Journal of Artificial Intelligence Research 62 (2018) 233-268 Submitted 12/2017; published 6/2018

Solving Large Problems with Heuristic Search: General-Purpose Parallel External-Memory Search

Matthew Hatem MHATEM AT CS.UNH.EDU
Ethan Burns EABURNS AT CS.UNH.EDU
Wheeler Ruml RUML AT CS.UNH.EDU
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA

Abstract

Classic best-first heuristic search algorithms, like A*, record every unique state they encounter in RAM, making them infeasible for solving large problems. In this paper, we demonstrate how best-first search can be scaled to solve much larger problems by exploiting disk storage and parallel processing and, in some cases, slightly relaxing the strict best-first node expansion order. Some previous disk-based search algorithms abandon best-first search order in an attempt to increase efficiency. We present two case studies showing that A*, when augmented with Delayed Duplicate Detection, can actually be more efficient than these non-best-first search orders. First, we present a straightforward external variant of A*, called PEDAL, that slightly relaxes best-first order in order to be I/O efficient in both theory and practice, even on problems featuring real-valued node costs. Because it is easy to parallelize, PEDAL can be faster than in-memory IDA* even on domains with few duplicate states, such as the sliding-tile puzzle. Second, we present a variant of PEDAL, called PE2A*, that uses partial expansion to handle problems that have large branching factors. When tested on the problem of Multiple Sequence Alignment, PE2A* is the first algorithm capable of solving the entire Reference Set 1 of the standard BAliBASE benchmark using a biologically accurate cost function. This work shows that classic best-first algorithms like A* can be applied to large real-world problems. We also provide a detailed implementation guide with source code both for generic parallel disk-based best-first search and for Multiple Sequence Alignment with a biologically accurate cost function. Given its effectiveness as a general-purpose problem-solving method, we hope that this makes parallel and disk-based search accessible to a wider audience.

1. Introduction

Best-first graph search algorithms such as A* (Hart, Nilsson, & Raphael, 1968) are widely used for solving problems in artificial intelligence. Graph search algorithms typically maintain an *open list*, containing nodes that have been generated but not yet expanded, and a *closed list*, containing all expanded nodes,[1] in order to prevent duplicated search effort when the same state is generated via multiple paths. As the size of problems increases, the memory required to maintain the open and closed lists makes algorithms like A* impractical. For example, an application of A* to random instances of the 15-puzzle using the

1. This data structure's name is indeed unfortunate, as it often holds more than just "closed" nodes in order to catch duplicates on the frontier and is rarely implemented as a list!

Manhattan distance heuristic will exhaust 8 GB of RAM in approximately two minutes on a modern computer (Burns, Hatem, Leighton, & Ruml, 2012).

The A* algorithm, as it is described in most literature, cannot scale beyond what are considered easy problems today. This motivates linear-space variants of A* that are able to solve problems that A* cannot solve while using only a fraction of the memory. Iterative Deepening A* (IDA*, Korf, 1985) and Recursive Best-first Search (RBFS, Korf, 1993) achieve linear-space complexity by eliminating the open and closed lists. As a result, they are limited to a narrow class of problems: those that do not form highly connected spaces. Without a closed list, these algorithms are not able to detect duplicate paths to the same state and are doomed to repeatedly explore the same states multiple times before finding a solution. For example, a depth-first search to depth three on a grid with four-way movement generates 52 states while a breadth-first search that recognizes duplicates generates only 26. Furthermore, in the absence of an open list these methods use a depth-first search. For IDA*, this means a best-first search order is only possible if the heuristic is admissible. RBFS simulates a best-first search order even with an inadmissible heuristic but, like IDA*, it suffers from unbounded node regeneration overhead for problems that do not exhibit a narrow range of edge costs (however see Hatem, Kiesel, & Ruml, 2015; Burns & Ruml, 2013; Russell, 1992). IDA* and RBFS work well when there are few duplicates and a narrow range of edge costs. However, real-world problems such as Multiple Sequence Alignment form highly connected search spaces and require a wide range of edge costs to model biologically plausible results.

In order to apply heuristic search to large problems that form highly connected spaces, we need scalable techniques for processing duplicates. In this paper we define *scalable techniques* as those that are capable of exploiting external memory and multiple CPUs to solve larger problems efficiently. External memory search algorithms take advantage of cheap secondary storage, such as magnetic disks, to solve much larger problems than algorithms that only use main memory. A naïve implementation of disk-based A* would exhibit poor performance because it relies on random access in order to process duplicates. The closed list, normally stored as a hash-table in RAM, provides quick random access to states that have already been explored by the search. While sequential access to disk can take upwards of two orders of magnitude longer than accessing RAM, random access to disk can take several orders of magnitude more time than sequential access. Storing the closed list as a hash table on disk is impractical. To implement an efficient disk-based best-first search, great care must be taken to access data sequentially to minimize seeks and exploit caching. The same techniques used by external search can be used to distribute search effort across multiple CPUs.

This paper presents simple modifications to classic A* search and demonstrates that they result in a general scalable algorithm: one that can exploit external storage and additional CPUs to solve larger problems efficiently. In section 2, we discuss the technique of delayed duplicate detection in detail and present empirical results for an efficient external memory variant of A* (A*-HBDDD). As far as we are aware, we are the first to present results for HBDDD using A* search, other than the anecdotal results mentioned briefly by Korf (2004). These results provide evidence that A*-HBDDD performs well on unit-cost domains and that efficient parallel external memory search can surpass serial in-memory search. Although many regard disk-based search as slow and unwieldy, we hope this result encourages practitioners to take another look at these techniques. In section 3, we show that previous approaches are unable to solve problems that exhibit a wide range of edge costs. To this end, we introduce Parallel External Dynamic A* Layering (PEDAL Hatem, Burns, & Ruml, 2011), an extension of A*-HBDDD that is able to solve problems with arbitrary costs. In section 4, we introduce the problem of Multiple Sequence Alignment (MSA). Previous approaches do not scale to the hardest instances of a popular MSA benchmark. In this section, we introduce a second extension to A*-HBDDD that uses the technique of partial expansions to solve the entire benchmark set.

This work demonstrates that parallel external-memory search does not need to completely abandon the best-first search principle. Instead only a small relaxation is needed to significantly improve its efficiency. We hope that it demystifies scalable search and encourages wider use of these techniques, which match so well with modern multi-core commodity hardware.

2. External Memory Search

Delayed Duplicate Detection (DDD, Korf, 2003) is a simple way to make use of external storage that places newly generated nodes in external memory and then processes them at a later time. The original description of DDD, also referred to as sorting-based DDD (SBDDD), divides the search process into two phases, an expand phase and a merge phase. The expand phase writes newly generated nodes directly to a file on disk. The merge phase performs a disk-based sort on the file so that duplicate nodes are brought together. Duplicate merging is accomplished by performing a linear scan of the sorted file, writing only unique nodes to a new file. This newly merged file becomes the search frontier for the next expand phase. The search continues, interleaving expand and merge phases, until a goal node is expanded. Files can in theory be made arbitrarily small and only one file needs to be kept in memory at a time. Unfortunately, the time complexity of this technique is $\mathcal{O}(n \log n)$ where n is the total number of nodes encountered during search. For large problems, this technique incurs more overhead than is desirable.

Structured Duplicate Detection (SDD, Zhou & Hansen, 2004) is an alternative to DDD that exploits connectivity in the state space to avoid writing duplicates to disk. SDD uses a projection function to localize memory references and performs duplicate merging immediately in main memory. Unlike DDD, SDD does not store duplicate states to disk and requires less external storage. However, this efficiency comes at the cost of increased time complexity, as SDD can read and write the same states to disk multiple times during duplicate processing (Zhou & Hansen, 2009). The benefits of SDD are limited to the amount of main memory available on a single machine and it is not obvious how to deploy SDD in a distributed setting. In this paper we focus on DDD because it is simple to implement and has been shown to easily scale beyond a single machine.

External A* (Edelkamp, Jabbar, & Schrdl, 2004) combines A* with a variant of SBDDD whereby nodes with the same g and h values are grouped together in a bucket which maps to a file on external storage. The search proceeds by iteratively expanding layers of buckets for which the g and h values sum to the minimum f value among the search frontier. Delayed duplicate detection is performed by appending nodes to their respective buckets and later sorting and scanning each bucket to eliminate duplicate nodes. In External A*, the g and h

buckets must be expanded in lowest-g-value-first order which is equivalent to the A* search order with worst-case tie breaking and can therefore result in many more node expansions than a regular A* search. Moreover, because of the way buckets are organized according to two values, it is not obvious how to dynamically relax the best-first search order.

To avoid the overhead of disk-based sorting, Korf (2004) presents an efficient form of DDD called Hash-Based Delayed Duplicate Detection (HBDDD). HBDDD uses two hash functions, one to assign nodes to buckets (which map to files on disk) and a second hash function to identify duplicate states within a bucket. Because duplicate nodes will hash to the same value, they will always be assigned to the same file. When removing duplicate nodes, only those nodes in the same file need to be in main memory. This technique increases the minimum memory requirements over SBDDD, requiring that the size of the largest bucket fit in main memory. However, this is easily achieved by using a hash function with an appropriate range. HBDDD has been shown to perform better than SBDDD when a search is limited by time rather than available storage (Korf, 2016).

Korf (2008a) described how HBDDD can be combined with A* search (A*-HBDDD). A*-HBDDD proceeds in two phases: an expansion phase and a merge phase. In the expansion phase, all nodes that have an f that is equal to the minimum solution cost estimate $f_{\min}$ of all open nodes are expanded. Unlike External A*, nodes are not grouped according to their g and h values so each bucket containing qualifying nodes must be scanned. The expanded nodes and the newly generated nodes are stored in their respective files. We define *recursive expansion* to be an expansion that is performed immediately to a generated node, without performing any duplicate checking. If a generated node has an $f \leq f_{\min}$, then it is recursively expanded. Once all nodes within $f_{\min}$ are expanded, the merge phase begins: each file is read into a hash-table in main memory and duplicates are removed in linear time. During the expand phase, HBDDD requires only enough memory to read and expand a single node from the open file; successors can be stored to disk immediately. During the merge phase, it is possible to process a single file at a time to reduce main memory requirements.

HBDDD may also be used as a framework to parallelize search (Korf, 2008a). Because duplicate states will be located in the same file, the merging of delayed duplicates can be done in parallel, with each file assigned to a different thread. Expansion may also be done in parallel. As nodes are generated, they are stored in the file specified by the hash function. It is possible that two threads might generate nodes that need to be placed in the same file. Therefore, a lock (often provided by the OS) must be placed around each file so that a thread can obtain exclusive access to the file while writing. A carefully constructed hash function, one that bounds the number of buckets than need to be written to when expanding a node, can help minimize lock contention. See literature on SDD, for example, the work by Burns, Lemons, Ruml, and Zhou (2010) for discussion on abstraction based hashing and balance between locality and parallelism. For our experiments we verified that a lock was provided by examining the source code for the I/O modules. For example, the source code for the Glibc standard library 2.12.90 does contain such a lock.

Because the main contributions of this paper build on the framework of A*-HBDDD. We will discuss A*-HBDDD in detail and present empirical results.

```
SEARCH($initial$)
 1. $bound \leftarrow f(initial)$; $bucket \leftarrow hash(initial)$
 2. $write(OpenFile(bucket), initial)$
 3. while $\exists bucket \in Buckets : min\_f(bucket) \leq bound$
 4.   for each $bucket \in Buckets : min\_f(bucket) \leq bound$
 5.       $ThreadExpand(bucket)$
 6.   if $incumbent$ break
 7.   for each $bucket \in Buckets : NeedsMerge(bucket)$
 8.       $ThreadMerge(bucket)$
 9.   $bound \leftarrow min\_f(Buckets)$

THREADEXPAND($bucket$)
10. for each $state \in Read(OpenFile(bucket))$
11.   if $f(state) \leq bound$
12.       $RecurExpand(state)$
13.   else $append(NextFile(bucket), state)$

RECUREXPAND($n$)
14. if $IsGoal(n)$ $incumbent \leftarrow n$; return
15. for each $succ \in expand(n)$
16.   if $f(succ) \leq bound$
17.       $RecurExpand(succ)$
18.   else
19.       $append(NextFile(hash(succ)), succ)$
20.   $append(ClosedFile(hash(n)), n)$

THREADMERGE($bucket$)
21. $Closed \leftarrow read(ClosedFile(bucket))$; $Open \leftarrow \emptyset$
22. for each $n \in NextFile(bucket)$
23.   if $n \notin Closed \cup Open$ or $g(n) < g(Closed \cup Open[n])$
24.       $Open \leftarrow (Open - Open[n]) \cup \{n\}$
25. $write(OpenFile(bucket), Open)$
26. $write(ClosedFile(bucket), Closed)$
```

Figure 1: Pseudocode for A*-HBDDD.

