External Memory Bidirectional Search

Nathan R. Sturtevant
Department of Computer Science
University of Denver
Denver, USA
sturtevant@cs.du.edu

Jingwei Chen
Department of Computer Science
University of Denver
Denver, USA
jingwei.chen@du.edu

Abstract

This paper studies external memory bidirectional search. That is, how bidirectional search algorithms can run using external memory such as hard drives or solid state drives. While external memory algorithms have been broadly studied in unidirectional search, they have not been studied in the context of bidirectional search. We show that the primary bottleneck in bidirectional search is the question of solution detection – knowing when the two search frontiers have met. We propose a method of *delayed solution detection* that makes external bidirectional search more efficient. Experimental results show the effectiveness of the approach.

1 Introduction

There is a long history of research into bidirectional search algorithms [Nicholson, 1966; Pohl, 1969], justified by the potential for an exponential reduction in the size of a bidirectional search over a unidirectional search. Despite this, bidirectional search is not the dominant approach for solving many search problems. There are many reasons behind this having to do with the quality of heuristics, the understanding of search algorithms, and the properties of the domains. Recent theoretical work has explained this in more detail.

Barker and Korf [Barker and Korf, 2015] suggested that the distribution of states in a search can be used to predict whether a bidirectional brute-force search should be used versus a unidirectional heuristic search. If the majority of states expanded are deeper than the solution midpoint, then the heuristic is weak and a bidirectional brute-force search should be used. If the majority of states expanded have depth that is prior to the solution midpoint, then the heuristic is strong, and unidirectional heuristic search should be used.

Holte et. al. [Holte *et al.*, 2016] refined this model, providing deeper insights into conditions under which different search algorithms will be successful. Their results suggest that there is a middle ground of problems where bidirectional heuristic search will be effective. Scaling the difficulty of a problem or the strength of a heuristic will alter the best approach — that is, what algorithm and heuristic will solve the problem most efficiently.

More research is needed to resolve these theoretical issues surrounding heuristics and the selection of the best algorithm to solve a particular problem. But, there is general agreement that on the largest problems, or those with weak heuristics, bidirectional search provides the best performance. So, if we wish to study and improve the state of the art in bidirectional search, it is important to develop techniques that scale the size of problems that can be solved.

Main memory is the primary bottleneck to solving larger problems with bidirectional search; algorithms are needed that can use external memory. External memory resources, such as hard drives and solid state drives, are often an order of magnitude or more larger than RAM, but require different algorithmic approaches since random access is not effective on these devices. When designed well, external memory algorithms have the potential to scale the size of problems solved without significant overhead from the external memory.

This paper makes the following contributions. First, this is, to our knowledge, the first paper that studies external memory bidirectional search, where the data structures for the bidirectional search are stored in external memory such as disk. Our work reveals that solution detection (testing whether the search frontiers have met) is an important component of external memory bidirectional search. We formalize the idea of delayed solution detection (DSD), which is necessary for efficient external-memory search. While our approach to external-memory search is general, we apply it specifically to the MM algorithm [Holte *et al.*, 2016], building parallel external-memory MM (PEMM) which can perform external-memory heuristic or brute-force search. We use PEMM to solve the Rubik's cube 'superflip' position [Rokicki *et al.*, 2014], one of the positions that is maximally distant (20 moves) from the goal. This is the first time, to our knowledge, that a 20-move Rubik's cube position has been solved with brute force search.

This paper combines ideas from two bodies of work. The first body of work includes algorithms for performing external memory search. The second body of work looks at algorithms for bidirectional search. We begin with background in these areas.

2 Background

To understand the challenges of external memory search, we can begin with a simple look at best-first search algorithms.

