Eager vs. Lazy Duplicate Detection in A*

Yuki Suzuki, Alex Fukunaga

The University of Tokyo
suzuki-yuki0631@g.ecc.u-tokyo.ac.jp, fukunaga@idea.c.u-tokyo.ac.jp

Abstract

A^* must perform duplicate detection on generated nodes. Eager duplicate detection checks for duplicates immediately after node generation, while lazy duplicate detection defers the duplicate check until a node is selected for expansion. We compare the state expansion orders induced by these two policies under standard priority queue and tie-breaking strategies. We show that, under FIFO tie-breaking, eager duplicate detection without decrease-key expands the same sequence of states as lazy duplicate detection, and the same holds with decrease-key if updates refresh timestamps. In contrast, under LIFO tie-breaking, the number of expanded nodes can differ by an arbitrarily large amount in either direction. Experiments in domain-independent planning benchmarks confirm that these differences occur in practice.

1 Introduction

A^* is a standard heuristic search algorithm for computing shortest paths in graphs [8], with many applications such as path finding, planning, and scheduling. In domain-independent planning, A^*-based algorithms are among the state-of-the-art for finding cost-optimal plans. At a high level, A^* is a straightforward algorithm which repeatedly selects a node from a priority queue and generates its successors until a goal node is selected for expansion. However, subtleties in the implementation can significantly affect its search behavior (the order in which states are explored) and runtime performance. For example, the tie-breaking policy for selecting among nodes with the same evaluation value can significantly affect the search behavior [2].

Another implementation choice which can significantly affect the performance of A^* is the duplicate detection policy. A^* repeatedly selects and expands a node n from a priority queue $Open$ with the best evaluation value f. When A^* generates a node n, it must determine whether n is a duplicate node. The duplicate detection can be done eagerly or lazily. *Eager duplicate detection* (EAGER), checks whether n is a duplicate as soon as it is generated, and if so, n is discarded. *Lazy duplicate detection* (LAZY), inserts n into $Open$ without checking whether n is a duplicate, but later, if n is popped from $Open$, it checks whether n is a duplicate, and if so, n is discarded.

Previous work compared EAGER vs LAZY primarily in terms of runtime overhead, duplicate-detection effort, and resulting differences in expansion rate [5]. EAGER incurs a duplicate detection check for each state generated. In the case of LAZY, when a solution is found and search terminates, there may be nodes in $Open$ for which the checks have been deferred, so LAZY tends to incur fewer duplicate detection checks. In some cases, many generated nodes remain in $Open$ at termination, so LAZY avoids duplicate checks for a large fraction of generated nodes, resulting in faster runtime on problems where duplicate detection accounts for a significant fraction of the total runtime.

It is important to understand whether the choice of EAGER vs. LAZY is merely an implementation-level decision whose impact is limited to runtimes and memory usage, or whether the choice can also impact search efficiency (number of states expanded). More specifically, do EAGER and LAZY expand nodes in the same order, and if not, how different are their search behaviors? To our knowledge, possible differences in node expansion order of EAGER and LAZY have not been investigated in previous work.

This paper compares the search behavior of eager and lazy duplicate detection both theoretically and experimentally. We first define duplicate detection mechanisms in Section 2. We then theoretically compare the search behaviors of eager vs lazy duplicate detection in Section 3. The results depend on the tie-breaking policy for prioritizing nodes with the same f and h values, as well as how improved paths to previously explored states are handled. We show that with a FIFO tie-breaking policy, where improved $Open$ entries are assigned a fresh (or refreshed) timestamp, eager and lazy duplicate detection expands the exact same sequence of states. On the other hand, we show that with a LIFO tie-breaking policy, the number of nodes expanded using these duplicate detection policies can differ by an arbitrarily large value. Section 4 compares eager vs lazy duplicate detection experimentally. In domains where some operators have a cost of 0, significant differences in the number of nodes expanded by eager and lazy duplicate detection are observed, with lazy duplicate detection tending to expand more nodes than eager duplicate detection.

2 Duplicate Detection in A^*

Given a start state s_0 and a goal test, A^* seeks a minimum-cost path from s_0 to a goal state in a state-space graph. A search node n represents a particular path to state $s(n)$,

so multiple nodes may represent the same state. There is a priority queue called $Open$, where nodes are prioritized according to $f(n) = g(n) + h(s(n))$, where $g(n)$ is the cost of the path represented by n, and $h(s(n))$ estimates the cost from $s(n)$ to a closest goal. A^* repeatedly selects a node from $Open$, and expands it, generating its successors, successors(n) by applying all applicable actions. For each successor node $n' \in$ successors(n), the algorithm computes its path cost $g(n') = g(n) + c(n, n')$, where $c(n, n')$ is the cost of the transition. If this new path to n' is better than any previously found (or if n' has not yet been discovered), the node is added to $Open$ (or updated in $Open$) with priority $f(n') = g(n') + h(s(n'))$.

To avoid redundant work, A^* maintains a set of states (called $Closed$) that have already been expanded.

This process continues until either a goal node is selected for expansion, at which point the algorithm returns the corresponding path as an optimal solution, or $Open$ is empty, in which case no solution exists.

2.1 Duplicate Detection

Duplicate detection determines how the algorithm handles generated nodes whose states have already been seen.

In EAGER (Algorithm 1, A^* with *eager duplicate detection*), duplicates are checked when successors are generated. For generated states, the algorithm maintains a partial map $g_{\text{best}} : S \rightharpoonup \mathbb{R}_{\geq 0}$, where $g_{best}(s)$ is the best path cost to s among generated nodes. If a successor state has not been generated before, then the successor is accepted and inserted into or updated in $Open$. If the new path is not better (a non-improving duplicate), the successor is discarded immediately.

There are two approaches for handling new, improved paths. In EAGER with a *decrease-key* operation, improving a node already in $Open$ can be handled by updating its g-value and parent pointer and decreasing its priority to the new value $f(n)$. In this case, there is typically only one queue entry per state in $Open$.