2.1 A*-HBDDD in Detail

To understand the algorithm in more detail, we present pseudocode of A*-HBDDD in Figure 1. Search nodes are mapped to *buckets* using a hash function. Each bucket is backed by a set of three files [2] on disk: 1) a file of frontier nodes that have yet to be expanded, 2) a file of newly generated nodes (and possibly duplicates) that have yet to be checked against the closed list and 3) a file of closed nodes that have already been expanded.

A*-HBDDD begins by placing the initial node in its respective bucket based on the supplied hash function (lines 1–2). The cost bound for the first iteration is set to the f value of the initial state (line 1). All buckets that contain a state with f less than or equal to the minimum *bound* are divided among a pool of threads to be expanded (lines 4–20). Alternatively, references to these buckets can be stored in a work queue, guarded by a lock. Free threads would acquire exclusive access to this queue for jobs.

Recall that each bucket is backed by three files: *OpenFile*, *NextFile* and *ClosedFile*. The *OpenFile* contains all open nodes for a bucket. The set of *OpenFile*s among all buckets collectively represent the open list for the search. When processing an expansion job for a given bucket, a thread proceeds by expanding all of the frontier nodes with f values that are within the current bound from the *OpenFile* of the bucket (lines 10–13). Nodes that are chosen for expansion are appended to the *ClosedFile* for the current bucket (line 20). The set of *ClosedFile*s among all buckets collectively represent the closed list for the search. Nodes that were not chosen for expansion and successor nodes that exceed the bound are appended to the *NextFile* for the current bucket (lines 13 & 19). The set of *NextFile*s collectively represent the search frontier and require duplicate detection in the following merge phase. Finally, if a successor is generated with an f value that is within the current bound then it is expanded immediately as a recursive expansion (lines 12 & 17). To improve efficiency, individual states are not written to disk immediately upon generation. Instead each bucket has an internal buffer to hold states. When the buffer becomes full, the states are written to disk.

If an expansion thread generates a goal state (line 15) within the bound (lines 16 and 11), a reference to the incumbent solution is updated (line 14) and (assuming the heuristic is admissible) the search terminates (line 6). If the heuristic is admissible, then the incumbent is admissible because of the strict best-first search order on f. Solution recovery is performed by walking backward from the goal state using an inversion operator to generate each parent state along the path to the initial state. This requires storing an inversion operator for each node. Each parent state generated during this solution recovery process needs to be mapped and loaded from its respective bucket. If a solution has not been found, then all buckets that require merging are divided among a pool of threads to be merged in the next phase (lines 7–8).

In order to process a merge job, each thread begins by reading the *ClosedFile* for its bucket into a hash-table (line 21) called *Closed*. A*-HBDDD requires enough internal memory to store all closed nodes and unique nodes on the frontier in all buckets currently being merged by active threads. The size of a bucket can be easily tuned by varying the granularity of the hash function. Next, all frontier nodes in the *NextFile* are streamed in and checked for duplicates against the closed list (lines 22–26). The nodes that are not

2. With the exception of the init file our files roughly correspond to those described by Korf (2008b)

duplicates or that have been reached via a better path and therefore have a lower g value are written back out to *OpenFile* so that they remain on the frontier for latter phases of search (lines 23–25). The hash-table is updated to contain these nodes as well. All other duplicate nodes are ignored. Finally, the open and closed nodes are flushed to disk (lines 25 and 26).

To save external storage, Korf (2008a) suggests that instead of proceeding in two phases, merge jobs may be interleaved with expansion jobs. With this optimization, a bucket may be merged if all of the buckets that contain its predecessor nodes have been expanded. An undocumented ramification of this optimization for HBDDD, however, is that it does not permit recursive expansions. Because of recursive expansions, one cannot determine the predecessor buckets and therefore all buckets must be expanded before merges can begin. Our variant of A*-HBDDD implements recursive expansions and therefore it does not interleave expansions and merges. One technique for detecting when predecessor nodes have been expanded is Structured Duplicate Detection (SDD, Zhou & Hansen, 2004). SDD is an alternative to DDD that exploits connectivity in the state space to avoid writing duplicates to disk.

2.2 Empirical Results

We evaluated the performance of A*-HBDDD on the sliding-tile puzzle. We compared A*-HBDDD with highly optimized implementations of internal A*, IDA* and Asynchronous Parallel IDA* (AIDA*, Reinefeld & Schnecke, 1994). AIDA* is a parallel version of IDA* that works by performing a breadth-first search to some specified depth and the resulting frontier is then divided evenly among all available threads. Threads perform an IDA* search in parallel for each node in its queue. The upper bounds for all IDA* searches are synchronized across all threads so that a strict best-first search order is achieved given an admissible and consistent heuristic. AIDA* can be seen as a parallel approximation to Simplified Memory-Bounded A* (SMA*, Russell, 1992) with large f layers.

To verify that we had efficient implementations of these algorithms, we compared our implementations (in Java) to highly optimized versions of A* and IDA* written in C++ (Burns et al., 2012). The Java implementations use many of the same optimizations. In addition we use the High Performance Primitive Collection (HPPC) in place of the Java Collections Framework (JCF) for many of our data structures. This improves both the time and memory performance of our implementations (Hatem, Burns, & Ruml, 2013).

We also compared A*-HBDDD to an alternative external algorithm, breadth-first heuristic search (BFHS, Zhou & Hansen, 2006) with delayed duplicate detection (BFHS-DDD). BFHS attempts to reduce the memory requirement of search, in part by removing the need for a closed list. BFHS proceeds in a breadth-first ordering by expanding all nodes within a given upper bound on f at one depth before proceeding to the next depth. To prevent duplicated search effort Zhou and Hansen (2006) use a strategy first introduced by Korf (1999), which guarantees that, in an undirected graph, checking for duplicates against the previous depth layer and the frontier is sufficient to prevent the search from leaking back into previously visited portions of the space. While BFHS is able to do away with the closed list, for many problems it will still require a significant amount of memory to store the exponentially growing search frontier. This motivates combining BFHS with HBDDD.

	Machine	Threads	Time	Expanded	Nodes/Sec
A* (Java)	A	1	925	1,557,459,344	1,683,739
A* (C++)	A	1	516	1,557,459,344	3,018,332
IDA* (Java)	B	1	1,104	18,433,671,328	16,697,166
IDA* (C++)	B	1	634	18,433,671,328	29,075,191
AIDA* (Java)	B	24	**222**	14,994,333,240	**67,542,041**
BFHS-DDD (Java)	B	24	3,355	10,978,208,032	3,272,193
A*-HBDDD (Java)	B	24	1,014	3,492,457,298	3,444,237
A*-HBDDD$_{tt}$ (Java)	B	24	433	**1,489,553,397**	3,440,077

Table 1: Performance summary on the 100 random 15-puzzle instances from (Korf, 1985). Times reported in wall clock seconds for solving all instances.

Like IDA*, BFHS uses an upper bound on f values to prune nodes. If a bound is not available in advance, iterative deepening can be used. However, since BFHS does not store a closed list, the full path to each node from the root is not maintained and it must use divide-and-conquer solution reconstruction (Korf, Zhang, Thayer, & Hohwald, 2005) to rebuild the solution path. Our implementation of BFHS-DDD does not perform solution reconstruction and therefore the results presented give a lower bound on its actual solving times.

The 15-puzzle is a standard search benchmark. We used the 100 instances from Korf (1985) and the Manhattan distance heuristic. For the algorithms using HBDDD, we selected a hash function that maps states to buckets by ignoring all except the position of the blank, one and two tiles. This hash function results in 3,360 buckets and the number of buckets that need to be considered for writing newly generated nodes when expanding a node is bound by the maximum number of actions applicable in any given state. A random hash function would probably provide even better load balancing among files. We use the minimum f value of any generated node greater than the current bound to update the cost bounds for both A*-HBDDD and BFHS-DDD.

The first set of rows in Table 1 summarizes the performance of internal A*, IDA* and AIDA*. The results for A* were generated on *Machine-A*, a dual quad-core (8 cores) machine with Intel Xeon X5550 2.66 GHz processors and 48 GB RAM. A* needs roughly 30 GB of RAM to solve all 100 instances. All other results were generated on *Machine-B*, a dual hexa-core machine (12 cores) with Xeon X5660 2.80 GHz processors, 12 GB of RAM and 12 320 GB disks. In-memory A* is not able to solve all 100 instances on this machine due to memory constraints. Our version of AIDA* used 24 threads and generated a frontier of 24,000 nodes, using an A* search, to seed the parallel phase of the search. From these results, we see that the Java implementation of A* is just a factor of 1.7 slower than the most optimized C++ implementation known. These results provide confidence that our comparisons reflect the true ability of the algorithms rather than misleading aspects of implementation details.

The second set of rows in Table 1 shows a summary of the performance results for A*-HBDDD compared to in-memory search. We used 24 threads and the states generated by the external algorithms were distributed across all 12 disks. A*-HBDDD outperforms BFHS-DDD because it expands fewer nodes. We discuss this in more detail in section 3. The results show that the base Java implementation of A*-HBDDD is just 1.7× slower than the C++ implementation of IDA* but slightly faster than the Java implementation. Note that A*-HBDDD expanded almost 3.5 billion nodes while A* expanded fewer than 1.6 billion. We believe this is due to duplicate states generated during recursive expansion, when the closed list is not consulted. We can improve the performance of A*-HBDDD by exploiting available RAM with the simple technique of using transposition tables to avoid expanding duplicate states during recursive expansions (A*-HBDDD$_{tt}$). With this improvement, A*-HBDDD is 1.4× faster than the highly optimized C++ IDA* solver and 2.5× faster than the optimized Java IDA* solver. A*-HBDDD$_{tt}$ is within a factor of two of a highly optimized implementation of parallel AIDA*, which cannot cope with state spaces with many duplicate nodes. Moreover, it is possible for AIDA* to expand more nodes than serial IDA* since it can expand parts of the tree that would not be reached by serial IDA* if serial IDA* finds a solution early, or it can expand fewer nodes than serial IDA* if the first solution that serial IDA* would find comes late in the search. AIDA* is able to outperform A*-HBDDD and A*-HBDDD$_{tt}$ even when it expands 5 to 10 times as many nodes because node expansion in the sliding-tiles domain is cheap. For many practical problems node expansion is much more expensive, and A*-HBDDD and A*-HBDDD$_{tt}$ may outperform AIDA*. While A*-HBDDDtt running on 12 cores (last line of table) has only 2x speed up over serial A* running on 1 core (first line of table), note that it is an external algorithm that trades slow access to disk for the ability to solve problems beyond the confines of RAM. Given that disk is millions of times slower than RAM, it is exciting to see that external-memory search can be faster than internal-memory search.

While these results show that A*-HBDDD performs well compared to IDA* on problems like the sliding-tile puzzle, the strictly best-first layered search does not work well for other domains, preventing it from serving as a general-purpose search method for large problems. In the next two sections we discuss two important limitations of A*-HBDDD that motivate the main contributions of this paper.

