than A according to $\mathcal{O}$, which means B comes after A on $\mathcal{O}$. If A and B are two operator sequences then their concatenation (AB) will sometimes be written as $A \bigcup B$. f^* is the cost of an optimal path from $start$, the start state, to $goal$.

Definition 2 *A total order on operator sequences $\mathcal{O}$ is "nested" iff $\varepsilon <_\mathcal{O} Z$ for all $Z \neq \varepsilon$, and $B >_\mathcal{O} A$ implies $XBY >_\mathcal{O} XAY$ for all A, B, X, and Y.*

Definition 3 *A "length-lexicographic order" $\mathcal{O}$ is a total order on operator sequences based on a total order of the operators $o_1 <_\mathcal{O} o_2 <_\mathcal{O}$ .... For operator sequences A and B, $B >_\mathcal{O} A$ iff $|B| > |A|$ or $|B| = |A|$ and $o_b >_\mathcal{O} o_a$ where o_b and o_a are the leftmost operators where B and A differ (o_b is in B and o_a is in the corresponding position in A).*

Definition 4 *For a nested order, $\mathcal{O}$, on operator sequences, and states s and t (t reachable from s), $min(s,t)$ is the least-cost path from s to t that is smallest according to $\mathcal{O}$.*

Theorem 1 *Let $\mathcal{O}$ be any nested order on operator sequences and B any operator sequence. If there exists an operator sequence A such that B is redundant with A and $B >_\mathcal{O} A$, then B does not occur as a consecutive subsequence in $min(s,t)$ for any states s, t.*

The preceding are from Holte and Burch (2014). Theorem 1 is the basis for Holte and Burch's move pruning technique. If $B = b_1 \dots b_k >_\mathcal{O} A$ and analysis determines that $B \geq A$, then move pruning refuses to apply b_k after $b_1 \dots b_{k-1}$. Theorem 1 implies that this will not prune any portion of $min(s,t)$ if a depth-first search is performed from s with a move sequence history that is initially empty.

Definition 5 *If $min(s,t)$ is the operator sequence $\sigma_1, ..., \sigma_k$ then we say node n is on $min(s,t)$ if $n = s$ or if $n = \sigma_i(\sigma_{i-1}(...(\sigma_1(s))...))$ for some i, $1 \leq i \leq k$.*

Lemma 1 *For any nested order $\mathcal{O}$, any states s and t (reachable from s), and any node n on $min(s,t)$, then $min(s,t) = min(s,n) \bigcup min(n,t)$.*

Correctness Proof for AL_1^*

Definition 6 *Ω^t is the set of states n at the start of iteration t that satisfy these three conditions:*

1. *n is on $min(start, goal)$.*
2. *the g-value for n is optimal.*
3. *n is on OPEN with admissible $h_u(n)$, or n is on CLOSED.*

AL_1^* sets $h_u(start) = h_s(start)$ so $start \in \Omega^t \; \forall t \geq 0$.

Definition 7 *Φ^t is the subset of states in Ω^t on OPEN.*

The nodes in Φ^t are precisely those with the three properties needed to prove that AL_1^* returns an optimal solution (if one exists). What we must prove, therefore, is that Φ^t is non-empty for all $0 \leq t \leq T$.

Definition 8 *If s is a state on $min(start, goal)$, $L(s)$ is the number of operators in $min(s, goal)$. $L(s) = \infty$ for states not on $min(start, goal)$.*

Definition 9 *L_Ω^t is the minimum $L(s)$ for $s \in \Omega^t$.*

Definition 10 *L_Φ^t is the minimum $L(s)$ for $s \in \Phi^t$. L_Φ^t is undefined if Φ^t is empty.*

Definition 11 $\Psi^t = \{s \in \Phi^t | L(s) = L_\Phi^t\}$.

Φ^t can contain several states but they will all have different L-values so Ψ^t contains exactly one state when Φ^t is non-empty.

