Heuristic Search for Large Problems With Real Costs

Matthew Hatem and **Ethan Burns** and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
mhatem, eaburns and ruml at cs.unh.edu

Abstract

The memory requirements of basic best-first heuristic search algorithms like A* make them infeasible for solving large problems. External disk storage is cheap and plentiful compared to the cost of internal RAM. Unfortunately, state-of-the-art external memory search algorithms either rely on brute-force search techniques, such as breadth-first search, or they rely on all node values falling in a narrow range of integers, and thus perform poorly on real-world domains with real-valued costs. We present a new general-purpose algorithm, PEDAL, that uses external memory and parallelism to perform a best-first heuristic search capable of solving large problems with real costs. We show theoretically that PEDAL is I/O efficient and empirically that it is both better on a standard unit-cost benchmark, surpassing internal IDA* on the 15-puzzle, and gives far superior performance on problems with real costs.

Introduction

Best-first graph search algorithms such as A* (Hart, Nilsson, and 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 generated states, in order to prevent duplicated search effort when the same state is generated via multiple paths. As the size of problems increases, however, the memory required to maintain the open and closed lists makes algorithms like A* impractical.

External memory search algorithms take advantage of cheap secondary storage, such as hard disks, to solve much larger problems than algorithms that only use main memory. A naïve implementation of A* with external storage has poor performance because it relies on random access and disks have high latency. Instead, great care must be taken to access memory sequentially to minimize seeks and exploit caching.

As we explain in detail below, previous approaches achieve sequential performance in part by dividing the search into layers. For heuristic search, a layer refers to nodes with the same lower bound on solution cost f. 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. The main contribution of this paper is a new strategy for external memory search that performs well on graphs with real-valued edges. Our new approach, Parallel External search with Dynamic A* Layering (PEDAL), combines A* with hash-based delayed duplicate detection (HBDDD, Korf 2008), however we relax the best-first ordering of the search in order to perform a constant number of expansions per I/O.

We compare PEDAL to IDA*, IDA*$_{\text{CR}}$ (Sarkar et al. 1991), A* with hash-based delayed duplicate detection (HBDDD-A*) and breadth-first heuristic search (Zhou and Hansen 2006) with delayed duplicate detection (BFHS-DDD) using two variants of the sliding tile puzzle 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 advances the state of the art by demonstrating that heuristic search can be effective for large problems with real-valued costs.

Previous Work

We begin by reviewing the previous techniques developed for internal and external memory search that PEDAL builds upon.

Iterative Deepening A*

Iterative-deepening A* (IDA*, Korf 1985) is an internal 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 size of iterations grows geometrically, then the number of nodes expanded by IDA* is $O(n)$, 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 schedule of IDA* may lead to only a few new nodes being expanded in each iteration. The number of nodes expanded by IDA* can be $O(n^2)$ (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 the pruned nodes during an iteration of search and uses it 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. If the successors of these pruned nodes are not expanded in the next iteration then this scheme is 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, branch-and-bound must be used on the final iteration of search to ensure optimality.

While IDA*$_{\text{CR}}$ can perform well on domains with real-valued edge costs, its estimation technique may fail to properly grow the iterations in some domains. IDA*$_{\text{CR}}$ also suffers on search spaces that form highly connected graphs. Because it uses depth-first search, it 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*.

Delayed Duplicate Detection

One simple way to make use of external storage for graph search is to place newly generated nodes in external memory and then process them at a later time. Korf (2008) presents an efficient form of this technique called Hash-Based Delayed Duplicate Detection (HBDDD). HBDDD uses a hash function to assign nodes to files. 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 considered.

Korf (2008) describes how HBDDD can be used with A* search (HBDDD-A*). The search proceeds in two phases: an expansion phase and a merge phase. In the expansion phase all nodes that have the current minimum solution cost estimate, f_{min}, are expanded then these nodes and their successors are stored in their respective files. If a generated node has an f less than or equal to f_{min} then it is expanded immediately instead of being stored to disk. This is called a *recursive expansion*. As we will see below, these are an important performance enhancement. Once all nodes with 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.

HBDDD may also be used as a framework to parallelize search (Korf 2008). 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. If two threads need to write nodes in to the same file, Korf (2008) states that this does not require an explicit lock at the user level as the operating system provides locking for file operations. While we could not find documentation specifying this behavior, the source code for the `glibc` standard library 2.12.90 does contain such a lock.

