Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence

Minimizing Writes in Parallel External Memory Search

Nathan R. Sturtevant and **Matthew J. Rutherford**
University of Denver
Denver, CO, USA
{*sturtevant, mjr*}@*cs.du.edu*

Abstract

Recent research on external-memory search has shown that disks can be effectively used as secondary storage when performing large breadth-first searches. We introduce the Write-Minimizing Breadth-First Search (WMBFS) algorithm which is designed to minimize the number of writes performed in an external-memory BFS. WMBFS is also designed to store the results of the BFS for later use. We present the results of a BFS on a single-agent version of Chinese Checkers and the Rubik's Cube edge cubes, state spaces with about 1 trillion states each. In evaluating against a comparable approach, WMBFS reduces the I/O for the Chinese Checkers domain by over an order of magnitude. In Rubik's cube, in addition to reducing I/O, the search is also 3.5 times faster. Analysis of the results suggests the machine and state-space properties necessary for WMBFS to perform well.

1 Introduction and Motivation

While it was once thought that hard disk drives are too slow for practical computational usage, work on external-memory search algorithms [Robinson *et al.*, 2007; Korf, 2008b; Zhou and Hansen, 2011] has shown that careful design of algorithms can avoid random read and writes and effectively use disks for large breadth-first searches (BFS). In particular, sequential disk access is far faster than random access, so disk can be effectively used as long as all disk accesses are performed sequentially. The use of disk extends the limits of problems that can be solved, as available disk is usually one to two orders of magnitude larger than available RAM, and the cost of disk is less than that of RAM.

This work describes a new algorithm for external memory search, Write-Minimizing Breadth-First Search (WMBFS), which makes the following contributions. (1) WMBFS works to minimize writes to external storage. (2) WMBFS stores the results of the computation for later use. (3) Several general techniques for reducing CPU usage are introduced, including a low-resolution open list, called a "coarse" open list. (4) Experimental results compare our WMBFS to the the existing state-of-art algorithm showing that we reduce I/O by over and order of magnitude. WMBFS has comparable execution speed in Chinese Checkers, and is 3.5 times faster in Rubik's Cube.

WMBFS is motivated by several observations:

First, most large-scale BFSs have been performed to verify the diameter (the number of unique depths at which states can be found) or the width (the maximum number of states at any particular depth) of a state space, after which any computed data is discarded. There are, however, many scenarios in which the results of a large BFS can be later used for other computation. In particular, a breadth-first search is the underlying technique for building pattern databases (PDBs), which are used as heuristics for search. PDBs require that the depth of each state in the BFS be stored for later usage. The largest PDBs have used both lossy and non-lossy compression [Felner *et al.*, 2004; Breyer and Korf, 2010] to bring the size of the PDB into that of available memory. The use of external memory directly for search has been limited [Schaeffer, 1997; Hatem *et al.*, 2011], but is an important potential application for this work.

Second, hard disk drives (HDD) have been the dominant form of media used for external memory search. But, prices have dropped and capacities have grown on solid states drives (SSDs), suggesting that SSDs may eventually replace HDDs, or they may also form an intermediate layer between external memory solutions, much as caches are used between the CPU and main memory. Because SSDs are not rotating disks, they do not have the same sequential restrictions as HDDs. So, their random read access is significantly faster than HDDs, but they also have a limited number of writes before they wear out [Ajwani *et al.*, 2009]. It is unclear if these limits will grow in the future, but they provide motivation behind the question of how to minimize writes in external memory search. Writes are strongly correlated with reads, as everything written to disk must later be read again, so minimizing writes is an effective way to reduce overall I/O. Data integrity is another concern, where increasing I/O increases the chance of a bit error and an incorrect computation.

Finally, there have been significant changes in available parallelism in the last few years. The number of processors available for computation is quickly growing, and efficient parallel algorithms are important for extending the size of problems that can be solved.

2 Background and Related Work

We describe a few existing external-memory approaches to performing a BFS in this section. We often describe the BFS approach inductively. That is, we provide the starting conditions of each iteration and show how the starting conditions of the next iteration are reached. Applied repeatedly this would complete an entire BFS.

We use the following definitions. *Expanding* a state means finding and applying all legal operators in order to *generate* all of its successors. A *ranking* function (or perfect hash function) converts a state into a into a unique integer, and an *unranking* function converts an integer back into the representative state. We assume that we have a perfect ranking function – if there are k possible states, then these states will rank to the values $0 \dots k - 1$.