Theorem 2 *Let $\mathcal{O}$ be a nested order on operator sequences, $start$ a state, $goal$ a state reachable from $start$, h_s an admissible heuristic, and T the index of the first iteration on which UB becomes equal to f^*. If AL_1^* is run with $start$, $goal$, h_s and move pruning based on $\mathcal{O}$, then at the start of iteration t, Φ^t is non-empty and $L_\Omega^t = L_\Phi^t$ $\forall t, 0 \leq t \leq T$.*

Proof The proof is by induction on t.
Base case ($t = 0$): $\Omega^0 = \Phi^0 = \Psi^0 = \{start\}$, $L_\Omega^0 = L_\Phi^0 = L(start)$.
Inductive step: We need to show the theorem is true for iteration $t+1$ given that it is true for iteration t, $0 \leq t < T$. Let P be the node expanded on iteration t.
Case 1. $P \notin \Psi^t$. *Proof sketch:* If no child of P meets all the criteria for being in Φ^{t+1} then $\Phi^{t+1} \supseteq \Psi^{t+1} = \Psi^t$ is not empty and $L_\Omega^{t+1} = L_\Phi^{t+1} = L_\Phi^t$. If a child B of P meets all the criteria for being in Φ^{t+1} then Φ^{t+1} will not be empty (B will be in it) and $L_\Omega^{t+1} = L_\Phi^{t+1} = min(L(B), L_\Phi^t)$. □
Case 2. $P \in \Psi^t$. Let B be the child of P generated by applying the first operator b on $min(P, goal)$ to P. Then the following are true:
Fact 1. B is on $min(start, goal)$.
Fact 2. $b = min(P, B)$.
Fact 3. $L(B) = L(P) - 1$.
Fact 4. $g(B)$, after applying operator b to P, is optimal.
Fact 5. If lookahead is performed when B is generated by applying b to P, it will return an admissible h_u-value for B.
Proof: Let UB_b be the value of UB when B is generated by applying b to P. The lookahead from B, with the move sequence history initialized to be b and $UB = UB_b$, generates exactly the same search tree as the tree below b generated by lookahead from P with a move sequence history initialized to be empty and $UB = UB_b$. By Theorem 1, no portion of $min(P, goal)$ will be pruned by move pruning during these lookaheads. In particular, no portion of $min(B, goal)$ will be pruned during lookahead from B with a move sequence history initialized to b. In addition, because $t < T$, $g(B)$ is optimal, and h_s is admissible, $f_s(n) < UB$ for every node n on $min(B, goal)$. Therefore lookahead will expand all nodes on $min(B, goal)$ except for those with $f_s > LHB$ (line 9 of Algorithm 2). Let F be the shallowest node on $min(B, goal)$ that is not expanded during lookahead. Then $h_u(B) = fmin - g^*(B) \leq f_s(F) - g^*(B) \leq f^*(F) - g^*(B) = h^*(B)$, i.e. $h_u(B)$ is admissible.
Fact 6. B will be on OPEN with an admissible h_u-value at the end of iteration t.
Proof: Because $L_\Omega^t = L_\Phi^t$, B is not, at the start of iteration t, on CLOSED with an optimal g-value. If lookahead from B is not performed because $f_s(B) > LHB$ then $h_u(B) = h_s(B)$ is admissible. If lookahead from B is performed then $h_u(B)$ is admissible (Fact 5). Because AL_1^* keeps the minimum $h_u(B)$ for the current g-value, B will be added to OPEN, or updated if it is already on OPEN with a larger g- or h-value, with this admissible h_u-value.

These six facts allow us to prove Case 2. By Facts 1, 4, and 6, B meets all the criteria for being in Φ^{t+1}, therefore Φ^{t+1} will not be empty. It is possible that other children of P also meet all the criteria, in which case $L_\Omega^{t+1} = L_\Phi^{t+1} =$ the minimum L-value of all such children. □

Correctness Proof for AL_0^*.