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). While, as we will see below, HBDDD-A* performs well on unit-cost domains, it suffers from excessive I/O overhead when there are many unique f values. HBDDD-A* reads all open nodes from files on disk and expands only the nodes within the current f bound. If there are 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 merging 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. This means that the number of I/O-efficient recursive expansions will be greatly reduced.

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. This is exactly what PEDAL does.

Parallel External Dynamic A* Layering

The main contribution of this paper is a new heuristic search algorithm that exploits external memory and parallelism and can handle arbitrary f cost distributions. It can be seen as a combination of HBDDD-A* and the estimation technique inspired by IDA*$_{\text{CR}}$ to dynamically layer the search space. We call the algorithm Parallel External search with Dynamic A* Layering (PEDAL).

Like HBDDD-A*, PEDAL proceeds in two phases: an expansion phase and a merge phase. PEDAL maps nodes to *buckets* using a hash function. Each bucket is backed by a set of four files on disk: 1) a file of frontier nodes that have yet to be expanded, 2) a file of newly generated nodes that have yet to be checked against the closed list, 3) a file of closed nodes that have already been expanded and 4) a file of nodes that were recursively expanded and must be merged with the closed file during the next merge phase. During the expansion phase, PEDAL expands the set of frontier nodes that fall within the current f bound. During the following merge phase, it tracks the distribution of the f values of the frontier nodes that were determined not to be duplicates. 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.

To save external storage with HBDDD, Korf (2008) suggests that instead of proceeding in two phases, merges may be interleaved with expansions. 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, 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. PEDAL uses recursive expansions and therefore it does not interleave expansion and merges.

```
SEARCH($initial$)
 1. $bound \leftarrow f(initial)$
 2. $bucket \leftarrow hash(initial)$
 3. $OpenFile(bucket) \leftarrow OpenFile(bucket) \cup \{initial\}$
 4. while $\exists bucket \in Buckets : min\_f(bucket) \leq bound$
 5.   for each $bucket \in Buckets : min\_f(bucket) \leq bound$
 6.     $JobPool \leftarrow JobPool \cup \{ThreadExpand(bucket)\}$
 7.   $ProcessJobs(JobPool)$
 8.   if $incumbent$ break
 9.   for each $bucket \in Buckets : NeedsMerge(bucket)$
10.     $JobPool \leftarrow JobPool \cup \{ThreadMerge(bucket)\}$
11.   $ProcessJobs(JobPool)$
12.   $bound \leftarrow NextBound(f\_dist)$

THREADEXPAND($bucket$)
13. for each $state \in OpenFile(bucket)$
14.   if $f(state) \leq bound$
15.     $RecurExpand(state)$
16.     $ClosedFile \leftarrow ClosedFile(bucket) \cup \{state\}$
17.   else $NextFile \leftarrow NextFile(bucket) \cup \{state\}$

RECUREXPAND($state$)
18. $f\_distribution\_remove(f\_dist, state)$
19. $children \leftarrow expand(state)$
20. for each $child \in children$
21.   $f\_distribution\_add(f\_dist, child)$
22.   if $is\_goal(child)$ and $f(incumbent) > f(child)$
23.     $incumbent \leftarrow child$
24.   else if $f(child) \leq bound$
25.     $RecurExpand(child)$
26.     $RecurClosedFile(hash(child)) \leftarrow$
            $RecurClosedFile(hash(child)) \cup \{state\}$
27.   else $NextFile \leftarrow NextFile(hash(child)) \cup \{state\}$

THREADMERGE($bucket$)
28. $Closed \leftarrow read(ClosedFile(bucket))$
29. for each $state \in RecurClosedFile(bucket)$
30.   if $state \notin Closed$
31.     $Closed \leftarrow Closed \cup \{state\}$
32.     $ClosedFile(bucket) \leftarrow$
            $ClosedFile(bucket) \cup \{state\}$
33. $OpenFile(bucket) \leftarrow \emptyset$
34. for each $state \in NextFile(bucket)$
35.   if $state \notin Closed$ or $g(state) < g(Closed[State])$
36.     $OpenFile(bucket) \leftarrow$
            $OpenFile(bucket) \cup \{state\}$
37.   else if $f(state) \leq f(Closed[state])$
38.     $Closed \leftarrow Closed - Closed[state] \cup \{state\}$
39.     $f\_distribution\_remove(f\_dist, state)$
```

Figure 1: Pseudocode for the PEDAL algorithm.

