Direction-Optimizing Breadth-First Search with External Memory Storage

Shuli Hu[1] and **Nathan R. Sturtevant**[2]
[1]Northeast Normal University, Changchun, Jilin, China
[2]University of Alberta, Edmonton, AB, Canada
husl903@nenu.edu.cn, nathanst@ualberta.ca

Abstract

While computing resources have continued to grow, methods for building and using large heuristics have not seen significant advances in recent years. We have observed that direction-optimizing breadth-first search, developed for and used broadly in the Graph 500 competition, can also be applied for building heuristics. But, the algorithm cannot run efficiently using external memory – when the heuristics being built are larger than RAM. This paper shows how to modify direction-optimizing breadth-first search to build external-memory heuristics. We show that the new approach is not effective in state spaces with low asymptotic branching factors, but in other domains we are able to achieve up to a 3x reducing in runtime when building an external-memory heuristic. The approach is then used to build a 2.6TiB Rubik's Cube heuristic with 5.8 trillion entries, the largest pattern database heuristic ever built.

1 Introduction

Breadth-first search is a simple and commonly-used algorithm in many fields of Computer Science. Within Artificial Intelligence, breadth-first searches (BFS) have been used for verifying the properties of state spaces [Korf and Shultze, 2005; Korf and Felner, 2007] and, more commonly, for building heuristics [Culberson and Schaeffer, 1998]. For many domains, a breadth-first search in an abstract state space can be used as a heuristic in the full state space. Because large heuristics generally have better performance [Korf *et al.*, 2001], one line of research has been on building larger heuristics [Felner *et al.*, 2007; Sturtevant and Rutherford, 2013; Döbbelin *et al.*, 2013], such as pattern databases (PDBs) [Culberson and Schaeffer, 1998; Holte *et al.*, 2004]. Researchers have adapted specific BFS variants for use with external memory [Korf, 2008; Zhou and Hansen, 2011; Sturtevant and Rutherford, 2013] or domains with non-unit edge costs [Hatem *et al.*, 2011].

Outside of Artificial Intelligence, breadth-first searches are also commonly used, for example in the Graph500 competition. *Direction optimizing* BFS (DOBFS) [Beamer *et al.*, 2012] is a popular idea used in this area [Ueno *et al.*, 2016; Buluç *et al.*, 2017]. A typical BFS tests states with known depth to see if their depth can be propagated to neighboring states with unknown depth. DOBFS reverses this process by testing states with unknown depth and computing their depth if they have a neighbor (or predecessor in the case of a directed graph) with known depth.

This paper extends DOBFS by developing a novel external-memory variant that uses external memory resources (e.g. hard disks) to extend the size of problems that can be solved when using a single computer. When a problem being solved is larger than fits into memory, alternate methods, such as delayed duplicate detection [Korf, 2004], must be used to arrange the computation so it can be performed efficiently – a straightforward BFS is no longer possible because detecting duplicates requires random access to disk, which is too slow.

We analyze the design choices available for external memory algorithms and then develop direction-optimizing external-memory breadth-first search (DEBFS). The key challenge for DEBFS is how to randomly access the predecessors of a state when states are found on disk instead of in RAM. Our approach uses a *changed list* to load predecessors from the last iteration into RAM for efficient processing.

We then experiment with DEBFS in several different domains. Our results indicate that DEBFS is most effective in state spaces that grow and shrink quickly, and we are able to achieve up to a 3x speedup on the domains tested. This gain is significant because all states are still visited by the BFS, just more efficiently. We additionally show that in domains with high locality it may be possible to achieve further speedups. Finally, we use this technique to build the 2.6 TiB 6-edge 4-corner Rubik's Cube heuristic, which is, to our knowledge, the largest heuristic built.