Definition 12 *AL_0^* is "safe" with move pruning using move sequence order $\mathcal{O}$ for state space S iff $\mathcal{O}$ is a length-lexicographic order and either of the following holds:*

1. *In Algorithm 1, AL_0^* applies operators in the order specified by $\mathcal{O}$.*
2. *There does not exist a state s and an operator o different than o_s, the first operator on $min(s, goal)$, such that:*
   (a) s is on a least-cost path from $start$ to $goal$,
   (b) $cost(o) = cost(o_s)$,
   (c) o applies to s and $o(s) = o_s(s)$.

Definition 13 *Ω^t is the set of states n at the start of iteration t that satisfy these three conditions:*

1. *n is on a least-cost path from $start$ to $goal$.*
2. *the g-value for n is optimal.*
3. *n is on OPEN or CLOSED.*

Definition 14 *Φ^t is the set of states n at the start of iteration t that satisfy these two conditions:*

1. *$n \in \Omega^t$.*
2. *n is on OPEN with an admissible h_u-value.*

AL_0^* sets $h_u(start) = h_s(start)$ so $start \in \Phi^0$.

The nodes in Φ^t are precisely those with the three properties needed to prove that AL_0^* returns an optimal solution (if one exists). What we must prove, therefore, is that Φ^t is non-empty for all $0 \leq t \leq T$.

Definition 15 *If s is a state, $L(s)$ is the minimum number of operators in a least-cost path from s to $goal$. $L(s) = \infty$ if $goal$ is not reachable from s.*

Note: if $\mathcal{O}$ is length-lexicographic, $L(s) = |min(s, goal)|$.

Lemma 2 *Let p be a state such that $p \in \Omega^t$, and a length-lexicographic order is used for move pruning. If during the expansion of p, p generates c by operator e, c is on a least-cost path from p to $goal$ and its g-value is optimal, then $L(c) \geq L(p) - 1$. If e is the first operator in $min(p, goal)$, then $L(c) = L(p) - 1$.*

Definition 16 *L_Ω^t is the minimum $L(s)$ for $s \in \Omega^t$.*

Definition 17 *L_Φ^t is the minimum $L(s)$ for $s \in \Phi^t$. L_Φ^t is undefined if Φ^t is empty.*

Definition 18 $\Psi^t = \{s \in \Phi^t | L(s) = L_\Phi^t\}$.

From the definitions we know $\Psi^t \subseteq \Phi^t \subseteq \Omega^t$, and Ψ^t is non-empty iff Φ^t is non-empty.

Theorem 3 *Let $\mathcal{O}$ be a length-lexicographic order on operator sequences, S a state space, $start$ a state, $goal$ a state reachable from $start$, h_s an admissible heuristic, and T the index of the first iteration on which UB becomes equal to f^*. If AL_0^* is run with $start$, $goal$, h_s and move pruning based on $\mathcal{O}$, and is safe to use with move pruning using $\mathcal{O}$ for S, then at the start of iteration t, Φ^t is non-empty and $L_\Omega^t = L_\Phi^t$ $\forall t, 0 \leq t \leq T$.*