An alternative approach is *reinsertion*, where an improved node is instead inserted into $Open$ as a new queue entry. The older entry remains in the priority queue but is stale.

In LAZY (Algorithm 2, A^* with *lazy duplicate detection*), duplicate checking is deferred until nodes are popped from $Open$. Successor nodes are inserted into $Open$ without first checking whether their states have already been generated. Duplicate entries are filtered later when they are selected for expansion (see stale-check description below).

The state-cost information maintained by the two algorithms has different meanings. In EAGER, $g_{best}(s)$ stores the best path cost to s among generated nodes, because duplicate detection is performed during successor generation. In LAZY, $g_{acc} : S \rightharpoonup \mathbb{R}_{\geq 0}$ instead stores the best path cost to each state among entries that have already been popped and accepted. Generated but not yet accepted entries do not update g_{acc}. Therefore, a popped LAZY entry for state s is discarded if s has already been accepted with cost at most $g(n)$, i.e., if $s \in \text{dom}(g_{acc})$ and $g(n) \geq g_{acc}(s)$.

In EAGER implementations without decrease-key, improved paths are handled by inserting a fresh entry into $Open$, while older entries for the same state become stale. Therefore, when an EAGER entry is popped, it is discarded if its path cost is larger than the current generated best cost, i.e., if $g(n) > g_{best}(s(n))$.

In LAZY, every generated node is inserted into $Open$ as a fresh entry. Since generated but unaccepted entries do not update g_{acc}, duplicates are detected only when entries are popped. A popped LAZY entry is discarded if $s(n)$ has already been accepted with cost at most $g(n)$, i.e., if $s(n) \in \text{dom}(g_{acc})$ and $g(n) \geq g_{acc}(s(n))$. Otherwise, the entry is accepted, $g_{acc}(s(n))$ is updated, and $s(n)$ is expanded. If $s(n)$ had previously been closed with a larger cost, it is reopened under the common reopening policy.

In the rest of the paper, we use the term *expansion* to refer to the entire process of removing a node from $Open$, accepting it as the current representation of its state $s(n)$, and generating all of its successors. Nodes that are popped from $Open$ but discarded by duplicate or stale checks are not counted as expanded.

The choice of eager vs. lazy duplicate detection offers several trade-offs. EAGER incurs a larger number of $Closed$ lookups, as EAGER will look up every generated node in $Closed$, while LAZY will only perform $Closed$ lookups for nodes which are popped from $Open$ and become candidates for expansion, and some nodes which remain in $Open$ when A^* terminates will never have been looked up in $Closed$. Looking up a node in $Closed$ typically requires a hash table query. In applications such as domain-independent planning using relatively expensive heuristic evaluation functions, this lookup is a relatively cheap operation. On the other hand, in applications with very fast node generation rates such as domain-specific solvers for the sliding-tile puzzle, this lookup can be a significant overhead [5].

LAZY may consume more memory than EAGER. This is because more nodes corresponding to the same state can be in $Open$ with LAZY compared to EAGER (which would eliminate such nodes immediately after generation).

Many textbooks present A^* with duplicate detection corresponding to eager duplicate detection with decrease-key [14, 6, 13, 12, 7]. To our knowledge, the first explicit description of an implementation which corresponds to lazy duplicate detection is [5]. They showed that EAGER was significantly faster than LAZY on the sliding tiles puzzle due to the overhead of $Closed$ hash table lookups. The terms "eager" and "lazy" duplicate detection were used to distinguish between these two types of duplicate detection in [11]. For an external memory search using an SSD, they argued that lazy duplicate detection was preferable to eager duplicate detection due to the very high cost of $Closed$ lookups when $Closed$ was implemented on an SSD.

3 Theoretical Comparison

We compare the nodes expanded by EAGER (with and without decrease-key) and LAZY. Specifically, we analyze the following concrete policies: (1) EAGER with reinsertion (no decrease-key), (2) EAGER with decrease-key and timestamp refresh, (3) EAGER with decrease-key and no timestamp refresh, and (4) LAZY with reinsertion of all generated entries.

Algorithm 1: A^* with eager duplicate detection

Require: start state s_0
1: $Open \leftarrow$ empty priority queue
2: $Closed \leftarrow \emptyset$
3: $g_{best} \leftarrow$ empty map
4: $n_0 \leftarrow$ new node with $s(n_0) = s_0$, $g(n_0) = 0$, $parent(n_0) = \perp$
5: $g_{best}(s_0) \leftarrow 0$
6: insert n_0 into $Open$ with priority $f(n_0) = g(n_0) + h(s_0)$
7: **while** $Open$ is not empty **do**
8: $n \leftarrow$ pop-min($Open$)
9: **if** $g(n) > g_{best}(s(n))$ **then**
▷ old entry without decrease-key
10: **continue**
11: **if** $s(n)$ is a goal state **then**
12: **return** path obtained by following parent
13: pointers from n
14: $Closed \leftarrow Closed \cup \{s(n)\}$
▷ mark $s(n)$ as expanded
15: **for all** $n' \in$ successors(n) **do**
16: $s' \leftarrow s(n')$
17: $g' \leftarrow g(n) + c(n, n')$
18: **if** $s' \notin \mathrm{dom}(g_{best})$ **or** $g' < g_{best}(s')$ **then**
▷ eager duplicate detection
19: $g_{best}(s') \leftarrow g'$
20: $g(n') \leftarrow g'$
21: $parent(n') \leftarrow n$
22: **if** $s' \in Closed$ **then**
23: $Closed \leftarrow Closed \setminus \{s'\}$
▷ reopen if a better path is found
24: **if** $Open$ supports decrease-key **then**
25: **if** a node for s' is already in $Open$ **then**
26: decrease-key that node to
27: priority $g' + h(s')$
28: **else**
29: insert n' into $Open$ with
30: priority $g' + h(s')$
31: **else**
32: insert n' into $Open$ with
33: priority $g' + h(s')$
▷ old entries, if any, become stale
34: **return** failure