2 Background and Related Work

Pattern Databases (PDBs) are a common form of heuristic that are built by abstracting the full state space of a problem and computing exact distances between states in the abstract state space. Distances in the PDB are typically computed using a BFS. The core operation in a BFS is to expand a state and to mark the depth of the successors of the state. When a PDB heuristic is typically built using a BFS, there are two common assumptions. The first is that the full PDB fits into memory. The second is that states in the PDB are stored *implicitly*. That is, instead of using a general hash table

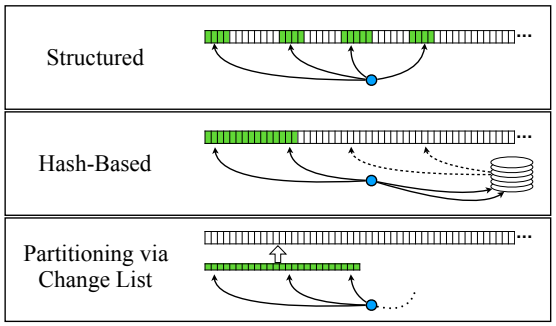

Figure 1: Approaches to writing values in external memory search.

to map abstract states to their depth, a perfect hash function, also called a ranking function [Myrvold and Ruskey, 2001], is used to uniquely and compactly map states into memory. This means that only the depth of a state needs to be stored, not the explicit representation of a state. If the depths are stored in an array, and the array is smaller than RAM, there are no significant complications to a simple BFS. However, if the array is larger than RAM, then the random access required to write the depth of the successors of a given state will break standard caching and virtual memory assumptions, causing the approach to fail. A BFS must be re-structured in order to efficiently load from and write data to disk during search.

External memory approaches to writing the successors of a state to disk are shown in Figure 1. The primary array at the top of each subfigure represents the array states in the PDB which are on disk. The subset of marked states in this array (green) represent states that are loaded in RAM. The blue circle represents the current state that is being expanded, and the arrows represent the successors of that state and where they belong on disk.

In *structured duplicate detection* [Zhou and Hansen, 2004], the PDB array is dynamically loaded and unloaded from memory in a way that ensures that all successors of state will always be in memory when they are generated, so the depth of the successors can be immediately written to disk. This approach requires a state space with suitable structure, something not addressed in this paper.

In *hash-based* duplicate detection [Korf, 2004; 2008] a portion of the PDB array on disk is loaded into RAM. Any successors that cannot be written to the portion of the PDB in RAM are temporarily written to files on disk. Later, when the memory associated with a given file is in RAM, the file is loaded and the successors are written to RAM. This approach uses additional disk storage to avoid expanding a state more than once. The hash-based approach was used by Two-Bit Breadth-First Search (TBBFS) [Korf, 2008]. TBBFS was originally designed only to perform a BFS, not to store the result of a BFS as a heuristic, but the general approach can still be used for building heuristics.

Partitioning approaches [Zhou and Hansen, 2007; Sturtevant and Rutherford, 2013] discard successors that cannot be written to memory immediately, re-expanding these states later. This approach in implemented in WMBFS with a *change list*, an implicit one-bit representation that marks states generated at the current level of the search. That is, one bit per state represents whether the state is generated at the current depth. Although the change list requires less memory than a depth array, it still may not fit in RAM. So, the change list is split into buckets and each bucket is handled separately. When the change list of one bucket is loaded into RAM, WMBFS scans all the states on disk at the current depth being expanded. The successors that do not fall into the change list of the current bucket are discarded. After all the states at the current depth are expanded, the entire change list is written back to disk. The full process is then repeated for the change list of each bucket. This approach avoids the additional disk used in the hash-based approach at the cost of expanding states multiple times. Despite the computational overhead, WMBFS was 3.5x faster than TBBFS when building the 12-edge Rubik's Cube PDB heuristic [Sturtevant and Rutherford, 2013].

3 External-Memory BFS

Let $N(d)$ be the number of states in a BFS at depth d. In a regular BFS, $N(d)$ states are expanded in order to find the $N(d+1)$ states at depth $d+1$. In a state space with a branching factor of b, this requires $b \cdot N(d)$ generations. However, if $N(d) \gg N(d+1)$, then this process is very inefficient, because either the same states at depth $d+1$ are generated many times, or states at depth d or $d-1$ are generated instead. In this case, it can be more efficient to expand the $N(d+1)$ states at depth $d+1$ to look for predecessors with known depth.

Consider, for instance, Table 4, which contains the distribution of states in the Rubik's Cube 6-edge 4-corner PDB. At depth 14 there are only 49 million states, while at depth 13 there are 847 billion states. Thus, it is much cheaper to expand the 49 million states that have unknown value and lookup the depth of their predecessors than it would be to expand the 847 billion states with depth 13 and check their 15 trillion successors. This paper addresses how to efficiently look up the depth for predecessors when they are stored in external memory.