Algorithm 1 Generic Best-First Search

```
1: procedure BEST-FIRST SEARCH($start$, $goal$)
2:     Push($start$, OPEN)
3:     while OPEN not empty do
4:         Remove best state $s$ from OPEN
5:         if $s$ == $goal$ then return $success$
6:         end if
7:         Move $s$ to $closed$
8:         for each successor $s_i$ of $s$ do
9:             if $s_i$ on OPEN then
10:                Update cost of $s_i$ on OPEN if shorter
11:            else if $s_i$ not on $closed$ then
12:                Add $s_i$ to OPEN
13:            end if
14:        end for
15:    end while
16:    return $failure$
17: end procedure
```

A generic best-first search algorithm is illustrated in Algorithm 1. Best-first search uses some measure of 'best' that is used to order state expansions. Example measures of best are low g-cost (Dijkstra's algorithm) and low f-cost (A*).[1]

Best-first searches maintain two data structures: an OPEN list which holds states which have been generated but not yet expanded and a CLOSED list which holds states which have already been expanded. OPEN is usually a priority queue enhanced with a hash table for constant-time lookups, while CLOSED is also typically a hash table. (The same hash table can be used for both OPEN and CLOSED.)

The common operations on OPEN are to (1) check to see if a state is already on OPEN (line 9), (2) updates states on OPEN when a shorter path is found to a state already on OPEN (line 10), and (3) add new states to OPEN (line 12). The first operation is performed most often, motivating the need for hash tables and constant-time lookups.

Similarly speaking, common operations on CLOSED are to (1) check to see if a state is on CLOSED (line 11) and (2) add a state to CLOSED (line 7). With inconsistent heuristics [Felner *et al.*, 2011] states may also have to be moved from CLOSED back to OPEN, something we do not address.

For a best-first search to be efficient, the operations on OPEN and CLOSED must be efficient, which usually means constant-time random access to look up states. This is problematic if we run out of internal memory and want to use disk or other external memory to increase the size of a search, as it is not efficient to perform random access on external memory devices, even on solid state drives.

The primary bottleneck to using external memory is access latency; the time to start loading data. The latency of hard disks is currently at least an order of magnitude slower than RAM. But, once we begin reading, throughput is generally faster than we can perform computations. Thus, external memory access is usually batched so that the cost of the computation performed on the data retrieved from disk far outweighs the latency of loading that data.

[1]We assume the reader is familiar with the definitions of g-cost, h-cost, f-cost and node/state expansion and generation.

There are two general approaches that have been used to achieve this in practice. The first is delayed duplicate detection [Korf, 2004]. The second is a method of dividing the state space into buckets that fit into memory so that memory-sized chunks of the state space can be handled independently [Korf, 2004]. These approaches work together as follows.

Delayed duplicate detection works by skipping the duplicate detection steps in lines 9-13 of Algorithm 1. Instead of immediately performing duplicate detection, states are written to temporary files after being generated. Then, after all states of a given cost have been generated, they are loaded into memory and duplicate detection is performed. The duplicate detection can be done via sorting [Korf, 2004], hash tables [Korf, 2004], or other structured approaches that use RAM [Zhou and Hansen, 2004; Sturtevant and Rutherford, 2013]. Algorithms like frontier search [Korf *et al.*, 2005] make this process more efficient by eliminating the need to perform duplicate detection against the closed list.

Hash-based delayed duplicate detection has two general requirements. First, there needs to be a sufficient number of states with each cost to make it efficient to process each layer of the search independently. With a large number of states at each depth, the cost of the latency of reads is amortized over all of these states. If each layer in the search is small, the latency of reading the states may dominate the cost of the duplicate detection. In this case the best-first nature of the search can be relaxed to maintain efficiency [Hatem *et al.*, 2011]. The second requirement is that there be sufficient memory to perform the duplicate detection. Structured duplicate detection and hash-based delayed duplicate detection both use in-memory hash tables to find duplicates. Clearly, we cannot use these techniques on the whole state space at once, as this would eliminate the need for external memory. Instead, the state space must be divided into smaller buckets which can be processed independently. The bucket for a given state is usually determined via a hash function. The lower 5 bits of the hash, for instance, can be used to determine a state's bucket, dividing the whole state space into 32 buckets. Using such a schema, all duplicates will be written to the same bucket, guaranteeing that only a single file need be analyzed to find duplicates. Buckets can be further subdivided by other metrics such as g-cost.