3. External Memory Search With Non-Uniform Edge Costs

A*-HBDDD achieves sequential I/O behavior by dividing the search into f layers. Each layer refers to nodes with the same lower bound on solution cost f. At each iteration of search, nodes are read sequentially from external memory and expanded only if their f value is within the current lower bound on solution cost. Many real-world problems have real-valued costs, giving rise to a large number of f layers with few nodes in each, substantially eroding performance. A*-HBDDD reads all open nodes from files on disk and expands only the nodes within the current f bound. If there is only a small number of nodes in each f layer, the algorithm pays the cost of reading the entire frontier only to expand a few nodes. Then in the merge phase, the entire closed list is read only to merge the same few nodes. Additionally, when there are many distinct f values, the successors of each node tend to exceed the current f bound, resulting in fewer I/O-efficient recursive expansions. Korf (2004) speculated that the problem of many distinct f values could be remedied by somehow expanding more nodes than just those with the minimum f value. In this section we present an algorithm, Parallel External Dynamic A* Layering (PEDAL) that does exactly this. PEDAL improves on A*-HBDDD by relaxing the strictly best-first ordering of the search in order to perform a constant number of expansions per I/O operation.

We begin by reviewing previous work in section 3.1. In section 3.2 we describe PEDAL in more detail and prove that it is I/O efficient. In an empirical evaluation in section 3.3, we compare PEDAL to IDA*, IDA*$_{\text{CR}}$ (Sarkar, Chakrabarti, Ghose, & Sarkar, 1991), A*-HBDDD and BFHS-DDD using a variant of the sliding-tile puzzle with non-unit edge costs and a more realistic dockyard planning domain. The results show that PEDAL gives the best performance on the sliding-tile puzzle and is the only practical approach for the real-valued problems among the algorithms tested in our experiments. PEDAL demonstrates that relaxed best-first heuristic search can be effective for large problems with arbitrary costs.

3.1 Previous Work

In this section, we present relevant previous work that PEDAL builds on, as well as alternative techniques. IDA* and BFHS were introduced in a previous section but we include descriptions here with further details.

3.1.1 Iterative Deepening A*

Iterative-deepening A* (IDA*, Korf, 1985) is an internal memory technique that requires memory only linear in the maximum depth of the search. This reduced memory complexity comes at the cost of repeated search effort. IDA* performs iterations of a bounded depth-first search where a path is pruned if $f(n)$ becomes greater than the bound for the current iteration. After each unsuccessful iteration, the bound is increased to the minimum f value among the nodes that were generated but not expanded in the previous iteration.

Each iteration of IDA* expands a super-set of the nodes in the previous iteration. If the number of nodes expanded in each iteration grows geometrically, then the total number of nodes expanded by IDA* is $\mathcal{O}(n)$, where n is the number of nodes that A* would expand (Sarkar et al., 1991). In domains with real-valued edge costs, there can be many unique f values and the standard minimum-out-of-bound bound layering of IDA* may lead to only a few new nodes being expanded in each iteration. Because of this, the number of nodes expanded by IDA* can be $\mathcal{O}(n^2)$ (Sarkar et al., 1991) in the worst case when the number of new nodes expanded in each iteration is constant. To alleviate this problem, Sarkar et al. introduce IDA*$_{\text{CR}}$. IDA*$_{\text{CR}}$ tracks the distribution of f values of pruned nodes (the nodes that were generated but not expanded during an iteration of search). This distribution used to find a good threshold for the next iteration. This is achieved by selecting the bound that will cause the desired number of pruned nodes to be expanded in the next iteration. To guarantee efficiency, the desired number must follow a geometric progression (at least doubling). If the successors of these pruned nodes are not expanded in the next iteration then this scheme is often able to accurately double the number of nodes between iterations. If the successors do fall within the bound on the next iteration then more nodes may be expanded than desired. Since the threshold is increased liberally, nodes are not expanded in a strict best-first order. Therefore, branch-and-bound must be used on the final iteration of search to ensure optimality. In branch-and-bound, we continue the search after finding a solution until all nodes whose lower bounds are less than the incumbent solution cost have been expanded, ensuring that the solution is optimal. Any nodes whose lower bound is equal or greater than the incumbents cost can be pruned, as they cannot lead to a better solution. IDA*$_{\text{CR}}$ is effective for problems that exhibit a wide range of f values but may still achieve poor performance for domains where the branching does not allow for doubling the number of expanded nodes for each iteration.

IDA* and IDA*$_{\text{CR}}$ suffer from an additional source of node regeneration overhead on search spaces that form highly connected graphs. Because they use depth-first search, they cannot detect duplicate search states except those that form cycles in the current search path. Even with cycle checking, the search will perform extremely poorly if there are many paths to each node in the search space. This motivates the use of a closed list in classic algorithms like A*.

3.1.2 Breadth-First Heuristic Search

In this section we provide more details for BFHS, introduced in section 2.2. BFHS attempts to reduce the memory requirement of search by removing the need for a closed list. BFHS proceeds in a breadth-first ordering by expanding all nodes within a given upper bound on f at one depth before proceeding to the next depth. If a bound is not available in advance, iterative deepening can be used, however, as discussed earlier, iterative-deepening fails on domains with many distinct f values. To provide a suitable comparison to PEDAL, we propose a novel variant of BFHS that uses the same technique of IDA*$_{\text{CR}}$ for updating the upper bound at each iteration of search. One side effect of the breadth-first search order is that BFHS is not able to break ties among nodes with the same f value. A* with optimal tie-breaking (expanding nodes with highest g first) expands nodes with higher g values first (deeper nodes first in domains with uniform edge costs). BFHS needs to expand all nodes n with $f(n) \leq C^*$ at all depth-layers prior to the depth layer that contains the goal. The search order of BFHS is equivalent to the search order of A* with worst-case tie breaking (expanding nodes with lower g first) and can expand up to twice as many unique nodes as A* with optimal tie breaking. When combined with iterative deepening, BFHS can expand up to four times as many nodes as A* (Zhou & Hansen, 2006). Furthermore, when combined with the bound setting technique of IDA*$_{\text{CR}}$, it can expand many nodes with f values greater than the optimal solution cost which are not strictly necessary for optimal search. BFHS is not able to benefit substantially from branch-and-bound in the final iteration because goal states are generated in the deepest layers of the search and it must expand all nodes within the final inflated upper bound whose depths are less than the goal depth.

3.2 Parallel External Dynamic A* Layering

A*-HBDDD suffers from excessive I/O overhead when there are a small number of nodes in each f layer. PEDAL solves this problem by relaxing the best-first search order, allowing it to solve problems with arbitrary f cost distributions. PEDAL can be seen as a combination

```
SEARCH($initial$)
27. $bound \leftarrow f(initial)$; $bucket \leftarrow hash(initial)$
28. $write(OpenFile(bucket), initial)$
29. while $\exists bucket \in Buckets : min\_f(bucket) \leq bound$
30.    for each $bucket \in Buckets : min\_f(bucket) \leq bound$
31.        $ThreadExpand(bucket)$
32.    if $incumbent$ break
33.    for each $bucket \in Buckets : NeedsMerge(bucket)$
34.        $ThreadMerge(bucket)$
35.    $bound \leftarrow NextBound(f\_dist)$

THREADMERGE($bucket$)
36. $Closed \leftarrow read(ClosedFile(bucket))$; $Open \leftarrow \emptyset$
37. for each $n \in NextFile(bucket)$
38.    if $n \notin Closed \cup Open$ or $g(n) < g(Closed \cup Open[n])$
39.        $Open \leftarrow (Open - Open[n]) \cup \{n\}$
40.        $f\_distribution\_add(f\_dist, f(n))$
41. $write(OpenFile(bucket), Open)$
42. $write(ClosedFile(bucket), Closed)$
```

Figure 2: Pseudocode for PEDAL.

of A*-HBDDD and an estimation technique inspired by IDA*$_{\text{CR}}$ to dynamically layer the search space.

Like HBDDD-A*, PEDAL proceeds in two phases: an expansion phase and a merge phase. However, during the merge phase, it tracks the distribution of the f values of the frontier nodes that were determined not to be duplicates. As we explain in detail below, this distribution is used to select the f bound for the next expansion phase that will give a constant number of expansions per node I/O. The pseudo-code for PEDAL, given in Figure 2, is adapted from the pseudo-code for A*-HBDDD given in Figure 1. The main difference is at lines 35 and 40 where PEDAL records the f value of all nodes that are added to the frontier and uses this distribution to select the next bound for the following expansion phase. Another critical difference is that, since PEDAL relaxes the best-first search order, it must perform branch-and-bound after an incumbent solution is found.

3.2.1 Overhead

PEDAL maintains a layering such that the number of nodes expanded in each layer is at least a constant fraction of the amount of I/O (the number of nodes read and written to external memory) at each iteration. It keeps a histogram of f values for all nodes on the open list and a count of the total number of nodes on the closed list. The cost bound for each layer is selected so that a constant fraction of the sum of nodes on the open and closed lists will be expanded. We found a value of 1/2 worked well in practice for the domains tested. Unlike IDA*$_{\text{CR}}$ which only provides a heuristic for the desired doubling behavior, the technique used by PEDAL is guaranteed to give only bounded I/O overhead. That is,

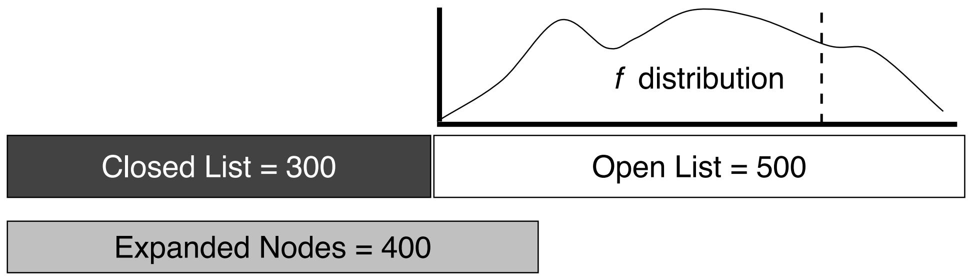


Figure 3: PEDAL keeps a histogram of f values on the open list and uses it to update the threshold to allow for a constant fraction of the number of nodes on open and closed to be expanded in each iteration.

the number of nodes expanded is at least a constant fraction of the number of nodes read from and written to disk. We assume a constant branching factor b and that the number of frontier nodes remaining after duplicate detection is always large enough to expand the desired number of nodes. We begin with a few useful lemmata. Let o be the number of nodes on the open list, c be the number of nodes on the closed list, e be the number of nodes expanded in an iteration and r be the number of recursively expanded nodes in an iteration.

Lemma 1 *The number of I/O operations during the expand phase is at most* $2o+eb+rb+r$.

Proof: During the expand phase we read o open nodes from disk. We write at most eb nodes plus the remaining $o-e$ nodes, that were not expanded, to disk. We also write at most rb recursively generated nodes and $e+r$ expanded nodes to disk. □

Lemma 2 *The number of I/O operations during the subsequent merge phase is at most* $c+e+2(r+eb+rb)$.

Proof: During the merge phase we read at most $c+e+r$ nodes from disk and $eb+rb$ newly generated nodes from disk. We write at most r recursively expanded nodes to the closed list and $eb+rb$ new nodes to the open list. □

Lemma 3 *The total number of I/O operations is at most* $2o+c+e(3b+1)+r(3b+3)$.

Proof: From Lemma 1, Lemma 2 and

$$\begin{aligned}
&\textit{total I/O} \\
&\quad = \textit{expanded I/O} + \textit{merged I/O} \\
&\quad = (2o+eb+rb+r)+(c+e+2(r+eb+rb)) \\
&\quad = (2o+eb+rb+r)+(c+e+2r+2eb+2rb)) \\
&\quad = 2o+c+(3eb+e)+(3rb+3r) \\
&\quad = 2o+c+e(3b+1)+r(3b+3)
\end{aligned}$$

□

	Threads	Time	Expanded	Nodes/Sec
IDA*$_{\mathrm{CR}}$	1	14,009	80,219,537,668	5,726,285
AIDA*$_{\mathrm{CR}}$	24	**1,052**	48,744,622,573	**46,335,192**
BFHS-DDD	24	3,147	7,532,248,808	2,393,469
PEDAL	24	1,066	**6,585,305,718**	6,177,585

Table 2: Performance summary for 15-puzzle with square root costs. Times reported in seconds for solving all instances.