In existing external-memory algorithms, the operation of writing the depth of a new state to disk has one-way information flow. Since all states being written share the same depth, only the identity of the new state is needed. Other information, such as the identity of the parent and the parent's location in memory, can be discarded. Thus, in both the hash-based duplicate detection and change-list based approaches, states are temporarily written to a data structure (either a file on disk or a change list) that discards parent information. This decoupling greatly simplifies the BFS algorithms.

DOBFS requires four steps: (1) finding a state s with unknown depth, (2) generating the predecessors of s, (3) looking up the depth of the predecessors, and (4) writing the depth of s if one of the predecessors of s has known depth. This is a simple approach when everything fits into RAM. But, when using external memory, things are more complex. This is because, unlike in other BFS approaches, the writing to disk in step (4) is contingent on the parent of the states that are being looked up in step (3). The search must be designed to efficiently keep all of this information in RAM.

Consider two possible approaches to DOBFS with exter-

nal memory. The first approach follows the structure of hash-based duplicate detection and has the aim of minimizing computation by only expanding each state once. This approach requires the following steps: (1) Find each state s with unknown depth. (2) Generate the predecessors of s. (3) If a predecessor is in RAM with known depth, then write the depth of s back to disk. (4) Otherwise, write the predecessors of s out to temporary file(s) along with the identity of s. (5) When processing the temporary files later in the search, if a predecessor with known depth is found then s can be written back out to disk. (6) Later, when s is in memory, the algorithm can write the final depth of s. This process is inefficient because predecessors of a state may not be readily available in RAM, and then when they are in RAM, the original state may no longer be in RAM. So, the efficiency depends crucially on whether predecessors will be found in RAM in steps (3) and (5). Additionally, it requires writing information about the parent of a state in step (4), which is inefficient on disk. If the state space was guaranteed to have suitable structure, it would be possible to dynamically load all data into RAM when needed. But, this isn't possible in many state spaces.

The main issue in adapting DOBFS to use external memory is that the predecessors of a state will not be in memory when needed. Thus, loading as many predecessors into memory as possible may avoid this bottleneck. Since the search only needs to know if a predecessor is at the previous depth, it is possible to use one bit per state to indicate whether a state has this property. We call any data structure that does this a *changed* list, as it marks states changed in the last iteration. This approach follows the structure of the *change* list in WMBFS and has the aim of minimizing writes to disk at the cost of extra computation. Each state will be written to disk at most once, but may be expanded multiple times.

Thus, we propose the following steps for external memory DOBFS: (1) Break the state space up into buckets, where each bucket fits as a changed list in RAM. (2) Load the next bucket from disk and mark states at the previous depth in the changed list in RAM. (3) Find all states with unknown value in the PDB and check if any of their predecessors can be found in the changed list. (4) If they are, write them back to disk at the next depth. (5) If not, discard the state and continue.

This approach is I/O efficient because states are read from disk linearly. The overall efficiency depends on how large the full change list is compared to RAM and the distribution of predecessors for each state. But, there is one advantage in the backward phase of a DOBFS over the forward phase. When doing a forward search, all successors must be generated to ensure they are given the next possible depth. When generating predecessors, however, we can stop as soon as a single predecessor has known depth, because that is sufficient to prove the depth of the state. For instance, if each state has 100 predecessors that are uniformly distributed in the PDB, then even if the changed list has to be broken up into 10 pieces, there is still a high likelihood that one of these successors can be found in the first changed list bucket that is processed, meaning that the states doesn't have to be re-expanded for the other 9 buckets.

Note that invertible operators are required for checking predecessors in step (3), but a perfect hash function is not required if illegal states can be identified from a state's rank.