A perfect ranking function enables efficient storage of information for all states in the state space using an *implicit* representation. Because the ranking is contiguous, an array or block of memory can store data for every state without the need to store the states themselves. The offset of data in the array or memory can be unranked to produce the state to which the data corresponds. This is contrasted by an explicit representation, where the rank of a state must be stored explicitly along with any associated data. An implicit representation will use less space when information about all (or a majority) of the states in the state space is saved. An explicit representation will use less space when only a small fraction of the state space is stored at any time. As an example, typical implementations of A* would use an explicit open list. Robinson et al. (2007) use the term *implicit open list* to refer to an open list which is stored implicitly.

2.1 Delayed Duplication Detection

Two delayed duplicate detection variants [Korf, 2004; 2008a] (DDD) have been proposed for external-memory frontier search. Frontier search [Korf, 2004] stores one bit per operator to mark operators that generate states in previous layers of the search. Avoiding these operators means that duplicates will only be found when generating states in the next layer of the BFS. DDD begins each iteration of the BFS with all states in the current layer stored explicitly on disk (in multiple files) without duplicates. The files for this layer are successively loaded, and states in the next layer are generated and written to new files on disk, with the previous files being removed. Duplicates are then removed from the new files on disk through sorting, or through loading into a memory-based hash table. A hash function is used to determine which file a state is stored in, meaning that all duplicates will be in the same file. From this point forward we refer to the hash-based DDD variant. After this processes is complete, the next layer is on disk without duplicates ready for further processing.

Structured Duplicate Detection. Structured Duplicate Detection [Zhou and Hansen, 2004] (SDD) is very similar to DDD, in that states are stored explicitly on disk in multiple files. But, instead of performing expansions and duplicate detection as separate steps, SDD performs these steps together. A state abstraction mechanism is used to detect where duplicates can occur. When a set of states on disk are ready for processing, all possible duplicates states are loaded into a memory-based hash table so that duplicates can detected before writing back to disk. Thus, SDD saves extra I/O, but it requires a state space with structure that can be exploited.

2.2 Two-Bit Breadth-First Search

Two-Bit Breadth-First Search (TBBFS) is a more general BFS technique that relies on fewer state-space properties than previous approaches [Korf, 2008b]. TBBFS begins each iteration of the BFS with all states in the state space stored on disk implicitly using two bits. Each state has one of four values: *closed*, *open*, *new* or *unseen*. *Closed* states are those that were already found in a previous layer of the search. *Open* states correspond to the states at the current depth that are being expanded. *New* states are the new successors of states marked *open*. *Unseen* states are states which have not been seen during search. Initially the start state is marked *open* and all other states are marked *unseen*. In general, an iteration begins with states marked either *closed*, *open*, or *unseen*; no states will be marked *new*.

If there is sufficient RAM, TBBFS loads the entire two-bit data from disk into RAM and then successively generates the successors of all *open* states. If any of these successors are currently marked *unseen*, they are changed to *new*; otherwise the successors are duplicates and can be ignored. After this process is complete, all *open* states are marked as *closed*, and all *new* states are marked as *open*, completing the iteration.

We use Figure 1 to illustrate how the search proceeds when there is insufficient RAM for the whole state space. The state space is broken into buckets; each bucket is the size of available RAM, assuming 8 bits per state in RAM. (This avoids the use of locks in the parallel implementation.) In step (1), a bucket is copied from external storage to RAM. In step (2) all states in RAM marked *open* are sent to threads to be expanded. The successors of these states are *local successors* if they fall into the same bucket that is currently in RAM. In step (3) the local successors are written directly back to RAM if they are marked *unseen* in RAM. The non-local successors are (4) written explicitly to temporary external storage, divided into files by buckets. If there are states in temporary external storage for the current bucket, written when a different bucket was in RAM, then these can be loaded and processed in step (5). After all states are processed, the updated information in RAM is (6) copied back onto external storage. This entire process is repeated for each bucket, after which any remaining states in temporary external storage are processed. Finally, all *open* states are marked as *closed*, and all *new* states are marked as *open*, completing the iteration.

While the two-bit portion of TBBFS is of fixed size, there are many states which are written explicitly to temporary external storage, depending on the locality of the successors of a state. Depending on the available size of disk, this may use up all available storage. If this occurs, the search must be paused while the disk files are immediately processed to free additional space on disk.

If the search crashes while in progress, it can be restarted as sufficient information to restart the search is contained in the two-bit representation. (As long as these files are not corrupted during a crash.) More details are in [Korf, 2008b].