2.1 Bidirectional Search

We consider a generic bidirectional search in Algorithm 2. This algorithm has the same overall structure as a best-first search, with the following changes. First, it maintains OPEN and CLOSED lists for each direction, which we distinguish with a subscript f (forward) or b (backward). States originating from the start are in the forward direction, while those originating from the goal are in the backwards direction. Second, it must check for solutions by finding the same state on the opposite open list, instead of performing a goal test against a goal state. Third, it does not always terminate immediately upon finding a solution. The search must continue until the best solution found thus far has been proven to be optimal. Example algorithms that follow this general structure include BS* [Kwa, 1989] and MM [Holte *et al.*, 2016].

BS* has other enhancements to prevent the search frontiers from passing through each other, which we do not show here.

Algorithm 2 Generic Bidirectional Search

```
1:  procedure BIDIRECTIONAL SEARCH($start$, $goal$)
2:      Push($start$, OPEN$_f$)
3:      Push($goal$, OPEN$_b$)
4:      while OPEN$_f$ and OPEN$_b$ not empty do
5:          Remove best state $s$ from OPEN$_f$ / OPEN$_b$
6:          if Can terminate search then return $success$
7:          end if
8:          if $s$ was on OPEN$_f$ then
9:              Move $s$ to CLOSED$_f$
10:             for each successor $s_i$ of $s$ do
11:                 if $s_i$ on OPEN$_f$ then
12:                     Update cost of $s_i$ on OPEN$_f$ if shorter
13:                 else if $s_i$ not on CLOSED$_f$ then
14:                     Add $s_i$ to OPEN$_f$
15:                 end if
16:                 if $s_i$ on OPEN$_b$ then // ISD
17:                     Update best solution
18:                 end if
19:             end for
20:         else
21:             // Analogous code in backwards direction
22:         end if
23:     end while
24: return $failure$
25: end procedure
```

The key difference from the point of view of the data structures used during search is the lookup that is required in OPEN in the opposite direction to check for solutions (Alg. 2 line 16). This lookup requires random access to the open list, which is fine for in-memory search, but will require changes for external-memory search. We call this approach *immediate solution detection* (ISD). We contrast this with *delayed solution detection* (DSD), which can span a broad range of different approaches. DSD refers to any approach that does not immediately check for solutions, but checks for solutions by the time a state is expanded and written to CLOSED.

Several previously proposed bidirectional search algorithms [Nicholson, 1966; Arefin and Saha, 2010; Sadhukhan, 2012] already perform DSD, checking for a solution when a state is expanded. This mimics how best-first search algorithms do not terminate until the goal is expanded. However, checking for a solution earlier in the search can allow earlier termination and improve performance [Holte *et al.*, 2016]. There is no reason not to use ISD when it can be implemented efficiently, because it can only cause the search to terminate earlier. As we will discuss, however, ISD is not efficient in external-memory search, and thus we must use DSD.

3 External Memory Bidirectional Search

As noted above, the key algorithmic difference between best-first and bidirectional search is that the bidirectional search must perform an additional lookup to check for states that are in OPEN in both the forward and backwards directions. If OPEN is stored on disk, this latency of this check will be too expensive to perform on a state-by-state basis.

Since, in general, the successors of a state can be found in any bucket of the state space, we design an external-memory algorithm to perform solution detection for many states at once to amortize the latency of reading from disk. Thus, the best time to perform solution detection is not when successors are being generated, as the successors will fall into many different buckets (and thus require solution detection across many files on disk). Instead, solution detection should be performed when a bucket of states is expanded, as we can perform DSD on all of these states simultaneously against the appropriate buckets in the opposite direction.

3.1 Parallel External-Memory MM (PEMM)

We can now describe our external-memory bidirectional search algorithm in detail. We base this algorithm on the recently introduced *meet in the middle* (MM) algorithm [Holte *et al.*, 2016], but similar modifications could be made to other external-memory bidirectional search algorithms. MM expands states by their priority, but uses a priority of $max(f, 2g)$ to ensure that each of the bidirectional searches do not expand any states past the midpoint of the search.[2] This priority function eliminates the need for BS*'s enhancements that ensure that a state is never expanded in both directions, because the MM search frontiers cannot pass through each other.