Algorithm 2: A^* with lazy duplicate detection

Require: start state s_0
1: $Open \leftarrow$ empty priority queue
2: $Closed \leftarrow \emptyset$
3: $g_{acc} \leftarrow$ empty map
4: $n_0 \leftarrow$ new node with $s(n_0) = s_0$, $g(n_0) = 0$, $parent(n_0) = \perp$
5: insert n_0 into $Open$ with priority $f(n_0) = g(n_0) + h(s_0)$
6: **while** $Open$ is not empty **do**
7: $n \leftarrow$ pop-min($Open$)
8: **if** $s(n) \in \mathrm{dom}(g_{acc})$ **and** $g(n) \geq g_{acc}(s(n))$ **then**
9: **continue** ▷ lazy duplicate/stale check
10: **if** $s(n) \in Closed$ **then**
11: $Closed \leftarrow Closed \setminus \{s(n)\}$
▷ mark $s(n)$ as reopened
12: $g_{acc}(s(n)) \leftarrow g(n)$ ▷ the popped entry is accepted
13: **if** $s(n)$ is a goal state **then**
14: **return** path obtained by following parent
15: pointers from n
16: $Closed \leftarrow Closed \cup \{s(n)\}$
▷ mark $s(n)$ as expanded
17: **for all** $n' \in$ successors(n) **do**
18: $s' \leftarrow s(n')$
19: $g' \leftarrow g(n) + c(n, n')$
20: $g(n') \leftarrow g'$
21: $parent(n') \leftarrow n$
22: insert n' into $Open$ with priority $g' + h(s')$
▷ duplicates are allowed
23: **return** failure

Throughout this section, we assume: (1) deterministic successor ordering, (2) nonnegative edge costs, (3) heuristic values depend only on the state, and (4) if a better path to a previously expanded state is found, that state is reopened, and both EAGER and LAZY use the same reopening policy (to handle inconsistent heuristics).

The results are summarized in Table 1. We discuss each case in detail below.

3.1 Analysis Under FIFO Tie-Breaking

We first consider First-In-First-Out (FIFO) tie-breaking, which orders nodes lexicographically by (f, h, t), where

tie-break	EAGER update policy	Result
FIFO	Reinsertion	Same
	Decrease-key with timestamp refresh	Same
	Decrease-key without timestamp refresh	possibly different
LIFO	any	Unbounded expansion-count gap

Table 1: Summary of theoretical results: comparison of expanded nodes

$t(n)$ (also referred to below as the "timestamp" of n) is an effective FIFO tie-breaking key representing the relative newness (with respect to insertion order) of n among $Open$ entries with the same (f, h) key. In FIFO tie-breaking, smaller t (earlier insertion) is preferred.

First, consider FIFO tie-breaking in implementations where every improved $Open$ entry receives a fresh effective timestamp. This happens automatically if improved paths are handled by reinsertion (inserting a new node), and also holds for decrease-key implementations that refresh the entry's timestamp when its key is decreased. This holds in a common implementation strategy where $Open$ is implemented as a two-level bucket priority queue: In such an implementation, $t(n)$ is not explicitly recorded, and instead the position in the bucket serves as the effective timestamp $t(n)$ (newly inserted elements in the bucket have newer $t(n)$ than elements inserted earlier). When node n is updated, it will move from its previous bucket to a new bucket, and will be inserted as the most recent element of that bucket (i.e., effectively the same as updating $t(n)$).

Definition 1 (Live $Open$ representative)**.** *Consider a particular run of an A^* algorithm A using lexicographic node ordering by $(f(n), h(s(n)), t(n))$. A node n for state s is live if A's pop-time stale and duplicate checks would not discard it before expansion if n were to be popped from $Open$ at the current point in the run (equivalently, n is live if it is a currently acceptable representative of s according to A's state-cost information and reopening policy)*

In EAGER, n is live only if $g(n) = g_{best}(s)$ and there is no closed representative of s with cost at most $g(n)$. In LAZY, n is live only if $s \notin \mathrm{dom}(g_{acc})$ or $g(n) < g_{acc}(s)$. Entries that fail these conditions are *stale* or *non-improving duplicates* and cannot be expanded before being discarded.

Definition 2 (Minimum live $Open$ representative)**.** *For a state s, the minimum $Open$ representative of s in A, denoted $m_A(s)$, is the live $Open$ entry n with $s(n) = s$ that is minimum according to the lexicographic $Open$ ordering by $(f(n), h(s(n)), t(n))$, if such an entry exists, and is undefined otherwise.*

Definition 3 (Representative-order equivalence)**.** *Let A and B be two runs of A^* using lexicographic $Open$ ordering by (f, h, t). For each run, let m_A and m_B denote the corresponding minimum live $Open$ representatives. We say that m_A and m_B are* representative-order equivalent*, written*

$$m_A \equiv_{\mathrm{ord}} m_B$$

if m_A and m_B have the same domain and, for all states s, s_1, s_2 in this domain, $g(m_A(s)) = g(m_B(s))$ and

$$\big(m_A(s_1) \prec_A m_A(s_2)\big) \iff \big(m_B(s_1) \prec_B m_B(s_2)\big),$$

where $\prec_A$ and $\prec_B$ are the Open-list orders in the two runs. Since h is state-based, equality of g-values for the same state also implies equality of the corresponding (f, h) keys.

Theorem 1 (Equivalence under FIFO)**.** *Consider* EAGER *and* LAZY*, two versions of A^* given the same start state, edge costs, heuristic h, successor function, and lexicographic $Open$ ordering by $(f(n), h(s(n)), t(n))$ (ties are broken by smaller f, then smaller h, then smaller t), where $f(n) = g(n) + h(s(n))$ and $t(n)$ is an effective FIFO tie-breaking key in each run. In* EAGER *implementations with decrease-key, update operations may refresh this timestamp.*

Assume: (1) A stale-check is applied when a node is popped from $Open$, and stale nodes are discarded. (2) EAGER *either handles improved $Open$ entries by reinsertion, or, if it uses decrease-key, assigns the updated entry the same effective FIFO timestamp it would have received had the improved node been inserted as a fresh $Open$ entry at that point.*

Then EAGER *and* LAZY *perform the same sequence of expansions, including reopenings.*

Proof. We prove by induction on the number of expansions that after every expansion: (1) both EAGER and LAZY have expanded the same states, in the same order, and (2) $m_{eager} \equiv_{\mathrm{ord}} m_{lazy}$.

