The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Revisiting Immediate Duplicate Detection in External Memory Search

Shunji Lin, Alex Fukunaga
Graduate School of Arts and Sciences
The University of Tokyo

Abstract

External memory search algorithms store the open and closed lists in secondary memory (e.g., hard disks) to augment limited internal memory. To minimize expensive random access in hard disks, these algorithms typically employ delayed duplicate detection (DDD), at the expense of processing more nodes than algorithms using immediate duplicate detection (IDD). Given the recent ubiquity of solid state drives (SSDs), we revisit the use of IDD in external memory search. We propose segmented compression, an improved IDD method that significantly reduces the number of false positive access into secondary memory. We show that A*-IDD, an external search variant of A* that uses segmented compression-based IDD, significantly improves upon previous open-addressing based IDD. We also show that A*-IDD can outperform DDD-based A* on some domains in domain-independent planning.

1 Introduction

Graph search algorithms, such as breadth first search, Dijkstra's algorithm, and A*, are constrained by the memory available to store the *open list* and the *closed list*. The open list acts as a priority queue for nodes to be expanded, while the closed list keeps track of previously expanded nodes in order to prevent the reexpansion of nodes via an operation known as *duplicate detection*. Without duplicate detection, there can be an exponential increase in the number of states visited by graph search, as the same states can be reached by many different paths.

In order to deal with large search problems that exhaust internal memory, *external memory search* algorithms have been developed using secondary memory such as hard disk drives (HDDs). The main hurdle in designing such algorithms is the need to overcome/avoid the latency of random read/write access into secondary memory, which is orders of magnitude slower than accessing RAM. Most previous external memory search algorithms were developed with the use of HDDs, using the method of *delayed duplicate detection* (DDD) (Korf 2003). In DDD, all generated nodes are first stored on disk and duplicate detection is performed using a separate duplicate elimination phase that removes duplicates in a single batch operation. This is in contrast to RAM-based graph search, where duplicate detection is performed on each node expansion or generation by *immediate duplicate detection* (IDD). However, in view of recent improvements in solid state drive (SSD) performances, capacities and cost per byte, it is natural to consider the use of SSDs as an alternative to HDDs in external memory search. The key difference between SSDs and HDDs is that SSDs allow much faster random access because they do not use a physical read/write head. Although there have been some past work on SSD-based external memory search (Edelkamp and Sulewski 2008), to our knowledge, DDD and IDD have not been compared empirically using SSDs, and the feasibility of IDD in SSD-based external memory search remains an open question (Edelkamp 2016).

We propose a method for IDD in external memory search, *segmented compression*, which improves upon the hash-based IDD method of *compression* (Edelkamp, Schrödl, and Koenig 2010) by adding a mapping structure that allows us to significantly reduce expensive lookups incurred by false positive probes into secondary memory. Using this method, we implement and evaluate A*-IDD, an external memory search variant of A*.

This paper is organized as follows. First, we review duplicate detection in external memory search. Then, we describe the new segmented compression strategy and A*-IDD. We evaluate A*-IDD on domain-independent planning using International Planning Competition (IPC) benchmarks, and first show that segmented compression significantly outperforms compression. Then, we compare A*-IDD to External A* (Edelkamp, Jabbar, and Schrödl 2004) and A*-DDD (Korf 2004; Hatem 2014), two DDD-based A* algorithms, and show that A*-IDD is competitive with DDD-based approaches in domain-independent planning.

2 Background: Duplicate Detection in External Memory Search

Secondary memory can be used in duplicate detection when the number of nodes expanded exceeds main memory capacity. However, a naive implementation (e.g., simply storing the closed list as a file on disk) is impractical, as each query of the closed list incurs a random I/O access into secondary memory, which is orders of magnitude slower than accessing an in-memory closed list. To circumvent this problem, several external memory search duplicate detection techniques have been proposed.

2.1 Delayed Duplicate Detection (DDD)

Delayed duplicate detection (DDD) (Korf 2003) defers duplicate checking so that it can be conducted sequentially on batches of nodes, instead of immediately on each generated node, thereby minimizing costly random access to disk. In general, DDD alternates between an *expansion phase* and a *duplicate elimination phase*. In each expansion phase, all nodes of the current *expansion layer* are expanded, and child nodes are appended to their respective files in secondary memory based on some partitioning of the search space. In the duplicate elimination phase, the removal of duplicate nodes through DDD is performed on files containing nodes in the next expansion layer. There are two main approaches to DDD: *sorting* and *hashing* (Korf 2004).

In sorting-based DDD, the file containing nodes in the next expansion phase is first sorted using external mergesort (time complexity of $\mathcal{O}(n \log n)$, where n is the # of nodes sorted) so that duplicate nodes are consecutive. Then in one pass, duplicate elimination is done both within nodes in the selected file and between nodes in the selected file and previously sorted files which contain closed list nodes of appropriate scope. Sorting-based duplicate detection has been used in complete breadth-first search (Korf 2003), as well as External A* (Edelkamp, Jabbar, and Schrödl 2004).

In hash-based DDD, two hash functions h_1 and h_2 are employed. In the expansion phase, h_1 is used to hash generated nodes to files according to their states, such that duplicate nodes reside in the same file. In the duplicate elimination phase, h_2 is used to hash all nodes from a file to entries of an in-memory hash table, such that duplicate entries are eliminated. h_1 has to be designed such that the largest file fits into main memory. Hash-based duplicate elimination has an amortized time complexity of $\mathcal{O}(n)$. Hash-based duplicate detection has been used in complete breadth-first search (Korf 2016), best-first frontier search (Korf 2004), as well as in A*-DDD (Korf 2004; Hatem 2014).