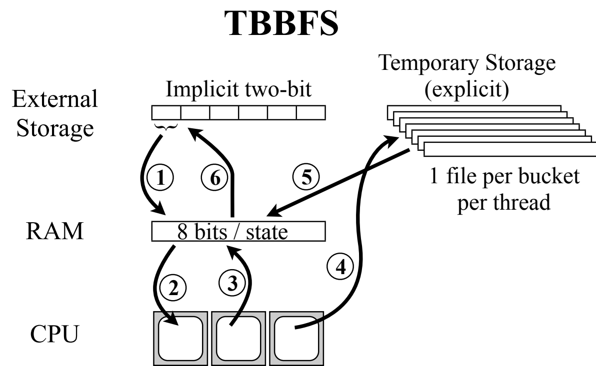

Figure 1: The TBBFS algorithm.

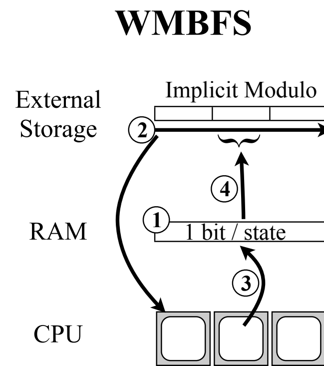

Figure 2: The WMBFS algorithm.

3 Write-Minimizing Breadth-First Search

We now propose a new algorithm, Write-Minimizing Breadth-First Search (WMBFS), which is designed to minimizing writes to external storage and to store the depth of each state in the BFS in addition to computing the diameter and width of the state space.

WMBFS begins each iteration with all states stored on disk implicitly using two or more bits per state. For illustration we will assume that four bits are used. The value stored on disk is the depth of the state modulo 15 ($0 \ldots 14$) or the reserved value, 15, indicating the state is *unseen* and not yet seen in the search. Any state that has a depth modulo 15 equal to the current depth of the search is considered to be *open*. States at all other depths are considered to be *closed*. Once all depths are computed, two bits are sufficient for recovering the true depth of any state [Breyer and Korf, 2010].

Like TBBFS, WMBFS divides the state space into buckets in order to accommodate a state space which is too large to be loaded directly into RAM. But, WMBFS uses a different in-memory representation and avoids writing successors to temporary external storage.

Figure 2 illustrates WMBFS. WMBFS begins (1) by initializing a 1-bit array, called a *change list* in RAM which is used for marking the successors of the current iteration; these states are potentially at the next depth of the BFS. The bits in the array implicitly correspond to states in one of the buckets on disk. After states in the array are initialized to *false*, (2) all states on disk are scanned sequentially. Any states which are *open* (at the current depth modulo 15) are sent to threads to be expanded. The successors of these states which fall into the bucket currently in RAM are (3) marked as *true* in the change list in RAM. All other successors are discarded. After all *open* states are expanded, the results can (4) be written back to a bucket on disk.

The change list does not use the same representation as the modulo representation on disk, so it cannot be copied directly back to disk. This means that duplicate detection takes place partially in RAM and partially when writing to disk. When many states share the same successor, it will only be stored in the change list once. But, the states marked in the change list can actually be in the previous, current, or next level of the search; this isn't detected until writing to disk. Writing occurs as follows. First, a sector of disk is read to a cache. Next, any states in the cache which are (1) marked in the change list and (2) have the value *unseen* are updated with the next depth in the search (modulo 15). Finally, the sector is written back to disk. Keeping track of the number of states actually written to disk at the next depth, as opposed to in the change list, indicates when the BFS is complete. (When no new states are written, the search is complete.)

At each depth of the search WMBFS must make one iteration through external storage (step 2) for every bucket (when the associated change list is in RAM). So, *open* states will be expanded as many times as there are buckets. But, WMBFS requires fewer buckets than TBBFS, because the representation of states in RAM is more efficient, reducing the number of iterations required. Note that states at depth 1 and 16 have the same modulo 15 representation, so in the most basic implementation of WMBFS all states at depth 1 will also be expanded at depth 16 and likewise for similar depth pairs.

3.1 WMBFS Enhancements

Two approaches are used to reduce the overhead of the modulo representation and the multiple passes through disk: a two-level ranking function and a *coarse open list*.

One common test performed in WMBFS is to see if a state falls into the bucket currently in RAM. Most ranking functions can be divided into multiple levels. The first level might contain the first few bits of the full ranking function, with the second level containing the remaining bits. The resulting first-level function can be used to map states to buckets. As this first-level ranking is much faster to compute than the full ranking, we can quickly determine whether a successor is stored in the current change list in RAM. If so, we then use the more expensive second-level ranking to determine its location in the change list; otherwise the state is discarded.

