Evaluation of Baseline Methods for IDD-based SSD External Memory Search

Yuki Suzuki, Alex Fukunaga

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

Abstract

Many difficult search problems cannot be solved by algorithms such as A^* using only RAM. Search algorithms which use external memory such as SSDs and HDDs with much higher capacity than RAM have been proposed in previous work, but previous work has focused on delayed duplicate detection approaches, as well as complex immediate duplicate detection (IDD) methods, and relatively simple methods for IDD have not been systematically studied. In addition, the effect of OS-level mechanisms for managing and speeding up accesses to external memory, such as page caches, has not been studied. This paper addresses these gaps in the literature by evaluating and analyzing the performance of simple baseline approaches for IDD-based A^*.

Code — https://github.com/Yuki0631/SIDD-A
Extended version — https://arxiv.org/abs/2606.01840

1 Introduction

Heuristic search algorithms such as A^* are limited by the memory available to store nodes. One approach to overcoming memory limitations is to use external memory such as hard disk drives (HDDs) or solid state drives (SSDs). Since accessing external memory is significantly slower than accessing RAM, previous work on search algorithms investigated methods to overcome this performance bottleneck.

Early work on external-memory search focused on Delayed Duplicate Detection (DDD) methods that reorganized the structure of the search algorithm in order to replace random accesses with sequential reads and writes, an approach which was especially effective in overcoming the massive (6 orders of magnitude) latency difference between RAM and HDDs. More recently, the ubiquity of SSDs, which are 3 orders of magnitude faster than HDDs, made Immediate Duplicate Detection (IDD), which allows standard search strategies (node expansion orders) to be easily implemented, a practical alternative to DDD.

Although SSDs are much faster than HDDs, they are orders of magnitude slower than RAM, so algorithms that use SSD as external memory were designed to reduce accesses to the SSD. Early work on external-memory search for model checking showed that multi-layer data structures which partly reside in RAM and partly on the SSD are effective in improving performance by reducing SSD accesses (Edelkamp et al. 2011). A*-IDD (Lin and Fukunaga 2018) is an implementation of A^* with IDD, which uses *compression*, a multi-layer technique for implementing the CLOSED hash table in order to reduce the cost of duplicate detection (Edelkamp et al. 2011). Compression seeks to minimize read accesses to the SSD by implementing CLOSED as a two-level hash table, where an internal hash table in RAM points to entries in the external hash table on the SSD. A*-IDD was shown to outperform DDD-based A^* on IPC domain-independent planning benchmarks.

However, IDD-based external-memory A^* is not well understood. Although it was shown that A*-IDD, a relatively complicated algorithm, could be competitive with DDD approaches, the factors underlying the performance of A*-IDD were not investigated in depth. For example, it was assumed that multi-layered techniques such as compression were necessary, and Lin and Fukunaga (2018) compared two variants of compression (Edelkamp et al. 2011), where RAM is used to store a compressed index of pointers to the CLOSED data structure on external memory, but did not evaluate simpler, baseline implementations of IDD. As another example, Lin and Fukunaga (2018) allude to the significant effects of the OS page cache mechanism on overall algorithm performance, but to our knowledge, previous work has focused on the algorithmic issues, without investigating systems (OS and device) level aspects that affect IDD performance.

Thus, despite the ubiquity of SSDs, some basic questions regarding IDD have not been investigated, including: What are reasonable baseline approaches to IDD, how do they perform, and where are the bottlenecks that need to be addressed by more sophisticated methods (such as compression)? This paper seeks to fill this gap through a focused re-evaluation of IDD baselines.

After a review of preliminaries and related work, we start by evaluating the behavior of the simplest possible baseline approach to IDD, which is to simply run A^* and rely on the OS virtual memory subsystem to use external memory when RAM is exhausted. Then, we propose SIDD-A^*, a simple, baseline algorithm for IDD-based external A^* search. Although previous work focused on open addressing implementations of hash tables for IDD, which necessitated meth-

ods such as compression, SIDD-A^* uses a straightforward separate chaining based implementation without user-level caching, buffering, or multi-layer data structures. We use SIDD-A^* to investigate the effects of the page cache and to better understand the system-level bottlenecks in IDD. We find that under high memory pressure, SIDD-A^* can achieve node expansion rates comparable to A*-IDD, while achieving expansion rates significantly higher than A*-IDD and closer to RAM-based A^* in low memory-pressure situations.

2 Preliminaries and Related Work

2.1 SSD File Access in Linux

I/O access in a standard OS such as Linux involves several layers of abstraction. In Linux, when a C++ program writes binary data to a file, the request enters the kernel through a system call, where the Virtual File System routes it to the correct filesystem. The data is first copied into the kernel's page cache (in RAM), and the filesystem later translates the file offset into logical block addresses. The block layer sends these as I/O requests to the NVMe driver, which submits NVMe commands over PCIe to the SSD device. The SSD's controller then maps the logical addresses to physical NAND flash cells and programs the data into flash memory. Each of these layers (page cache, block layer, NVMe SSD device) potentially incurs performance costs.

We consider two I/O interfaces: `pwrite`, which performs explicit system calls and copies data through the page cache, and `mmap`, which accesses files via memory mapping and page faults. The latter reduces syscall overhead but provides less explicit control over write timing.

Page Cache The Linux page cache stores file data in RAM, allowing reads and writes to be served from memory. Writes are buffered and flushed asynchronously, and cached pages are evicted under memory pressure.