2.2 Issues With DDD-Based A*

In order to implement duplicate detection as an I/O-efficient, batch operation, DDD-based approaches make the following tradeoffs compared to standard, RAM-based A*:

Compatibility with Various Classes of Domains and Heuristics It is sometimes difficult to adapt DDD-based A* algorithms to new domains and heuristic functions because DDD variants often make strong assumptions about the search space. External A* works by expanding nodes in (f, g) expansion layers, and limits its duplicate detection scope to a select few (f, g) expansion layers. Although efficient for small-integer cost domains, the number of (f, g) expansion layers, and hence files, that have to be read during the duplicate detection phase is quadratic in the maximum edge cost of the problem graph (Edelkamp, Jabbar, and Schrödl 2004), rendering External A* inefficient for domains with a large range of edge costs. In addition, External A* does not allow for the reopening of nodes, and thus can only use consistent heuristics (in the case of optimal search). In DDD-based A*, nodes that are generated within the same expansion layer have to be *recursively expanded*. Thus, if there are duplicates within an expansion layer, both External A* and A*-DDD have to expand more nodes than necessary, and may even traverse infinite paths (e.g. in domains with zero-cost reversible actions) as duplicate detection is not performed within an expansion phase in DDD. In such cases, an in-memory *transposition table* can be used to cache and prune duplicate nodes within the expansion layer.

Preprocessing of Nodes in DDD A* progresses by expanding nodes in expansion layers of increasing f values. In DDD, all nodes in the final expansion layer must be processed for duplicate detection. As a result, in External A* and A*-DDD[1] all nodes in the final f layer — that is the set of expansion layers that have an f value equivalent to the optimal solution cost — are processed for duplicate detection, assuming no recursive expansions. In contrast, the number of nodes expanded in the final f layer by RAM-based A* varies significantly according to the tie-breaking rule (Asai and Fukunaga 2016). This penalty is sizeable in domains where duplicate nodes make up a significant proportion of generated nodes, and consequently of the final f layer.

Furthermore, in the duplicate elimination phase of A*-DDD, all nodes in the open list are processed whether or not they belong to the next f layer to be expanded, and this overhead is pronounced in domains that have many unique f values (Hatem 2014).

Tie-breaking Due to constraints imposed by DDD, DDD-based A* algorithms do not prioritize tie-breaking rules during the expansion phase. External A* breaks ties among minimum f value nodes by selecting nodes with the lowest g value, which is often a very poor tie-breaking policy (in general, nodes with higher g values tend to be closer to the goal). A*-DDD does not perform tie-breaking other than selecting nodes with the minimum f value for expansion[1]. The consequence of poor tie-breaking on the final f layer is that DDD-based A* typically expands and generates more nodes in the final f layer than RAM-based A*.

2.3 Immediate Duplicate Detection (IDD)

In contrast to DDD, immediate duplicate detection (IDD) does not perform duplicate processing as a separate, batch operation and instead performs duplicate detection as each node is processed. Unlike DDD, the search behavior of external memory search using IDD can mimic standard, RAM-based A*, and thus, IDD does not suffer the issues described above for DDD. Instead, IDD poses the problem of expensive random access to secondary memory.

Although IDD is impractical for external memory search using HDDs, the advent of SSDs has made IDD a feasible

[1]It is possible to implement A*-DDD with tie-breaking on g values such that not all nodes in the final f layer have to be processed in the duplicate elimination phase, but this comes at the expense of processing all open list nodes on each (f, g) expansion layer instead of each f expansion layer.

proposition, as SSDs provide up to 100x faster random access compared to HDDs, while providing similar sequential read/write performance.

Although SSDs are significantly faster than HDDs, they are still orders of magnitude slower than RAM, so a naive IDD method which simply maps the closed list to the SSD is impractical. In addition, care is required when writing to SSDs. While reads and writes can be done at the page ($2\,\mathrm{KiB}$–$16\,\mathrm{KiB}$) level, overwrites are performed at the block (128–256 pages) level. Excessive random writes thus cause internal fragmentation, and lead to a phenomenon known as *write-amplification* (Hu et al. 2009), whereby the physical size of writes is much larger than the logical size of writes requested by the client. Write-amplification is undesirable as it reduces write performance and the life of flash cells.

Edelkamp, Schrödl, and Koenig (2010, Chapter 8) propose three IDD methods for external memory search using SSDs — *mapping*, *compression* and *flushing* — that use open address hashing for *external hash tables* located on SSDs. These methods use in-memory write buffers and/or hash tables to avoid unnecessary access to the external hash tables. The main issue with these methods is inherent in open addressing — as the load factor of the hash table grows, the expected number of probes into the hash table for each query operation increases due to collisions, and each probe into the external hash table incurs the cost of an expensive I/O access into the SSD. We focus on compression rather than flushing and mapping, because of the three methods, compression is the only method that mitigates write-amplification by ensuring that nodes are written to the external hash table in a buffered and contiguous manner. To our knowledge, there is no published empirical comparisons of these three methods, and a deeper investigation of flushing and mapping is an avenue for future work.