Theorem 1 *If the number of nodes expanded e is chosen to be $k(o + c)$ for some constant $0 < k \leq 1$, and there is a sufficient number of frontier nodes, $o \geq e$, then the number of nodes expanded is bounded from below by a constant fraction of the total number of I/O operations for some constant q.*

Proof:

$$\begin{aligned}
&\textit{total I/O}\\
&\quad= e(3b+1) + r(3b+3) + 2o + c \quad \text{by Lemma 3}\\
&\quad< e(3b+3) + r(3b+3) + 2o + c\\
&\quad= ze + zr + 2o + c \qquad \text{for } z = (3b+3)\\
&\quad= zko + zkc + zr + 2o + c \qquad \text{for } e = ko + kc\\
&\quad= o(zk+2) + c(zk+1) + zr\\
&\quad< o(zk+2) + c(zk+2) + zr\\
&\quad< qko + qkc + qr \qquad \text{for } q \geq (zk+2)/k\\
&\quad= q(ko + kc + r)\\
&\quad= q(e + r) \qquad \text{because } e = k(o+c)\\
&\quad= q \cdot \textit{total expanded}
\end{aligned}$$

Because $q \geq (zk + 2)/k = (3b + 3) + 2/k$ is constant, the theorem holds. □

3.3 Empirical Evaluation

We evaluated the performance of PEDAL on two domains with a wide range of edge costs: the square root sliding-tile puzzle and a dockyard robot planning domain. For these experiments, we implemented a novel variant of BFHS-DDD that uses the IDA*$_{\mathrm{CR}}$ technique for setting the upper bound at each iteration. As in the previous experiments, our implementation of BFHS-DDD does not perform solution reconstruction and therefore the results presented give a lower bound on its actual solution times. All algorithms were written in Java as described in section 2.2 and were run on *Machine-B*.

3.3.1 The Square Root 15-Puzzle

The classic sliding-tile puzzle lacks an important feature that many real-world applications of heuristic search have: real-valued costs. In order to evaluate PEDAL on a domain with

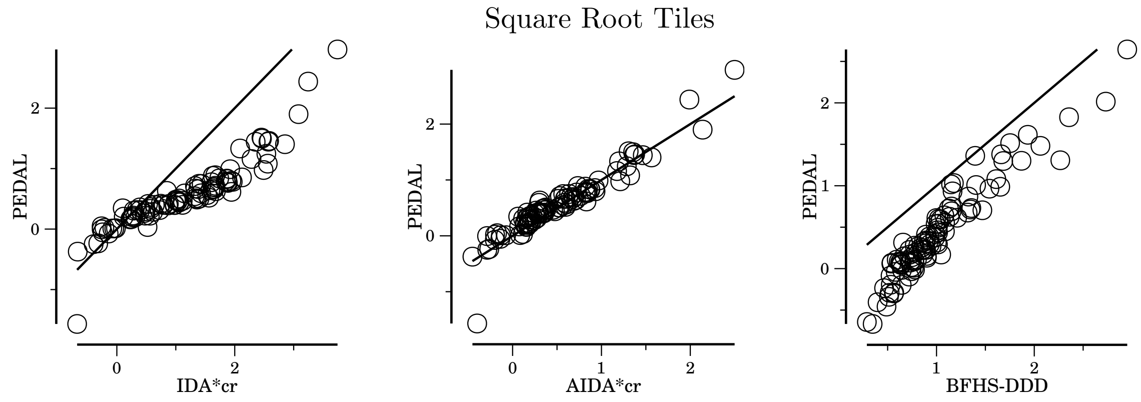
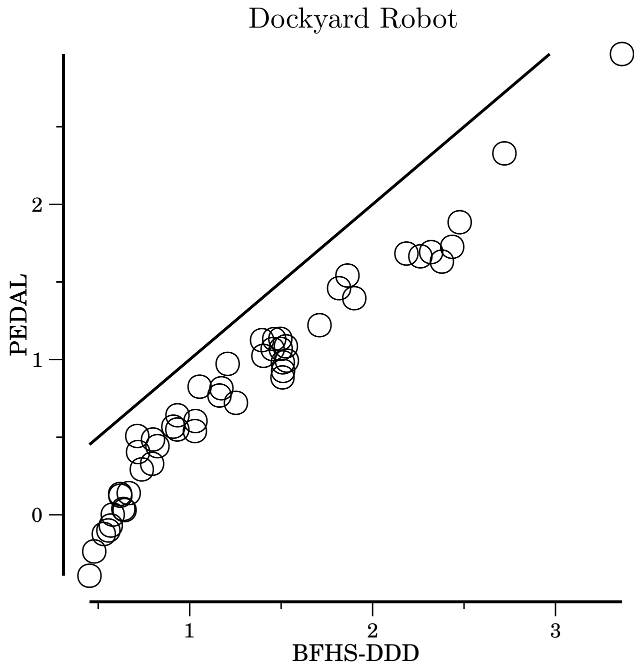


Figure 4: Comparison between PEDAL, IDA*$_{\mathrm{CR}}$, AIDA*$_{\mathrm{CR}}$, and BFHS-DDD. The axes show $\log_{10}$ CPU time.

real-valued costs that is simple, reproducible and has well understood connectivity, we use a variant of the puzzle proposed by Hatem et al. (2011), in which each move costs the square root of the number on the tile being moved. This gives rise to many distinct f values.

The plots in Figure 4 show a comparison between PEDAL, IDA*$_{\mathrm{CR}}$, AIDA*$_{\mathrm{CR}}$and BFHS-DDD on the square root version of the 100 tiles instances used by Korf (1985). The x axes show $\log_{10}$ CPU time in seconds: points below the diagonal $y = x$ line represent instances where PEDAL solved the problems faster than the respective algorithm. The first square root tiles plot shows a comparison between PEDAL and IDA*$_{\mathrm{CR}}$. We can see from this plot that IDA*$_{\mathrm{CR}}$ solved the easier instances faster because it does not have to go to disk, however PEDAL greatly outperformed IDA*$_{\mathrm{CR}}$ on the more difficult problems. The advantage of PEDAL over IDA*$_{\mathrm{CR}}$ grew quickly as the problems required more time. The center plot shows a comparison between PEDAL and AIDA*$_{\mathrm{CR}}$ and Table 2 includes results for AIDA*$_{\mathrm{CR}}$. The node expansion rate of AIDA*$_{\mathrm{CR}}$ is nearly 7.5 times that of PEDAL but has roughly the same solving time because it cannot detect duplicates. Both algorithms achieve a speedup of approximately 13x when run on this 24-core machine.

The square root tiles plot on the left compares PEDAL to BFHS-DDD. PEDAL was much faster on easier instances and gave consistently superior performance throughout the range of problem difficulties. As discussed above, the search order of BFHS is equivalent to A* with worst-case tie breaking and when combined with iterative deepening, it can expand up to four times as many nodes as A* (Zhou & Hansen, 2006). Moreover, since BFHS-DDD and PEDAL use a loose upper bound, they can expand many nodes with f values greater than the optimal solution cost which are not strictly necessary for optimal search. However, unlike PEDAL, BFHS is not able to effectively perform branch-and-bound in the final iteration and must expand all nodes within the final inflated upper bound that are shallower than the goal.



	Time	Expanded	Nodes/Sec
BFHS-DDD	4,993	4,695,394,966	940,395
PEDAL	**1,765**	**1,983,155,888**	**1,123,601**

Table 3: Performance summary for Dockyard Robot. Times reported in seconds for solving all instances using 12 cores and 24 threads. The axes show $\log_{10}$ CPU time.

3.3.2 Dockyard Robot Planning

The sliding-tile puzzle does not have many duplicate states and it is, for some, perhaps not a practically compelling domain. We implemented a planning domain inspired by the dockyard robot example used throughout the textbook by Ghallab, Nau, and Traverso (2004). In the dockyard robot domain, which is NP-hard, containers must be moved from their initial locations to their desired destinations via a robot that can carry only a single container at a time. The containers at each location form a stack from which the robot can only access the top container by using a crane that resides at the given location. Accessing a container that is not at the top of a stack therefore requires moving the upper container to a stack at a different location. The available actions are: *load* a container from a crane into the robot, *unload* a container from the robot into a crane, *take* the top container from the pile using a crane, *put* the container in the crane onto the top of a pile and *move* the robot between locations. There is a connection between all locations.

The load and unload actions have a constant cost of 0.01, accessing a pile with a crane costs 0.05 times the height of the pile plus 1 (to ensure non-zero-cost actions) and movement between locations costs the distance between locations. For these experiments, the location graph was created by placing random points on a unit square. The length of each edge was the Euclidean distance between the locations. Every location is connected to all other locations - the location graph is fully connected. All connections are undirected. The heuristic lower bound sums the distance of each container's current location from its goal location. We conducted these experiments on a configuration with 5 locations, cranes, and piles and 8 containers. We used a total of 50 instances and selected a hash function that maps states to buckets by ignoring all except the position of the robot and three containers. We used IEEE double-precision floating point to represent all costs.

The search space for dockyard robot planning forms a highly connected graph, and thus there are many ways to reach the same state. For example moving the robot from location A to B to C then back to A forms a cycle of length 3. Search algorithms that cannot remember duplicates will perform extremely poorly. Such is the case for IDA*$_{\mathrm{CR}}$, which failed to solve any instance within the time limit so we do not show results for it (in the sliding-tiles puzzle domain where IDA*$_{\mathrm{CR}}$is merely slow rather than failing catastrophically the shortest cycles are of length 12). PEDAL and BFHS-DDD were able to solve all instances. Table 3 shows a performance comparison between PEDAL and BFHS-DDD. Again, points below the diagonal represent instances where PEDAL had the faster solution time. We can see from the plot that all of the points lie below the $y = x$ line and therefore PEDAL outperformed BFHS-DDD on every instance.

These results provide evidence to suggest that a relaxed best-first search order is competitive in an external memory setting. It significantly reduces the number of nodes generated which corresponds to many fewer expensive I/O operations. BFHS uses a breadth-first search strategy to reduce the space complexity by removing the closed list. However, the performance bottleneck for the search problems examined in this section is the rapidly growing search frontier, not the closed list. Thus, the breadth-first search order of BFHS provides no advantage. In the next section, we show how PEDAL can be extended to outperform alternative approaches for large problems that have large branching factors and thus achieve a new state-of-the-art for the problem of Multiple Sequence Alignment.

4. External Memory Search With Large Branching Factors

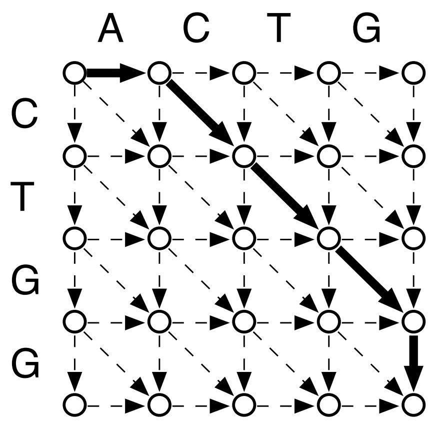
The branching factor for the sliding-tile puzzle is relatively small since there are few actions that can be taken from any state. Each time a node is expanded, the search generates at most 3 new nodes if the parent of a node is not generated as one of its children. In some domains there can be many possible actions to take at every state, resulting in a rapidly increasing search frontier. For domains with practical relevance, these actions can take on a wide range of costs and many of the new nodes that are generated are never expanded by the search because their costs exceed the cost of an optimal solution. For external memory search, this results in a lot of wasted I/O overhead as these nodes that are never expanded are read from and written to disk at each iteration of the search.

One real-world application of heuristic search with practical relevance (Korf, 2012) is Multiple Sequence Alignment (MSA). MSA can be formulated as a shortest path problem where each sequence represents one dimension in a multi-dimensional lattice and a solution is a least-cost path through the lattice. To achieve biologically plausible alignments, great care must be taken in selecting the most relevant cost function. The scoring of *gaps* is of particular importance. Altschul (1989) recommends *affine* gap costs, described in more detail below, which increase the size of the state space by a factor of 2^k for k sequences. Whereas the dockyard problem has a fixed set of actions, MSA has a large branching factor of $2^k - 1$, a value which increases rapidly as the number of sequences to be aligned grows. This means that the performance bottleneck for MSA is the memory required to store the frontier of the search.