It is possible to bypass the page cache with Direct I/O, a file access mode in which data transfers occur directly between user-space buffers and the storage device, bypassing the kernel's page cache. In Linux, this behavior is typically enabled by opening a file with the `O_DIRECT` flag.

2.2 Previous work on external-memory A^*

The primary bottleneck in external-memory search is duplicate detection, which determines whether a generated node is a duplicate of a previously generated node. This is a bottleneck because duplicate detection requires determining whether any previous record of the node exists in either RAM or external memory.

Previous work on external-memory search can be broadly classified based on how they perform duplicate detection. Most previous work focused on Delayed Duplicate Detection (DDD), an approach which does *not* perform duplicate detection on individual nodes when they are generated, and instead periodically performs a duplicate detection phase on large batches of nodes. For example, sorting-based DDD writes all newly generated nodes to a temporary file, sorts the file, and removes duplicates in a linear scan of the sorted file (Korf 2003; Edelkamp, Jabbar, and Schrödl 2004). Another approach is hash-based DDD (Korf 2004, 2008, 2016; Hatem, Burns, and Ruml 2018; Siag et al. 2024).

DDD has been shown to be highly effective in some domains, but DDD is not a simple modification of A^*. In DDD, node expansion order (among nodes with the same f-value) is strongly influenced by the specific DDD mechanism (e.g., the sort order in sorting-based DDD), making expansion order and duplicate detection *non-orthogonal*. For example, it is not trivial to implement complex tie-breaking strategies among nodes with the same f and h values which have been shown to influence search efficiency in RAM-based A^* (Asai and Fukunaga 2017) with DDD.

Immediate Duplicate Detection (IDD) is an approach where the expansion order can, in principle, be exactly the same as standard RAM-based A^*, except that some/most of the data structures are stored on external memory instead of RAM. In *eager* IDD, the duplicate detection step (checking CLOSED) is performed immediately after node generation. In *lazy* IDD, duplicate detection is performed before a node is expanded. Eager IDD requires less RAM but incurs more duplicate detection checks, whereas lazy IDD uses more memory but incurs fewer duplicate detection checks.

In IDD, as with standard A^*, node expansion order and duplicate detection are much more orthogonal than in DDD, and IDD preserves the ability to implement the same best-first priority orders as A^* (including tie-breaking among nodes with the same f and h values). However, the large gap in random access latency between RAM and external memory can result in significantly slower node expansion rates compared to RAM-based A^*. Previous work proposed approaches to address this problem using in-memory (RAM) buffers and hash tables to minimize external-memory accesses (Edelkamp et al. 2011; Edelkamp, Schrödl, and Koenig 2010; Lin and Fukunaga 2018).

Open addressing and Separate Chaining for CLOSED In standard A^*, CLOSED is typically one of the largest data structures and is represented as a hash table to support the query, "has state s been seen before?" Hash table implementations can be broadly categorized into approaches based on *separate chaining* or *open addressing*. In separate chaining, each table index entry holds a chain of states that have the same hash value, and hash collisions are resolved by traversing the chain until an entry that matches the state is found.

In contrast, in open addressing, each table entry contains only one item. Collisions are resolved by *probing* additional slots in some sequence until a match is found.

Previous work on IDD has focused on open addressing. Early theoretical work on IDD considered separate chaining as an option for representing CLOSED, but dismissed it in favor of open addressing because chaining incurs storage overhead (an explicit pointer to the next element in the chain, which is unnecessary in open addressing), and processing overhead for reading the next element (compared to open addressing with linear probing, which can read multiple entries in a single read I/O operation) (Edelkamp, Schrödl, and Koenig 2010; Edelkamp et al. 2011).

In a basic implementation of CLOSED as an open ad-

dressed hash table on the SSD, each table entry contains all data for the node, and each insertion of a node into this table incurs a relatively expensive random write operation into some location in the file representing the table (Edelkamp, Schrödl, and Koenig 2010), with little access locality.

Compression and A*-IDD Previous work on IDD-based A^* is based on a technique called compression (Edelkamp et al. 2011), which separates a hash table into a RAM-resident portion and an external portion.

Compression represents CLOSED as follows. The (uncompressed) external table holds the actual node data (state values and f, g values) and is an array where entries are appended sequentially. The (compressed) internal table is an open addressed hash table in RAM. This design greatly alleviates the random write bottleneck of the basic open addressing table design described above by always sequentially appending to the external table, as well as reducing reads using the internal table.

Each entry in the internal table is a pointer to an entry in the external table on the SSD. Given a node n for state s, duplicate detection in CLOSED works as follows. First, the internal table is probed for s . If this probe fails, then we know there is no duplicate in CLOSED (without having to access the external table). If the probe returned a valid pointer, then we check external[p]. If external[p].state = s, then we found a duplicate. Otherwise, external[p].state was a hash collision, so we continue by probing the next entry in the internal table (according to the open addressing). This continues until either the probe fails (n is not a duplicate), or we find a pointer to an external table entry whose state is equal to s (n is a duplicate). In addition, a small buffer in RAM can be used as a cache, as well as a means to batch the writes to reduce the number of writes to SSD.

A*-IDD (Lin and Fukunaga 2018) introduced *segmented compression*, where states are mapped to a segment (among p partitions) of the external table. False positive probes into the external table would require both a segment and entry collision, thus reducing the probability of false positives (and the accompanying expensive read access to SSD) by a factor of p. A*-IDD uses lazy duplicate detection.