The pseudo code for PEDAL is given in Figure 1. PEDAL begins by placing the initial node in its respective bucket based on the supplied hash function (lines 2–3). The minimum bound is set to the f 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 5–7).

Recall that each bucket is backed by four files: $OpenFile$, $NextFile$, $RecurClosedFile$ and $ClosedFile$. 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 13–27). Nodes that are chosen for expansion are appended to the $ClosedFile$ for the current bucket (line 16). The set of $ClosedFile$s among all buckets collectively represent the closed list for the search. Successor nodes that exceed the bound are appended to the $NextFile$ for the current bucket (lines 17 & 27). 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 15 & 25). Nodes that are recursively expanded are appended to a separate file called the $RecurClosedFile$ (line 26) to be merged with the closed list in the next expansion phase.

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 flushed to disk.

If an expansion thread generates a goal state, its f value is compared with an incumbent solution, if one exists (line 22). If the goal has a better f than the incumbent, then the incumbent is replaced (line 23). If a solution has been found and there are no frontier nodes with f values less than the incumbent solution then PEDAL can terminate (line 8). 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 9–11).

In order to process a merge job, each thread begins by reading the $ClosedFile$ for the bucket into a hash-table called $Closed$ (line 28). Like HBDDD, PEDAL requires enough memory to store all closed nodes in all buckets being merged. The size of a bucket can be easily tuned by varying the granularity of the hash function. Next, all of the recursively expanded nodes from the $RecurClosedFile$, which were saved on disk in the previous expansion phase, are streamed into memory and merged with the closed list (lines 29–32). Then all frontier nodes in the $NextFile$ are streamed in and checked for duplicates against the closed list (lines 33–39). The nodes that are not duplicates or that have been reached via a better path are written back out to $NextFile$ so that they remain on the frontier for latter phases of search (lines 35–36). All other duplicate nodes are ignored.

Overhead

PEDAL uses a technique inspired by IDA*$_{CR}$ to maintain a bound schedule such that the number of nodes expanded is at least a constant fraction of the amount of I/O at each iteration. We keep 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 next bound is selected to be a constant fraction of the sum of nodes on the open and closed lists. Unlike IDA*$_{CR}$ which only provides a heuristic for the desired doubling behavior, the technique used by PEDAL is guaranteed to give only bounded I/O overhead.

We now confirm that this simple scheme ensures constant I/O overhead, that is, 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.

Lemma 1 *If e nodes are expanded and r extra nodes are recursively expanded then the number of I/O operations during the expand phase is $2o + eb + rb + r$.*

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 *If e nodes are expanded and r extra nodes are recursively expanded during the expand phase, then the number of I/O operations during the merge phase is at most $c + e + 2(r + eb + rb)$.*

Proof: During the merge phase we read at most $c + e$ closed nodes from disk. We also read r recursively expanded nodes and $eb + rb$ 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 *If e nodes are expanded during the expansion phase and r nodes are recursively expanded, then 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 algebra. □

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 by a constant fraction of the total number of I/O operations for some constant q.*

Proof:

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

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

Experiments

We evaluated the performance of PEDAL on three domains: the sliding tiles puzzle with two different cost functions and a dock robot planning domain. All experiments were run on a dual quad-core machine with Xeon X5550 2.66Ghz processors, 8Gb of RAM and seven 1Tb WD RE3 drives. The files for DDD-based algorithms were distributed uniformly among the 7 drives to enable parallel I/O.

We compared PEDAL to HBDDD-A*, BFHS-DDD, IDA*, IDA*$_{\mathrm{CR}}$, and an additional algorithm, breadth-first heuristic search (BFHS, Zhou and Hansen 2006). BFHS is a reduced memory search algorithm that 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 f bound at one depth before proceeding to the next depth. To prevent duplicate search effort Zhou and Hansen (2006) prove 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.

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, as discussed earlier, this technique fails on domains with many distinct f values. In the following experiments, we implemented a novel variant of BFHS using DDD and the IDA*$_{\mathrm{CR}}$ technique for the bound schedule. Also, since BFHS does not store a closed list, the full path to each node from the root is not maintained in memory and it must use divide-and-conquer solution reconstruction (Korf et al. 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 solution times.