Proof The proof is by induction on t.
Base case ($t = 0$): $\Omega^0 = \Phi^0 = \Psi^0 = \{start\}$, $L^0_\Omega = L^0_\Phi = L(start)$.
Inductive step: We need to show the theorem is true for iteration $t+1$ given that it is true for iteration t, $0 \leq t < T$. Let P be the node expanded on iteration t.
Case 1. $P \notin \Phi^t$. By Definition 14, there are three possible reasons for $P \notin \Phi^t$: (a) P is not on a least-cost path from $start$ to $goal$, (b) $g(P)$ is not optimal, and (c) $h_u(P)$ is not admissible. If (a) or (b) is true, then no child of P that is not in Φ^t can be on a least-cost path with its optimal g-value, otherwise P is on a least-cost path and $g(P)$ is optimal. Thus, if (a) or (b) is true, $\Phi^{t+1} \supseteq \Psi^{t+1} = \Psi^t$ is not empty and $L^{t+1}_\Omega = L^{t+1}_\Phi = L^t_\Phi$. Finally, it is impossible for (c) to be true if (b) is false, because in this situation P would not have been chosen for expansion. Φ^t is non-empty, and each node in Φ^t has an admissible h_u-value, hence an admissible f_u-value. For each $n \in \Phi^t$, $g^*(n) + h_u(n) \leq f^* < g^*(P) + h_u(P)$, so n would have been chosen for expansion, not P, if (c) were true and (b) false. □
Case 2. $P \in \Phi^t$, $P \notin \Psi^t$. If no child of P that is not in Φ^t meets all the criteria for being in Φ^{t+1} then $\Phi^{t+1} \supseteq \Psi^{t+1} = \Psi^t$ is not empty and $L^{t+1}_\Omega = L^{t+1}_\Phi = L^t_\Phi$. If a child B of P that is not in Φ^t meets all the criteria for being in Φ^{t+1} then Φ^{t+1} will not be empty (B will be in it) and $L^{t+1}_\Omega = L^{t+1}_\Phi = L^t_\Phi$ (by Lemma 2, L^{t+1}_Φ cannot be less than L^t_Φ since $L(P) > L^t_\Phi$). □
Case 3. $P \in \Psi^t$. Let B be the child of P generated by applying the first operator b in $min(P, goal)$ to P. Then the following are true:
Fact 1. B is on a least-cost path from $start$ to $goal$.
Fact 2. $b = min(P, B)$.
Fact 3. $L(B) = L(P) - 1$.
Fact 4. $g(B)$ after applying operator b to P is optimal.
Fact 5. If lookahead is performed when B is generated by applying b to P, it will return an admissible h_u-value for B.
Proof: This proof is same as the proof for Fact 5 of Theorem 2's Case 2, and is omitted here due to space limitation.
Fact 6. B will be on OPEN with an admissible h_u-value at the end of iteration t.
Proof: Because of Facts 1 and 3, Definition 13, and $L^t_\Omega = L^t_\Phi$, B is not, at the start of iteration t, on OPEN or CLOSED with an optimal g-value. If lookahead from B is performed when B is generated by applying b to P, then, by Fact 5, h_u will be admissible. In Algorithm 1, there are three reasons why lookahead from B, when it is generated by applying b to P, might not be performed. We will consider each in turn.
(a) $f_s(B) \geq UB$ (line 5) is impossible because $t < T$, $g(B)$ is optimal, and h_s is admissible.
(b) Lookahead would not be performed if $g(B)$, as calculated when B is generated by applying b to P, was equal to $g_{stored}(B)$ (line 5). By Fact 4 we know this $g(B)$ is optimal, and, as noted above, we know $g_{stored}(B)$ was not optimal at the start of this iteration. It is possible, however, that $g_{stored}(B)$ was updated to an optimal value during this iteration. That would happen if there was an operator different than b but having the same cost as b, that was applied to P before b and generated B. This cannot have happened because AL^*_0 is safe with move pruning using $\mathcal{O}$ for state space S (Definition 12).
(c) The only other reason lookahead would not be performed from B is that $f_s(B) > LHB$ (line 8). In this case $h_u(B) = h_s(B)$ is admissible (line 13).
Thus, B will be added to OPEN, or updated if it is already on OPEN with a larger g-value, with an admissible h_u-value.

These six facts allow us to prove Case 3. By Facts 1, 4, and 6, B meets all the criteria for being in Φ^{t+1}, therefore Φ^{t+1} will not be empty and $L^{t+1}_\Omega = L^{t+1}_\Phi = L(B)$ (by Lemma 2, L^{t+1}_Φ cannot be less than $L(B)$). □

Experimental Comparison – Unit Costs

As mentioned above, it is not possible to predict the exact effect on performance of the changes made to AL^*_0 to create AL^*_1 and ELH. We expect AL^*_1 to be slower than AL^*_0, but we do not know by how much. ELH could potentially be the fastest, or the slowest, of them all. The main purpose of the experiment in this section is to compare the performance of these methods on a wide range of problems. In addition, because our implementations of AL*, A*, and IDA*, are completely different than those of Stern et al. (2010), this experiment serves to test which of their results are implementation-independent and which are not.