2.3 Experimental Setup and Preliminaries

Experimental Platform. All experiments were conducted on a machine equipped with an Intel Core i7-14700KF CPU, 32 GiB of DDR5-5600 memory (2×16 GiB Micron DIMMs), and a PCIe Gen4 NVMe SSD (Samsung 990 series, manufacturer-rated 4 KiB random IOPS: 1400K read and 1550K write). Unless otherwise noted, experiments were performed on Ubuntu 24.04.3 LTS under identical hardware and software configurations.

Memory Restriction and Measurement Memory is restricted using the cgroup v2 memory controller, `memory.max` setting which limits all RAM used by the process, including the page cache. For memory usage, we report the peak memory consumption using the cgroup v2 `memory.peak` metric, which captures all memory charged to the cgroup during execution, including anonymous memory, file-backed page cache, and relevant kernel-side allocations. To reduce inter-run interference, we drop the page cache before each measurement run.

Baseline Search Algorithm and Heuristics We evaluate IDD-based A^* for heuristic-search based domain-independent, classical planning. As the RAM-based baseline, we use the Fast Downward (FD) implementation of A^* (Helmert 2006), which performs eager duplicate detection and uses a (f, h, FIFO) tie-breaking policy.

We evaluate the search algorithms with two representative heuristics: `blind` and `merge-and-shrink` (Helmert et al. 2014). The `blind` heuristic has negligible computation cost, allowing the experiments to expose differences in external I/O performance more clearly. Both heuristics are admissible in our unit-cost STRIPS setting; `merge-and-shrink` is also consistent and lookup-table based. For this study, we ignore the time cost of precomputing the heuristic tables, although under RAM-limited settings the RAM limit is increased by a domain-dependent amount ($\sim$ 100 MiB) for precomputation.

Benchmark Instances We used STRIPS planning tasks from the Autoscale benchmark suite (Torralba, Seipp, and Sievers 2021), derived from IPC benchmarks. In order to evaluate all of the algorithms in a controlled experimental environment using the same SSD device, all experiments are executed on the same machine, and only one run is executed at a time. Given that some of the external memory methods sometimes run much slower than standard RAM-based A^*, it is not feasible to run all of the algorithms on a large set of instances as is commonly done for RAM-based search algorithms for classical planning. Therefore, we carefully selected a set of benchmark instances to match the experiment design. The detailed process and criteria for selecting an appropriate set of benchmark instances are described in (Suzuki and Fukunaga 2026).

3 A^*: A Trivial Baseline for IDD

The simplest possible IDD-based baseline for A^* is to run the standard in-memory A^* algorithm under a strict RAM limit and rely on the operating system's virtual memory mechanism once that limit is reached. When the search requires more memory than fits in RAM, the OS must continually evict and reload pages, effectively using external storage as backing memory.

Figure 1 shows the expansion rate of Fast Downward (Helmert 2006) running A^* with the `blind` heuristic on blocksworld-p23 under a RAM limit of 2 GiB. At the beginning of the search, the expansion rate is high, at roughly several 10^5 nodes per second. RAM usage reaches the 2 GiB limit within the first few dozen seconds, but the sharp slowdown occurs only after a lag of roughly 100 seconds. Around 200 seconds, the search undergoes a brief near-stall, then recovers to a much lower steady-state regime of roughly several 10^4 nodes per second. For the remainder of the run, RAM usage stays near the imposed limit while the search remains in this degraded low-throughput state.

This behavior is consistent with swap thrashing under severe memory pressure. Once the working set no longer fits

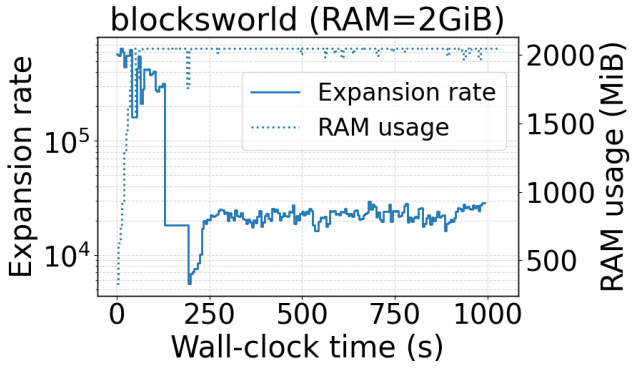

Figure 1: Expansion rate and RAM usage of Fast Downward A^* on blocksworld-p23 with the `blind` heuristic under a 2 GiB RAM limit. Expansion rate is based on nominal 5-second wall-clock sampling; during stall periods, values may reflect expansions accumulated over longer intervals.

in RAM, accesses to OPEN and CLOSED trigger frequent page eviction and reloading, so the search spends much of its time waiting for memory transfers rather than expanding nodes. Similar qualitative behavior was observed across other PDDL domains under RAM pressure; see Section 5.4 and (Suzuki and Fukunaga 2026).

Observation 1. This trivial approach is practical if the amount of RAM available is almost enough to solve the problem, but the orders of magnitude slowdown after RAM is exhausted leaves much room for improvement by more sophisticated IDD-based methods for problems that require much more memory than the RAM available.

4 SIDD-A^*

This section presents *Simple Immediate Duplicate Detection* A^* (SIDD-A^*), an external-memory implementation of A^* with lazy duplicate detection, which places OPEN and CLOSED on external storage. SIDD-A^* is intended as a minimal baseline for IDD-style external-memory heuristic search, and exposes the core interaction between best-first search and external storage through simple external-memory data structures.