Algorithm 1 Direction-Optimizing External-Memory BFS pseudo-code

```
DEBFS()
 1: Initialize disk files and other data structures
 2: while new states left on disk do
 3:    for all buckets do
 4:       Clear changed list, scan the disk files and mark open states
          from the current bucket in changed list
 5:       for every group G in coarse open list (in RAM) do
 6:          if G has unseen states then
 7:             for each state s_i in group G do
 8:                if s_i is unseen (on disk) then
 9:                   SendToWorkerQueue(s_i)
10:                end if
11:             end for
12:          end if
13:       end for
14:       Wait for worker threads to finish
15:    end for
16: end while

WORKERQUEUE()
 1: while true do
 2:    s ← GetWorkFromQueue()
 3:    for each predecessor p_i of s do
 4:       if p_i is marked in the changed list then
 5:          Write s with the next depth to local cache
 6:          break
 7:       end if
 8:    end for
 9:    if cache is full then
10:       Write local cache to disk files and update next coarse list
11:    end if
12: end while
```

3.1 Direction-Optimizing External-Memory BFS

We now provide a complete description of Direction-Optimizing External-Memory BFS (DEBFS). Any external-memory algorithm can be used for the layers until the point where the direction of the BFS switches. Thus, we consider DEBFS to just be the algorithm that runs in the reverse direction. Because of its efficiency in Rubik's Cube, we use WMBFS for the forward search.

Building on WMBFS, DEBFS uses two or more bits to represent each state; the experiments in this paper use four bits per state, so the following discussion makes that assumption. The value of each state is equal to the depth of the state modulo 15 or the reserved value, 15, which indicates that the state is *unseen*, meaning the depth has yet to be discovered. A state is *open* if its value equals the current depth modulo 15. States at the other depths are *closed*. Initially, all states are stored on disk with the reserved value 15 except the start state. The start state is initialized to depth 0, meaning it is *open*.

WMBFS breaks the state space into multiple buckets, with a *change list* for each bucket. Additionally, it stores a *coarse open list*. The coarse open list contains one bit for every group of N states in the PDB. A bit is set in the coarse open list if any of the N states it represents are currently open. In the main loop of the search the coarse open list is used to quickly

skip over portions of the PDB that do not contain open states. In each iteration, states that are open are expanded. If any of their successors fall into the change list of the current bucket, the successors are written there; other successors are discarded. After each pass, the change list is written to disk and then the process is repeated for the change list of the next bucket. When all buckets have been processed, the next depth is processed in the same manner.

When it would be efficient to switch directions, DEBFS converts the change list into a *changed list* for marking predecessors with known depths. While states on disk use 4 bits each, the changed list only requires one bit to indicate that a state was changed in the last iteration. Thus, the relevant states can be more efficiently loaded into memory. The process of finding states at the next depth is divided into one iteration per bucket. At the beginning of each iteration, a bucket is initialized in RAM and DEBFS scans the disk files to find and mark any states from the previous depth in the changed list. When generating the predecessors of a state with unknown depth, a two-level ranking function is applied to locate states in the changed list. The first-level ranking maps states to buckets and the second-level ranking maps to an offset in the changed list for a given bucket. The first-level ranking can be computed quickly; the second-level ranking is only computed if the state falls into the current bucket in RAM.

Pseudo-code for a parallel version of DEBFS is shown in the Algorithm 1. After loading the changed list, DEBFS scans all the states stored on disk sequentially, expanding unseen states (with value 15). The coarse open list marks whether there are *unseen* states among each group of N states on disk, one bit per group. These *unseen* states will be sent to a queue to be expanded. WMBFS expands the known states at current depth and generates the potential states at the next depth, whereas DEBFS expands the *unseen* states to see if they are a successor of a state with known depth. In DEBFS, if one predecessor of a state is marked in the changed list, we then know the depth of the unseen state, and that state can be written back to disk with the next depth modulo 15. Otherwise, the state will be discarded.

At each depth, both WMBFS and DEBFS need to make one iteration through external storage for the change(d) list of every bucket. With WMBFS, open states will be expanded as many times as there are buckets. In addition, since states at depth 1 and 16 have the same modulo 15 representation, states at depth 1 will also be expanded at depth 16 and likewise for similar depth pairs. (The coarse open list will prevent some of these re-expansions, but not all of them.) DEBFS performs differently depending on the nature of the unseen states. If a state is unseen but is at the current depth, then the state will be expanded one or more times. As soon as a single predecessor is found in the changed list, the depth of the state will be known and it will not be expanded again. States at later depths will be expanded as many times as there are buckets. The modulo representation does not impact DEBFS, because it only expands unseen states.