We use 9 domains in which all operators cost 1. We use the same two domains as Stern et al., the 15-puzzle and (16,4)-TopSpin, with exactly the same heuristics (Manhattan Distance, and a pattern database (PDB, (Culberson and Schaeffer 1996)) defined by the same 8 tokens), and 6 new domains, all with PDB heuristics. The new domains were chosen to be small enough that A* could solve all the test instances without running out of memory. Finally, we used the 15-puzzle with the 7-8 additive PDB and an additional lookup defined by the reflection about the main diagonal (Korf and Felner 2002).

For test instances, we used 100 solvable start states for each domain, generated uniformly at random, except as follows: for the 15-puzzle with Manhattan Distance we used the 74 instances Stern et al.'s A* could solve with 2 Gigabytes of memory; for the 15-puzzle with the 7-8 PDB we used the complete set of 1000 standard test instances (Korf and Felner 2002); for the 3x3x3 Rubik's Cube our start states were all distance 9 from the goal ("easy"); for "Work or Golf" we generated the instances by random walks backwards from the goal; and for Gripper we used only 2 start states, one with all the balls in the wrong room, and one with all but one of the balls in the wrong room.

All experiments were run on a 2.3 GHz Intel Core i7 3615QM CPU with 8 GB of RAM. The results presented for each domain are averages over its test instances, and the times reported do not include the preprocessing time for the move pruning analysis and building the PDBs. The value of L used for move pruning is shown in parentheses after the domain name in Table 1. The results for IDA*, using the same move pruning as AL*'s lookahead search, and for A* are shown below the domain names in Table 1 with the faster

of the two in bold.

For all algorithms other than A* a time limit for solving all the instances in a domain was set to be five times the time required by A* to solve all the instances. If the time limit was exceeded, "n/a" is shown for "# Stored Nodes" and the time shown in Table 1 is the time limit, divided by the number of instances, followed by a + (e.g. ELH on the Arrow puzzle for $k \geq 3$). If the largest value of k for a domain is less than 6 it is because all the AL* variants exceeded the time limit for the larger values of k.

We focus first on the results when immediate expansion is used (columns labeled ELH, AL_0^*, and AL_1^*).

The primary objective of AL* is to reduce A*'s memory requirements without substantially increasing solving time, and this it achieves admirably. The number of stored nodes is almost identical for the three AL* variants and is only shown for AL_0^*. As Stern et al. observed, the number of stored nodes decreases substantially for every increase in k (except in "Work or Golf" and Towers of Hanoi, where the decrease is moderate). In most domains AL* achieves up to three orders of magnitude reduction in memory. In all domains except Gripper, memory reduction is achieved with only a modest increase in solving time in the worst case. $k = 0$ stores substantially fewer nodes than A* only in 2 domains (TopSpin, Arrow Puzzle).

AL_0^* is almost always the fastest (ELH is slightly faster than AL_0^* on the Pancake puzzle when $k = 6$, and on Rubik's Cube when $k \geq 4$; AL_1^* is slightly faster than AL_0^* on Rubik's Cube when $k = 2$) but the times required to solve an average instance by the different AL* algorithms are all within a factor of two of one another except in the Arrow Puzzle where AL_0^* is much faster than ELH and AL_1^* and in Rubik's Cube with $k = 4$ where ELH is slightly more than twice as fast as the others. ELH is sometimes faster than AL_1^* and sometimes slower. In other words, there is usually not much difference between the algorithms' solving times, and when there is, AL_0^* is usually fastest.

Solving time always increases as k increases, except: on TopSpin, k=4, 5 (all methods); on the Arrow Puzzle, k=1–3 for AL_0^*, k=1 for AL_1^*, and k=2 for ELH; on Rubik's Cube, k=3 (all methods); and on the 15-puzzle with Manhattan Distance, $k = 2, 4$ (all methods).

There are few cases where an AL* variation is faster than A* (these are marked in bold in the table). AL_0^* on the Arrow Puzzle for $k = 2, 3$ is the only case where an AL* variation is more than twice as fast as A*. When IDA* is faster than A* it is also faster than AL*, except on the 15-puzzle with Manhattan Distance ($k > 0$).