The *coarse open list* is a type of implicit open list, but instead of using one bit per state, it uses one bit for multiple states. This bit can be seen as marking sectors that contain states to be expanded in the next iteration, as opposed to marking individual states. The sector must then be read to find the individual states to be expanded. This significantly reduces the memory overhead of the implicit open list. In our later experiments, for instance, we use 1 bit per 256 states. So,

if any of the first 256 states in the state space were changed in the previous iteration, the first bit in the coarse open list will be set.

The coarse open list offers several advantages. First, WMBFS can avoid reading any sectors from disk which are not marked in the coarse open list. This is particularly noticeable on the first iteration when a single state is written to disk, but a naive implementation would read through all states on disk looking for states at the current depth. Next, the coarse open list reduces the overhead associated with a modulo representation of the open list. There can be many states with the same modulo depth, but any states with the same modulo depth not in the coarse open list will be ignored. Finally, the coarse open list can be updated during iterations through each bucket, reducing the cost of multiple iterations. If all successors of states represented by one bit of the coarse open list fall inside the current or previous buckets, then the associated bit in the coarse open list can be cleared.

3.2 Parallel WMBFS

The parallel implementation of WMBFS uses a single master thread and multiple worker threads. The master thread initializes the data structures for search and then reads through the buckets on disk looking for *open* states. These states are sent to worker threads to be expanded. The worker threads are responsible for taking these states, expanding them, and writing the results back to the change list. Because the change list is shared between threads, the results are temporarily cached in RAM in order to reduce the overhead used by the locks. Threads are re-synchronized after each pass through disk (step 2 in Figure 2) is complete.

Pseudo-code for parallel WMBFS is found in Algorithm 1. This high-level code does not show some details of the implementation, such as load balancing for threads, and the full details of caching and writing to the change list.

Algorithm 1 WMBFS pseudo-code

```
WMBFS()
 1: Initialize disk files and other data structures
 2: while new states left on disk do
 3:   for all buckets do
 4:     Clear change list
 5:     for every sector S in coarse open list do
 6:       if S not marked as changed in last iteration then
 7:         Continue to next sector
 8:       end if
 9:       for each state s_i in sector S do
10:         if s_i is open then
11:           SendToWorkerQueue(s_i)
12:         end if
13:       end for
14:     end for
15:     Wait for workers to finish
16:     Write current change list and update next coarse list
17:   end for
18: end while

WORKERTHREAD()
 1: while true do
 2:   s ← GetWorkFromQueue()
 3:   for each successor s_i of s do
 4:     if s_i in current bucket then
 5:       Write s_i to local cache
 6:     end if
 7:   end for
 8:   Write local cache to change list
 9: end while
```

3.3 Restarting WMBFS

A WMBFS can be restarted as long as no files are corrupted when the program was interrupted and we know the last depth written. This is works because a WMBFS stores the modulo depth of each state on disk, but no other information about the progress of the search. The downside to restarting is that the coarse open list is lost, so there may be extra work involved in the first iteration after the restart.

4 Comparison and Analysis

At the highest level, WMBFS trades off computation and reading from disk to reduce the number of writes to disk. WMBFS is most effective when the cost of expanding a state is low and the locality of the successors is low. WMBFS also performs well when the size of RAM is close to the size of the problem being solved. TBBFS is more effective when the cost of expanding states is high and the successors of a state exhibit high locality, or when RAM is very limited, but large, fast disks are available. We provide a high-level comparison of three external BFS approaches in Table 1, focusing on representation.

Hash-based delayed duplicate detection (DDD) does not store all states in memory at any point in the search, while TBBFS uses a implicit 2-bit representation and WMBFS uses an implicit representation based on the modulo depth.

DDD stores the current and next levels explicitly. TBBFS uses the implicit representation to distinguish states which are *open* and in the *new* iteration. States in *new* are also stored explicitly when they are outside the current bucket being processed. WMBFS uses the modulo representation to determine which states are *open* and in the *new* iteration.

DDD uses frontier search to avoid duplicates from previous levels and an in-RAM hash table to perform duplicate detection on the current level of the search. TBBFS uses its two-bit representation in RAM to perform duplicate detection between current and previous levels of the search. WMBFS uses a one-bit representation in RAM to detect duplicates in the current level of the search, and detects duplicates in previous levels as states are written to disk.

Overall, the maximum disk usage by DDD is the largest explicit layer with forbidden operators, although this depends partly on when duplicate detection is performed. The maximum disk usage by TBBFS is the size of the implicit representation plus the size of the explicit representation, although this again depends on when and how the explicit representation is processed. WMBFS only stores the implicit representation on disk.

This table should help make it clear that WMBFS and TBBFS form two distinct choices with different optimizations. WMBFS minimizes writes while computing the modulo depth of each state, while TBBFS minimizes the number of node expansions while computing the width/diameter of

Table 1: A comparison of BFS representational approaches.