Initially, the above claim is true, since both algorithms start with only the start node in $Open$.

Assume that the invariant holds after some number of expansions. Since $m_{eager} \equiv_{\mathrm{ord}} m_{lazy}$, the minimum live representatives induce the same $Open$-minimum live state x. EAGER chooses x as its next expansion. LAZY may first pop entries that are not live, because it may contain deferred duplicates that EAGER discarded at generation time. Each such entry fails LAZY's pop-time duplicate/stale check and is not expanded, so removing it does not change m_{lazy}. Hence LAZY's next expansion is also x. Both algorithms generate the same successors of x in the same order.

Consider one such generated successor node n, with state $s = s(n)$. We show that the invariant is preserved after processing n. There are two cases.

Case 1: n becomes the new minimum live $Open$ representative of s. Then n represents a path to s that is strictly better than any existing live representative of s, or else s has no previous live representative and n is not stale. Thus n becomes minimum live $Open$ representative of s in both algorithms. LAZY inserts the generated entry in $Open$. EAGER either inserts a new fresh $Open$ entry for s or updates the existing one. In the reinsertion case, the EAGER entry receives a fresh FIFO timestamp at the same point in successor processing as the corresponding LAZY insertion. In the decrease-key case, Assumption (2) gives the updated EAGER entry the same FIFO timestamp it would have received under the reinsertion case.

Therefore, in both algorithms, the resulting representatives for s have the same state, same path cost, same heuristic value, the same f-value, and the same FIFO position relative to the other minimum live representatives. If this path improves a previously expanded state, both algorithms apply the same reopening policy. In LAZY, the improvement will trigger reopening when the entry is popped, while in EAGER the state may be marked reopened immediately, but this difference does not change the representative order. Thus, $m_{eager} \equiv_{\mathrm{ord}} m_{lazy}$ after processing n.

Case 2: n does not become the new minimum live $Open$ representative of s. Then either n is stale or non-improving with respect to an already accepted representative of s, or it

is dominated by some existing live *Open* representative of s.

If n is stale or a non-improving duplicate because s has already been accepted or expanded with cost at most $g(n)$, then EAGER may discard it immediately, while LAZY may insert it temporarily. In LAZY, such an entry would fail the pop-time duplicate/stale check and therefore is not live, so it does not affect $m_{lazy}(s)$.

Otherwise, n may be live in LAZY. Since n does not become the new minimum live *Open* representative of s, let r be the existing minimum live *Open* representative of s before processing n. Then r is no worse than n in the FIFO (f, h, t) order. Because r and n represent the same state, $g(r) \leq g(n)$. If $g(r) < g(n)$, then, $f(r) < f(n)$ (because r and n have the same h-value), so n cannot be the minimum representative. If $g(r) = g(n)$, then r and n have the same f and h values, but r has an earlier effective FIFO position. Under FIFO tie-breaking, r is preferred to n, so n cannot change the minimum live *Open* representative of s.

Thus, after processing n, the minimum live *Open* representative maps m_{eager} and m_{lazy} remain representative-order equivalent. Since the algorithms process the same successors in the same order, the invariant is preserved after x is fully expanded. This completes the induction. □

If decrease-key is not used, then whenever an improved path is found, it is inserted into *Open* as a new node with a fresh timestamp, so based on Theorem 1, it follows that EAGER and LAZY expand the same sequence of states.

Corollary 1 (FIFO equivalence without decrease-key)**.** *Under FIFO tie-breaking,* EAGER *without decrease-key and* LAZY *expand the same sequence of states, including reopenings.*

Proof. When EAGER does not use decrease-key, whenever an improved path is accepted, it is inserted into *Open* as a fresh node, exactly as in LAZY. EAGER may discard generated duplicates immediately, while LAZY may keep them in *Open* until they are selected/popped. However, such discarded or deferred duplicate entries cannot change $m_{eager}(s)$ or $m_{lazy}(s)$.

Thus, EAGER and LAZY satisfy the representative-order condition of Theorem 1: each accepted improved path is inserted as a fresh *Open* entry, and deferred LAZY-only duplicates cannot change the minimum live representative under FIFO, so they expand the same sequence of states. □

When decrease-key is used, EAGER with decrease-key and LAZY will expand the same sequence of states if the decrease-key operation is implemented such that the updated node also has an updated timestamp $t(n)$.

Corollary 2 (FIFO equivalence with timestamp-refreshing decrease-key)**.** *Under FIFO tie-breaking,* EAGER *with decrease-key and* LAZY *expand the same sequence of states, including reopenings, provided that each decrease-key operation assigns the updated Open entry the same effective timestamp that a fresh insertion of the improved node would receive.*

Proof. Consider an improved path to a state s already in *Open*. LAZY inserts the improved node as a fresh *Open* entry with a fresh timestamp. EAGER decreases the key of the existing *Open* entry for s and places it in the same effective FIFO position that the corresponding fresh insertion would occupy among live representatives. Thus, after the update, both algorithms have the same minimum live *Open* representative $m_A(s)$.

For non-improved (duplicate) paths to a state s already in *Open*, LAZY inserts a fresh *Open* entry with a fresh timestamp. EAGER discards it immediately. Such an entry cannot change $m_A(s)$.

Thus, after every *Open* update, the conditions for Theorem 1 are satisfied. Therefore, EAGER and LAZY expand the same sequence of states. □

FIFO non-equivalence when decrease-key does not refresh timestamp Theorem 1 makes the assumption that whenever a node for a state which is already represented in *Open* is generated, it receives a fresh timestamp when inserted into *Open*.