WMBFS writes the change list to disk after each bucket is completed, performing duplicate detection in the process. DEBFS, however, writes states to disk as soon as they are discovered, with no need for duplicate detection, since DEBFS cannot reach the same state by two different paths as other external-memory BFS algorithms do.

3.2 Parallel DEBFS

External memory search algorithms are typically parallelized, due to the scale of the computation being performed. The changed list is built sequentially at the beginning of each iteration. DEBFS then uses a main thread to read through states on disk looking for unseen states to be expanded. Unseen states are placed in a task queue for threads to handle. The worker threads retrieve states from the task queue, expand them, and then write them to local cache if they are successors of a state with known depth. When the cache is full these states are flushed to disk. This approach works efficiently because states are read and processed from disk linearly, and so they will also be written back to disk in an approximately linear order.

3.3 Switching Direction

The primary question that remains is when to switch directions from the forward to backwards calculation. Switching too early will be inefficient. Clearly if the search switched immediately, then all of the states in the state space would be expanded in the first iteration. Switching too late will reduce the gain from the approach. Previous analysis has allowed for switching back and forth between approaches [Beamer *et al.*, 2012], but due to the characteristics of PDBs, which typically grow and then shrink a single time, we build a simpler policy that will just switch to DEBFS when it looks advantageous. This policy has two considerations; the search switches direction if either criteria hold.

Expanding fewer states. The simplest policy is to switch to DEBFS if there are more states to be expanded at the current depth than there are remaining states in the state space with unknown depth.

Looking at fewer successors. At each iteration, a BFS must expand open states, generate their successors, and write any previously unseen successors to disk. Therefore, a BFS needs to generate and check all successors. DEBFS expands the unseen states, but it will stop generating predecessors once it finds a predecessor with known depth in the changed list. DEBFS generates all predecessors in the worst case that the given unseen state is not a successor of any states with known depth, or if the known state is the last predecessor. So, we estimate the effective branching factor in the search from the number of states at the previous two levels $(N(d)/N(d-1))$. The lower the effective branching factor, the higher the chance that fewer successors will be analyzed by DEBFS. When the effective branching factor is less than one, we switch to DEBFS.

3.4 Locality Optimization

The locality of a state space is determined by how close the successors of a state are to the parent in memory. In state spaces with high locality the depths of different states may not be randomly distributed across memory, and may instead be clustered in different parts of the state space. If a state space has this property, it is possible to take advantage of it

in DEBFS. In particular, suppose that most of the new states in the last iteration were written into bucket 3. Then, it would be advantageous to process bucket 3 first, since states at the next depth are more likely to have a predecessor in bucket 3. In general we can process buckets in order of the number of states written in the previous iteration. We will illustrate the benefit of this approach in the experimental results. Although the overall impact on the search is small, it is a simple optimization that may be beneficial when used in other domains.

4 Experimental Results

To evaluate the effectiveness of DEBFS we experimented with Rubik's Cube, Chinese Checkers, Top Spin, and the Pancake puzzle. We ran our experiments on a 16-processor 2.4GHz Intel Xeon E5 server with 128GB of RAM, two 8TB disk drives configured as a RAID drive, and one 1.5 TB SSD. We vary the memory and number of buckets used in each experiment, to simulate performance on machines with less RAM. The SSD was used for storage when possible, but the largest PDB required using the RAID.

An overview of our experiments can be found in Table 1. The first column is the domain studied. The second column is the search configuration for that domain. The third column is the number of buckets used in the search. The fourth column is the number of states in the largest level divided by the number of states at the next depth. The fifth column is the RAM used during the search. The sixth column is the disk storage required for search. This is followed by the running time of WMBFS and DEBFS in seconds and the overall speedup.

We will analyze the Rubik's Cube and Chinese Checkers results in detail, but we can make the following broad observations about the results. First, the larger the ratio between the largest layer and the following layer, the larger the performance increase from DEBFS. Although this is related to the branching factor, state spaces like Chinese Checkers and the pancake puzzle have large branching factors too; but with large numbers of duplicate states, the effective branching factor is smaller. Second, the relative gain of DEBFS seems to grow with more buckets. This is also observed in other experiments not presented here. Because some states have their depth resolved in earlier buckets, they do not need to be expanded again when later buckets are processed as in WMBFS.