Algorithm	State Space	Open	New	Dup. Detection	Max. Disk
DDD	-	explicit	explicit	frontier + RAM hash	explicit layer + forbidden operators
TBBFS	implicit (2 bits)	implicit	implicit + explicit	implicit (state space)	state space + explicit new
WMBFS	implicit (modulo)	implicit	implicit	implicit (changed list + disk)	state space

the state space. Many variants could be created by combining features of both algorithms, although a full exploration of these parameters is outside of the scope of this paper.

4.1 Theoretical Analysis

To better understand the tradeoffs that come into play when choosing between TBBFS and WMBFS, we analyze the number of bits written per state by each algorithm. We simulate TBBFS as previously described, storing 2-bits per state on disk, and storing non-local states as 64-bit explicit states[1]. We assume that TBBFS has enough external storage for all states written externally noting that WMBFS uses a fixed amount of disk space and therefore avoids this issue altogether.

Both TBBFS and WMBFS should have nearly identical characteristics with respect to initializing disk and writing the final values for each state, except that WMBFS must perform an extra read before writing to disk.

TBBFS will expand each state once, so we analyze the I/O operations of TBBFS with respect to a single state expansion. For each state expanded, the following occurs:

- The 2-bit value for the state is read once to get the initial value;
- All successors of that state are generated. Those that fall in the same bucket are updated in memory.
- Any other successors are explicitly written to disk.
- Each of these successors will later be read and then written to their own memory segments as a 2-bit value (unless the state was a duplicate).
- The state will be written as *closed* on disk.

Under the WMBFS algorithm, the following I/O operations take place for each expansion:

- The 4-bit value is read n times for each state (where n is the number of buckets);
- All successors of that state are generated. Those that fall in the current bucket are written to memory.
- The modulo value is both read and written for each state that has its depth updated.

Assume for TBBFS that the branching factor is b and p is the percent of states fall inside the memory segment (the locality). For each state, TBBFS is expected to write $64 \cdot b \cdot (1-p)$ bits explicitly to temporary storage. All other writing will be approximately equal for the two algorithms. TBBFS will read each 2-bit state once. All explicit states will be read at a cost of $64 \cdot b \cdot (1-p)$ bits. WMBFS will read each 4-bit state $n+1$ times. Thus, if $(1-p) > 0$ WMBFS must perform strictly fewer writes than TBBFS. If $64 \cdot b \cdot (1-p) + 2 > 4 \cdot (n+1)$, then WMBFS will also perform fewer reads.

[1]We could sub-divide buckets into smaller files so that only 32 bits need to be written to disk, but this would significantly increase the number of simultaneous open files; it is unclear how it would change performance

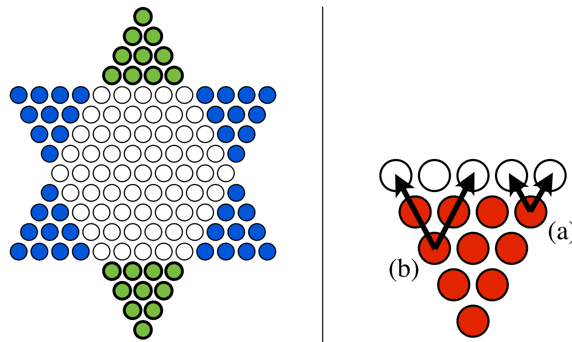

Figure 3: A Chinese Checkers board.

5 Experimental Domains

Two domains are used in this work, Chinese Checkers and Rubik's cube.

5.1 Chinese Checkers

We use the game of Chinese Checkers as the primary domain for this work, as the results of a full BFS are novel. Although it can be played competitively with 2-6 players, there are also interesting single-agent questions such as the number of moves required to move all of one's pieces across the board. A domain-specific solution [Bell, 2008] showed that 27 moves suffice, but did not store the results or compute the width and diameter of the state space.

The complete breadth-first search data can be used for playing Chinese Checkers. Chinese Checkers begins and ends as a single-agent problem, and the optimal set of moves to reach the goal for a single player can be used as a heuristic for the full game. This has been applied successfully to a small version of the game with only 13 million positions [Sturtevant, 2002; Schadd and Winands, 2011]. Variations on the idea have also been explored [Samadi *et al.*, 2008].

A sample Chinese Checkers board is shown on the left side of Figure 3. The board is shaped as a 6-sided star, with start locations on each of the arms of the stars. For the single-agent search we use the bolded arms at the top and bottom as the start and goal location for the search. The other shaded locations are excluded from the board, as players are not allowed to move into these locations. This leaves 81 passable locations; the board can also be drawn as a 9x9 grid with 6-way movement, following [Bell, 2008].