This is a key constraint for ensuring that EAGER and LAZY expand the same sequence of states. If we break this assumption, the equivalence guarantee is lost. For example, if EAGER with FIFO tie-breaking uses decrease-key, but the decrease-key operation does not update $t(n)$, then EAGER and LAZY can expand different state sequences.

Consider the graph in Figure 1. Assume the blind heuristic. After expanding the start state s, *Open* contains c (cost 4), a (cost 5), and b (cost 6). Assume these successors were opened in the order b, a, c, so $t(b) < t(a) < t(c)$. Next, we expand c. This generates a better path to b with cost 5. In LAZY, this creates a fresh node b' (cost 5), newer than a, so a will be expanded before b according to FIFO tie-breaking policy, and the expansion order is s, c, a, b. However, EAGER with decrease-key will change the old node b to cost 5, without updating $t(b)$. Thus, according to FIFO, EAGER will expand b before a, and the expansion order is s, c, b, a.

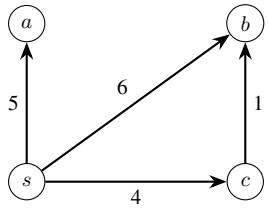


Figure 1: FIFO decrease-key counterexample.

3.2 Analysis Under LIFO Tie-Breaking

Next, we consider Last-In-First-Out (LIFO) tie-breaking, which orders nodes lexicographically by $(f, h, -t)$, where larger t (later insertion) is preferred. Under LIFO tie-breaking, the equivalence results for the sequence of states expanded by EAGER and LAZY do not hold. In fact, the number of expanded states can differ by an arbitrarily large amount. The reason is that a non-improving duplicate with the same cost as the previous representative of a state will

be immediately discarded by EAGER before it enters *Open*, but LAZY would insert it as a fresh entry in *Open*, giving it the most recent timestamp (and hence highest priority among nodes with the same f and h values under LIFO tie-breaking). This can lead to different expansions from the same (f, h) plateau. The following theorem shows that the difference in the number of state expansions is unbounded.

In standard A^* with admissible heuristics, any live *Open* node with $f < C^*$ must be selected before an optimal goal with $f = C^*$. Therefore, differences in expansion between the policies can only arise among nodes with $f = C^*$, modulo stale/duplicate entries and reopenings. However, the number of such states can be very large.

Theorem 2 (Unbounded expansion gap under LIFO tie-breaking)**.** *For A^* with LIFO tie-breaking, the difference in the number of states expanded by* EAGER *and* LAZY *can be arbitrarily large in either direction. For every integer N, there exists an instance on which* EAGER *expands at least N more nodes than* LAZY*, and there exists an instance on which* LAZY *expands at least N more nodes than* EAGER*.*

Proof. We construct two families of instances, both based on the following gadget construction. Let the optimal solution cost be $C^* = 1$. Let the subgraph rooted at q_e be a zero-cost chain $q_e \rightarrow q_1 \rightarrow q_2 \cdots \rightarrow q_N$, where edges have cost 0, none of $q_e, \ldots q_N$ is a goal, and h=0 for every state in the chain. Then every state reachable from q_e has $f = g + h = 1 = C^*$, and therefore belongs to the final plateau. If one duplicate-detection policy enters the subgraph rooted at q_e before reaching the goal, while the other strategy reaches the goal without entering the subgraph rooted at q_e, the number of expansions by the two strategies differ by at least N. N can be chosen to be arbitrarily large, so the expansion gap is unbounded.

First, consider the instance in Figure 2. Suppose successors are generated in an order such that after expanding s and p, $t(y) > t(q_e)$, so according to LIFO tie-breaking, both EAGER and LAZY expand y before q_e. When y is expanded, it generates another path to x ($s \rightarrow p \rightarrow y \rightarrow x$) with the same cost as the existing *Open* entry for x, a non-improving duplicate.

At this point, the behaviors of the two duplicate detection policies diverge. LAZY inserts this newly generated copy of x into *Open*. According to LIFO tie-breaking, this copy of x is selected next. Since x has not yet been accepted, the lazy duplicate check does not discard it, and expanding x immediately generates and reaches the goal g without entering the subgraph rooted at q_e.

In contrast, EAGER detects that the newly generated copy of x is a duplicate of the existing *Open* node for x and is discarded. Thus, after expanding y, the node q_e is selected before the original copy of x. Since q_e is the entry to an arbitrarily large zero-cost subgraph whose states all satisfy $f = C^*$, EAGER expands the entire subgraph containing at least N nodes before returning to x and then reaching g.

Therefore, for the family of instances represented by Figure 2, EAGER expands at least N more nodes than LAZY.

Second, consider the instance in Figure 3. Suppose successors are generated in an order such that after expanding s, its successors x, g, and p are generated in that order so that $t(x) < t(g) < t(p)$. According to LIFO ordering, p is expanded next. Suppose this is followed by y, which results in another path to x ($s \rightarrow p \rightarrow y \rightarrow x$) (cost 1, so an equal cost, non-improving duplicate).

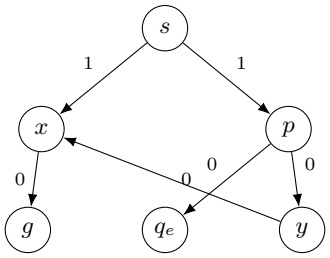


Figure 2: Under LIFO tie-breaking, EAGER can expand arbitrarily more nodes than LAZY. $h = 0$ everywhere. q_e is the entry to an arbitrarily large subgraph with all 0-cost edges. Successors of s are generated in the order x, p; successors of p are generated in the order q_e, y.

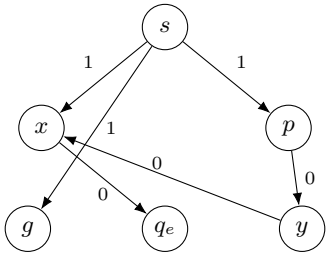


Figure 3: Under LIFO tie-breaking, LAZY can expand arbitrarily more states than EAGER. $h = 0$ everywhere. q_e is the entry to an arbitrarily large subgraph with all 0-cost edges. Successors of s are generated in the order x, g, p.