A key design choice in SIDD-A^* is the use of *separate chaining* for CLOSED. As discussed in the preliminary section, prior work on IDD-based external-memory A^* has primarily focused on open addressing (Edelkamp, Schrödl, and Koenig 2010; Edelkamp et al. 2011; Lin and Fukunaga 2018). In contrast, SIDD-A^* keeps a compact RAM-resident head table and stores collided records as linked chains on external storage. The resulting implementation is simple and yields a different external-memory I/O pattern from open addressing-based designs, as discussed below in Section 4.5.

Algorithm 1 summarizes the main search loop and the CLOSED operation. Nodes are extracted from OPEN, and duplicate detection is performed only when a node is removed from OPEN. In Algorithm 1(b), CLOSEDFINDORINSERT returns **true** iff n is the first extracted node for its state

Algorithm 1: SIDD-A^*

(a) Main loop

$n_0 \leftarrow (s_0, 0, h_0, \perp)$
OPENINSERT(n_0, KEY(n_0))
while OPEN $\neq \emptyset$ **do**
  $n_u \leftarrow$ OPENEXTRACTMIN()
  if CLOSEDFINDORINSERT(n_u) **then**
    if ISGOAL($s(n_u)$) **then**
      return EXTRACTPLAN(n_u)
    for all (op, s') $\in$ SUCCESSORS($s(n_u)$) **do**
      $g' \leftarrow g(n_u) +$ COST(op)
      $h' \leftarrow h(s')$
      $n_v \leftarrow (s', g', h', n_u, \text{op})$
      OPENINSERT(n_v, KEY(n_v))
return failure

(b) CLOSEDFINDORINSERT

procedure CLOSEDFINDORINSERT(n)
  $b \leftarrow$ BUCKET(HASH($s(n)$))
  $i \leftarrow$ HEAD[b] ▷ RAM-resident bucket head
  while $i \neq$ NULL **do**
    $(x, i_{\text{next}}) \leftarrow$ READRECORD(i)
    if $s(x) = s(n)$ **then**
      if $g(n) < g(x)$ **then**
        UPDATEHEADER(i, n)
        OPENINSERT(n, KEY(n))
      return false
    $i \leftarrow i_{\text{next}}$
  $u \leftarrow$ APPENDNODE(n, HEAD[b])
  HEAD[b] $\leftarrow u$
  return true

to be inserted into CLOSED; in that case the node is expanded. If the state is already present, the procedure returns **false**; if n improves the stored g-value for that state, the procedure also updates the corresponding header and reinserts n into OPEN. Under nonnegative operator costs and a consistent heuristic, SIDD-A^* returns an optimal solution and expands the same states as standard A^* modulo tie-breaking and duplicate-handling timing. The main novelty is therefore in the external-memory realization of OPEN and especially CLOSED, rather than in the high-level control flow.

4.1 OPEN Implementation

OPEN is implemented as a simple external-memory two-level bucket queue. For a node n, we define KEY(n) = $(f(n), h(n))$, where $f(n) = g(n) + h(n)$. Nodes are ordered by increasing f, ties are broken by increasing h, and nodes with equal (f, h) are extracted in FIFO order.

Because duplicate detection is deferred until node extraction, OPEN stores complete search nodes rather than node identifiers. Each node record contains the packed state representation together with its parent pointer, generating operator, and g- and h-values.

For each (f, h) pair, SIDD-A^* maintains a separate bucket file on external storage. Insertion appends a node record to the file corresponding to its (f, h) value, and OPENEXTRACTMIN removes a node from the lexicographically smallest non-empty (f, h) bucket in FIFO order.

Because multiple OPEN entries for the same state may coexist, duplicate entries may remain in OPEN until extraction. Such entries are handled by CLOSEDFINDORINSERT.

Only lightweight metadata is kept in RAM, namely the set of non-empty f-layers and, within each layer, the set of non-empty h-buckets. In the implementation used here, bucket files are managed without user-level buffering.

4.2 CLOSED with External Separate Chaining

CLOSED supports three operations: exact duplicate detection, insertion of newly extracted states, and access to per-state metadata, including the parent pointer, generating operator, and the stored g- and h-values.

A key design choice of SIDD-A^* is to implement CLOSED as an external-memory hash table with separate chaining. Only the bucket-head table is kept in RAM, requiring just one 4-byte pointer per bucket; the collision chains themselves are stored externally. Each CLOSED record consists of an index part (hash value and next pointer), a fixed-size node header, and the packed state representation. The node header stores the parent pointer, generating operator, and the g- and h-values associated with the current best path recorded for that state.

Given a state s, SIDD-A^* computes a 32-bit hash value, maps it to a bucket in $\{0, \ldots, B-1\}$, where B is the number of buckets in the head table, and follows the linked chain starting from the RAM-resident head pointer. For each visited record, the packed state is read from external storage and compared exactly against s. If a matching state is found, the stored header provides the current metadata for that state. When the new path is cheaper, SIDD-A^* updates only the header of the existing record and reinserts the improved node into OPEN. If no match is found, a new record is appended to the external record file and linked at the head of the corresponding chain.

As in OPEN, the implementation used here does not employ user-level buffering for CLOSED. New states are appended to an external record file, so the dominant CLOSED write pattern is append-only, except for occasional header updates for improved duplicates. Separate chaining therefore keeps the bucket directory compact and RAM-resident, while making duplicate detection depend on chain traversal and external record reads.