Example legal moves are shown on the right side of Figure 3. Pieces can either move to an adjacent empty location using six-way movement, as shown for the piece labeled (a). They can alternately jump over adjacent pieces, provide that each jump lands in an empty cell, demonstrated by piece marked (b). These jumps can be chained together to allow a piece to move far across the board. The jumping

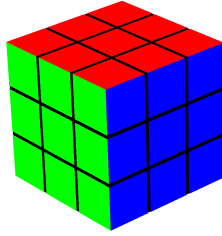

Figure 4: Rubik's Cube

process makes successor generation significantly more expensive than in domains like Rubik's Cube. Generating all legal moves requires a recursive search with duplicate detection to avoid generating the same move twice via different paths. A scan of the state space has shown that the branching factor of single-agent Chinese Checkers varies from 14 in the starting state to 99 moves in the mid-game.

Ranking. A Chinese Checkers board is not a permutation, as in Rubik's cube or the sliding-tile puzzle. [Edelkamp *et al.*, 2010] describe a lexicographical ranking function that works for Chinese Checkers; we use caches to rank states in linear time. Our two-level ranking first ranks the initial two pieces on the board, and then ranks the remaining pieces. The first two pieces are also used to determine buckets and symmetry.

Symmetry. A Chinese Checkers board is symmetric between the left and right halves. Ideally, we would like to avoid storing both a state and its mirror image, but testing and storing states precisely is difficult and expensive. Instead, we inspect just the first two pieces in the board. If the first piece is on the right portion of the board or the first piece is on the center line and the second piece is on the right side of the board the state is considered symmetric and not written to disk.

This reduces the number of states stored on disk from 1,878,392,407,320 to 1,072,763,999,648. During the search, however, every successor and its mirrored counterpart are both generated because neither a state or its mirror image may be symmetric according to our definition of symmetry.

5.2 Rubik's Cube

Rubik's cube, shown in Figure 4, has been a popular domain for study, including for use by TBBFS. The full state space is currently too large to exhaustively search, however there are $12! \cdot 2^{11} = 980,995,276,800$ possible combinations of the corner cubes, which is just smaller than the Chinese Checkers board after symmetry is removed. We used a linear-time ranking function [Myrvold and Ruskey, 2001] and did not exploit any symmetry present in the problem.

6 Experiments

We have performed a significant number of experiments using both WMBFS and TBBFS. A selection of experimental results are provided here to highlight the salient results. Experiments were run on three different machines. The first is a 2-core 2.66 GHz Intel i7 laptop with 8GB of RAM. Our first server is a 8-core 2.4GHz Xeon machine with 12GB of RAM, but only 500GB of hard disk. Our second server is a 16-core 2.6GHz AMD Opteron server with 64GB of RAM. This machine has a 500GB system disk, a 2TB work disk, and a 500GB SSD.

Table 2: Chinese Checkers States (Full Game)

Depth	States	Depth	States
0	1	17	37693202588
1	14	18	62046216886
2	156	19	92359898257
3	1331	20	123602266465
4	9477	21	147727791811
5	58643	22	156370829128
6	319561	23	144977817552
7	1540658	24	116054592894
8	6625563	25	78782115036
9	25566703	26	44355908984
10	88849561	27	20131633161
11	278909319	28	7079256201
12	793407418	29	1815972817
13	2053473432	30	308256210
14	4853812532	31	29204439
15	10504849377	32	1088715
16	20820518489	33	6269

6.1 Chinese Checkers

The primary result of this work is a full BFS through the single-agent version of Chinese Checkers, with 1.8 trillion states, of which approximately 1 trillion are stored on disk. Given 4 bits per state, approximately 500 GB of storage is required. We chose this because it maximizes the usefulness of the results for later applications. The WMBFS uses 1 bit per state in the change list, meaning that approximately 126GB is needed to store all states in RAM. With a 64GB machine, two buckets/iterations are required for the BFS. Our coarse open list has 256 entries per bit, meaning that 500 MB are used for each of the coarse open lists. The 4-bit data for the search is stored on the SSD, and the other hard drives are not used.

We also performed the BFS using TBBFS. TBBFS uses 8 bits per state in RAM, so we divided the state space into 20 buckets. We tested parameters for TBBFS using a board with 8 checkers on it. Our results suggested that the best configuration was to store the main TBBFS data on the SSD (using 256 GB), and to use the 2 TB disk for writing out additional states to disk. We chose not to modify TBBFS to store the results of the search, instead giving a conservative estimate of its performance.