In EAGER, the newly generated copy of x is immediately detected as a duplicate of the existing node for x in *Open*, so it is discarded. Then, *Open* contains the original x and goal g. According to LIFO policy, g is selected for expansion next, and EAGER terminates without entering the subgraph rooted at q_e.

In contrast, LAZY inserts the new copy of x into *Open*. According to LIFO policy, it is selected next for expansion, which leads to q_e. Since q_e is the entry to an arbitrarily large zero-cost subgraph whose states all satisfy $f = C^*$, LAZY expands the entire subgraph containing at least N nodes before returning to x and then reaching g.

Therefore, for the family of instances represented by Figure 3, LAZY expands at least N more nodes than EAGER.

In both constructions, the path that causes the search order divergence is a non-improving duplicate with the same path cost as an existing *Open* representative, so this argument does not depend on whether decrease-key is used or not. Since N can be arbitrarily large, the expansion gap between EAGER and LAZY under LIFO tie-breaking is unbounded in both directions. □

4 Experimental Results

We experimentally compare EAGER and LAZY in domain-independent cost-optimal planning using A^*, based on the implementation in the widely used Fast Downward (FD) planner [9]. The EAGER configuration is based on FD's `astar` search plugin. Since the standard FD `astar` configuration uses FIFO tie-breaking, we extended the open-list implementation to also support LIFO tie-breaking within each (f, h) bucket. The LAZY configuration is implemented as a new FD search plugin using the same (f, h) ordering and the same FIFO/LIFO bucket policy.

Both configurations use the same evaluator structure: nodes are ordered lexicographically by (f, h), where $f = g + h$, and remaining ties are broken either by FIFO or LIFO order. For EAGER, we use FD's `TieBreakingOpenList`. For LAZY, we implement an equivalent bucketed open list whose keys are (f, h); each bucket stores all generated entries with that key and removes entries from the front for FIFO tie-breaking and from the back for LIFO tie-breaking.

In the EAGER implementation, the search space stores the current best g-value and parent for each generated state, and non-improving duplicates are discarded during successor generation. In the LAZY implementation, each generated path is stored as a separate open-list entry containing its own g-value and parent information; the state-level search-space record is updated only when an entry is popped and accepted for expansion.

All tested configurations use *Open* without decrease-key support and with reinsertion/stale-checks. This matches the "no decrease-key" cases in Table 1.

For the heuristic function, we use the admissible and consistent h_{max}[4], and the admissible but inconsistent h_{LMcut}[10].

All experiments were run on a machine with Ubuntu 22.04.2 LTS, equipped with an Intel Xeon E5-2670 v3 @2.30GHz processor. Each run was given a time limit of 5 minutes and a memory limit of 4 GiB.

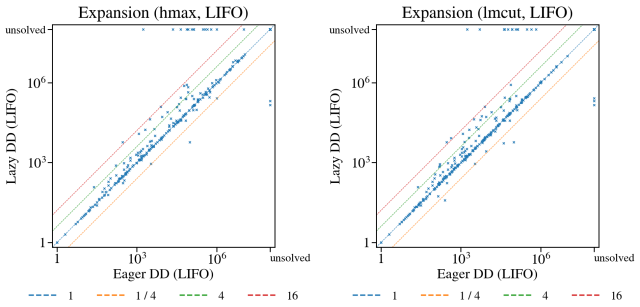
4.1 Results on Standard IPC Benchmarks

The first set of experiments is with standard IPC domains. We used the `optimal_strips` benchmark suite from the Fast Downward benchmark repository [1], consisting of 1847 planning tasks from 66 domains.

We compare four configurations for each heuristic: EAGER with FIFO tie-breaking, EAGER with LIFO tie-breaking, LAZY with FIFO tie-breaking, and LAZY with LIFO tie-breaking.

Figure 5 shows pairwise comparisons between EAGER and LAZY on the standard benchmark suite.

The expansion results are consistent with our theoretical analysis. With FIFO tie-breaking, EAGER and LAZY expand the same number of states on all solved tasks. With LIFO tie-breaking, differences in the number of expanded nodes were observed. With h_{max}, there were 5 tasks where the expansion counts differed by more than a factor of 2: mprime(p26) and openstacks-opt08(p12, p22, p07, p10). With h_{LMcut}, there were 9 tasks where the expansion counts differed by more than a factor of 2: mprime(p08, p12, p16, p26), mystery(p19), openstacks-opt08(p12, p22), and openstacks-opt11(p07, p10). The differences do not show a systematic advantage for either duplicate-detection policy.



(a) h_{max} (b) h_{LMcut}

Figure 4: Comparison of state expansions on zero-cost benchmarks with LIFO tie-breaking. Points above the diagonal indicate instances where LAZY expands more states.

We also compared the number of duplicate detection checks, i.e., the number of times the implementation queries the relevant data structure to decide whether a generated successor or popped *Open* entry is a duplicate. The duplicate detection column in Figure 5 shows that LAZY incurs significantly fewer checks than EAGER. This is because LAZY checks nodes only when they are popped from *Open*; nodes left in *Open* at termination are never checked. The reduction is larger with h_{LMcut}, likely because its stronger guidance and inconsistency leave more generated or stale entries unexpanded in *Open*.

The search time column in Figure 5 shows that fewer duplicate-detection calls do not necessarily yield faster search: with the cheaper h_{max}heuristic, the extra *Open* operations of LAZY often make it slower than EAGER, whereas with the more expensive h_{LMcut}, the two variants have similar runtimes.

Finally, Figure 5 shows that LAZY generally uses more memory. This is because more duplicate and stale entries remain in *Open* until they are popped, and these entries must retain state identifiers and parent information.