4.3 SIDD-A^*_p and SIDD-A^*_m (`pwrite` vs. `mmap`)

Although the algorithmic behavior of SIDD-A^* is defined above, performance also depends on the I/O interface used to access the external OPEN and CLOSED files. These files can be accessed either via explicit file I/O using `pread`/`pwrite` or via a memory-mapped interface using `mmap`. We therefore consider two implementations: SIDD-A^*_p, which uses `pread`/`pwrite`, and SIDD-A^*_m, which uses `mmap`. Both implementations use the same search algorithm and data structures, differing only in the mechanism used to access external storage. We do not use hinting interfaces such as `posix_fadvise` or `madvise`. Experiments with `O_DIRECT` necessarily use `pread`/`pwrite`, since `mmap` operates through the page cache. Below, we use SIDD-A^* when the text applies to both SIDD-A^*_p and SIDD-A^*_m.

4.4 SIDD-A^*/OA: A Baseline Implementation with Open Addressing

To better understand the design tradeoffs in external-memory duplicate detection, we also implemented a baseline variant of SIDD-A^* in which the external CLOSED is realized using *open addressing* rather than separate chaining. This variant, SIDD-A^*/OA, uses the same search algorithm and external-memory OPEN as SIDD-A^*, and differs only in the representation of CLOSED.

In SIDD-A^*/OA, CLOSED is a fixed-size external hash table with M slots, no deletion, and no resizing during search. Collisions are resolved by linear probing. Each slot stores one complete CLOSED record: an index part (occupancy flag and 32-bit hash), a fixed-size node header, and the packed state representation. Lookup reads the index part of each probed slot and reads the full record only when the stored hash matches; insertion writes the complete record to the first empty slot in the probe sequence, while improved duplicates update only the header. The implementation uses no RAM-resident auxiliary index and no user-level buffering or batched writes. Additional implementation details and load-factor statistics are in (Suzuki and Fukunaga 2026).

4.5 Separate Chaining vs. Open Addressing

In our baseline implementations, the most important difference between separate chaining and open addressing is the write path for newly inserted CLOSED records. Both implementations perform file-backed reads during duplicate detection, so probing itself is not the main distinction in our comparison. With separate chaining, each new record is written by appending it to the end of the external record file, while the bucket-head table is updated only in RAM. This yields sequential full-record writes, except for occasional small random header updates for improved duplicates. With open addressing, by contrast, insertion writes the complete record directly into the first empty slot found along the probe sequence. Because the destination slot depends on the hash value and collision pattern, this results in random full-record writes. We therefore expect the dominant I/O disadvantage of straightforward open addressing to come primarily from record insertion, rather than from probing alone. Sequentially appending such records is generally favorable for file-system and storage-layer behavior, whereas random placement of full records is more likely to incur higher overhead, especially when memory pressure is high and the page cache cannot absorb most accesses.

4.6 SIDD-A^*-Based Classical Planner

We implemented a domain-independent planner based on SIDD-A^*. The implementation is derived from Fast Downward (FD)(Helmert 2006), replacing the standard A^* component with SIDD-A^*. Much of the FD code is reused, including successor generation, task representation, and heuristic functions. States use the same packed encoding and 32-bit hash computation as FD, ensuring identical state-to-key mappings. The differences lie in the implementation of the underlying data structures for OPEN and CLOSED.

5 Experimental Evaluation of SIDD-A^*

We evaluate and compare: (1) the standard, RAM-based implementation of A^* in Fast Downward, with virtual memory page-swapping enabled, (2) A*-IDD, (3) SIDD-A^*_p, (4) SIDD-A^*_m, (5) SIDD-A^*_p/OA, (6) SIDD-A^*_m/OA.

The A*-IDD implementation is based on the original implementation by Lin (2017). Although the original implementation uses LIFO tie-breaking, we modified A*-IDD to use FIFO tie-breaking to match the tie-breaking policies of A^* and SIDD-A^*. We verified that A*-IDD search performance with FIFO tie-breaking was very similar to search performance with LIFO tie-breaking (Suzuki and Fukunaga 2026).

Unless otherwise noted, both the RAM-resident head table of SIDD-A^* (separate chaining) and the internal table of A*-IDD were set to 512 MiB. For SIDD-A^*/OA, the external hash table size was fixed to 2^{27} slots, matching the bucket count of the separate chaining variant.

We verified that when executed to completion without RAM or runtime limits, all the algorithms expanded the same number of nodes on all benchmark instances used in this paper (Suzuki and Fukunaga 2026). Since all implementations expanded the same number of nodes in unrestricted runs, expansion rate is an appropriate implementation-level comparison metric for this study.

To reduce variability caused by file growth during search, SIDD-A^* preallocates its external files at the beginning of the first run using `fallocate`.

5.1 Comparison Under High Memory Pressure

We compared the algorithms on 8 instances each for `blind` and `merge-and-shrink` heuristics. All 6 algorithms were compared with a 1GiB RAM limit. This is a low RAM limit compared to the amount of storage required to solve these problem instances, i.e., memory pressure is high. Each run was executed with a 600 second time limit.

In addition, the two algorithms which performed best under 1GiB (SIDD-A^*_p and A*-IDD) were compared under even higher memory pressure (600MiB RAM limit).

To focus on the expansion rates when the memory pressure is highest, Table 1 shows the number of node expansions per second on the last 60 seconds of the run (the expansion rates over the entire run are in (Suzuki and Fukunaga 2026)). Runs that solved the instance are marked with a "$*$".

Observation 2. The straightforward open addressing baseline performs extremely poorly for IDD-based A^*.