MM terminates when one of four termination conditions are met with respect to the best solution found thus far. If U is the cost of the best solution so far, the search can terminate when:

$$U \leq \max(min(prmin_F, prmin_B), \\ fmin_F, fmin_B, gmin_F + gmin_B + \epsilon)$$

where $prmin_F$ and $prmin_B$ are the minimum priority on OPEN in each direction, $fmin_F$ and $fmin_B$ are the minimum f-cost in each direction, $gmin_F$ and $gmin_B$ are the minimum g-cost in each direction, and ϵ is the smallest edge cost in the state space. The final termination condition is novel to MM, but cannot be applied with DSD; we will explain this in more detail later in the paper.

We call our algorithm Parallel External-Memory MM (PEMM). The changes required for MM to search efficiently with external memory are as follows. First, we divide OPEN and CLOSED into buckets and use hash-based delayed duplicate detection (DDD) [Korf, 2004] to detect and remove duplicates when a bucket is loaded into RAM. The buckets for OPEN and CLOSED are stored in files on disk; writes to OPEN are buffered and then appended to the appropriate bucket when the buffer is full. We only write to CLOSED once – after expanding the associated OPEN bucket.

PEMM stores a summary of the states in OPEN and CLOSED in an array in RAM. This array has one entry for each bucket on disk, and also maintains file pointers and other

[2]Roughly speaking, all states exactly half-way between the start and goal will have priority equal to the cost of the optimal solution due to the $2g$ component of the priority. Thus, all such states will be expanded, and the goal found, before states with higher priority (past the midpoint) are expanded. See [Holte *et al.*, 2016] for complete details.

structures. The array summarizing the OPEN list is monolithic, containing information about buckets for both directions of search. States are divided into buckets by (1) the priority of a state, (2) the g-cost of the state, (3) the search direction, (4) the lower i bits of the state hash function, and (5) the h-cost of a state. Every state in a bucket will have the same values for each of these attributes. Bucket expansions are ordered by priority (low to high), g-cost (low to high), search direction (forward then backward), and then by the hash function and h-cost. The ordering by hash and heuristic value does not influence correctness or efficiency of search. Ordering buckets by low to high g-cost may seem counter-intuitive, since it is the opposite of a typical A* ordering. This is important for PEMM, however, as it ensures that once we expand a bucket we will not generate any new states back into that bucket. Without this ordering we would be forced to process buckets multiple times as new states were re-added to the bucket, which is inefficient.

The second major change of PEMM from MM is the use of DSD in PEMM. Instead of performing ISD when a state is generated, PEMM performs solution detection when a bucket of states are expanded. DSD is performed by comparing against states in the open list in the opposite direction. We provide more details on this in the next section.

The final modifications of PEMM are that we cannot use the ϵ in MM's termination condition (see section 3.3). Also, while MM can search with inconsistent heuristics, the reopening of closed nodes and f-cost reductions associated with inconsistent heuristics may be inefficient with external memory search. Similarly, MM can handle 0-cost actions, which will be inefficient for PEMM (for the same reason why we process buckets from low to high g-cost). We leave a deeper study of these issue to further work; our implementation assumes a consistent heuristic and positive edge costs.

Algorithm 3 Parallel External-Memory MM

```
 1: procedure PEMM($start$, $goal$)
 2:     Push($start$, OPEN)
 3:     Push($goal$, OPEN)
 4:     while OPEN$_f$ and OPEN$_b$ not empty do
 5:         Choose best bucket $b$ from OPEN
 6:         if Can terminate search then return $success$
 7:         end if
 8:         Remove $b$ from OPEN
 9:         ReadBucket($b$);
10:         RemoveDuplicates($b$); // DDD
11:         WriteToClosed($b$);
12:         CheckForSolution($b$) // DSD done in parallel
13:         ParallelExpandBucket($b$)
14:     end while
15:     return $failure$
16: end procedure
```

The basic pseudo-code for PEMM can be found in Algorithm 3. We begin by pushing the start and the goal onto the OPEN list (on disk and in the summary data structure in RAM). In the primary search loop, the best bucket is chosen first. When selecting this bucket, we update the conditions for termination, and thus may discover that we can terminate with the best solution found thus far. (We may, for instance, discover that the minimum priority, f- or g-costs have changed.)