Evaluation of Immediate Expansion (IE)

With "immediate expansion" (IE), if a node is generated (in line 2 of Algorithm 1) with the same f-value as its parent it is immediately expanded (it is put on CLOSED, its children are generated, and the same process is applied to them), so genuine lookahead only takes place from nodes that have f-values different than their parents. Without IE, by contrast, a normal lookahead is done from all children, even if a child's f-value is the same as its parent's.

	Avg. # Stored Nodes		Avg. Time (secs)			
k	AL_0^*	No IE	ELH	AL_1^*	AL_0^*	No IE
(16,4)-TopSpin (L=5) A*: 8,580,684 stored nodes, 4.05s. IDA*: **1.72**s						
0	2,371,540	1,750,085	**3.13**	**3.13**	**3.13**	**3.97**
1	1,129,039	575,435	5.50	5.04	4.41	5.00
2	335,512	121,278	5.69	5.76	4.43	5.27
3	94,134	21,552	7.55	6.61	5.49	4.58
4	13,706	3,898	7.12	5.38	4.86	**3.56**
5	2,280	237	6.57	4.60	4.45	**3.32**
6	1,489	19	7.66	5.53	5.45	4.18
28-Arrow Puzzle (L=2) A*: 2,495,747 stored nodes, 2.68s. IDA*: **0.38**s						
0	1,096,924	875,124	**2.57**	**2.57**	**2.57**	4.54
1	622,168	550,731	3.16	**1.97**	**1.40**	5.40
2	377,820	312,222	2.88	13.4+	**1.18**	7.70
3	131,367	68,898	13.4+	13.4+	**1.10**	3.79
4	63,995	3,430	13.4+	13.4+	**2.29**	**0.80**
5	60,486	25	13.4+	13.4+	2.74	**0.43**
6	60,486	25	13.4+	13.4+	3.86	**0.68**
14-Pancake Puzzle (L=5) A*: 3,135,552 stored nodes, **1.44s**. IDA*: 2.56s						
0	3,134,333	1,342,999	1.46	1.46	1.46	1.61
1	600,127	249,838	2.45	2.52	2.22	2.90
2	99,534	28,391	2.62	3.03	2.58	3.83
3	19,006	3,504	3.67	3.49	3.22	4.76
4	3,282	282	4.75	4.15	4.00	5.58
5	1,139	14	4.92	4.30	4.28	4.13
6	1,097	14	6.90	7.03	7.02	6.85
16 ball Gripper (L=4) A*: 9,592,688 stored nodes, **8.34s**. IDA*: 41.7+s						
0	9,592,686	n/a	8.38	8.38	8.38	41.7+
3x3x3 Rubik's Cube (L=4) A*: 2,314,413 stored nodes, 0.98s. IDA*: **0.42**s						
0	2,236,741	1,174,385	1.02	1.03	1.00	**0.95**
1	670,255	302,035	1.98	1.81	1.69	1.54
2	99,075	36,116	2.57	2.03	2.12	1.61
3	7,797	2,703	1.76	1.61	1.53	1.40
4	1,475	n/a	2.13	4.52	4.47	4.9+
5	n/a	n/a	4.69	4.9+	4.9+	4.9+
Work or Golf (L=3) A*: 1,093,754 stored nodes, **0.93s**. IDA*: 4.65+s						
0	1,093,761	928,222	0.93	0.93	0.93	1.36
1	961,237	791,317	1.58	1.33	1.11	1.89
2	544,604	439,802	2.61	3.50	1.87	3.90
3	472,982	n/a	3.46	4.60	2.50	4.65+
4-Peg Towers of Hanoi 12 Disks (L=3) A*: 915,544 stored nodes, **1.03s**. IDA*: 5.15+s						
0	915,536	885,506	1.04	1.04	1.04	1.42
1	853,416	815,468	2.54	2.24	1.96	3.43
2	732,162	n/a	5.15+	5.15+	4.64	5.15+
15-Puzzle (L=2) Manhattan Distance A*: 4,780,390 stored nodes, 1.63s. IDA*: **1.58**s						
0	4,780,410	2,625,665	1.66	1.66	1.66	1.63
2	1,366,333	709,795	**1.47**	**1.52**	**1.40**	**1.46**
4	364,396	180,518	**1.37**	**1.41**	**1.34**	**1.42**
6	91,315	41,949	**1.40**	**1.45**	**1.39**	**1.54**
15-Puzzle (L=2) 7-8 additive PDB A*: 11,723 stored nodes, **0.012s**. IDA*: 0.025s						
0	11,702	4,153	0.012	0.012	0.012	0.014
2	2,269	739	0.016	0.016	0.016	0.019
4	407	120	0.019	0.019	0.019	0.022
6	79	18	0.023	0.023	0.022	0.025