4.2 Results on Zero-Cost Benchmarks with LIFO Tie-Breaking

The results with standard IPC benchmark tasks showed that significant differences in node expansions were relatively rare. However, significant differences in expanded nodes occurred in openstacks, a domain where some operators have cost 0, and a large final plateau region with $f = C^*$. This is because EAGER and LAZY must expand the same number of nodes with $f < C^*$. Any differences in the number of expanded nodes can only arise in the final plateau where nodes have $f = C^*$, and domains such as openstacks which have zero-cost operators can have very large final plateaus.

To further investigate the search behavior of EAGER vs. LAZY on problems with large final plateau regions, we compared the algorithms on the set of "Zero-cost benchmarks"

Heuristic	Cov. E	Cov. L	Both	L>E	L<E	GM L/E
h_{max}	245	232	230	127	51	1.252
h_{LMcut}	280	269	266	139	44	1.249

Table 2: Zero-cost benchmark results with LIFO tie-breaking. Cov. E and Cov. L denote the coverage of EAGER and LAZY over all 620 tasks. "Both" is the number of tasks solved by both. L>E and L<E are the number of tasks solved by both where LAZY expands more or fewer states than EAGER, and vice versa (the remaining tasks had identical expansion counts). GM L/E is the geometric mean of the LAZY-to-EAGER expansion ratio over tasks solved by both.

by [3]. These are based on IPC tasks, except that some operators have been modified to have cost 0, to model tasks where the actual cost of some actions are much larger than others. For example, in the original IPC driverlog domain, all operators have cost 1, but in the zero-cost benchmark suite version, load-package is modified to have cost 0, while the other operators have the original cost of 1. We used the full set of 620 zero-cost tasks from 28 domains.

Figure 4 shows that on these instances, LAZY tends to expand more states than EAGER under LIFO tie-breaking, for both $h_{max}$and h_{LMcut}. As summarized in Table 2, coverage is slightly lower for LAZY than for EAGER for both heuristics. Among tasks solved by both, LAZY expands more states than EAGER on substantially more instances than the reverse. The geometric mean expansion ratio is about 1.25 for both heuristics, indicating a consistent multiplicative increase in expansions for LAZY on the zero-cost benchmarks.

A possible explanation is that lazy duplicate insertion changes the meaning of LIFO tie-breaking inside large plateaus. Asai and Fukunaga [3] observed that zero-cost actions create large plateaus in which many nodes have identical f- and h-values, so the search behavior is largely determined by the tie-breaking rule. They also note that LIFO tends to behave like a depth-first traversal of the plateau, because it repeatedly selects the most recently generated node.

Under EAGER, this depth-first-like behavior is applied to newly accepted state representatives: equal-cost duplicates are removed before insertion into $Open$, and therefore do not affect the LIFO order. Under LAZY, however, equal-cost duplicates are inserted into $Open$ and receive the most recent LIFO priority. Thus, LIFO is applied to generated path entries rather than to unique states. These duplicate entries can interrupt the depth-first-like plateau traversal and redirect the search toward regions that EAGER would not prioritize. Since zero-cost benchmarks contain large final plateaus, this effect is amplified, providing a plausible explanation for the observed increase in expansions for LAZY.

5 Conclusion

We compared EAGER and LAZY duplicate detection for A^*, and showed that, under FIFO tie-breaking, EAGER without decrease-key and LAZY expand the same sequence of states. The same equivalence holds for EAGER with decrease-key if decrease-key refreshes the effective timestamp. However, under LIFO tie-breaking, the number of nodes expanded may differ by an arbitrarily large amount in either direction. Experimentally, we showed that the divergence of search behavior with LIFO tie-breaking occurs in practice, especially on problems where a significant amount of search among nodes with f-values equal to the optimal solution cost is required. These results show that duplicate-detection policy is not merely an implementation choice which affects runtime, and can significantly change the search behavior of A^*.

While we focused on A^*, the theoretical results can be generalized to other best-first search such as greedy best-first search (GBFS), and the experimental gap between EAGER and LAZY we observed for A^* may also be found in other algorithms such as GBFS. This is a direction for future work.

References

[1] AI Basel. 2024. Fast Downward Benchmark Collection. https://github.com/aibasel/downward-benchmarks. GitHub repository, accessed 2026-05-06.

[2] Asai, M.; and Fukunaga, A. 2017. Tie-breaking strategies for cost-optimal best first search. *Journal of Artificial Intelligence Research*, 58: 67–121.

[3] Asai, M.; and Fukunaga, A. S. 2016. Tiebreaking Strategies for A* Search: How to Explore the Final Frontier. In *AAAI*, 673–679.

[4] Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence*, 129(1–2): 5–33.

[5] Burns, E. A.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing fast heuristic search code. In *Fifth Annual Symposium on Combinatorial Search*.

[6] Edelkamp, S.; Schrödl, S.; and Koenig, S. 2010. *Heuristic Search: Theory and Applications*. Morgan Kaufmann Publishers Inc.

[7] Ghallab, M.; Nau, D.; and Traverso, P. 2025. *Acting, Planning, and Learning*. Cambridge University Press.

[8] Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

[9] Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Int. Res.*, 26(1): 191–246.

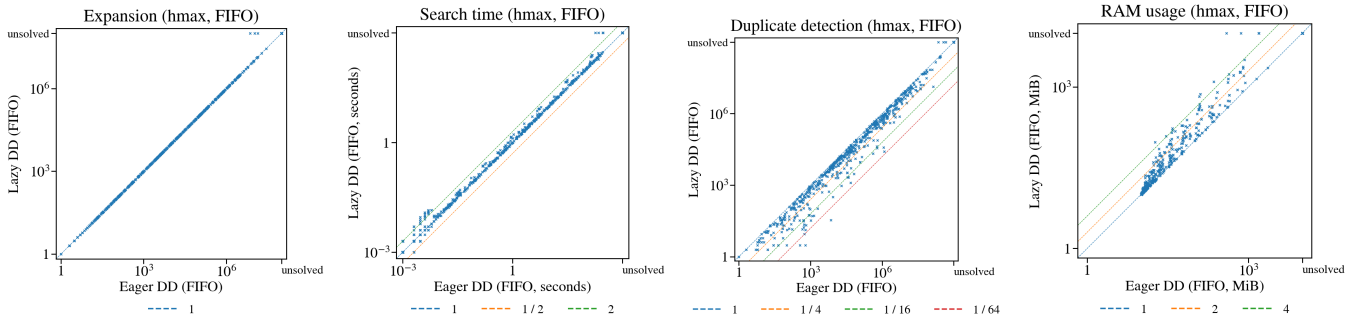
[10] Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proc. ICAPS*.