In compression, an in-memory hash table (*internal hash table*) stores pointers to nodes in the external hash table, which can be thought of as a contiguous array stored on the SSD. When nodes are added to the closed list, they are first inserted into an in-memory buffer. When the buffer is full, its nodes are written sequentially (appended) to the external hash table, and pointers to the nodes are inserted into the internal hash table according to an appropriate hash function on the nodes' states. Checking whether a node n with state s is a duplicate involves the following: first the in-memory buffer is checked, and if a node with state s is not found, the internal hash table is probed (according to an appropriate open addressing scheme). If the probe into the internal hash table returns an invalid pointer (empty slot), we come to the end of our probe sequence and we know that a duplicate node does not exist in the closed list. In this case we avoid a *false positive* probe into the external hash table. On the other hand, if the probe into the internal hash table returns a valid pointer, we access the external hash table via the pointer. If a node with state s is found at the pointer's location, we found a duplicate node and the query terminates. Otherwise, the procedure proceeds to alternate between probing the internal and external hash tables until either the internal hash table returns an invalid pointer (duplicate not found) or a node with state s is found in the external hash table (duplicate found). In case of an unsuccessful query (duplicate not found), the internal hash table eliminates one false positive probe into the external hash table, as compared to using a single hash table on SSD.

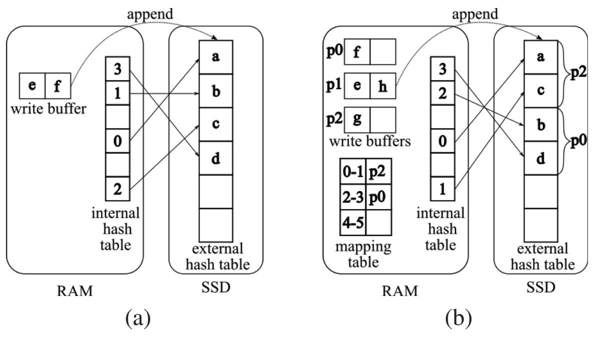

Figure 1: (a) Compression and (b) segmented compression. The alphabets represent nodes, whose ordering reflect a possible sequence of insertions. When a write buffer becomes full, its nodes are written sequentially (appended) to the external hash table, while their corresponding pointers are hashed into the in-memory internal hash table. Additionally, in segmented compression, the mapping table is updated to reflect the partition value of the appended write segment.

If the external hash table has a size of m, a single pointer requires $\lceil \lg m \rceil$ bits, and the internal hash table requires $m\lceil \lg m \rceil$ bits in total. The *compression ratio* reflects the degree of space saving achieved by storing pointers in memory instead of nodes, and is given by the size of a node divided by the size of a pointer; the greater the size of the node, the greater the magnitude of compression.

As an additional optimization, Edelkamp, Schrödl, and Koenig (2010, Chapter 8) propose the use of cuckoo hashing (Pagh and Rodler 2004) for the internal hash table to limit the maximum number of probes into the external hash table to 2 for any query operation. While effective in reducing the cost of queries, this introduces new issues. Firstly, the load factor has to be kept low ($< \frac{1}{2}$) in order to prevent frequent rebuilding of the internal hash table due to cycles (Pagh and Rodler 2004). Secondly, an insertion may require a significant number of probes into the external hash table through the eviction process, in order to compute new hash values for evicted nodes. In contrast, unoptimized compression requires no probes into the external hash table during insertion.

3 Segmented Compression: Reducing False Positive Probes in IDD

We introduce a method for improving SSD-based duplicate detection by further limiting the number of false positive probes into the external hash table per query operation. This idea is inspired by the mapping tables used in an SSD's *flash translation layer*, which allows for space-efficient logical-to-physical address mapping (Chung et al. 2009).

We exploit the fact that in compression, each write to the external hash table is done sequentially and has a fixed size (the size of the write buffer). By using multiple write buffers, we can partition writes to the external hash table into *write segments*. We introduce a *partition function* which takes as its input a node, and outputs a *partition value*. The partition values are defined over a fixed, discrete interval corresponding to the number of write buffers (i.e. the number of partitions). Every time a node is inserted into the closed list, it is first inserted into its corresponding write buffer, according to its partition value. By maintaining an in-memory *mapping table* which maps write segments to partition values, we can keep track of the partition value of a node in the external hash table by using its pointer as an index into the mapping table. When a write buffer becomes full, its nodes are appended into the external hash table as a write segment, and the mapping table is updated with the partition value corresponding to the write buffer. The mapping table then acts as an additional filter for false positive probes into the external hash table. If a probe into the internal hash table returns a valid pointer, we check the partition value returned by the mapping table on the pointer's write segment. If this value corresponds to the node of interest's partition value, we proceed to probe the external hash table. Otherwise, we avoid a false positive probe into the external hash table and we continue with the probe sequence. It can be seen that for segmented compression to work, for any two nodes with the same state, the nodes must have the same partition values, such that false negative probes do not occur.

We can evaluate the reduction in false positive probes by segmented compression compared to compression as follows. Let the partition function be a uniformly distributed random variable with its domain the set of all states, and its codomain the interval $[1, p]$, where p is the number of partitions. Furthermore, assume that the partition function and the hash function are independent. In compression, a false positive probe is caused by two nodes with different states hashing to the same slot in the internal hash table. In compression with partitioning, a false positive probe results from the subsequent collision of the above two nodes on their partition values. The probability of collision on partition values for two randomly chosen nodes that have different states is $\frac{1}{p}$, due to uniformity. Hence we see that with segmented compression, the number of false positive probes into the external hash table can be reduced by a factor of p.