Table 1: Experimental results with unit costs.

The "No IE" columns in Table 1 are AL_0^* without IE. In all domains, AL_0^* stores more nodes with IE than without it. This is because all the nodes expanded by IE are stored in CLOSED, but they might not be stored at all if IE is not done (because search terminates before they are generated in line 2 of Algorithm 1). In the Pancake puzzle and 15-puzzle with the 7-8 PDB, and in several domains for larger values of k, the difference in the number of stored nodes is substantial.

IE tends to be faster but there are exceptions: $k \geq 3$ for TopSpin, $k \geq 4$ for the Arrow Puzzle, $k \geq 5$ for the Pancake puzzle, and $k \leq 3$ for Rubik's Cube. However, in all domains except Gripper and the Arrow Puzzle, and "Work or Golf" when $k \geq 2$, the times with or without IE are within a factor of two of one another. When k is chosen optimally, IE produces the fastest time in all domains except the Arrow Puzzle, where $k = 5$ without IE is fastest, and Rubik's Cube, where $k = 0$ without IE is fastest.

IE always requires more memory and is slightly faster in most domains but not all. The No IE variant tends to give a better memory/time tradeoff; orders of magnitude in memory at the cost of a slight increase in time in some cases.

Stern et al. reported a 48% reduction in A*'s time on the 15-puzzle (with Manhattan Distance) when IE ($k = 0$) was added, and a 33% reduction on TopSpin. On these domains we get -2% and 23% reductions, respectively. Only in one of the new domains, the Arrow Puzzle, does IE ($k = 0$) reduce A*'s time, and then only slightly (4%). On the other domains A*'s time remains the same or increases with IE ($k = 0$). We believe this difference is the result of our using a different OPEN list implementation (Sun et al. 2009).

Experimental Comparison – Non-unit Costs

The purpose of this experiment is to illustrate situations in which AL* is superior to both A* and IDA*. We use Rubik's Cube, (16,4)-TopSpin, and larger versions of the Arrow and Pancake puzzles. Each operator is assigned a cost between 1 and 100 uniformly at random. We used AL_0^* with IE and $k = 20$. Note that 20 is a relatively small value of k considering the average cost of a single operator is 50.5. For A*, and for AL_0^*, the combined memory for OPEN and CLOSED was limited to 1 GB. For test instances, we used 100 solvable start states for each domain, generated uniformly at random, except for the 3x3x3 Rubik's Cube, where our start states were all 5 moves from the goal, independent of cost. We repeated the experiment 5 times with different cost assignments on each run. IDA* was given a per-instance time limit for each run on each domain equal to five times the maximum time needed by AL_0^* to solve an instance on that run in that domain. Table 2 reports the totals over the 5 runs.