[11] Lin, S.; and Fukunaga, A. 2018. Revisiting Immediate Duplicate Detection in External Memory Search. In *Proc. AAAI*, 1347–1354.

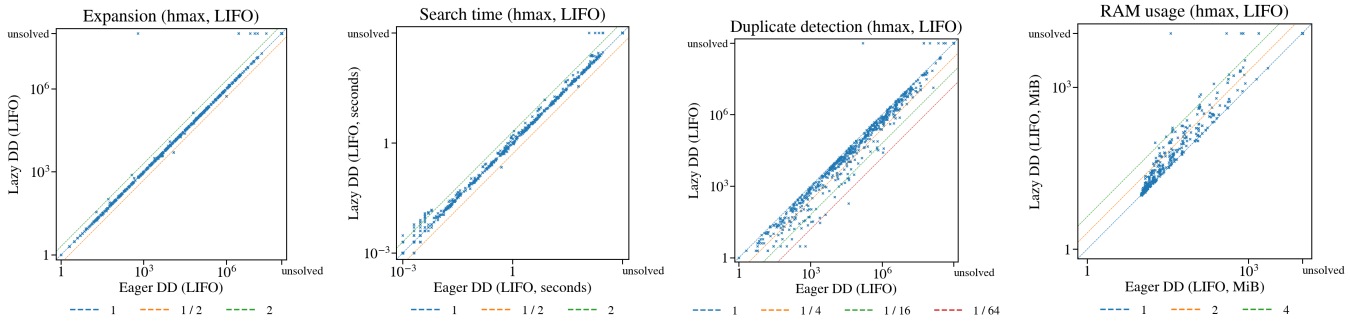
[12] Nilsson, N. 1971. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill.

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

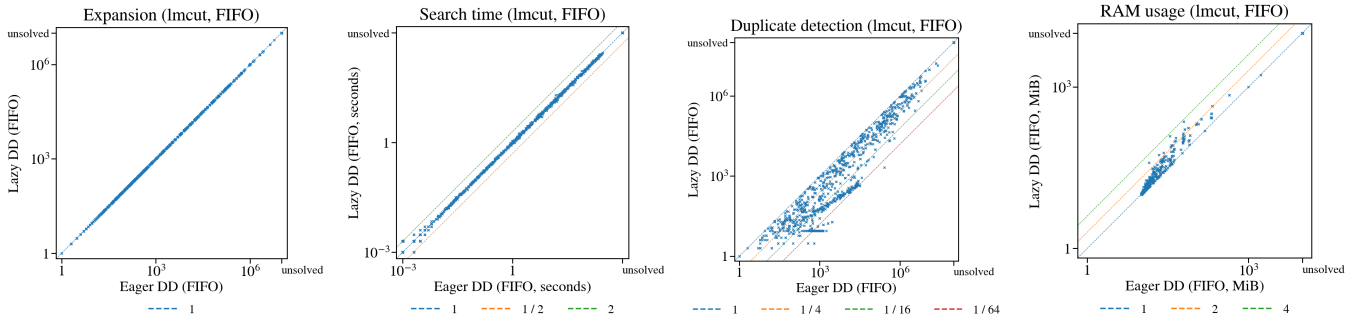
[14] Russell, S.; and Norvig, P. 2021. *Artificial Intelligence, a Modern Approach, 4th edition*. Prentice Hall.



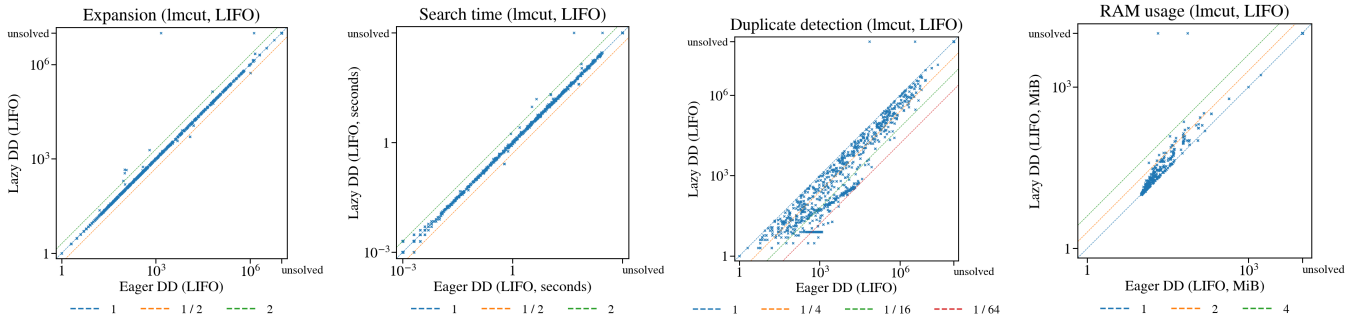
(a) h_{max}heuristic with FIFO tie-breaking



(b) h_{max}heuristic with LIFO tie-breaking



(c) h_{LMcut}heuristic with FIFO tie-breaking



(d) h_{LMcut}heuristic with LIFO tie-breaking

Figure 5: Comparison of EAGER and LAZY on standard IPC benchmarks for $h_{max}$and h_{LMcut}heuristics, under FIFO and LIFO tie-breaking strategies. Columns show expansions, search time, duplicate detection checks, and peak RAM usage.