If the search does not terminate, we load the states from the best bucket into RAM, perform duplicate detection, and write the states back to the closed list. Duplicate detection within a bucket occurs when the bucket is loaded into a hash table, since the hash table will only store one copy of each state. Previous layers of the closed list are read sequentially, and any duplicate states are removed from the hash table.

We then, in parallel, expand all the states in the bucket, writing their successors to their respective buckets on disk[3], and perform solution detection on the states in the bucket (Algorithm 4). The expansion process uses multiple threads, with each thread using a simple hash function to determine what states from the bucket to expand. An additional enhancement, not shown in the pseudo-code, is that we can perform DDD for the next bucket concurrently with the expansion of the previous bucket. We do this as long as the g-cost and search direction of the buckets are the same. This approach is correct because the successors of the states in each of these buckets will have higher g-cost, and thus we will not fall into any bucket with the same g-cost as their parent.

Algorithm 4 Parallel Bucket expansion pseudo-code (Executed in parallel by many threads)

```
 1: procedure PARALLELEXPANDBUCKET($b$)
 2:     for every state $s$ in $b$ do
 3:         if current thread should expand $s$ then
 4:             for each successor $s_i$ of $s$ do
 5:                 add $s_i$ to open cache
 6:             end for
 7:             if open cache is full then
 8:                 flush cache to disk
 9:             end if
10:         end if
11:     end for
12:     flush cache to disk
13: end procedure
```

3.2 Delayed Solution Detection

Our pseudo-code does not provide the details of how DSD is performed; we provide these details here. Recall that our buckets are, in addition to other measures, determined by both the g-cost and a hash of the states in the bucket. We have a monolithic OPEN that stores this bucket information (but not the states) for all states in both the forward and backward direction. Given a particular bucket we only need to perform DSD with other buckets that are (1) in the opposite frontier, (2) have the same bucket hash, and (3) would form a complete path that is at least as long as the current lower-bound on the solution. For example, if the current priority is 10 and we are looking at a state with g-cost of 5, we only need to perform DSD with states in the opposite frontier with g-cost 5 or higher. Any bucket with g of 4 can be ignored during DSD; a duplicate between these buckets would lead to a solution cost 9, which is not possible given the current priority.

During DSD we have the current bucket stored in RAM in a hash table that we can use to test for membership. We

[3]States are first cached in memory.

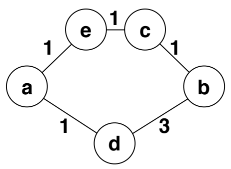

Figure 1: Failure of MM's ϵ rule in PEMM.

then load the states in the bucket in the opposite frontier that we are testing for DSD and individually check if they are in the current bucket. If they are, we have found a potential solution. One drawback of this approach is that we might not have performed DDD on the opposite frontier, so there may be duplicates or states from previous depths in the frontier, increasing the cost of DSD. Our experimental results suggest that it is better to wait and perform DDD later in the search.

DSD can be performed in parallel to node expansion while ensuring that all states have DSD performed appropriately. When the search is expanding in one direction we have two guarantees. First, since we expand from low to high g-cost and assume a consistent heuristic and non-zero edge costs, there cannot be new states added to the current bucket being expanded. Thus, when the current bucket is chosen for expansions, it contains all states that will ever be in that bucket. Second, because we only expand one direction at a time, the opposite frontier is completely static. Thus, DSD, which compares the current bucket to the opposite frontier, can be performed in parallel to the expansion process.

3.3 Termination

MM introduced a new termination condition that allows the search to stop early based on the minimum edge cost (ϵ) and minimum g-cost in OPEN in each direction. Unfortunately, this rule does not work in general with DSD. We illustrate this in Figure 1, where states a and b are the start and goal. All edges are marked with their costs; no heuristic is used.

In this example the search begins with a and b on OPEN. After each of these states are expanded, e and d will be on OPEN in the forward direction and c and d will be on OPEN in the backwards direction. Although d is on both OPEN lists, this has not yet been detected. Suppose we then expand e and then d in the forward direction. DSD will find the potential solution a-d-b with cost 4. At this point the minimum g-cost in the forward direction is 2 (c), and the minimum g-cost in the backward direction is 1 (also c). Using the ϵ rule we would conclude that we could terminate with an optimal solution cost 4, which is incorrect. The ϵ rule is justified with ISD because there are no paths through OPEN that have not been discovered already. Thus, new paths can only be found by adding new edges to existing paths. With DSD there can be undiscovered solutions on OPEN with cost $gmin_F + gmin_B$ such as c in this example, hence the ϵ rule cannot be used.