Although dynamic programming is the classic technique for solving MSA (Needleman & Wunsch, 1970), heuristic search algorithms can achieve better performance than dynamic programming by pruning much of the search space, computing alignments faster and using less memory (Ikeda & Imai, 1999). Unfortunately, for challenging MSA problems, the memory required to store the open list makes A* impractical. Yoshizumi, Miura, and Ishida (2000) present a variant of A* called Partial Expansion A* (PEA*) that reduces the memory needed to store the open list by storing only the successor nodes that appear most promising. This technique can significantly reduce the size of the open list. However, like A*, PEA* is limited by the memory required to store the open and closed list and for challenging alignment problems PEA* can still exhaust memory.

One previously proposed alternative to PEA* is Iterative Deepening Dynamic Programming (IDDP, Schroedl, 2005), a form of bounded dynamic programming that relies on an uninformed search order to reduce the maximum number of nodes that need to be stored during search. The memory savings of IDDP comes at the cost of repeated search effort and divide-and-conquer solution reconstruction. IDDP forgoes a best-first search order and, as a result, it is possible for IDDP to visit many more nodes than a version of A* with optimal tie-breaking. Moreover, because of the wide range of edge costs found in the MSA domain, IDDP must rely on the bound setting technique of IDA*_{CR} (Sarkar et al., 1991). With this technique, it is possible for IDDP to visit four times as many nodes as A* (Schroedl, 2005). And, even though IDDP reduces the size of the frontier, it is still limited by the amount of memory required to store the open nodes. For large MSA problems, this can exhaust main memory.

Rather than suffer the overhead of an uninformed search order and divide-and-conquer solution reconstruction, we propose solving large MSA problems by using external memory search. In this section we present an extension of PEDAL, called Parallel External Partial Expansion A* (PE2A*), that combines the external memory search of PEDAL with the best-first partial expansion technique of PEA*. We compare PE2A* with in-memory A*, PEA* and IDDP for solving challenging instances of MSA. As in the previous section, the results show that parallel external memory best-first search can outperform serial in-memory search and is capable of solving large problems that cannot fit in main memory. Contrary to the assumptions of previous work, we find that storing the open list is much more expensive than storing the closed list. We also demonstrate that PE2A* is capable of solving, for the first time, the entire Reference Set 1 of the BAliBASE benchmark for MSA (Thompson, Plewniak, & Poch, 1999) using a biologically plausible cost function that incorporates affine gap costs. And just as with PEDAL, PE2A* shows that a relaxed external best-first search can effectively use heuristic information to surpass methods that rely on uninformed search orders.



Score Matrix:

	A	C	T	G	_
A	0	4	2	2	3
C		1	4	3	3
T			0	6	3
G				1	3
_					0

Alignment:

```
A C T G _
_ C T G G
```

Cost: 3+1+0+1+3 = 8

Figure 5: Optimal sequence alignment using a lattice and scoring matrix.

4.1 Multiple Sequence Alignment

We first discuss the MSA problem in more detail, including computing heuristic estimates. Then we review the most popular applicable heuristic search algorithms.

4.1.1 MSA as a Shortest Path Problem

DNA sequences are composed of just four nucleotides. A sequence of three nucleotides form a triplet codon. Protein sequences are composed of a sequence of amino acids, each of which are coded by one or more codons but the alignment techniques we discuss in this paper make no distinction. We use nucleotide sequences in our figures and examples for simplicity. The sequences used in our experiments are protein sequences.

In bioinformatics, alignments are computed in order to identify the most similar regions between two or more sequences of DNA. An alignment also represents the most likely set of changes necessary to transform one sequence into the other. One way to compute an alignment is to find an optimal placement of *gaps* in each of the sequences that maximizes the size of overlapping regions.

In the case of aligning two sequences, an optimal (most plausible) *pair–wise* alignment can be represented by a shortest path between the two corners of a two-dimensional lattice where columns and rows correspond to the sequences being aligned. An example is given in Figure 5. A move vertically in the lattice represents the insertion of a *gap* in the sequence that runs along the horizontal axis of the lattice. A move horizontally represents the insertion of a gap in the vertical sequence. Biologically speaking, a gap represents a mutation whereby one nucleotide has either been inserted or deleted; commonly referred to as an *indel.* A diagonal move represents either a conservation or mutation whereby one nucleotide has either been conserved or substituted for another.

In a shortest-path formulation, the indels and substitutions have associated costs; represented by weighted edges in the lattice. A solution is a least-cost path through the lattice. The cost of a path is computed by summing all indel and substitution costs along the path. The biological plausibility of an alignment depends heavily on the cost function used to compute it. A popular technique for assigning costs that accurately models the mutations observed by biologists is with a Dayhoff scoring matrix (Dayhoff, Schwartz, & Orcutt, 1978) that contains a score for all possible amino acid substitutions and gaps. Each value in the matrix is calculated by observing the differences in closely related proteins. Edge weights are constructed accordingly. It is straightforward to transform a score matrix into costs for shortest path solving. Figure 5 shows how the cost of an optimal alignment is computed using a cost matrix. This technique can be extended to a multiple alignment by taking the sum of all pair-wise alignments induced by the multiple alignment, also referred to as the standard *some-of-pairs* cost function.

The scoring of gaps is particularly important. Altschul (1989) found that assigning a fixed cost for gaps did not yield the most biologically plausible alignments. Biologically speaking, a mutation of n consecutive indels is more likely to occur than n separate mutations of a single indel. Altschul et al. construct a simple approximation called *affine* gap costs. In this model the cost of a gap is $a + b * x$ where x is the length of a gap and a and b are some constants. Figure 6 shows an example that incorporates affine gap costs. The cost of the edge E depends on the preceding edge; one of h, d, v. If the preceding edge is h then the cost of E is less because the gap is being extended.

A pair-wise alignment is easily computed using dynamic programming (Needleman & Wunsch, 1970). This method could be extended to multiple alignments (alignments of k sequences where $2 < k$) in the form of a k dimensional lattice. However, for alignments of higher dimensions, the N^k time and space required render dynamic programming infeasible. This motivates the use of heuristic search algorithms that are capable of finding an optimal solution by pruning much of the search space with the help of an admissible heuristic. While affine gap costs have been shown to improve accuracy, they also increase the size of the state space. This is because each state is uniquely identified not only by the lattice coordinates but also by the incoming edge, indicating whether a gap is being extended. This increases the state space by a factor of 2^k for aligning k sequences. Affine gap costs also make the MSA domain implementation more complex and require more memory for storing the heuristic.

4.1.2 Admissible Heuristics

To apply heuristic search to the most challenging MSA problems, it is imperative that we develop good heuristics. Admissible heuristics guarantee that A* will find optimal solutions. For a heuristic to be admissible it must always provide a lower bound on the cost to reach the goal from any state. Carrillo and Lipman (1988) show that the cost of an optimal alignment for k sequences is greater than or equal to the sum of the optimal alignments of a disjoint partitioning of the k sequences. Partitions are disjoint if any two sequences do not appear in more than one subset. Therefore, we can construct a lower

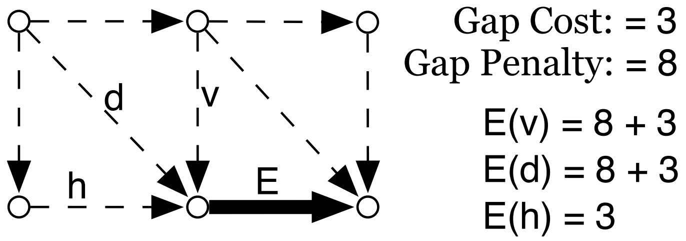


Figure 6: Computing edge costs with affine gaps. The cost of edge E depends on the incoming edge; one of h, d or v. If the incoming edge is v or d, then E represents the start of a gap, and incurs a gap start penalty of 8. However, if the incoming edge is h then E represents the continuation of a gap and incurs no gap start penalty.

bound on the cost of an optimal alignment of k sequences by partitioning the sequences into n disjoint subsets, computing an optimal alignment of each subset, and taking the sum of the n optimal alignments. Partitions with larger subsets yield stronger heuristics. For example, one possible partitioning of six sequences has a maximum size of two sequences per subset. A stronger heuristic uses a maximum size of three sequences per partition. Larger subsets require more memory to compute optimal alignments and thus a feasible partitioning depends on the amount of memory available.

Using the results above, we can construct a lower bound on the cost to complete a partial optimal alignment of k sequences by computing the optimal alignments of each subset in reverse; starting in the lower right corner of the lattice and finding a shortest path to the upper left corner using dynamic programming. The lattice computed by the reverse alignment stores the cost of a path from the goal to every coordinate and thus the cost-to-go from every coordinate to the goal. The forward lattice coordinates of a partial alignment are then used in all reverse lattices to look up the cost-to-go estimate. Figure 7 shows how to construct a cost-to-go estimate for aligning 3 sequences. All pairwise alignments ($[Seq_1, Seq_2]$, $[Seq_1, Seq_3]$ and $[Seq_2, Seq_3]$) are computed using dynamic programming in reverse order, starting in the bottom right corner and finishing in the top left. The values computed at each location in the corresponding lattices represent the cost to go from that location to the bottom right corner. The bottom right corner of Figure 7 shows how the cost-to-go is computed from the sum of the three reverse alignments (right) for the state in the lattice that corresponds to the partial alignment (left). A lower bound on the cost-to-go for completing the partial alignment of all 3 sequences is given by the sum of values in the corresponding locations in each lattice. Lermen and Reinert (2000) use this technique in a heuristic function for solving MSA with the classic A* algorithm. This heuristic is referred to as $h_{all,m}$.

Higher quality admissible heuristics can be obtained by computing optimal alignments with larger subsets. Unfortunately the time and space complexity of this technique make it challenging to compute and store optimal alignments of subsets of size $m > 2$. Kobayashi and Imai (1998) show that splitting the k sequences into two subsets and combining the

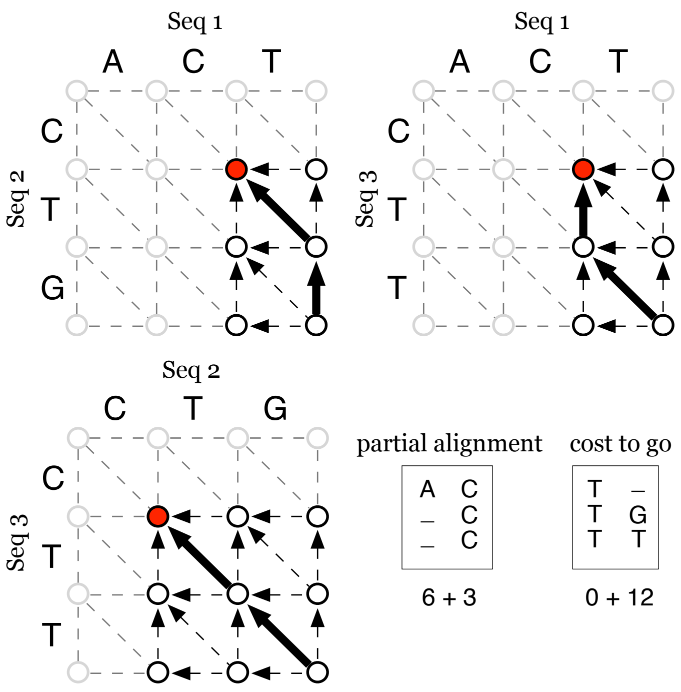


Figure 7: Computing cost-to-go by solving the lattice in reverse.

costs for the optimal alignments of the subsets with all disjoint pair-wise alignments between subsets is admissible. For example, given a set of sequences S we can define two subsets $S_1 \subset S$, $S_2 \subset S$ such that $S_1 \cap S_2 = \emptyset$. A lower bound on the cost of an optimal alignment of all sequences in S can be computed by taking the sum of the optimal alignments for S_1, S_2 and all pair-wise alignments for sequences $x \in S_1, y \in S_2$. This heuristic is referred to as the $h_{one,m}$ heuristic where *one* refers to the number of *splits*. The accuracy of the $h_{one,m}$ heuristic is similar to $h_{all,m}$ but it requires much less time and memory to compute. An example of the $h_{one,m}$ heuristic for six sequences would compute two 3-fold alignments, one for each subset, and nine pairwise alignments, one for each pair of sequences that are not contained in the same subset.