The result of the BFS is found in Table 2. The results with the WMBFS and the TBBFS were identical, suggesting that the results are correct. The implementations only share the Chinese Checkers code. Both runs confirmed that the goal is at depth 27 [Bell, 2008]. A comparison of the major statistics of each approach can be found in Table 3. The WMBFS is about 10% slower than the TBBFS, but writes 18 times less data and reads 8 times less data.

Through the whole search TBBFS wrote 80TB to temporary storage (over 10 trillion states). Adding to this the cost of flushing temporary storage when filled results in 88TB of reading and writing. We did not precisely measure the other read/write costs, so 88TB is a lower-bound on the total reading and writing. The disk was filled and had to be flushed a total of 45 times during the search. Our code didn't check if

Table 3: TBBFS and WMBFS in Chinese Checkers

	WMBFS (15 threads)	TBBFS (16 threads)
Total Time	2,632,266 sec 30.5 days	2,410,966 sec 27.9 days
Nodes Expanded	1,837,185,821,822	1,072,763,999,648
Total Writes	4.85 TB	88 TB*
Total Reads	10.80 TB	88 TB*

Table 4: WMBFS parallel speedups on a Chinese Checkers board with 49 locations and 6 pieces.

Machine	Base Time	Thread Speedup Factor 2	4	8	16	Final Speed
Intel Laptop	55.1s	1.5	1.7	-	-	32.3s
Intel Server	44.0s	1.8	3.3	5.4	6.4	6.9s
AMD Server	60.2s	1.9	3.5	6.1	7.8	7.7s

the disk was full quite often enough, and it was filled beyond capacity during the largest iteration, crashing the program and our server. But, we were able to restart the search. This overall reduction in I/O means fewer opportunities for disk errors and increased disk life.

Although WMBFS is 10% slower, it is also performing a more interesting computation. At the end of its computation, TBBFS has no data stored besides the numbers from Table 2. WMBFS, however, has the modulo 15 distance for every state stored. It would adversely affect the performance of TBBFS if we limited it to use the same disk space as the WMBFS or required TBBFS to store the same data as the WMBFS.

Successor Locality. We had expected WMBFS to be faster than TBBFS on the full game, because on our experiments on the board with just 8 pieces WMBFS had performed better than TBBFS. This is explained by the locality of the state space. TBBFS is very efficient if successors fall into the same bucket in RAM, and inefficient if they fall external to the bucket and must be written explicitly to disk. Sampling the board with 8 pieces shows that percent of local successors is 84.9%. On the full game the locality rises to 87.3%; this higher locality improved the performance of TBBFS. WMBFS was faster than TBBFS on the first half of the search, and slower in the second half. TBBFS in the largest depth (22) wrote 10.1 states per expansion externally. At depth 23 and 24 only 9.6 and 8.9 states per expansion respectively were written externally. This change shifted the speed of the second half of the search in favor of TBBFS.

Concurrency. Concurrency statistics for our implementations are important, as they can significantly influence the overall results. We present WMBFS data from solving a small board with 49 locations and 6 pieces in Table 4; TBBFS results are similar. Our implementations both use a single thread for reading from disk and sending work to worker threads. The number of threads in this table is the number of worker threads; the total threads is one more. The Intel machines both have hyper-threading. The Intel i7 laptop (2 cores + hyper-threading) achieved a 1.7 times speedup with 4 worker threads versus one. The Intel server (8 cores + hyper-threading) achieved a 6.4 times speedup. The AMD server achieved a 7.8 time speedup on 16 real cores, but overall was slower than the Intel server despite a faster clock speed. Overall, the AMD machine is more sensitive to global memory access.

Table 5: Time overhead of using multiple buckets.

	Number of buckets 1	2	3	6	9
Chinese Checkers (49/6)	1.00	1.46	1.91	3.11	4.51
Rubik's Cube Corners	1.00	1.26	1.54	2.14	3.33

Table 6: TBBFS and WMBFS in Rubik's Cube

	WMBFS (15 threads)	TBBFS (16 threads)
Total Time	586,433 sec 6.8 days	2,099,746 sec 24.3 days
Nodes Expanded	1,961,990,553,104	980,995,276,800
Total Writes	2.68 TB	60.5 TB*
Total Reads	6.85 TB	60.5 TB*

Bucket Overhead. The overhead of WMBFS grows with the addition of more buckets in the search. But, using twice as many buckets will not double the cost of the search. The relative slowdown of multiple buckets is illustrated for smaller Chinese Checkers and Rubik's cube problems in Table 5.