The maximum additional memory cost of segmented compression compared to compression is equivalent to the sum of the maximum total size of the write buffers and the maximum size of the mapping table.

4 A*-IDD

We propose A*-IDD, an external memory search algorithm which uses segmented compression to perform IDD. Algorithm 1 shows the pseudocode for A*-IDD. The overall structure of the A*-IDD pseudocode is almost identical to that of RAM-based A*, except that checking whether a node is in the closed list and inserting nodes into the closed list is performed by the *find-insert* routine, which implements external memory IDD using segmented compression (Algorithm 2, described more fully below). The key features of A*-IDD are:

Lazy Duplicate Detection Duplicate detection in A* can be conducted *lazily* or *eagerly*. In lazy duplicate detection, nodes are checked for duplicates before they are expanded, against nodes in the closed list. In eager duplicate detection, nodes are checked for duplicates as they are generated, against nodes in both the open and closed list in order to ensure completeness. Although memory efficient, this requires more duplicate checks, as the number of generated nodes is usually much larger than the number of expanded nodes in A*. We implement lazy duplicate detection in A*-IDD because: (1) duplicate detection is expensive given that the SSD has be to accessed; (2) as the open list is a priority queue backed by files on SSD, it is not efficient to access nodes in the open list by their state representations. Note that even though certain domains perform better with lazy duplicate detection (Burns et al. 2012), in the case where the memory savings brought about by eager duplicate detection allows for RAM-based A*, eager duplicate detection should be used as the latency of accessing secondary memory is certain to outweigh any gains from lazy duplicate detection.

Open List We arrange nodes in the open list into *buckets* represented by files on the SSD. Each bucket *Open*(i, j) holds all open list nodes u with $f(u) = i$ and $g(u) = j$, similar to External A* (Edelkamp, Jabbar, and Schrödl 2004). Unlike in External A* however, this allows us to select nodes with the highest g value, among nodes with the lowest f value. This is a tie-breaking rule that allows for a lower number of expansions in the final f layer. We also insert and remove nodes from the buckets in last-in-first-out order, which is not only cache-friendly but also an effective deterministic tie-breaking rule for A* (Asai and Fukunaga 2016).

Closed List The closed list uses segmented compression to handle duplicate detection, and supports two operations: *find-insert* and *construct-path*.

Find-insert takes a node u, and returns a boolean value indicating whether u should be expanded. It also inserts u into the closed list if a duplicate node with lower f value does not already exist. To handle inconsistent heuristic functions, find-insert updates an already expanded node v with u if v has a higher g value than u. In this case find-insert returns true so that reexpansion can occur.

Construct-path reconstructs the solution path when the goal node is found. It does so by recursively searching the hash tables for each node's parent node, from the goal node to the initial node. In order to do this efficiently, we store in each node either the parent node's state representation or the parent node's hash value (whichever uses less space).

To minimize collisions on the external hash table, we need a good open addressing scheme as well as a good hash function. We use double hashing, which avoids collisions due to clustering. We use *simple tabulation hashing* (Zobrist 1970) for the hash function as it is efficient, appropriate for hashing state representations and provides probabilistic guarantees similar to fully random hashing (Patrascu and Thorup 2010). Simple tabulation hashing works by initializing and storing a table of random bitstrings corresponding to each

Algorithm 1 A*-IDD

```
 1: procedure A*-IDD($s, e$)                    ▷ start state $s$, goal state $e$
 2:     Closed ← $\emptyset$                               ▷ closed list
 3:     Open ← $n_s$                      ▷ open list, initial node $n_s$
 4:     while Open $\neq \emptyset$ do
 5:         Remove from Open node $u$ with lowest $f$ value, tie-
            breaking on highest $g$ value
 6:         if State($u$) $= e$ then
 7:             return Construct-Path($u$)
 8:         if Find-Insert($u$) returns true then
 9:             Successors($u$) ← Expand($u$)
10:             for $v$ in Successors($u$) do
11:                 Insert $v$ into Open
12:     return $\emptyset$
```

Algorithm 2 Find-Insert

```
 1: procedure FIND-INSERT($u$)                               ▷ node $u$
    ▷ p_value is the partition value
 2:     if $u$ in write_buffers[$u$.p_value] then
 3:         if cheaper path to $u$ found then
 4:             update $u$ in write buffer
 5:             return true
 6:         else
 7:             return false
 8:     loop
 9:         pointer ← Probe(internal hash table)
10:         if pointer is invalid then                      ▷ empty slot
11:             return false
12:         else
13:             if $u$.p_value = mapping_table[pointer] then
14:                 if $u$ = Probe(external hash table) then
15:                     if cheaper path to $u$ found then
16:                         update $u$ in external hash table
17:                         return true
18:                     else
19:                         return false
```

value of each state variable. To obtain the hash value of a state, we XOR all the bitstrings corresponding to the state. For double hashing, we assign our hash table size m to be prime, and our initial probe value to be $h_1(k) = k \bmod m$, where k is the simple tabulation hash value of the node, and $\bmod$ is the modulo operator. Our secondary probe value is $h_2(k) = 1 + (k \bmod (m-1))$. This ensures that $h_2(k)$ is relatively prime to m and our probe sequence never cycles (Cormen et al. 2001, Chapter 11).