4.1.3 BAliBASE

Randomly generated sequences do not accurately reflect the sequences found in biology and provide no means of measuring the biological plausibility of the alignments that are produced. A popular benchmark for MSA algorithms is BAliBASE, a database of multiple sequence alignment problems specifically designed for the evaluation and comparison of multiple sequence alignment programs (Thompson et al., 1999). Of particular interest to our study is a set of instances known as Reference Set 1. Each instance in this set contains 4 to 6 protein sequences that range in length from 58 to 993. The sequences in this set are challenging for optimal MSA programs because they are highly dissimilar, requiring that much of the state space be explored to find an optimal alignment. To the best of our knowledge, no one has been able to compute optimal alignments for the entire Reference Set 1 using affine gap costs.

4.2 Previous Work

In this section we discuss previous work.

4.2.1 Iterative-Deepening Dynamic Programming

Iterative Deepening Dynamic-Programming (IDDP, Schroedl, 2005) is an iterative deepening search that combines dynamic programming with an admissible heuristic for pruning. IDDP uses a pre-defined search order much like the dynamic programming algorithm by Needleman and Wunsch (1970). The lattice state space is divided into levels. Levels can be defined by rows, columns or diagonals (lower-left to upper-right). IDDP proceeds by expanding nodes one level at a time. To detect duplicates, only the adjacent levels are needed. All other previously expanded levels may be deleted. This pre-defined expansion order reduces the amount of memory required to store open nodes: if levels are defined by antidiagonals then only k levels need to be stored in the open list during search, where k is the number of sequences being aligned. In this case the maximum size of the open list is $\mathcal{O}(kN^{k-1})$ for sequences of length N. This is one dimension smaller than the entire space which is $\mathcal{O}(N^k)$.

Because IDDP deletes closed nodes, divide-and-conquer solution reconstruction is required to recover the solution. As we will see later, deleting closed nodes provides a limited advantage since the size of the closed list is just a small fraction of the number of nodes generated during search.

IDDP uses a heuristic function, similar to A*, to prune unpromising nodes and in practice the size of the open list is much smaller than the worst case. At each iteration of the search, an upper bound b on the solution cost is estimated and only the nodes with $f \leq b$ are expanded. IDDP uses the same bound setting technique of IDA*$_{\text{CR}}$ (Sarkar et al., 1991) to estimate an upper bound that is expected to double the number of nodes expanded at each iteration.

In order to achieve memory savings, IDDP expands nodes in an uninformed pre-defined expansion order. In contrast to a best-first expansion order, a node in one level may be expanded before a node in another level with a lower f. As a result, it is possible for IDDP to expand many more nodes in the final f layer than A* with optimal tie-breaking. Like BFHS, the expansion order of IDDP approximates worst-case tie-breaking. The preferred tie-breaking policy for A* is to break ties by expanding nodes with higher g first. The g of a goal node is maximal among all nodes with equal f. Therefore, as soon as the goal node is generated it is placed at the front of its f layer in the open list and search can terminate on the next expansion from that layer. In contrast, IDDP will expand all non-goal nodes n with $f(n) \leq C^*$ where C^* is the cost of an optimal solution. This is because in the final iteration the bound is set to $b \geq C^*$ and all nodes n with $f(n) \leq C^* \leq b$ are expanded at each level. The level containing the goal is always processed last and therefore all non-goal nodes n with $f(n) \leq C^*$ must be expanded first.

The situation is even worse when relying on the bound setting technique of IDA*$_{\text{CR}}$. This technique estimates each bound in an attempt to double the number of expanded nodes at each iteration. Since the search order of IDDP is not best-first, it must visit all nodes in the final iteration to ensure optimality, which can include many nodes with $f > C^*$. Even if perfect doubling is achieved, it is possible for IDDP to visit four times as many nodes as A* (Schroedl, 2005).

Schroedl (2005) was able to compute optimal alignments for 80 of the 82 instances in Reference Set 1 of the BAliBASE benchmark using IDDP with a cost function that incorporated affine gap costs. Edelkamp and Kissmann (2007) extend IDDP to external memory search using sorting-based DDD but were only able to solve one additional instance, gal4 which alone required over 182 hours (7.5 days) of solving time.

4.2.2 Partial Expansion A*

When expanding a node, search algorithms typically generate and store all successor nodes, many of which have an f that is greater than the optimal solution cost. These nodes take up space on the open list, yet are never expanded. PEA* (Yoshizumi et al., 2000) reduces the memory needed to store the open list by pruning the successor nodes that do not appear promising i.e., nodes that are not likely to be expanded. A node appears promising to PEA* if its f does not exceed the f of its parent node plus some constant C. Partially expanded nodes are put back on the open list with a new f equal to the minimum f of the successor nodes that were pruned below it. We designate the updated f values as $F(n)$.

Yoshizumi et al. (2000) show that for MSA, PEA* is able to reduce the memory requirement of the classic A* algorithm by up to a factor of 100. The reduced memory comes at the cost of having to repeatedly expand partially expanded nodes until all of their successors have been generated. However, these re-expansions can be controlled by adjusting the constant C. With $C = 0$, PEA* will only store nodes with f value that is less than or equal to the cost of an optimal solution but will give the worst case overhead of re-expansions. With $C = \infty$ PEA* is equivalent to A* and does not re-expand any nodes. Yoshizumi et al. show that selecting a reasonable value for C can lead to dramatic reductions in the number of nodes in the open list while only marginally increasing the number of expansions.

PEA* is an effective best-first approach to handling problems with large branching factors such as MSA. However, it is still limited by the memory required to store open and closed nodes. This limitation motivates combining it with PEDAL.

4.3 Parallel External Memory PEA*

The second main contribution of this paper is an extension of PEDAL that combines the partial expansion technique of PEA* with HBDDD to exploit external memory and parallelism with bound estimation for robustness. We call this new algorithm Parallel External Partial Expansion A* (PE2A*). Like PEDAL, PE2A* maps nodes to buckets using a hash function and proceeds by iterating two phases: an expansion phase and a merge phase. PE2A* *partially* expands the set of frontier nodes n whose updated f values, $F(n)$ fall within the current upper bound. Partially expanded nodes are written back to the open list. Successor nodes that exceed the upper bound are generated but immediately discarded.

The pseudo code for PE2A* is given in Figure 8. PE2A* extends PEDAL by modifying the *ThreadExpand* and *RecurExpand* functions. An expansion thread proceeds by expanding all frontier nodes that fall within the current upper bound. Expansion generates

```
ThreadExpand(bucket)
43. for each $state \in Read(OpenFile(bucket))$
44.   if $F(state) \leq bound$
45.     $RecurExpand(state)$
46.   else $append(NextFile(bucket), state)$

RecurExpand(n)
47. if $IsGoal(n)$ $incumbent \leftarrow n$; return
48. $SUCC_p \leftarrow \{n_p | n_p \in succ(n), f(n_p) \leq F(n) + C\}$
49. $SUCC_q \leftarrow \{n_q | n_q \in succ(n), f(n_q) > F(n) + C\}$
50. for each $succ \in SUCC_p$
51.   if $f(succ) \leq bound$
52.     $RecurExpand(succ)$
53.   else
54.     $append(NextFile(hash(succ)), succ)$
55. if $SUCC_q = \emptyset$
56.   $append(ClosedFile(hash(n)), n)$
57. else
58.   $F(n) \leftarrow \min f(n_q), n_q \in SUCC_q$
59.   $append(NextFile(hash(n)), n)$
```

Figure 8: Pseudocode for the PE2A* algorithm.

two sets of successor nodes for each expanded node n; nodes with $f \leq F(n) + C$ and nodes with $f > F(n) + C$ (lines 48–49). Successor nodes that do not exceed the upper bound are recursively expanded. Nodes that exceed the upper bound but do not exceed $F(n) + C$ are appended to files that collectively represent the frontier of the search and require duplicate detection in the following merge phase. Partially expanded nodes that have no pruned successor nodes are appended to files that collectively represent the closed list (lines 55–56). Partially expanded nodes with pruned successor nodes are updated with a new F and appended to the frontier (lines 58–59). PE2A* approximates optimal tie-breaking by sorting buckets before each iteration according to the minimum F of all nodes in each bucket.

4.4 Empirical Evaluation

To determine the effectiveness of this approach, we compared A*, PEA*, IDDP, A*-HBDDD and PE2A* on the problem of multiple sequence alignment using the PAM 250 Dayhoff substitution matrix with affine gap costs and alignment problems from the BAliBASE Reference Set 1 benchmark. All experiments were run on *Machine-B*, a dual hexa-core machine (12 cores) with Xeon X5660 2.80 GHz processors, 12 GB of RAM and 12 320 GB disks. The files generated by the external memory search algorithms were distributed uniformly among all 12 disks to enable parallel I/O. We found that using a number of threads that is equal to the number of disks gave best performance. We tried a range of values for C from 0 to 500

	Threads	Expanded	Generated	Time
IDDP	1	163,918,426	474,874,541	1,699
A*	1	**56,335,259**	1,219,120,691	1,054
PEA*	1	96,007,627	**242,243,922**	1,032
A*-HBDDD	1	90,846,063	1,848,084,799	8,936
PE2A*	1	177,631,618	351,992,993	3,470
A*-HBDDD	12	90,840,029	1,847,931,753	1,187
PE2A*	12	177,616,881	351,961,167	**577**

Table 4: Results for the 75 easiest instances. Times reported in seconds for solving all 75 instances.

and found that 100 performed best. We found the bound setting technique of PEDAL did not provide a significant advantage over the standard layering used by A*-HBDDD because the use of the PAM 250 substitution matrix results in integer edge costs.

One advantage of an uninformed search order is that the open list does not need to be sorted. IDDP was implemented using an open list that consisted of an array of linked lists. We found IDDP to be sensitive to the number of bins used to represent the distribution that is used to estimate the next bound; 500 bins performed well. We stored all closed nodes in a hash table and to improve time performance did not implement the *SparsifyClosed* procedure (a procedure that removes unneeded nodes from the closed list) as described by Schroedl (2005). Since we did not remove any closed nodes, we did not have to implement divide-and-conquer solution reconstruction.

We used the pair-wise ($h_{all,2}$) heuristic to solve 80 of the 82 instances in the BAliBASE Reference Set 1 benchmark. This was primarily because the memory required to store the $h_{all,3}$ heuristic exceeded the amount of memory available. For example, instance 1taq has a maximum sequence length of 929. To store just one *3-fold* alignment we would need to store a matrix of 929^3 different values. Using affine gap costs each cell of the matrix requires storing $2^3 - 1$ values, one value for each possible gap opening in each sequence, for a total of approximately 21 GB, assuming 32-bit integers. The maximum sequence length for the hardest two instances is 573, allowing us to fit the $h_{one,3}$ heuristic in memory for those instances. For the 75 easiest instances, we used a hash function that used the lattice coordinate of the longest sequence. For all other instances, we used a hash function that used the lattice coordinates of the two longest sequences.

Table 4 shows results for solving the 75 easiest instances of BAliBASE Reference Set 1 (the instances that A* is able to solve) for comparison. The first set of rows are intended to show how the serial internal algorithms compare to each other and to show that our results are consistent with those reported elsewhere.

In the first set of rows we see that IDDP expands 1.7 times more nodes than PEA* and nearly 3 times more nodes than A*. The total solving time for IDDP is approximately 1.6 times longer than A* and PEA*. These results are consistent with results reported by Schroedl (2005) for alignments consisting of 6 or fewer sequences. We also see that the

	A*-HBDDD			PE2A*		
	Expanded	Generated	Time	Expanded	Generated	Time
2myr	**50**	761	38:31	122	**269**	**8:49**
arp	**64**	2,012	55:53	180	**417**	**17:05**
2ack	**209**	6,483	6:47:50	962	**2,884**	**1:45:32**
1taq	**1,328**	41,195	2:10:04:05	4,280	**8,856**	**8:19:04**
1lcf	**2,361**	148,793	3:02:13:07	11,306	**31,385**	**1:00:10:08**
ga14	**1,160**	35,963	2:01:31:55	3,632	**8,410**	**9:30:00**
1pamA	**14,686**	455,291	32:14:01:48	67,730	**173,509**	**9:08:42:39**

Table 5: Results for the 7 hardest instances of the BAliBASE Reference Set 1. Expanded and generated counts reported in millions and times reported in days:hours:minutes:seconds using 12 cores and 24 threads.

partial expansion technique is effective in reducing the number of nodes generated by a factor of 5. IDDP generates 1.96 times more nodes than PEA*.