4 Proof of Correctness

We analyze the correctness of DSD here. While previous work has used DSD, MM has only been proven to be correct with ISD [Holte *et al.*, 2016]. Thus, we begin with the assumption of MM's correctness, and then show that delaying the solution detection cannot lead to termination with a suboptimal solution.

It is clear that for each direction of a bidirectional search, a state will pass monotonically through three phases: $ungenerated \rightarrow open \rightarrow closed$. We assume that solution detection will be performed at some point during the $open$ phase, but do not distinguish when.

Consider all such states, s^*, such that s^* has optimal g-cost in both directions, s^* is on an optimal path, and s^* is found on OPEN in both directions. Performing solution detection on s^* would find the optimal solution immediately. MM performs solution detection at the earliest possible moment - when s^* is first placed on OPEN. We show that until s^* is removed from OPEN and a state with higher priority is expanded, MM (without the ϵ termination condition) cannot terminate with a suboptimal solution. Since we perform solution detection on s^* before it is placed on CLOSED, MM with DSD will terminate with the optimal solution.

Recall the PEMM termination conditions:

$$U \leq \max(min(prmin_F, prmin_B), \\ fmin_F, fmin_B, gmin_F + gmin_B)$$

Let C^* be the cost of the (optimal) solution through s^*. We show that as long as s^* is on OPEN in both directions, the search cannot terminate with a solution cost $> C^*$. We examine the termination conditions one at a time and show that they will not be met.

Since the heuristic is admissible and s^* is on OPEN, $fmin_F, fmin_B \leq C^*$. This handles the second and third termination conditions. Given that s^* has optimal g-cost in each direction $g_F(s^*) \leq \frac{C^*}{2}$ or $g_B(s^*) \leq \frac{C^*}{2}$ where g_F and g_B are the respective g-costs in the forward and backward directions. (This holds because any state on an optimal path must be closer to the start, the goal, or exactly half way in between.) Without loss of generality, assume $g_F(s^*) \leq \frac{C^*}{2}$. In this case s^*'s bucket in the forward direction must have $prmin_F \leq C^*$. (Because $2g_F(s^*) \leq C^*$ and $f_F(s^*) \leq C^*$.) This handles the first termination condition. Finally, since s^* is on OPEN in both directions, $gmin_F + gmin_B \leq C^*$.

Thus, until s^* is removed from OPEN, we cannot terminate with a suboptimal solution. Since we will perform solution detection when removing s^* from OPEN, we are guaranteed to find the optimal solution even when performing DSD.

5 Experimental Results

It is an ongoing research question to study the performance of bidirectional search and to build algorithms and heuristics that can be used for efficient bidirectional search. Holte et. al. [Holte *et al.*, 2016] discuss the conditions under which a bidirectional search will be preferred to a unidirectional search. Our goal is to demonstrate that we can perform large searches and to study the characteristics of these searches. Our experimental results are in the domain of Rubik's Cube. In addition to solving the standard 10 Korf instances [Korf, 1997], we also solve the 'superflip' position, one of the known problems at depth 20. Our bidirectional

Table 1: PEMM results on Rubik's cube instances from Korf (0-9) and the superflip position (S).