In order to ensure that our partition function approximates a uniformly distributed random variable that is independent of our open address hash function, we use a separately initialized simple tabulation hash function for our partition function. The partition value of the node is the simple tabulation hash value of the node modulo p, where p is the number of partitions.

5 Experiments

We compared compression to segmented compression by evaluating A*-IDD on a set of IPC optimal-track benchmark instances. We also compare A*-IDD with two sequential DDD-based A* variants (External A* and A*-DDD) on the 15-puzzle domain, as well as on a subset of the IPC optimal-track benchmark instances. We used a Xeon W3680 $3.3\,\mathrm{GHz}$ CPU, with a RAM limit of $1.33\,\mathrm{GiB}$ (which gives an available memory of $1.16\,\mathrm{GiB}$ when excluding background processes), running Ubuntu $16.04.3$ LTS. We used a $900\,\mathrm{GB}$ SATA 3.0 SanDisk Ultra II consumer-grade TLC SSD. All programs were implemented in C++ (g++ $5.4.0$, C++11).

For all open list I/O operations, we used C++'s file stream class, fstream, with buffers of size $16\,\mathrm{KiB}$ each. The actual writing of nodes to the files on SSD is handled by the kernel. For A*-IDD's external hash table, we used the POSIX mmap system call, which maps files on SSD to virtual memory. We supplied the madvise function with the MADV_RANDOM advice, which instructs the kernel not to prefetch more pages than necessary in each mmap page fault. We used a bit vector for A*-IDD's internal hash table in order to pack arbitrary-sized pointers into a contiguous array. Given a memory limit for the size of the internal hash table, the size of a pointer is dynamically chosen so as to maximize the capacity of the external hash table. We used the probabilistic Miller-Rabin primality test to initialize prime-sized hash tables for double hashing. The simple tabulation hash functions are initialized by the Mersenne Twister pseudorandom number generator.

5.1 Evaluating Segmented Compression

In order to compare compression and segmented compression, we implemented A*-IDD on top of *Fast Downward* (Helmert 2006), a heuristic search-based domain-independent, classical planner. We used the consistent, abstraction-based *merge-and-shrink* heuristic (Helmert et al. 2014) with the recommended parameters[2]. We selected candidate problems from the IPC optimal track domains (2008, 2011 and 2014), including only problems that exhaust $2\,\mathrm{GiB}$ to $10\,\mathrm{GiB}$ in RAM-based A* with eager duplicate detection. We excluded duplicate problems as well as problems that use more than $1\,\mathrm{GiB}$ during the merge-and-shrink initialization phase, resulting in a set of 34 instances. We ran compression and segmented compression with $p = [10, 100, 1000]$ on the set of problems, ordered by their memory consumption in RAM-based A*. We used an internal hash table of size $500\,\mathrm{MiB}$ for each run. The additional maximum cost incurred by segmented compression can be calculated to be less than $18.3\,\mathrm{MiB}$ (for $p = 1000$, assuming nodes of size $64\,\mathrm{bytes}$ each). With a $48\,\mathrm{h}$ time limit, compression solved 27, segmented compression with $p = 10$ solved 31, and segmented compression with $p = 100$ and $p = 1000$ solved 33 out of the 34 problems.

For the easier instances, the difference in search times between compression and segmented compression were marginal, as the entire closed list could fit in RAM and thus be cached. For the harder instances, segmented com-

[2](merge_and_shrink(shrink_strategy=shrink_bisimulation(greedy=false),merge_strategy=merge_sccs(order_of_sccs=topological,merge_selector=score_based_filtering(scoring_functions=[goal_relevance,dfp,total_order])),label_reduction=exact(before_shrinking=true,before_merging=false),max_states=50000,threshold_before_merge=1))

	compression			segmented compression ($p = 10$)				segmented compression ($p = 100$)				segmented compression ($p = 1000$)			
domain/problem	search time (s)	FP probes	$\frac{\text{FP probes}}{\text{total probes}}$	search time (s)	FP probes	$\frac{\text{FP probes}}{\text{total probes}}$	FOR in FP probes	search time (s)	FP probes	$\frac{\text{FP probes}}{\text{total probes}}$	FOR in FP probes	search time (s)	FP probes	$\frac{\text{FP probes}}{\text{total probes}}$	FOR in FP probes
citycar-opt14/p2-2-6-1-1	2663	295523	0.0167	2674	25778	0.0015	11.5	2586	2412	0.0001	122.5	2689	140	0.0000	2110.9
elevators-opt11/p19	2679	5644802	0.0347	2695	560052	0.0036	10.1	2668	54044	0.0003	104.4	2829	4809	0.0000	1173.8
sokoban-opt08/p27	2777	8982443	0.1282	2111	873913	0.0141	10.3	2019	87329	0.0014	102.9	2085	8506	0.0001	1056.0
citycar-opt14/p2-2-5-2-2	4215	1638425	0.0348	4304	150317	0.0033	10.9	4280	14318	0.0003	114.4	4385	1218	0.0000	1345.2
hiking-opt14/ptesting-2-3-5	6785	33021199	0.0684	6105	3239036	0.0072	10.2	5938	311231	0.0007	106.1	6016	29193	0.0001	1131.1
hiking-opt14/ptesting-2-2-7	9419	40879262	0.0943	7569	3972964	0.0100	10.3	7251	380885	0.0010	107.3	7493	32893	0.0001	1242.8
sokoban-opt08/p25	9852	46243553	0.2522	4748	4518384	0.0319	10.2	4100	449323	0.0033	102.9	4250	44549	0.0003	1038.0
transport-opt14/p08	11934	93749442	0.2282	4246	9214471	0.0282	10.2	3487	852041	0.0027	110.0	3506	74981	0.0002	1250.3
barman-opt14/p435-3	44886	276411337	0.4351	18532	27209675	0.0705	10.2	15936	2688840	0.0074	102.8	16451	256806	0.0007	1076.3
barman-opt11/pfile02-005	54440	372339695	0.4834	20466	36585139	0.0842	10.2	17373	3582056	0.0089	103.9	17783	327519	0.0008	1136.8