The OA runs had low load factors at the 600-second cutoff (below 0.05 in all reported RAM-limited runs), so this poor performance is not simply due to near-full-table behavior.

Observation 3. Straightforward separate chaining (SIDD-A^*) performs much better, and is comparable to the state-of-the-art approach (A*-IDD) on these benchmarks.

5.2 On the Importance of the Page Cache

The importance of the page cache in the context of external-memory search was alluded to by Lin and Fukunaga (2018), but not investigated. In general, more RAM available for the page cache will result in better I/O latency. Thus, there is a tradeoff between the amount of RAM used by data structures for improving performance in IDD-based search vs. the amount of RAM left available for the page cache.

Evaluation of SIDD-A^* with no RAM limit We evaluated A^* and SIDD-A^* on 8 instances using the `blind` heuristic. Each run was executed without a RAM limit (i.e., full usage of the 32GB RAM), and all runs were executed until completion. All of these problem instances are solvable by A^* in the available RAM, so this is a low memory pressure scenario. Results with the `merge-and-shrink` heuristic are in (Suzuki and Fukunaga 2026).

Table 2 shows expansions/second and the RAM (including page cache) used by each method. These results show that with sufficient RAM for the page cache such that the entire search can fit in the page cache, SIDD-A^* can expand nodes at rates which are comparable to A^* (within a factor of 2 for SIDD-A^*_m, and within a factor of 8 for SIDD-A^*_p).

Observation 4. Under low memory pressure, the page cache significantly boosts IDD performance; if the RAM available is sufficient for the entire search, expansion rates can be within an order of magnitude of RAM-based A^*.

A*-IDD internal table size vs page cache In (Lin and Fukunaga 2018), the authors sidestepped the interaction between page cache and A*-IDD by using almost all of the available RAM for the internal hash table for compression (they state that they chose the internal table size "to limit any speedups from page caching while provisioning enough space for Fast Downward" (Sec. 5.3, par.3)).

Table 3 compares A*-IDD with an internal table size of 900MiB vs A*-IDD with an internal table size of 512MiB, both running under a 1GiB RAM limit. The version using the *smaller* internal table has a significantly higher expansion rate than the one with the larger table.

Observation 5. IDD algorithm data structures compete with the page cache for RAM. Using *less* RAM for the IDD-related data structures leaves more RAM for the page cache, which can sometimes *improve* performance.

5.3 Where are the overheads for external-memory access?

In this section, we try to quantify how much of the I/O overheads is due to the SSD device, and how much of the overhead is due to operating-system level overheads.

In order to measure system-related overheads independent of the external-memory device, we use *memory-backed filesystem*, where a portion of system RAM is made accessible using the file system interface. A `brd` (block ram disk) is a block-layer abstraction where a fixed size block of RAM is made available as a block device (e.g., /dev/ram0). Another memory-backed filesystem is `tmpfs`, a filesystem-layer abstraction which implements a dynamically sized, memory-backed filesystem. These memory-backed filesystems can be used instead of an SSD to store the external-memory data structure, and can be used as a proxy for an ideal SSD with no device-level overheads.

Problem	1.0GiB RAM limit						600MiB RAM limit	
	A^*	A*-IDD	SIDD-A^*_p Sep. Chaining	SIDD-A^*_m Sep. Chaining	SIDD-A^*_p Open Address.	SIDD-A^*_m Open Address.	A*-IDD	SIDD-A^*_p Sep. Chaining
Blind heuristic								
blocksworld-p23	29,181	**70,565**	29,771	19,454	9,946	2,383	**45,051**	28,546
data-network-p08	3,471	19,056	**30,667***	11,976*	4,480	309	**12,108**	11,553
depots-p23	6,675	16,789	**27,586**	16,429	3,424	1,000	6,565	**18,842**
floortile-p03	3,543	**44,457**	43,042	23,561	5,975	231	19,070	**27,832**
mprime-p05	8,238*	25,194	**49,111***	29,558*	4,716	1,513	11,746	**20,880**
rovers-p03	27,881	28,693	**44,050**	20,353	4,655	1,346	14,951	**32,717**
snake-p22	26,564	**48,140**	30,560	22,658	7,381	2,412	29,104	**30,837**
storage-p05	25,231	**63,121**	30,192	19,660	6,208	2,825	**31,781**	24,419
Merge-and-shrink heuristic								
agricola-p09	**72,137***	30,015	32,856	27,451	11,167	2,933	28,003	**41,019**
blocksworld-p10	36,069*	51,131*	**158,287***	101,682*	6,857	2,519	40,398*	**72,227***
data-network-p18	5,371	11,120	**28,937**	23,223	1,347	685	5,435	**21,536**
depots-p05	15,083	14,422	**29,042**	11,777	4,059	1,877	12,730	**24,327**
driverlog-p26	4,436	**33,607***	104,731*	89,403*	5,131	1,852	19,572	**48,477***
floortile-p07	3,855	**37,331**	25,357	21,510	4,926	1,256	17,295	**21,769**
hiking-p17	**66,122**	25,892	50,747	27,486	5,251	2,136	24,384	**34,711**
zenotravel-p08	8,119	17,797	58,568*	**63,958***	3,691	2,008	13,891	**31,548***

Table 1: Exp. rates (states/sec) for the last 60 seconds of search, under high memory pressure. A "*" indicates search completed.