In all instances, A* generates and stores many more nodes than it expands. For example, for instance 1sbp A* generates approximately 64,230,344 unique nodes while expanding only 3,815,670 nodes. The closed list is a mere 5.9% of all nodes generated during search. This means that simply eliminating the closed list would not significantly reduce the amount of memory required by search. PEA* stores just 5,429,322 nodes and expands just 5,185,887 nodes, reducing the number of nodes generated by a factor of 11.8 while increasing the number of nodes expanded by a factor of just 1.3.

In the second set of rows of Table 4, we show results for serial and parallel versions of A*-HBDDD and PE2A*. PE2A* expands nearly twice as many nodes as serial A*-HBDDD but generates about 5 times fewer nodes; as a result PE2A* incurs less I/O overall and is over 2.6 times faster than serial A*-HBDDD. The parallel versions of A*-DDD and PE2A* show good speedup, solving instances faster on average by a factor of approximately 8 and 6 respectively. Parallel A*-HBDDD is faster than IDDP and just 1.1 times slower than serial in-memory A*. PE2A* outperforms all other algorithms and is nearly 1.8 times faster than serial in-memory PEA* despite using external memory. It achieves a speedup of 6x on this 12-core machine. Again, it is exciting to see that external search can be faster than internal given that disk can be millions of times slower than RAM.

The external memory algorithms expand and generate more nodes than their in-memory counterparts for two reasons: 1) the search expands all nodes within the current bound a bucket at a time and therefore is not able to perform optimal tie-breaking and 2) recursive expansions blindly expand nodes without first checking whether they are duplicates and as a result duplicate nodes are expanded and their successors are included in the generated counts. However, we approximate optimal tie-breaking by sorting buckets, and recursive expansions do not incur I/O, so their effect on performance appears minimal.

Finally, Table 5 shows results for solving the 7 most difficult instances of BAliBASE Reference Set 1 using the scalable external memory algorithms, Parallel A*-HBDDD and

PE2A*. We used *Machine-A*, a dual quad-core (8 cores) machine with Intel Xeon X5550 2.66 GHz processors and 48 GB RAM, to solve the hardest two instances (gal4 and 1pamA). The additional RAM was necessary to store the $h_{one,3}$ heuristic. For all instances PE2A* outperforms A*-HBDDD by a significant margin and only PE2A* is able to solve the hardest instance (1pamA) in less than 10 days. To the best of our knowledge, we are the first to present results for solving this instance optimally using affine gap costs or on a single machine. The most relevant comparison we can make to related work is to External IDDP, which took over 182 hours (7.5 days) to solve gal4 (Edelkamp & Kissmann, 2007) with a slightly stronger heuristic. PE2A* takes just 9.5 hours and even A*-HBDDD takes just 2 days.

5. Discussion

We have explored best-first heuristic search in a parallel external memory setting. The results presented in previous sections provide evidence that approximating a best-first search order is beneficial. Alternative algorithms attempt to improve performance by omitting the closed list with techniques, such as IDDP, that rely on a non-best-first search order. However for many problems, such as MSA, the performance bottleneck is the rapidly growing frontier, not the closed list. While BFHS and IDDP avoid costly I/O by using the previous depth layer for duplicate checking, this savings is minimal compared to the savings afforded by a relaxed best-first search order, as with PEDAL and PE2A*. A predefined search order results in many more node expansions than best-first search. Moreover, divide-and-conquer solution reconstruction is required when omitting the closed list, contributing overhead and complexity to the search.

Algorithms that use an upper bound such as IDDP and BFHS enjoy the advantage of not having to generate and store nodes that exceed the upper bound. This can avoid costly I/O for an entire frontier layer. However, the use of such an upper bound often requires iterative-deepening search which can increase search effort and I/O overhead by a constant factor. Moreover, this advantage is quickly diminished when combined with the bound estimation technique of IDA*$_{\text{CR}}$. With a best-first search order and optimal tie breaking, the goal node is often expanded before many of the nodes in the final frontier layer are generated. Furthermore, the partial expansion technique of PE2A* also avoids generating many nodes in this layer, saving unnecessary and costly I/O. Another advantage of PEDAL and PE2A* is that they enjoy the benefits of recursive expansions. These expansions do not incur any I/O and allow for much faster solving times. IDDP and BFHS cannot benefit from recursive expansions.

The recursive expansion technique can give poor performance on domains where there are many paths to the same node with equal or similar cost. Because duplicate checking is deferred, many duplicate nodes may be generated that need to be stored in external memory and later accessed to perform merging. This can result in significant I/O overhead. One way to deal with this is to bound the depth of recursive expansions, and in effect bound the number of duplicate nodes that can be generated during the expand phase. Another technique, demonstrated in section 2, is to employ transposition tables during the expansion phase. Each thread keeps a local and bounded-size closed list in order to avoid generating many duplicate nodes.

If the closed list is large compared to the size of the open list, then techniques like BFHS and IDDP may provide superior performance. BFHS and IDDP need only the current and previous depth layers to check duplicates. Structuring the I/O to support this is straightforward because of the depth-layered scheme of these algorithms. Algorithms based on A*-HBDDD may perform unnecessary I/O if many of the nodes that are stored in the closed list are not needed for duplicate merging. However, the pruning technique of Sparse Memory Graph Search (Zhou & Hansen, 2003a) may provide the same performance benefit for A*-HBDDD based algorithms. We leave this for future work.

6. Related Work

In this section, we discuss other related work.

6.1 Parallel Frontier A*

Niewiadomski, Amaral, and Holte (2006) combine parallel Frontier A* search with DDD (PFA*-DDD). A sampling based technique is used to adaptively partition the workload at run-time. PFA*-DDD was able to solve the two most difficult problems (gal4 and 1pamA) of the BAliBASE benchmark using a cluster of 32 dual-core machines. However, affine gap costs were not used, simplifying the problem and allowing for stronger heuristics to be computed and stored with less memory. The hardest problem required 16 machines with a total of 56 GB of RAM. In their experiments, only the costs of the alignments were computed. Because Frontier Search deletes closed nodes, recovering the actual alignments requires extending PFA*-DDD with divide-and-conquer solution reconstruction. Niewiadomski et al. report that the parallelization of the divide-and-conquer strategy with PFA*-DDD is non-trivial.

6.2 DDD with Map-Reduce

The two phase (expand and merge) framework of DDD bares a striking resemblance to the two step (map and reduce) framework of MapReduce (Dean & Ghemawat, 2008). Reinefeld and Schütt (2010) applied MapReduce to frontier search. In this setting, the map step is analogous to the expand phase and the reduce step is analogous to the merge phase. Map and reduce jobs are distributed across multiple processors and nodes are read and written to a distributed file system. This distributed variant of DDD was used to generate the complete search space of the 15-puzzle (approximately 10 trillion states) in 66 hours utilizing 128 processors and is bound only by the speed and capacity of the I/O subsystems.

6.3 Sweep A*

Sweep A* (Zhou & Hansen, 2003b) is a space-efficient algorithm for domains that exhibit a partial order graph structure. IDDP can be viewed as a special case of Sweep A*. Sweep A* differs slightly from IDDP in that a best-first expansion order is followed within each level in the partially ordered graph. This helps find the goal sooner in the final level. However, if levels are defined by antidiagonals in MSA, then the final level contains very few nodes and they are all goals. Therefore, the effect of best-first sorting at each level is minimal. Zhou and Hansen (2004) combine Sweep A* with SDD on a subset of the BAliBASE benchmark without affine gap costs. In our experiments we compared with IDDP as the algorithms are nearly identical and results provided by Schroedl (2005) and Edelkamp and Kissmann (2007) for IDDP used affine gap costs and are thus more relevant to our study.

6.4 Enhanced Partial Expansion A*

In practice, PEA* generates all successor nodes and discards the nodes that do not appear promising. In some cases, it is possible to avoid some of the overhead of generating the unpromising successor nodes. This observation motivates Enhanced PEA* (EPEA*, Felner, Goldenberg, Sharon, Stern, Beja, Sturtevant, Schaeffer, & Holte, 2012). EPEA* is an enhanced version of PEA* that improves performance by predicting the cost of nodes, generating only the promising successors. Unfortunately it is not clear how to efficiently predict successor costs in MSA since the cost of an operator cannot be known *a priori*.

7. Conclusion

Previous approaches to scaling heuristic search, such as BFHS and IDDP, have focused on eliminating the closed list and abandoning best-first search order in an attempt to increase efficiency. However, the performance bottleneck of many practically motivated domains, such as the MSA problem, is the memory requirement for storing the frontier of the search, not the closed list. We feel that best-first search techniques such as A* have largely been overlooked in combination with external search. In this paper we showed that a best-first and relaxed best-first search orders provide significant advantages over rigidly predefined search orders. We presented the first empirical results for A*-HBDDD, studied its limitations and presented two extensions to address these limitations, making it more practical.

In section 2 we discussed techniques for external memory search in detail. We presented pseudo-code for A*-HBDDD and presented the first empirical results for the sliding-tile puzzle, comparing A*-HBDDD with internal memory search algorithms and BFHS. These results provide evidence that external-memory search benefits from a best-first search order and performs well on unit-cost domains and that efficient general-purpose parallel external-memory search can surpass serial in-memory search. Although many regard disk-based search as slow and unwieldy, we hope this result encourages practitioners to take another look at these techniques.

In section 3 we presented PEDAL, a new parallel external-memory search that combines ideas from A*-HBDDD and IDA*$_{\text{CR}}$ to address performance limitations for real problems. We proved that a simple layering scheme allows PEDAL to guarantee a constant I/O overhead. In addition, we showed empirically that PEDAL gives very good performance in practice, solving a real-cost variant of the sliding-tile puzzle more quickly than both IDA*$_{\text{CR}}$ and BFHS and it surpasses BFHS on the more practically motivated dockyard robot planning domain. PEDAL demonstrates that a relaxed best-first heuristic search can outperform alternative approaches for large problems that have real costs.

In section 4 we presented PE2A*, an extension to PEDAL that combines the partial expansion technique of PEA* with hash-based delayed duplicate detection to deal with problems that have large branching factors. We showed empirically that PE2A* performs very well in practice, solving 80 of the 82 instances of the BAliBASE Reference Set 1 benchmark with the weaker $h_{all,2}$ heuristic and the hardest two instances using the $h_{one,3}$ heuristic. In our experiments, PE2A* outperformed serial PEA*, IDDP and parallel A*-HBDDD and was the only algorithm capable of solving the most difficult instance in less than 10 days using the more biologically plausible affine gap costs.

Finally, we invite the reader to examine our source code (available at Hatem, Burns, & Ruml, 2014) to see that parallel external-memory search need not be a scary exotic nightmare. Given its scalability and advantages over IDA*, we hope it becomes more widely taught and used. We also include source code for the MSA domain with affine gap costs, a non-trivial implementation that can be used to evaluate other search algorithms in a practical setting.

Acknowledgments

We gratefully acknowledge support from NSF (grants 0812141 and 1150068) and DARPA (grant N10AP20029). We also thank Gene Cooperman, Rong Zhou and Rich Korf for helpful discussions and sharing code. Some of these results were previously reported by Hatem et al. (2011) and Hatem and Ruml (2013). Material and results in sections 1 and 2 including results for AIDA* and AIDA*$_{\mathrm{CR}}$, and the material in the Appendix have not been previously reported.

Appendix A. Appendix A. Implementing an External Memory Search

Here we discuss additional implementation details for disk-based search. We wrote all algorithms in Java, however, many of the implementation details are applicable to other programming languages. Source code is available at Hatem et al. (2014).