Table 1: Results of the 10 most difficult instances from the set of IPC benchmark instances solvable by both A*-IDD with compression and with segmented compression ($p = [10, 100, 1000]$). *FP probes* refer to false positive probes into the external hash table. *FOR in FP probes* refers to the factor of reduction in false positive probes relative to compression. *Total probes* refer to the sum of write buffer hits, true positive probes and false positive probes.

pression showed significant speedups compared to compression. Table 1 shows the search times for the most difficult instances solvable by all algorithms. The most difficult instance solved by both compression and segmented compression (barman-opt11/pfile02-005) took $15.1\,\mathrm{h}$, $5.3\,\mathrm{h}$ ($p = 10$), $4.8\,\mathrm{h}$ ($p = 100$) and $4.9\,\mathrm{h}$ ($p = 1000$) respectively, and had an external hash table load factor of 0.75 ($4.7\,\mathrm{GiB}$) at the end of search. The differences in search times between compression and segmented compression can be attributed to the reduction in false positive probes[3] into the external hash table by the mapping table. The minimum and median values for the factor of reduction in false positive probes by segmented compression are: $(10.0, 10.3)$, $(100.0, 107.3)$ and $(988.0, 1173.8)$ for $p = 10$, 100 and 1000 respectively. The results are consistent with the predicted reduction factor for p. For the set of problems, a partition number of $p = 100$ was able to reduce the number of false positive probes from a significant to an insignificant proportion ($< 3\,\%$) of total probes (see Table 1), effectively suppressing wasted I/O access into the SSD. For $p = 1000$, the additional reduction in in false positive probes was not worth the overhead of managing additional buffers, as the average search time increased (compared to $p = 100$).

5.2 Comparing A*-IDD and DDD-based A* on the 15-puzzle domain

We compared A*-IDD to two sequential DDD-based A* algorithms, External A* (Edelkamp, Jabbar, and Schrödl 2004) (sorting-based) and A*-DDD (Korf 2004; Hatem 2014) (hash-based) on the 15-puzzle domain with the consistent, Manhattan distance heuristic. The 15-puzzle is a unit cost domain and hence compatible with External A*. We evaluated both algorithms on the set of 100 random 15-puzzle instances from Korf (1985). The largest instance in this set required $27\,\mathrm{GB}$ to solve in a highly optimized implementation of RAM-based A* using lazy duplicate detection (Burns et al. 2012), which is sufficiently large for our RAM limit. For A*-IDD, we used segmented compression ($p = 100$), with $950\,\mathrm{MiB}$ for the internal hash table, in order to achieve an external hash table size that is big enough to fit the closed list of the largest problem. For the external mergesort operation in External A*, we used a chunk size of $950\,\mathrm{MiB}$, and we perform a k-way mergesort, where k is simply the number of chunks read in that iteration. This is as random seeks are efficient in SSDs, and the dynamic k value helps to reduce the number of merge pass runs to just one. For A*-DDD, we used simple tabulation hashing for assigning nodes to files, with a hash value range of $[1, 20]$. Each hash value is backed by three files — *Open*, *Next* and *Closed* — for nodes in the current expansion layer, next expansion layer and closed list respectively.

To solve all 100 instances, A*-IDD took $17.6\,\mathrm{h}$, expanded 1.6×10^9 nodes and generated 3.1×10^9 nodes; External A* took $3.8\,\mathrm{h}$, expanded 5.7×10^9 nodes and generated 1.1×10^{10} nodes; A*-DDD took $3.0\,\mathrm{h}$, expanded 3.3×10^9 nodes and generated 6.4×10^9 nodes.

The results show that A*-IDD performs poorly compared to DDD-based A* in the 15-puzzle domain. The likely explanation is that in the 15-puzzle domain, not only is node generation inexpensive, the reduction in nodes processed in the final f layer by tie-breaking in A*-IDD is not significant enough as there are relatively few duplicates compared to other domains (Burns et al. 2012). Thus the overhead incurred by DDD-based A* in the final f layer is not substantial compared to the expensive random access probes into SSD by A*-IDD. Nevertheless, the results on the 15-puzzle domain give further evidence on segmented compression's ability to suppress false positive probes even on load factors that would cripple standard open addressing methods — the maximum fraction of false positive probes to total probes for the instances is 0.16 for a load factor of 0.97 (closed list of size $4.82\,\mathrm{GiB}$) at the end of search.

5.3 Comparing A*-IDD and DDD-based A* on Domain-independent Planning

Next we compared A*-IDD to External A* and A*-DDD on domain-independent planning. This is as in domain-independent planning, the cost of expanding a node is expensive compared to in the 15-puzzle domain, and certain problems may contain a significant number of duplicates (e.g. in grid pathfinding).