Problem	A^*		SIDD-A^*_m		SIDD-A^*_p	
	Exp. rate	RAM	Exp. rate	RAM	Exp. rate	RAM
blocksworld-p23	613,812	**5,952**	**619,260**	8,842	275,866	8,829
data-network-p08	**209,699**	**4,347**	127,829	7,483	61,915	7,466
depots-p23	**410,784**	**3,214**	226,325	8,993	95,376	8,977
floortile-p03	**442,277**	**3,859**	386,156	7,646	168,778	7,627
mprime-p05	**331,365**	**3,077**	235,846	9,197	101,557	9,181
rovers-p03	**599,220**	**3,420**	228,980	6,289	83,819	6,273
snake-p22	**337,423**	**5,524**	319,380	6,687	235,724	6,671
storage-p05	**664,610**	**3,249**	512,314	7,346	217,441	7,329

Table 2: Expansion rates (states/sec) and peak RAM usage (cgroup, MiB) on 8 instances (blind heuristic) with no RAM limit (i.e., full usage of 32GB system RAM).

This allows us to compare 3 types of storage for the external data structures, each of which incurs different overheads: (1) `ssd`: device overhead + block driver overhead + filesystem overhead, (2) `brd`: block driver + filesystem , and (3) `tmpfs`: filesystem. By comparing the performance of SIDD-A^* using `ssd`, `brd`, and `tmpfs`, we can better understand the relative contribution of each of the 3 layers of overhead (device, block driver, filesystem) in this setup.

With Page Cache - No memory pressure Table 4 compares the node expansion rates of `ssd`, `brd`, and `tmpfs` in a low memory pressure scenario (RAM limit=32GB) using instances which can all be solved using less than 6GB of RAM by A^*. Although `tmpfs` has the highest expansion rate by a very small margin, `brd` and `ssd` have almost the same expansion rates. Despite the fact that `ssd`, `brd`, and `tmpfs` incur different levels of overhead, their performance is almost indistinguishable in this scenario.

Problem	A*-IDD (512MiB)	A*-IDD (900MiB)
Blind heuristic		
blocksworld-p23	78,945	45,957
data-network-p08	13,742	6,134
floortile-p03	24,173	10,129
Merge-and-shrink heuristic		
agricola-p09	20,764	20,018
data-network-p18	9,359	6,710
floortile-p07	34,403	13,483

Table 3: A*-IDD comparison of internal table sizes (900MiB vs 512MiB) with 1GiB RAM limit

Without Page Cache (`O_DIRECT`) Next, we configure the I/O to bypass the page cache and use Direct I/O by using the `O_DIRECT` flag when opening the files on the storage layer. Table 4 shows that without a page cache to mask the overheads, `tmpfs` is 2-3 times faster than `brd`, which in turn is almost 2 orders of magnitude faster than `ssd`.

Observation 6. The device, block driver, and file system all incur significant overheads. However, in low-memory pressure conditions, the page cache can hide the block layer and device overheads.

5.4 More Detailed View of Expansion Rate Behavior vs. Time

Figure 2 plots running average expansion rate vs. wall-clock runtime for blocksworld-p23 (blind heuristic) and floortile-p07 (merge-and-shrink heuristic) with 4GiB, 2GiB, and 1GiB RAM limits, to observe how the amount of system RAM available affects expansion rate behavior.

All runs were executed to completion (the end of each line

	A^* (blind)	SIDD-A^*_p **with** page cache			SIDD-A^*_p **without** page cache		
		`tmpfs`	`brd`	`ssd`	`tmpfs`	`brd`	`ssd`
barman-p02	880,527	220,950	206,096	246,549	162,533	69,627	1,433
blocksworld-p04	1,009,026	361,513	340,223	337,634	224,297	76,626	1,944
gripper-p06	1,296,619	279,892	262,435	263,088	171,587	68,875	1,429
pegsol-p23	1,254,337	423,892	462,881	460,173	307,257	129,459	2,767
termes-p01	998,678	298,692	321,677	331,494	216,510	88,257	1,950
Geom. mean (5)	1,076,117	309,298	307,290	319,720	210,788	84,066	1,847

Table 4: Comparison of backing storage (`tmpfs`, `brd`, `ssd`) with and without page cache (expansions/second)