A.1 Data Structures

While the speed of an external search is dominated by the latency of I/O, the choice of internal data structures can have some effect on performance. The most critical data structures for an external search are the same for an internal search; a heap to manage the expansion order for buckets and a closed list to perform duplicate detection. For the open list we used a standard binary heap. For the closed list we used the High Performance Primitives Collection (HPPC) package, an alternative to the standard Java collections package. HPPC provides implementations of common data structures that make efficient use of memory and are written specifically for each primitive value type so there is no unnecessary overhead from casting. Specifically, we used LongByteOpenHashMap, a hash-table with open addressing and linear probing for collision detection. Keys are stored as long primitives and values as byte primitives in arrays.

A.2 Efficient I/O

For the disk-based algorithms we used Java's New I/O (NIO) for high performance I/O operations. Each bucket is backed by several files and each file has a corresponding NIO ByteBuffer. These buffers are not thread safe so access to the buffers must be synchronized.

We sized each buffer to be a multiple of the file system's block size. To minimize I/O we packed the state of each node into the fewest bits possible (without resorting to compression techniques). For example a 4x4 sliding-tiles puzzle state can be stored in just 60 bits. For each tile we record its location which falls within a range of 0 to 15, representable by just 4 bits. We can pack all values into a 64-bit integer. If the hash function uses the location of tiles (to ensure that duplicates fall within the same bucket) then all the states within a bucket share the locations of some common set of tiles. In this case you can reduce the number of bits required even further by storing the locations of the common tiles just once e.g., in the file name or file header. We found that reducing the amount of memory required per state on disk yielded some of the greatest speedups. These speedups benefited all external memory algorithms. We did not investigate the many possible compression techniques that might further reduce I/O per state. Many state spaces admit clever encoding or ranking techniques that reduce the number of bits required to store states (Schmidt & Zhou, 2011 and Bonet, 2008). Nor did we investigate optimizations at the device level such RAID arrays, short-stroking disks, solid-state disks or RAM clouds. We believe these are natural directions for future research.

A.3 Concurrency

We used Java's concurrency.utils package to manage multiple threads with instances of the ThreadPoolExecutor class. For each phase we create a list of tasks where each task corresponds to a bucket that needs to be processed. A ThreadPoolExecutor takes a list of task objects to process and each task is processed in parallel. For some algorithms, this list was sorted based on the cost of the nodes in the file — buckets with smaller f are processed first. The main search loop starts the expand phase by identifying all buckets that contain open nodes with an f equal to the current bound and adding an expand task to the list for each one. The expand task encapsulates information about which bucket needs to be expanded and keeps track of information such as the current minimum f of all nodes in the bucket, node expansion and generation counts and which other buckets have been given newly generated nodes. Buckets that received new nodes during the expand phase are marked with a dirty flag. Only dirty buckets are processed during the merge phase. A merge task is added to a new list for each dirty bucket. Following the expansion phase all dirty buckets are processed in a similar fashion. All threads are joined between phases.

A.4 Solution Recovery

Normally a solution is recovered by following parent pointers, starting from the goal node and continuing until the initial state is reached. However, with external memory search it is not possible to store pointers (in the form of memory references) to disk since the memory being referenced is freed once a node is written to disk. One way to recover solutions with external memory search is by recursively regenerating parent states, mapping the state to its respective bucket (file) and continuing until the initial state is reached. In order to regenerate parent states, the operator that generates each child state must be stored with its respective node. A parent state is regenerated by simply reversing the stored operator. The initial state is identified as not having a stored operator. Solution recovery cannot be done in parallel. However, the time spent recovering the solution is a tiny fraction of the total solving time.

Appendix B. Appendix B. Multiple Sequence Alignment

Multiple Sequence Alignment (MSA) is a challenging domain to implement, especially variants that incorporate state-of-the-art heuristics and scoring techniques like affine gap costs. We arrived at an efficient implementation after months of trial and error and borrowing some code from other implementations.

B.1 State Space Representation

We follow the popular choice of modeling the state space as an n-dimensional lattice, where n is the number of sequences being aligned. A state is simply an index into this lattice. Each component of the index can be interpreted as the number of elements in the sequence that have been considered. For example, imagine an alignment between the sequences AGTC and AGCC. The index $(2, 2)$ indicates that the first two elements from each sequence have been *aligned*. Additionally, the index $(1, 3)$ would indicate the first element from the first sequence has been aligned with the first three elements from the second sequence and thus contains two gaps. Identifying duplicate states is trivial.

Expanding a node involves computing the costs and heuristics for 2^d generated states where d is the number of sequences being aligned. This can become computationally intensive for large d. To reduce the total amount of computation required by each expansion, we use a state generation scheme based on grey codes. Each newly generated state is the result of a small incremental change to the previously generated state. This allows us to reuse cost and heuristic computations from previously generated states.

B.2 Computing Heuristics

We use the algorithm by Needleman and Wunsch (1970), a dynamic programming technique, to compute the heuristic, an estimate of the cost-to-go for every state. We used the popular table based implementation of Needleman-Wunsch whereby an n-dimensional table of values is computed one row at a time. Implementing this algorithm for two dimensional alignments is trivial and extending it to three dimensions is straight forward. However, incorporating affine gap costs can be challenging. Each entry in the lattice needs to store three different values, corresponding to whether a gap is being started or extended. A detailed explanation of how to implement affine costs is beyond the scope of this paper. For details we refer you to our source code.

References

Altschul, S. F. (1989). Gap costs for multiple sequence alignment. *Journal of Theoretical Biology*, *138*(3), 297–309.

Bonet, B. (2008). Efficient algorithms to rank and unrank permutations in lexicographic order. In *AAAI-Workshop on Search in AI and Robotics*.

Burns, E., Hatem, M., Leighton, M. J., & Ruml, W. (2012). Implementing fast heuristic search code. In *Proceedings of the Symposium on Combinatorial Search (SoCS-12)*.

Burns, E., Lemons, S., Ruml, W., & Zhou, R. (2010). Best-first heuristic search for multicore machines. *Journal of Artificial Intelligence Research*, *39*, 689–743.

Burns, E., & Ruml, W. (2013). Iterative-deepening search with on-line tree size prediction. *Annals of Mathematics and Artificial Intelligence*, *S68*, 1–23.

Carrillo, H., & Lipman, D. (1988). The multiple sequence alignment problem in biology. *SIAM J. on Applied Mathe- matics*, *48*(5), 1073–1082.

Dayhoff, M. O., Schwartz, R. M., & Orcutt, B. C. (1978). A model of evolutionary change in proteins. *Atlas of protein sequence and structure*, *5*(suppl 3), 345–351.

Dean, J., & Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107–113.

Edelkamp, S., Jabbar, S., & Schrdl, S. (2004). External A*. In *Advances in Artificial Intelligence*, Vol. 3238, pp. 226–240. Springer Berlin / Heidelberg.

Edelkamp, S., & Kissmann, P. (2007). Externalizing the multiple sequence alignment problem with affine gap costs. In *KI 2007: Advances in Artificial Intelligence*, pp. 444–447. Springer Berlin / Heidelberg.

Felner, A., Goldenberg, M., Sharon, G., Stern, R., Beja, T., Sturtevant, N. R., Schaeffer, J., & Holte, R. (2012). Partial-Expansion A* with selective node generation. In *Proceedings of AAAI-2012*.

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, *SSC-4*(2), 100–107.

Hatem, M., Burns, E., & Ruml, W. (2011). Heuristic search for large problems with real costs. In *Proceedings of AAAI-2011*.

Hatem, M., Burns, E., & Ruml, W. (2013). Faster problem solving in java with heuristic search..

Hatem, M., Burns, E., & Ruml, W. (2014). Research code for heuristic search. https://github.com/matthatem. Accessed June 28, 2014.

Hatem, M., Kiesel, S., & Ruml, W. (2015). Recursive best-first search with bounded overhead. In *Proceedings of AAAI-2015*.

Hatem, M., & Ruml, W. (2013). External memory best-first search for multiple sequence alignment. In *Proceedings of AAAI-2013*.

Ikeda, T., & Imai, H. (1999). Enhanced A* algorithms for multiple alignments: optimal alignments for several sequences and k-opt approximate alignments for large cases. *Theoretical Computer Science*, *210*(2), 341–374.

Kobayashi, H., & Imai, H. (1998). Improvement of the A* algorithm for multiple sequence alignment. In *Proceedings of the 9th Workshop on Genome Informatics*, pp. 120–130.

Korf, R. (2012). Research challenges in combinatorial search. In *Proceedings of AAAI-2012*.

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

Korf, R. E. (1993). Linear-space best-first search. *Artificial Intelligence*, *62*(1), 41–78.

Korf, R. E. (1999). Divide-and-conquer bidirectional search: First results. In *Proceedings of the Sixteenth International Joint Conference on Articial Intelligence (IJCAI-99)*, Vol. 99, pp. 1184–1189.

Korf, R. E. (2003). Delayed duplicate detection. In *Proceedings of the Eighteenth International Joint Conference on Articial Intelligence (IJCAI-03)*, pp. 1539–1541.

Korf, R. E. (2004). Best-first frontier search with delayed duplicate detection. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pp. 650–657. AAAI Press.

Korf, R. E. (2008a). Linear-time disk-based implicit graph search. *Journal of the ACM*, *55*(6).

Korf, R. E. (2008b). Minimizing disk I/O in two-bit breadth-first search. In *Proceedings of AAAI-08*, pp. 317–324.

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 Articial Intelligence (IJCAI-16)*, pp. 610–616.

Korf, R. E., Zhang, W., Thayer, I., & Hohwald, H. (2005). Frontier search. *Journal of the ACM*, *52*(5), 715–748.

Lermen, M., & Reinert, K. (2000). The practical use of the A* algorithm for exact multiple sequence alignment. *Journal of Computational Biology*, *7*(5), 655–671.

Needleman, S. B., & Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, *48*, 443–453.

Niewiadomski, R., Amaral, J. N., & Holte, R. C. (2006). Sequential and parallel algorithms for Frontier A* with delayed duplicate detection. In *Proceedings of AAAI-2006*, pp. 1039–1044. AAAI Press.

Reinefeld, A., & Schnecke, V. (1994). Aida*-asynchronous parallel ida*. In *Proceedings of the Biennial Conference-Canadian Society for Computational Studies of Intelligence*, pp. 295–302.

Reinefeld, A., & Schütt, T. (2010). Out-of-core parallel frontier search with mapreduce. In *High Performance Computing Systems and Applications*, pp. 323–336. Springer.

Russell, S. (1992). Effiicient memory-bounded search methods..

Sarkar, U., Chakrabarti, P., Ghose, S., & Sarkar, S. D. (1991). Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. *Artificial Intelligence*, *50*, 207–221.

Schmidt, T., & Zhou, R. (2011). Succinct set-encoding for state-space search. In *Proceedings of the Twenty-Fifth Conference on Artificial Intelligence (AAAI-2011)*.

Schroedl, S. (2005). An improved search algorithm for optimal multiple-sequence alignment. *Journal of Artificial Intelligence Research*, *23*, 587–623.

Thompson, Plewniak, & Poch (1999). BAliBASE: a benchmark alignment database for the evaluation of multiple alignment programs. *Bioinformatics*, *15*.

Yoshizumi, T., Miura, T., & Ishida, T. (2000). A* with partial expansion for large branching factor problems. In *Proceedings of AAAI-2000*, pp. 923–929.

Zhou, R., & Hansen, E. (2009). Dynamic state-space partitioning in external-memory graph search. In *The 2009 International Symposium on Combinatorial Search (SOCS-09)*.

Zhou, R., & Hansen, E. A. (2003a). Sparse-memory graph search. In *IJCAI*, pp. 1259–1268.

Zhou, R., & Hansen, E. A. (2003b). Sweep A*: Space-efficient heuristic search in partially ordered graphs. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI-03)*.

Zhou, R., & Hansen, E. A. (2004). Structured duplicate detection in external-memory graph search. In *Proceedings of AAAI-2004*.

Zhou, R., & Hansen, E. A. (2006). Breadth-first heuristic search. *Artificial Intelligence*, *170*(4–5), 385–408.