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.

Sliding Tile Puzzle

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 DDD, we used a thread pool with 16 threads and 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.

In the unit cost sliding tile problem, we use the minimum f value out-of-bound schedule for both PEDAL and BFHS-DDD. The number of nodes with a given cost grows geometrically for this domain. Therefore this schedule will result in the same schedule as the one derived by dynamic layering for PEDAL. Without this optimization BFHS-DDD would require branch-and-bound in the final iteration. For all other domains, PEDAL explicitly uses dynamic layers and BFHS-DDD uses the technique from IDA*$_{\mathrm{CR}}$. PEDAL with a minimum f value out-of-bound schedule is the same as HBDDD-A*.

The first plot in Figure 2 shows a comparison between PEDAL and IDA*. The x axis show CPU time in seconds: points below the diagonal $y = x$ line represent instances that PEDAL solved faster than the respective algorithm. Data points represented by the '×' glyph at the edge of the plot represent instances where the corresponding algo-

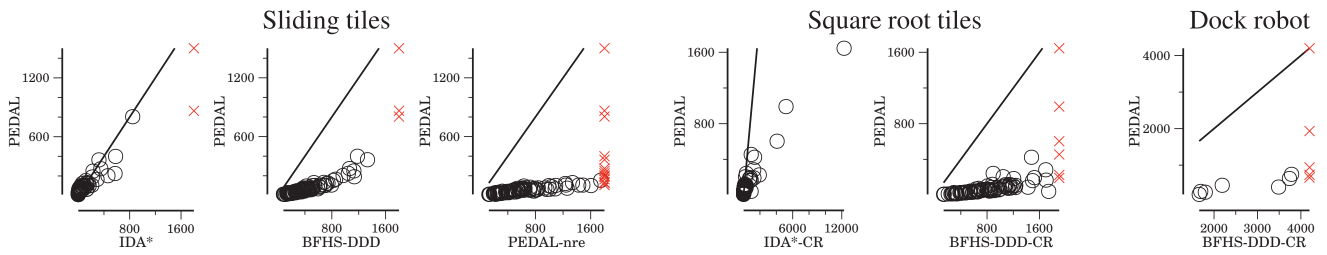

Figure 2: Comparison between PEDAL, IDA*, IDA*$_{CR}$, BFHS-DDD and PEDAL without recursive expansions.

rithm reached the time limit of thirty minutes without finding a solution. We can see from this plot that many of the data points reside below the $y = x$ line and therefore PEDAL outperformed IDA* on these instances. The distribution of the data points indicates that PEDAL had a larger advantage over IDA* as instances became more difficult. Finally, there are two points that reside on the extreme end of the x axis. These two points represent instances 82 and 88, which IDA* was unable to solve within the time limit. The main reason that PEDAL outperformed IDA* is because it is able to detect duplicate states and it benefits from parallelism.

The second plot for sliding tiles shows a comparison between PEDAL and BFHS-DDD. BFHS-DDD is unable to solve three instances within the time limit and the remaining instances are solved much faster by PEDAL. This can be explained by the performance on the last iteration of the search. In a unit cost domain with an admissible heuristic both PEDAL and BFHS can stop once a goal node is generated. However, because the h value of the goal is zero, BFHS will only generate goals at its deepest and last depth layer. Many nodes may have $f(n) = f^*$ that are shallower than the shallowest optimal solution. BFHS-DDD will expand all these nodes before it arrives at the depth layer of the shallowest goal. PEDAL has an equal chance of generating the goal as it expands any of its files in the last layer. PEDAL also benefits from recursive expansions, which require less I/O.

The third plot for sliding tiles shows a comparison between PEDAL with and without recursive expansions (PEDAL-nre). It is clear from this plot that recursive expansions are critical to PEDAL's performance.

The sliding tile puzzle is one of the most famous heuristic search domains because it is simple to encode and the actual physical puzzle has fascinated people for many years. This domain, however, 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 real-value costs that is simple, reproducible and has well understood connectivity, we created a new variant 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 center two plots in Figure 2 show results for a comparison with IDA*$_{CR}$ and BFHS-DDD using the technique from IDA*$_{CR}$ to schedule its bound on the square root version of the same 100 tiles instances.