In Table 2, for each domain there is a row for those instances that AL_0^* solved and a row for those instances that AL_0^* failed to solve because it exceeded the memory limit. For A* and IDA*, there is a column for the instances each solved and a column for the instances each failed to solve. Cells in a "failed" row or column contain the number of instances that fall into that cell (e.g. how many instances AL_0^* solved but A* failed to solve). Instances that fall into a cell in a row and column that are both "solved" are divided into

	A*		IDA*	
AL_0^*	solved	failed	solved	failed
30-Arrow Puzzle (L=3)				
solved	**36**W, 2L, 249D	**172**	**401**W, 4L, 27D	**27**
failed	0	41	7	34
15-Pancake Puzzle (L=5)				
solved	**0**W, 46L, 252D	**183**	**427**W, 0L, 6D	**48**
failed	0	19	0	19
3x3x3 Rubik's Cube (L=3)				
solved	**15**W, 77L, 273D	**36**	**134**W, 104L, 163D	**0**
failed	0	99	19	80
(16,4)-TopSpin (L=5)				
solved	**1**W, 3L, 326D	**159**	**459**W, 0L, 3D	**12**
failed	0	11	0	11

Table 2: Experimental results with non-unit costs.

"wins for AL_0^*" (W), "losses" (L), and "draws" (D). An instance solved by both AL_0^* and A* (IDA*) is a win if A*'s (IDA*'s) solving time is more than double AL_0^*'s, a loss if AL_0^*'s solving time is more than double A*'s (IDA*'s), and a draw otherwise. For example, "15W, 77L, 273D" in the "solved" row for Rubik's Cube and the "solved" column for A* means that of the instances solved by both AL_0^* and A* AL_0^* was at least twice as fast A* on 15, at least twice as slow on 77, and within a factor of 2 on the other 273.

In each domain A* fails on 27–42% of the instances and AL_0^* solves the vast majority of these, except for Rubik's Cube, where it "only" solves 27% of the instances A* fails on. By contrast, there are no problems in any domain that are solved by A* but not by AL_0^*. On most instances solved by both systems, the times for A* and AL_0^* are within a factor of 2 of each other. There are relatively few wins or losses. They favor A* on all domains except the Arrow Puzzle.

IDA* has relatively few failures (231 out of 2000), indicating that the time limits we set for it were reasonable. AL_0^* solves 38% of the instances IDA* fails on. By contrast, AL_0^* fails on fewer instances (170), and IDA* solves only 15% of those instances. The vast majority of instances are solved by both methods, and AL_0^* has more wins than losses in every domain, usually many more. We believe the poor performance of IDA* is because the wide variety of operator costs force it to do a large number of iterations before its cost bound reaches the optimal solution cost.

Conclusions

We have shown that the original version of AL* (AL_0^*) will sometimes return a suboptimal solution, described conditions in which it is guaranteed to return optimal solutions, and presented two new versions, AL_1^* and ELH, that we proved always return optimal solutions. All the AL* methods reduce the memory needed by A* substantially, with increasing savings as the lookahead bound is increased. In our unit-cost experiments, AL_0^* was usually the fastest, but its advantage was usually small. In our non-unit cost experiment we showed that AL_0^* substantially outperforms both A* and IDA*. We also showed experimentally that immediate expansion always causes AL* to require more memory, sometimes much more, and that its speedup benefits are limited; it even sometimes slows AL* down.

References

Burch, N., and Holte, R. C. 2011. Automatic move pruning in general single-player games. In *Proceedings of the 4th Symposium on Combinatorial Search (SoCS)*.

Burch, N., and Holte, R. C. 2012. Automatic move pruning revisted. In *Proceedings of the 5th Symposium on Combinatorial Search (SoCS)*.

Culberson, J., and Schaeffer, J. 1996. Searching with pattern databases. In *Proceedings of the 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, volume 1081 of *Lecture Notes in Computer Science*, 402–416. Springer.

Holte, R. C., and Burch, N. 2014. Automatic move pruning for single-agent search. *AI Communications*. (to appear).

Holte, R. C. 2013. Move pruning and duplicate detection. In *Proceedings of the 26th Canadian Conference on Artificial Intelligence*, 40–51.

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

Pearl, J. 1984. *Heuristics – Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Stern, R.; Kulberis, T.; Felner, A.; and Holte, R. 2010. Using lookaheads with optimal best-first search. In *AAAI*, 185–190.

Sun, X.; Yeoh, W.; Chen, P.-A.; and Koenig, S. 2009. Simple optimization techniques for A*-based search. In *AAMAS (2)*, 931–936.