4.1 Rubik's Cube Experiments

Rubik's Cube has a uniform branching factor of 18, and the state space grows and shrinks quickly, suggesting that this is an ideal domain for DEBFS. We built three PDBs for this domain, the 2.6 TiB 6-edge 4-corner PDB, the 0.4 TiB 12-edge PDB, and the 12 GiB 4-edge 4-corner PDB. The 6-edge 4-corner PDB encompasses half of the cubes, and is, to our knowledge, the largest PDB ever built. Due to the size of the PDB, we were unable to compare directly WMBFS, but analyzing the ratios in Table 1 suggests a 3-4x speedup over WMBFS. The distribution of states in this PDB is found in Table 4.

Detail about the 12-edge PDB can be found in Table 2. We report the number of expansions, successor checks, and time (in seconds) spent by DEBFS and WMBFS at depths 11-13. Both algorithms are running the same code from depths 0 to 10, so we omit these lines in the table.

There are 552,734,197,682 new states at depth 11, so WMBFS expands each of these states twice, once for each bucket. On the other hand, there are 305,116,412,392 remaining states at depth 11. DEBFS could potentially expand each of these twice, but in practice it only expands 346,712,437,361 states, 1.14 expansions per state. The performance is better because the majority of these states have their depth resolved once they are expanded the first time. Similarly, WMBFS has to check all 18 successors of each state, while DEBFS only has to check 7.26 predecessors of each state, on average, before the depth is found. WMBFS took 380,839s (4 days 10 hours) to build the PDB while DEBFS took 157,376s (1 day 20 hours), 2.4 times faster.

4.2 Chinese Checker Experiments

Next, we look in more detail at results in Chinese Checkers. Here, the goal is to find the shortest sequence to move a single player's pieces across the board. This can be used as an evaluation function for the two-player version of the game [Sturtevant, 2002; Roschke and Sturtevant, 2014; Schadd and Winands, 2011]. The board has 81 locations and 10 pieces, resulting in 1.8 trillion states. However, a symmetry reduction reduces the size of the state space to 1 trillion states. Chinese Checkers has several properties that are significantly different with Rubik's Cube. First, the successor generation is significantly more expensive. Second, the number of successors is significantly larger. While Rubik's Cube has a fixed branching factor of 18, the branching factor in Chinese Checkers can reach over 100. Finally, while there are relatively few duplicates in Rubik's Cube for most of the search, in Chinese Checkers there are far more duplicates at each level of the search, so the effective branching factor is much smaller.

Domain	Config.	Buckets	Ratio	RAM	Storage	Time (s) DEBFS	Time (s) WMBFS	Speed-up
Rubik's Cube	6 edges 4 corners (PDB)	6	4.33	112.42 GiB	2.6 TiB	4,341,500	-	-
	4 edges 4 corners (PDB)	2	3.38	1.51 GiB	12.0 GiB	41,456	127,933	3.09
	12 edges (PDB)	2	1.81	57.10 GiB	0.4 TiB	157,376	380,839	2.42
16 Top Spin	9 tiles (PDB)	6	1.93	0.08 GiB	1.93 GiB	8,248	17,820	2.16
		4	1.93	0.12 GiB	1.93 GiB	6,313	13,456	2.13
		2	1.93	0.24 GiB	1.93 GiB	4,326	9,038	2.11
15 Pancake	15 (full BFS)	2	1.45	76.12 GiB	0.6 TiB	515,622	871,463	1.69
Chinese Checkers	81 locations 10 pieces	2	1.08	62.44 GiB	0.5 TiB	1,730,669	1,717,056	0.99

Table 1: An overview of the experimental results

D	Expansions		Successor Checks		Time(sec)	
	DEBFS	WMBFS	DEBFS	WMBFS	DEBFS	WMBFS
11	346,712,437,361	1,105,468,395,364	2,518,799,329,265	19,898,431,116,552	38,575	178,419
12	331,065,363	609,572,153,252	1,949,741,604	10,972,298,758,536	15,130	99,930
13	249	660,671,036	1,402	11,892,078,648	1,813	5,311
Total	593,332,836,425	1,961,990,553,104	6,953,957,074,407	35,315,829,955,872	157,376	380,839