The first square root tiles plot shows a comparison between PEDAL and IDA*$_{CR}$. This experiment did not include a time limit. IDA*$_{CR}$ actually solved the easier instances faster than PEDAL because it does not have to go to disk, however PEDAL greatly outperformed IDA*$_{CR}$ on the more difficult problems. The advantage of PEDAL over IDA*$_{CR}$ grew quickly as the problems required more time.

The second square root tiles plot compares PEDAL to BFHS-DDD. PEDAL clearly gave superior performance and increasing benefit as problem difficulty increased. As before, because BFHS-DDD tends to generate the goal in the deepest layer it may expand many nodes with f values equal to the optimal solution cost whose expansion is not strictly necessary for optimal search.

Dock Robot Planning

The sliding tiles 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 dock worker robot example used throughout the textbook by Ghallab, Nau, and Traverso (2004). In the dock robot domain, 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 adjacent 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 end points. 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, piles and 8 containers. A* is unable solve these problems within 8Gb of RAM. We used a total of 12 instances and a time limit of 70 minutes.

Because of the large number of duplicate states IDA*$_{CR}$ failed to solve all instances within the time limit so we do not show results for it. PEDAL was able to solve all but one instance in the time limit and BFHS-DDD solved all except for five. The instances that timed out are, again, represented using the '×' glyph. The right-most plot in Figure 2 shows a 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.

Overall, we have seen that PEDAL was able to solve the unit cost sliding tiles problems more quickly than both alternative approaches and it far surpasses existing methods on practically motivated domains with real costs.

Related Work

We briefly review some of the other approaches that have been taken to scaling heuristic search to handle large problems.

Recursive Best-first Search: RBFS (Korf 1993) is an alternative to IDA* that also uses an amount of memory that is linear in the maximum depth of the search space. For this reason, it also cannot handle domains with many duplicate states.

Frontier Search: Like breadth-first heuristic search, the frontier search algorithm (Korf et al. 2005) is a search framework that eliminates the need for storing a closed list while still preventing leak-back. Also, like BFHS, frontier search uses divide-and-conquer solution reconstruction and may require a substantial amount of memory to store the search frontier.

Frontier search can also be combined with f layers and DDD, however, Korf (2008) points out a problem where recursive expansions prevent this combination from correctly eliminating duplicates. Niewiadomski, Amaral, and Holte (2006) solve this issue in a distributed DDD implementation by simply omitting recursive expansions. However, our results clearly show that recursive expansions are critical when using external storage.

External A*: External A* (Edelkamp, Jabbar, and Schrdl 2004) combines A* with sorting-based DDD. Nodes of the same g and h values are grouped together in a bucket which maps to a file on external storage. It is not obvious how to dynamically inflate each bucket to handle real-valued costs.

Structured Duplicate Detection: Instead of delaying duplicate detection to a separate merge phase, structured duplicate detection (SDD, Zhou and Hansen 2004) performs duplicate detection as nodes are generated. SDD uses an abstraction function to determine the portion of the state space that is necessary to have in RAM for duplicate detection. PEDAL may also be implemented using SDD.

Conclusion

We have presented a general-purpose parallel external-memory search with dynamic A* layering (PEDAL) that combines ideas from HBDDD-A* and IDA*$_{\text{CR}}$. 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. It surpassed IDA* on unit-cost sliding tiles. In our experiments PEDAL was the only practical algorithm for square root sliding tiles and the dockyard robot domain. PEDAL demonstrates that best-first heuristic search can scale to large problems that have duplicate states and real costs.

Acknowledgments

We gratefully acknowledge support from NSF (grant IIS-0812141) and the DARPA CSSG program (grant N10AP20029). We thank Gene Cooperman and Rich Korf for helpful discussions.

References

Edelkamp, S.; Jabbar, S.; and Schrdl, S. 2004. External A*. In *Advances in Artificial Intelligence*, volume 3238. Springer Berlin / Heidelberg. 226–240.

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

Hart, P. E.; Nilsson, N. J.; and 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.

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

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

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

Niewiadomski, R.; Amaral, J. N.; and Holte, R. C. 2006. Sequential and parallel algorithms for frontier A* with delayed duplicate detection. In *Proceedings of the 21st national conference on Artificial intelligence (AAAI-06)*, 1039–1044. AAAI Press.

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

Zhou, R., and Hansen, E. A. 2004. Structured duplicate detection in external-memory graph search. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*.

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