		No heuristic						888 PDB Heuristic					
#	Depth	Time(s)	% Exp	% I/O	% DSD	# Exp.	Disk	Time(s)	% Exp	% I/O	% DSD	# Exp.	Disk
0	16	1,063	78.19	21.81	12.28	1.00	133GB	415	92.03	7.97	7.09	0.10	13GB
1	17	3,683	45.93	54.07	47.12	2.13	281GB	3,256	95.62	4.38	2.53	0.87	116GB
2	17	6,031	36.47	63.53	58.86	2.78	367GB	4,412	92.54	7.46	5.61	1.14	151GB
3	17	3,362	48.32	51.68	44.03	2.02	266GB	1,452	94.98	5.02	3.62	0.37	50GB
4	18	11,681	49.70	50.30	36.06	5.77	774GB	15,761	68.58	31.42	29.83	2.89	382GB
5	18	8,245	40.33	59.67	49.37	3.69	487GB	15,495	67.74	32.26	30.60	2.87	278GB
6	18	8,031	44.19	55.81	43.57	3.85	506GB	18,715	66.18	33.82	32.18	3.23	428GB
7	18	8,276	45.78	54.22	42.85	3.98	539GB	19,457	66.86	33.14	31.51	3.41	450GB
8	18	6,386	38.42	61.58	55.00	2.88	388GB	17,745	62.31	37.69	36.01	2.99	396GB
9	18	22,643	56.69	43.31	17.33	12.23	1.6TB	16,154	66.15	33.85	32.23	2.90	382GB
S	20	100,816	33.04	66.96	56.97	38.08	5.0TB	321,827	40.59	59.41	22.26	38.08	5.0TB

search does not use any special enhancements that are specific to Rubik's cube, and thus is, to our knowledge, the first fully general purpose algorithm to solve a depth 20 Rubik's cube instance. Our experiments are run on a 2.4 GHz Intel Xeon E5 with dual 8-core processors. The machine has 128 GB of RAM and has two 8TB disk drives on which the experiments are performed.

We solved all of these instances using PEMM as both a bidirectional breadth-first search ($h = 0$ for all states) and a bidirectional heuristic search. Our heuristic was the maximum value of three pattern databases [Culberson and Schaeffer, 1996]. The first is the 8-corner PDB, while the second two use 8-edges each (four of the edges overlap). While we could have exploited the symmetry in the Rubik's cube problem to only build the forward heuristics, we built separate reverse heuristic as well. So, a total of 10GB of RAM was used for the heuristics. The time to build the heuristic is not included in the results. The brute-force results all use 128 buckets, except for the superflip position which used 512. The heuristic search had a baseline of 128 buckets that were further divided by heuristic values, so the exact number of buckets varied on each problem instance.

The results are found in Table 1. For each Korf instance (0-9) and the superflip position (S), we report the solution depth, the total time required to solve the problem, the percentage of time spent doing node expansions (% Exp), the percentage of time spent doing **all** I/O operations including DDD and DSD (% I/O), the percentage of time doing DSD not in parallel with node expansions, but possibly parallel with other I/O (% DSD), the total number of node expansions in billions, and the disk space used at the end of the search.

There are several trends in the data. First, the brute force search expands states significantly faster than the heuristic search. This is because there are costs associated with looking up heuristics that slow the search down. Furthermore, the heuristics are not strong enough on hard problem to lead to significant amounts of pruning; in PEMM only heuristic values greater than the solution depth are useful for pruning [Holte *et al.*, 2016]. Finally, using too few buckets during the search has a significant impact on performance. Our initial experiments on the superflip position used only 128 buckets and took over twice as long to solve. Using 512 buckets on the other problem instances does not significantly impact performance.

There is an important point of comparison here. Our parallel implementation of AIDA* [Reinefeld and Schnecke, 1994] using the 888 heuristic required 215,800 seconds to solve this same problem with 116 billion node expansions. This is 3x more node expansions than PEMM and more than 2 times slower. Using a 9-edge heuristic (19 GB) the AIDA* search required 166,231 seconds and 104 billion node expansions. It is only with the 114 GB 10-edge heuristic that AIDA* performance surpasses the brute-force PEMM results, requiring 40,048 seconds and 24.6 billion node expansions. As more research is performed, the efficiency of PEMM can be expected to grow significantly, especially as we build better heuristics for PEMM.

One question we looked into was whether it would be more efficient to perform DDD and DSD between the priority layers of the PEMM search. (That is, after expanding all nodes of a given priority and before starting the next priority.) While this does reduce the number of nodes expanded, it is not faster. On Korf problem 3, PEMM expanded 30% more states but was 8.5 times faster. There are two reasons for this. First, DDD and DSD are I/O bound and do not parallelize well, so this reduces the overall efficiency. More importantly, it is more efficient to load a small frontier into memory than a large frontier. If a solution exists in the existing frontiers, we should load the smaller frontier into RAM to do this check. PEMM does this as part of its DSD when expanding the opposite frontier.