We first evaluated External A*, A*-DDD and A*-IDD on only the subset of unit cost instances from the IPC benchmark instances used in Section 5.1. This is as External A* does not generalize well to non-unit cost domains (see Section 2.2). We then evaluated A*-DDD and A*-IDD on the

[3]Probe statistics are upper bounds as the kernel's page cache provides an additional layer of buffering given enough memory.

	External A*			A*-DDD			A*-IDD			A*-IDD with poor tie-breaking		
domain/problem	search time (s)	expanded	generated	search time (s)	expanded	generated	search time (s)	expanded	generated	search time (s)	expanded	generated
nomystery-opt11/p19	1902	61675963	459918590	1999	47585468	377746156	2123	22228851	168660014	3106	60813817	453369620
nomystery-opt11/p09	6770	86313769	660087182	2674	59116115	480645587	2929	27018183	207393051	5339	85296578	652244754
hiking-opt14/ptesting-2-3-5	43164	34173715	807807459	6756	52051539	1459011981	6552	26276862	624733774	6916	34173715	807807459
hiking-opt14/ptesting-2-2-7				9665	80503020	1984917277	7784	41198523	890211751	12134	74368649	1517350852
barman-opt14/p435-3				23562	282986075	1362756736	17750	105511695	481776292	11439	108191657	494280872
hiking-opt14/ptesting-2-4-5				38673	232898180	7915465779	36262	90966582	2699027855	44646	119116405	3528672811

Table 2: Results of External A*, A*-DDD and A*-IDD on the set of unit cost IPC benchmark instances. Includes A*-IDD with poor tie-breaking (i.e. select nodes with lowest g value among minimum f value nodes).

	A*-DDD			A*-IDD			A*-IDD with poor tie-breaking		
domain/problem	search time (s)	expanded	generated	search time (s)	expanded	generated	search time (s)	expanded	generated
floortile-opt11/opt-p02-003	1243	24289484	169984014	1303	12270544	84061147	852	15592326	105676348
transport-opt14/p08	14309	62337371	425026742	5354	58275211	392447045	5038	58364000	393073722
floortile-opt11/opt-p03-005	5937	109485715	810638870	5852	45144345	331711820	4178	65572481	476920099
barman-opt11/pfile02-006	29519	300049236	1440411576	18165	114433246	521645539	13342	115163358	525004590
barman-opt11/pfile02-005	29648	299602752	1438362774	18373	114433260	521645580	12163	115163358	525004590
barman-opt11/pfile02-008				19530	114668287	522755574	13278	115372409	525980278

Table 3: Results of A*-DDD and A*-IDD on the set of non-unit cost IPC benchmark instances (without zero cost actions). Includes A*-IDD with poor tie-breaking (i.e. select nodes with lowest g value among minimum f value nodes).

subset of non-unit cost instances, excluding instances with zero-cost actions, as these can lead to A*-IDD traversing infinite paths during the expansion phase (see Section 2.2). This gave us two sets of six instances, which we ordered by difficulty in terms of A*-IDD's search times. Additionally, on both evaluations, we included A*-IDD with poor tie-breaking, a modified version of A*-IDD that breaks ties among minimum f value nodes by selecting nodes with the lowest g value, in the manner of External A*.

We use the merge-and-shrink heuristic with the same parameters as in Section 5.1, and the same implementations of the search algorithms as in Section 5.2, with minor modifications. For A*-IDD, we used segmented compression ($p = 100$), with $900\,\mathrm{MiB}$ for the internal hash table, to limit any speedups from page caching while provisioning enough space for Fast Downward. For External A*, we used a chunk size of $900\,\mathrm{MiB}$. For A*-DDD, we used a hash value range of $[1, 20]$. Preliminary tests showed that A*-DDD was unable to solve the easiest of instances in a reasonable amount of time, as the number of duplicates due to recursive expansions is significant in many IPC benchmark instances. Hence we equipped A*-DDD with a $900\,\mathrm{MiB}$ transposition table (deallocated during the duplicate elimination phase) that used an always-evict policy. We set a $24\,\mathrm{h}$ time limit for each algorithm for both the unit cost and non-unit cost evaluations.

On the unit cost test set, External A* solved 3 instances, A*-DDD solved 5 instances while both versions of A*-IDD solved all 6 instances. For the non-unit cost test set, A*-DDD solved 5 instances while both versions of A*-IDD solved all 6 instances. The results are shown in Table 2 and Table 3 respectively.

For both the unit cost and non-unit cost evaluations, A*-IDD outperformed A*-DDD in terms of search times on the harder instances, as the difference between the number of expanded and generated nodes by A*-DDD and A*-IDD grew. This is in spite of the fact that A*-DDD had a higher expansion and generation rate for most of the instances (the exception being transport-opt14/p08, an instance that required exploring a large range of unique f values). Furthermore, except for the two easiest instances in the unit cost test set, A*-DDD expanded and generated more nodes than A*-IDD with poor tie-breaking, showing that recursive expansions can amount to a significant overhead in A*-DDD[4]. This overhead is likely to grow in proportion with the search tree — i.e. exponentially — as the factor of reduction in duplicates by the memory-limited transposition table is expected to diminish as the instances become harder.