Table 2: 12-edge Rubik's Cube Expansions, Successor Checks and Time

D	Expansions		Successor Checks		Time(sec)	
	DEBFS	WMBFS	DEBFS	WMBFS	DEBFS	WMBFS
23	487,943,905,521	289,955,666,742	38,075,925,088,399	29,749,623,224,756	280,405	212,061
24	277,307,550,980	232,109,207,884	20,128,933,569,335	23,451,165,025,904	163,176	164,592
25	134,942,864,251	157,564,242,350	9,056,782,282,996	15,639,372,812,432	82,937	109,572
26	54,382,693,574	88,711,821,592	3,359,629,878,085	8,627,867,950,716	36,696	62,265
27	17,368,565,426	40,263,267,444	984,437,612,223	3,825,672,820,280	14,215	30,315
28	4,126,353,828	14,158,512,558	214,160,087,149	1,309,716,025,460	4,902	13,165
29	660,404,627	3,631,945,654	31,376,587,707	325,687,774,036	993	5,547
30	60,007,484	616,512,420	2,613,870,399	53,296,727,624	193	2,619
31	2,186,308	58,408,878	87,605,969	4,830,254,976	72	1,953
32	12,537	2,177,430	457,186	170,201,420	44	823
Total	2,295,250,927,892	2,145,528,146,308	211,386,018,917,772	222,519,474,695,928	1,730,669	1,717,056

Table 3: Expansions, Successor Checks and Time on Chinese Checkers (81-10)

Depth	Time	States
0	0.00	1
1	5.49	15
2	19.74	198
3	184.76	2,626
4	1,493.01	34,442
5	7,405.69	447,447
6	24,855.64	5,778,256
7	90,904.57	73,958,273
8	169,070.89	933,669,413
9	74,627.41	11,462,981,889
10	136,356.31	130,448,801,079
11	347,475.94	1,135,763,335,981
12	1,804,614.22	3,667,543,789,100
13*	1,417,564.07	847,721,302,874
14*	259,227.43	49,252,006

Table 4: Distribution of states in the Rubik's Cube 6-edge 4-corner PDB. Depth 13 & 14 use the direction-optimizing search.

Depth	WMBFS	DEBFS	DEBFS (ordered)
24	232,109,207,884	277,307,550,980	242,784,997,068
25	157,564,242,350	134,942,864,251	110,002,246,903
26	88,711,821,592	54,382,693,574	40,788,584,104
27	40,263,267,444	17,368,565,426	11,878,938,590
28	14,158,512,558	4,126,353,828	2,562,338,457
29	3,631,945,654	660,404,627	373,851,598
30	616,512,420	60,007,484	31,516,993
31	58,408,878	2,186,308	1,101,609
32	2,177,430	12,537	6,269

Table 5: Expansions in Chinese Checkers with bucket ordering.

Table 3 shows the number of expansions, successor checks, and time for each level in Chinese Checkers. For depths 25 and beyond, DEBFS is significantly faster than WMBFS, but these levels do not take a significant fraction of the entire search, so overall there is very little gain in Chinese Checkers. This is because the state space grows and shrinks very slowly. (See Sturtevant and Rutherford [2013] for the distribution of states.) However, Chinese Checkers has significant locality, as only one piece moves at a time on the board. So, we implemented bucket ordering according to the number of states written at the previous depth.

Table 5 shows the improvement relative to node expansions by this enhancement. Ordering the buckets provides up to a 2x reduction in node expansions over DEBFS without bucket ordering. As there are only 2 buckets in the state space, this is close to the maximum improvement possible. These gains do not provide a significant overall benefit in Chinese Checkers, but they illustrate the potential improvement in a domain that grows and shrinks like Rubik's Cube but has the locality of Chinese Checkers.

5 Summary and Future Work

This paper illustrates how the ideas of DOBFS can be extended to use external memory, resulting in a new algorithm, DEBFS. DEBFS uses a *changed* list to efficiently lookup the predecessors of a given state. Using this approach we were able to build a 2.6 TiB PDB encompassing half of the cubes in the Rubik's Cube. Our analysis shows that the effectiveness of DEBFS depends primarily on the ratio between the number of states at the largest level and the number of states at the level following that.