6 Conclusion and Future Work

This paper presents PEMM, a parallel external-memory bidirectional search algorithm. We identify that efficient solution detection is important for external memory bidirectional search and show that the PEMM can be used to solve large search problems, such as the depth 20 Rubik's cube superflip position. There is significant work to be done, including experiments in different domains, improving the parallelism of node expansions and disk I/O, finding the best ways to perform DSD, and investigating how more effective heuristics can be built for solving large problem instances.

7 Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1551406.

References

[Arefin and Saha, 2010] Kazi Shamsul Arefin and Aloke Kumar Saha. A new approach of iterative deepening bi-directional heuristic front-to-front algorithm (IDBHFFA). *International Journal of Electrical and Computer Sciences (IJECS-IJENS)*, 10(2), 2010.

[Barker and Korf, 2015] Joseph Kelly Barker and Richard E. Korf. Limitations of front-to-end bidirectional heuristic search. In *Proc. 29th AAAI Conference on Artificial Intelligence*, pages 1086–1092, 2015.

[Culberson and Schaeffer, 1996] Joseph Culberson and Jonathan Schaeffer. Searching with pattern databases. In *Proceedings of the 11th Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, volume 1081 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 1996.

[Felner *et al.*, 2011] A. Felner, U. Zahavi, R. Holte, J. Schaeffer, N. Sturtevant, and Z. Zhang. Inconsistent heuristics in theory and practice. *Artificial Intelligence (AIJ)*, 175(9-10):1570–1603, 2011.

[Hatem *et al.*, 2011] Matthew Hatem, Ethan Burns, and Wheeler Ruml. Heuristic search for large problems with real costs. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011*, 2011.

[Holte *et al.*, 2016] Robert C. Holte, Ariel Felner, Guni Sharon, and Nathan R. Sturtevant. Bidirectional search that is guaranteed to meet in the middle. In *AAAI Conference on Artificial Intelligence*, 2016.

[Korf *et al.*, 2005] Richard E. Korf, Weixiong Zhang, Ignacio Thayer, and Heath Hohwald. Frontier search. *J. ACM*, 52(5):715–748, 2005.

[Korf, 1997] Richard Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the 14th AAAI Conference on Artificial Intelligence*, pages 700–705, 1997.

[Korf, 2004] Richard E. Korf. Best-first frontier search with delayed duplicate detection. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 650–657, 2004.

[Kwa, 1989] James B. H. Kwa. BS*: An admissible bidirectional staged heuristic search algorithm. *Artificial Intelligence*, 38(1):95–109, 1989.

[Nicholson, 1966] T. A. J. Nicholson. Finding the shortest route between two points in a network. *The Computer Journal*, 9(3):275–280, 1966.

[Pohl, 1969] Ira Pohl. Bi-directional and heuristic search in path problems. Technical Report 104, Stanford Linear Accelerator Center, 1969.

[Reinefeld and Schnecke, 1994] Alexander Reinefeld and Volker Schnecke. AIDA*-Asynchronous Parallel IDA*. In *Proceedings of the Biennial Conference-Canadian Society for Computational Studies of Intelligence*, pages 295–302. Canadian Information Processing Soceity, 1994.

[Rokicki *et al.*, 2014] Tomas Rokicki, Herbert Kociemba, Morley Davidson, and John Dethridge. The diameter of the rubik's cube group is twenty. *SIAM Review*, 56(4):645–670, 2014.

[Sadhukhan, 2012] Samir K. Sadhukhan. A new approach to bidirectional heuristic search using error functions. In *Proc. 1st International Conference on Intelligent Infrastructure at the 47th Annual National Convention Computer Soceity of India (CSI-2012)*, 2012.

[Sturtevant and Rutherford, 2013] Nathan R. Sturtevant and Matthew J. Rutherford. Minimizing writes in parallel external memory search. In *Proc. 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, 2013.

[Zhou and Hansen, 2004] Rong Zhou and Eric A. Hansen. Structured duplicate detection in external-memory graph search. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence, July 25-29, 2004, San Jose, California, USA*, pages 683–689, 2004.