On the unit cost test set, External A* performs poorly, as it pays both the price of ineffective tie-breaking and the $\mathcal{O}(n \log n)$ factor for preprocessing nodes in the duplicate elimination phase.

Interestingly, while A*-IDD with good (maximum g-value based) tie-breaking expanded and generated fewer nodes than A*-IDD with poor (minimum g-value based) tie-breaking in all of the instances, A*-IDD with poor tie-breaking had consistently higher node expansion and generation rates, which resulted in better search times in many of the instances. This suggests that the impact of tie-breaking strategies in A*-IDD is not as straightforward as in RAM-based A*, as a tie-breaking strategy which is "better" with respect to search effiency can result in poorer locality with regards to the page cache.

6 Discussion and Conclusion

We proposed and evaluated segmented compression, an improved approach to open addressing-based immediate duplicate detection in external memory search. Our evaluation of segmented compression showed that it is is possible to reduce the number of expensive, false positive probes into secondary memory by two orders of magnitude compared to compression, with minimal memory overhead. On the do-

[4]We also evaluated a version of A*-DDD that breaks ties among minimum f value nodes by selecting nodes with the highest g value. This version of A*-DDD performed poorly, solving only 1 and 0 problems from the unit cost and non-unit cost test sets respectively. In this case, the overhead of preprocessing every open list node on each (f, g) expansion layer outweighed the gains from better tie-breaking and the elimination of recursive expansions.

mains we evaluated, false positive probes into the external hash table are no longer a significant overhead, resulting in signficantly improved runtime performance of IDD.

We also showed that in domain-independent planning, A*-IDD, which performs IDD using segmented compression, can significantly outperform DDD-based external A* on some domains when an SSD is used as secondary memory. Given the increasing ubiquity of SSDs and the rapidly diminishing gap in the cost per byte of storage in SSDs vs. HDDs, our results suggest that although delayed duplicate detection-based approaches have been the primary focus of research in external memory search, immediate duplicate detection is a promising approach for external memory search in some domains, particularly when the overheads associated with DDD are significant. Our results with the 15-puzzle show that in some domains, DDD-based approaches continue to dominate IDD-based approaches. It should be noted that in the domains used for our experiments, the node sizes ($\leq 64\,\mathrm{bytes}$) are significantly smaller than the smallest SSD read unit of a page ($2\,\mathrm{KiB}$–$16\,\mathrm{KiB}$), resulting in high constant overheads associated with paging in unnecessary nodes in each probe into the external hash table. In applications with larger node sizes, this overhead is reduced, and should make IDD-based approaches more even more attractive. A deeper understanding of these tradeoffs is an interesting direction for future work.

A*-IDD tends to be simpler and more convenient to implement than previous DDD-based approaches because it requires fewer assumptions about the domain to which it is applied (e.g., consistent vs. inconsistent heuristic, unit cost vs. non-unit cost edges) in order to provide performant behavior. This is because A*-IDD is simply standard A*, except that the low-level management of the closed list in external memory is handled by segmented compression. Unlike DDD, IDD allows the search strategy to be mostly separated from the details of the duplicate detection strategy. That being said, there are still some significant limitations to the orthogonality of the search strategy in A*-IDD. For example, although 3-level tie-breaking strategies which have been shown to be effective for RAM-based A* (Asai and Fukunaga 2016) can be implemented efficiently in A*-IDD, implementations of cache-friendly tie-breaking strategies, as well as more complex strategies such as depth-based and distance-to-go-based tie-breaking (Asai and Fukunaga 2017) pose directions for future work.

Acknowledgments

We thank the reviewers for their helpful comments and suggestions. This work was supported by Kakenhi grants 2533053, 17K00296, and a scholarship from the Konosuke Matsushita Memorial Foundation.

References

Asai, M., and Fukunaga, A. S. 2016. Tiebreaking strategies for A* search: How to explore the final frontier. In *AAAI*, 673–679.

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

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

Chung, T.-S.; Park, D.-J.; Park, S.; Lee, D.-H.; Lee, S.-W.; and Song, H.-J. 2009. A survey of flash translation layer. *Journal of Systems Architecture* 55(5):332 – 343.

Cormen, T. H.; Stein, C.; Rivest, R. L.; and Leiserson, C. E. 2001. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition.

Edelkamp, S., and Sulewski, D. 2008. Flash-efficient LTL model checking with minimal counterexamples. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, 73–82.

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.

Edelkamp, S. 2016. *External-Memory State Space Search*. Cham: Springer International Publishing. 185–225.

Hatem, M. 2014. *Heuristic search with limited memory*. Ph.D. Dissertation, University of New Hampshire.

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.

Helmert, M. 2006. The fast downward planning system. *J. Artif. Int. Res.* 26(1):191–246.

Hu, X.-Y.; Eleftheriou, E.; Haas, R.; Iliadis, I.; and Pletka, R. 2009. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, 10:1–10:9. New York, NY, USA: ACM.

Korf, R. E. 1985. Depth-first iterative-deepening. *Artificial Intelligence* 27(1):97 – 109.

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: Morgan Kaufmann Publishers Inc.

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. 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.

Pagh, R., and Rodler, F. F. 2004. Cuckoo hashing. *J. Algorithms* 51(2):122–144.

Patrascu, M., and Thorup, M. 2010. The power of simple tabulation hashing. *CoRR* abs/1011.5200.

Zobrist, A. L. 1970. A new hashing method with application for game playing.