Now that we have built the largest PDB, our next challenge is how to use it. With the recent progress in machine learning, we plan to study how machine learning can be used to compress the data in this pattern database so that it can be used for solving problems on machines with limited memory.

Acknowledgements

This work was supported by the National Science Foundation under Grant No. 1551406, and the Fundamental Research Funds for the Central Universities under Grant No. 2412018ZD017 .

References

[Beamer *et al.*, 2012] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[Buluç *et al.*, 2017] Aydin Buluç, Scott Beamer, Kamesh Madduri, Krste Asanovic, and David A. Patterson. Distributed-memory breadth-first search on massive graphs. *CoRR*, abs/1705.04590, 2017.

[Culberson and Schaeffer, 1998] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[Döbbelin *et al.*, 2013] Robert Döbbelin, Thorsten Schütt, and Alexander Reinefeld. Building large compressed pdbs for the sliding tile puzzle. In *IJCAI Workshop on Computer Games*, pages 16–27, 2013.

[Felner *et al.*, 2007] Ariel Felner, Richard E. Korf, Ram Meshulam, and Robert Holte. Compressed pattern databases. *Journal of Artificial Intelligence Research*, 30:213–247, 2007.

[Hatem *et al.*, 2011] Matthew Hatem, Ethan Burns, and Wheeler Ruml. Heuristic search for large problems with real costs. In *AAAI Conference on Artificial Intelligence*, pages 30–35, 2011.

[Holte *et al.*, 2004] Robert C. Holte, Jack Newton, Ariel Felner, Ram Meshulam, and David Furcy. Multiple pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 122–131, 2004.

[Korf and Felner, 2007] Richard E. Korf and Ariel Felner. Recent progress in heuristic search: A case study of the four-peg Towers of Hanoi problem. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2324–2329, 2007.

[Korf and Shultze, 2005] Richard E. Korf and Peter Shultze. Large-scale, parallel breadth-first search. In *National Conference on Artificial Intelligence*, pages 1380–1385, 2005.

[Korf *et al.*, 2001] Richard E. Korf, Michael Reid, and Stefan Edelkamp. Time complexity of iterative-deepening-A*. *Artificial Intelligence*, 129(1–2):199–218, 2001.

[Korf, 2004] Richard E. Korf. Best-first frontier search with delayed duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, pages 650–657, 2004.

[Korf, 2008] Richard E. Korf. Minimizing disk I/O in two-bit breadth-first search. In *AAAI Conference on Artificial Intelligence*, pages 317–324, 2008.

[Myrvold and Ruskey, 2001] W. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79:281–284, 2001.

[Roschke and Sturtevant, 2014] Max Roschke and Nathan R. Sturtevant. UCT enhancements in chinese checkers using an endgame database. In Tristan Cazenave, Mark H.M. Winands, and Hiroyuki Iida, editors, *Computer Games*, pages 57–70, 2014.

[Schadd and Winands, 2011] Maarten P. D. Schadd and Mark H. M. Winands. Best reply search for multiplayer games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(1):57–66, 2011.

[Sturtevant and Rutherford, 2013] Nathan R. Sturtevant and Matthew J. Rutherford. Minimizing writes in parallel external memory search. *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 666–673, 2013.

[Sturtevant, 2002] Nathan R. Sturtevant. A comparison of algorithms for multi-player games. In *Computers and Games*, pages 108–122, 2002.

[Ueno *et al.*, 2016] Koji Ueno, Toyotaro Suzumura, Naoya Maruyama, Katsuki Fujisawa, and Satoshi Matsuoka. Extreme scale breadth-first search on supercomputers. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1040–1047, Dec 2016.

[Zhou and Hansen, 2004] Rong Zhou and Eric A. Hansen. Structured duplicate detection in external-memory graph search. In *National Conference on Artificial Intelligence (AAAI)*, pages 683–689, 2004.

[Zhou and Hansen, 2007] Rong Zhou and Eric A. Hansen. Edge partitioning in external-memory graph search. In *International Joint Conference on Artifical Intelligence (IJCAI)*, pages 2410–2416, 2007.

[Zhou and Hansen, 2011] Rong Zhou and Eric A. Hansen. Dynamic state-space partitioning in external-memory graph search. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 290–297, 2011.