6.2 Rubik's Cube

In Rubik's Cube every action moves many cubes at once, so the locality of the state space is relatively low (near 50%) [Korf, 2008b]. But, the cost of node expansions is much cheaper in Rubik's cube than in Chinese Checkers, as there is no need to search for legal moves – there are the same 18 legal moves in every state. These two factors suggested that a WMBFS would perform favorably in this domain.

Published results for a TBBFS required 35 days [Korf, 2008b] for the search, although these results are several years old. Our implementation of TBBFS on the AMD server, which used the 2TB disk for temporary storage and the 500GB SSD for the main data, required 24.3 days. The WMBFS required just 6.8 days to compute and store the same result, 3.58 times faster. The full statistics for the search are in Table 6. As before, the TBBFS data represents only the reading and writing used in storing and flushing states from temporary storage. The WMBFS performed approximately 20 times fewer writes and 10 times fewer reads.

7 Summary and Future Work

We have presented the write-minimizing breadth-first (WMBFS) algorithm, which works to minimize the number of writes performed in an external memory breadth-first search. The algorithm reduces I/O by over and order of magnitude. It has similar time performance to TBBFS in the domain of Chinese Checkers, and is significantly faster in Rubik's cube. In the future we plan to examine hybrids between TBBFS and WMBFS, and also plan to look at how the data we have computed can be effectively used to guide search algorithms from external memory.

Acknowledgments

We thank Ariel Felner for comments on the draft manuscript.

References

[Ajwani *et al.*, 2009] Deepak Ajwani, Andreas Beckmann, Riko Jacob, Ulrich Meyer, and Gabriel Moruz. On computational models for flash memory devices. In Jan Vahrenhold, editor, *SEA*, volume 5526 of *Lecture Notes in Computer Science*, pages 16–27. Springer, 2009.

[Bell, 2008] George I. Bell. The shortest game of chinese checkers and related problems. *CoRR*, abs/0803.1245, 2008.

[Breyer and Korf, 2010] Teresa Maria Breyer and Richard E. Korf. 1.6-bit pattern databases. In *AAAI*, 2010.

[Edelkamp *et al.*, 2010] Stefan Edelkamp, Damian Sulewski, and Cengizhan Yücel. Gpu exploration of two-player games with perfect hash functions. In *SOCS*, 2010.

[Felner *et al.*, 2004] A. Felner, R. Meshulam, R. C. Holte, and R. E. Korf. Compressing pattern databases. In *National Conference on Artificial Intelligence (AAAI-04)*, pages 638–643, 2004.

[Hatem *et al.*, 2011] Matthew Hatem, Ethan Burns, and Wheeler Ruml. Heuristic search for large problems with real costs. In Wolfram Burgard and Dan Roth, editors, *AAAI*. AAAI Press, 2011.

[Korf, 2004] R. E. Korf. Best-first frontier search with delayed duplicate detection. In *National Conference on Artificial Intelligence (AAAI-04)*, pages 650–657, 2004.

[Korf, 2008a] Richard E. Korf. Linear-time disk-based implicit graph search. *J. ACM*, 55(6):26:1–26:40, December 2008.

[Korf, 2008b] Richard E. Korf. Minimizing disk i/o in two-bit breadth-first search. In *AAAI*, pages 317–324, 2008.

[Myrvold and Ruskey, 2001] Wendy J. Myrvold and Frank Ruskey. Ranking and unranking permutations in linear time. *Inf. Process. Lett.*, 79(6):281–284, 2001.

[Robinson *et al.*, 2007] Eric Robinson, Daniel Kunkle, and Gene Cooperman. A comparative analysis of parallel disk-based methods for enumerating implicit graphs. In *PASCO*, pages 78–87, 2007.

[Samadi *et al.*, 2008] Mehdi Samadi, Jonathan Schaeffer, Fatemeh Torabi Asr, Majid Samar, and Zohreh Azimifar. Using abstraction in two-player games. In *ECAI*, pages 545–549, 2008.

[Schadd and Winands, 2011] Maarten P. D. Schadd and Mark H. M. Winands. Best reply search for multiplayer games. *IEEE Trans. Comput. Intellig. and AI in Games*, pages 57–66, 2011.

[Schaeffer, 1997] Jonathan Schaeffer. *One jump ahead - challenging human supremacy in checkers*. Springer, 1997.

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

[Zhou and Hansen, 2004] R. Zhou and E. Hansen. Structured duplicate detection in external-memory graph search. In *National Conference on Artificial Intelligence (AAAI-04)*, pages 683–689, 2004.

[Zhou and Hansen, 2011] Rong Zhou and Eric A. Hansen. Dynamic state-space partitioning in external-memory graph search. In *ICAPS*, 2011.