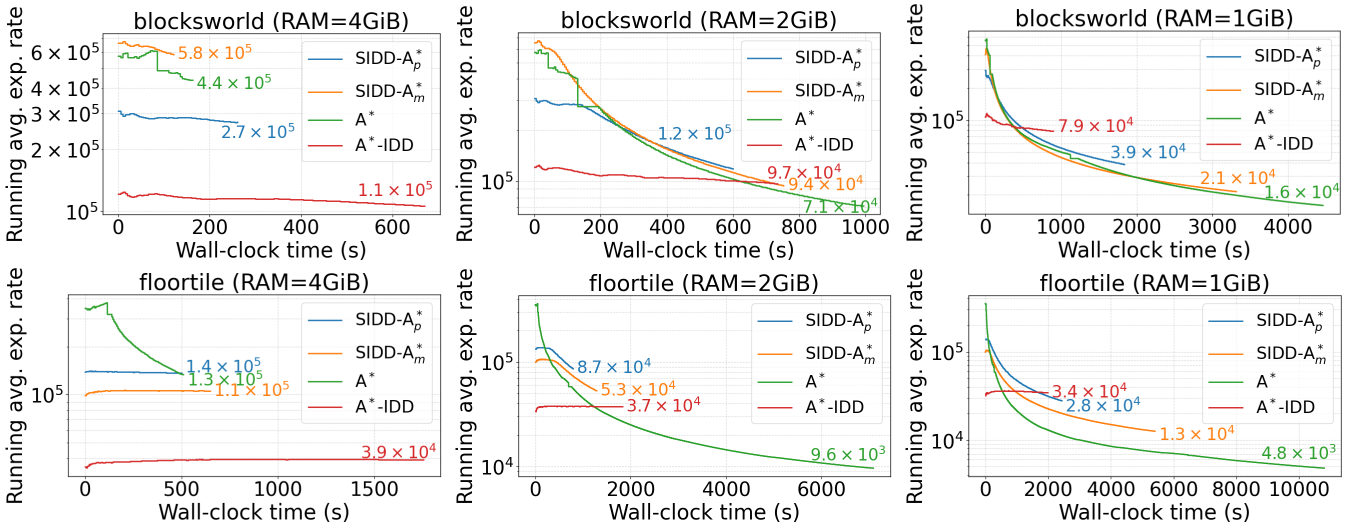

Figure 2: Running average expansion rate vs. wall-clock time for 4GiB, 2GiB, and 1GiB RAM limits on blocksworld-p23 (blind heuristic) and floortile-p07 (merge-and-shrink heuristic)

denotes the time when that algorithm completed its search). Running average expansion rate (total number of expansions up to time t divided by t) is shown instead of the instantaneous expansion rate at time t because the instantaneous expansion rates fluctuate and are difficult to see, especially with multiple plots per figure.

For all of the algorithms, the expansion rates decrease over time, and also as the RAM limit is decreased. A^* initially has a high expansion rate, but as RAM is depleted and exhausted, its expansion rate drops precipitously. SIDD-A^*_p and SIDD-A^*_m also have a relatively high initial expansion rate which drops over time. A*-IDD expansion rates are relatively stable compared to A^* and SIDD-A^*.

As the RAM limit is decreased, all algorithms exhibit degraded performance curves. A^* slows down as the RAM limit decreases because reducing RAM increases page swap thrashing. The other algorithms slow down because of increased page cache misses/eviction. Compared to SIDD-A^*_p, SIDD-A^*_m tends to degrade more both over time and also as the RAM limit is decreased.

Additional running-average, raw-rate, and total-expansion plots are in (Suzuki and Fukunaga 2026).

Observation 7. The performance of all evaluated algorithms depends on available RAM; even algorithms such as SIDD-A^* that use very little RAM explicitly run faster with additional RAM for the page cache.

Observation 8. `mmap`-based external data structure access can be more susceptible to degradation under RAM pressure than `pwrite`-based access.

6 Conclusion and Discussion

This paper experimentally evaluated baseline approaches for immediate duplicate detection in external-memory search. We proposed SIDD-A^*, which uses a simple CLOSED implemented with a straightforward separate chaining hash table, and showed that this is sufficient to achieve expansion rates comparable to the previous state-of-the-art algorithm, A*-IDD, without user-level caches or buffers. Building upon this simple baseline with more complex techniques to improve performance is a promising direction for future work.

We also investigated the performance impact of the OS page cache, and showed that in low memory pressure situations, a sufficient amount of page cache can significantly speed up IDD. Significant I/O overheads can be attributed not only to the SSD device, but also the OS block device driver and filesystem system call layers. Thus, approaches that bypass the OS layers in order to better control performance at the application level are a direction for future work.

References

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

Edelkamp, S.; Jabbar, S.; and Schrödl, S. 2004. External A*. *KI*, 4: 226–240.

Edelkamp, S.; Schrödl, S.; and Koenig, S. 2010. *Heuristic Search: Theory and Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 0123725127, 9780123725127.

Edelkamp, S.; Sulewski, D.; Barnat, J.; Brim, L.; and Šimeček, P. 2011. Flash memory efficient LTL model checking. *Science of Computer Programming*, 76(2): 136–157. Selected papers from the workshops on Formal Methods for Industrial Critical Systems.

Hatem, M.; Burns, E.; and Ruml, W. 2018. Solving Large Problems with Heuristic Search: General-Purpose Parallel External-Memory Search. *J. Artif. Intell. Res.*, 62: 233–268.

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

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *J. ACM*, 61(3): 16:1–16:63.

Korf, R. E. 2003. Delayed Duplicate Detection: Extended Abstract. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, 1539–1541. San Francisco, CA, USA.

Korf, R. E. 2004. Best-first Frontier Search with Delayed Duplicate Detection. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, 650–657. AAAI Press.

Korf, R. E. 2008. Linear-time disk-based implicit graph search. *J. ACM*, 55(6): 26:1–26:40.

Korf, R. E. 2016. Comparing Search Algorithms Using Sorting and Hashing on Disk and in Memory. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, 610–616. AAAI Press.

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

Lin S. 2017. Source code for A*-IDD. https://github.com/shunjilin/ExternalMemorySearchFastDownward.

Siag, L.; Shperberg, S. S.; Felner, A.; and Sturtevant, N. R. 2024. On Parallel External-Memory Bidirectional Search. In *ECAI 2024 - 27th European Conference on Artificial Intelligence*, volume 392 of *Frontiers in Artificial Intelligence and Applications*, 4190–4197. IOS Press.

Suzuki, Y.; and Fukunaga, A. 2026. Evaluation of Baseline Methods for IDD-based SSD External Memory Search (Extended Version). arXiv:2606.01840.

Torralba, A.; Seipp, J.; and Sievers, S. 2021. Automatic Instance Generation for Classical Planning. *Proceedings of the International Conference on Automated Planning and Scheduling*, 31(1): 376–384.