Minimizing Disk I/O in Two-Bit Breadth-First Search

Richard E. Korf
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
korf@cs.ucla.edu

Abstract

We present a breadth-first search algorithm, two-bit breadth-first search (TBBFS), which requires only two bits for each state in the problem space. TBBFS can be parallelized in several ways, and can store its data on magnetic disk. Using TBBFS, we perform complete breadth-first searches of the original pancake problem with 14 and 15 pancakes, and the burned pancake problem with 11 and 12 pancakes, determining the diameter of these problem spaces for the first time. We also performed a complete breadth-first search of the subspace of Rubik's Cube determined by the edge cubies.

Introduction and Overview

The Pancake Problems and Rubik's Cube

The pancake problem was introduced in (Dweighter 1975)[1]. When a sloppy chef prepares a stack of pancakes, they are all different sizes. The waiter wants to sort the pancakes by size, by repeatedly flipping the top k pancakes over as a group. What is the maximum number of flips needed to sort any stack of n pancakes? In other words, what is the diameter of this permutation group? In the burned pancake problem, each pancake is burned on one side, and in addition to sorting them, the waiter wants the burned sides face down.

A much more famous open problem of this form is what is the maximum number of moves needed to solve any scrambled Rubik's cube?

Breadth-First Search

The only general way to solve these problems exactly is systematic search. For example, the maximum depth reached in a complete breadth-first search (BFS) from any given state is the diameter of a permutation group.

Complete breadth-first searches are also used to compute pattern database heuristics.(Culberson & Schaeffer 1998) The larger the pattern database, the more accurate the heuristic. Very large pattern databases that exceed available memory can be compressed to fit in memory, often with little loss of information.(Felner *et al.* 2007)

What limits the size of a breadth-first search in practice is storage to detect when a state has previously been generated.

[1](Gates & Papadimitriou 1979) identify "Harry Dweighter" as Jacob E. Goodman

Overview

The main contribution of this paper is a new BFS algorithm that requires only two bits per state of the problem. This corrects an algorithm suggested by Robinson, Kunkle and Cooperman (2007). We also present several improvements to their work that significantly reduce the amount of disk I/O needed, and an alternative way to parallelize their algorithm. We first present previous work, including breadth-first frontier search, and the work of Kunkle and Cooperman (2007). We then present our two-bit BFS, beginning with a simple version that resides in memory, then parallelizing the algorithm, next extending it to use magnetic disk storage, and finally modifying it to allow it be resumed in the event of system crashes. Next we present experimental results on the pancake problems. We performed a complete BFS of the original 14 and 15 pancake problems, and the 11 and 12 burned pancake problems, computing the diameters of these problems for the first time. In the process, we discovered a surprising anomaly of the burned pancake problem. In addition, we performed a complete BFS of the subspace of Rubik's cube determined by the edge cubies.

Previous Work

Previous Work on the Pancake Problems

The first results on the diameter of the pancake problems are from (Garey, Johnson, & Lin 1977). They give the diameters of the original problem for up to seven pancakes. Gates and Papadimitriou (1979) give these values for up to nine pancakes, citing personal communication with David P. Robbins in 1977 for the last two. They also introduce the burned pancake problem, and focus on upper and lower bounds for arbitrary size problems. Finally, Heydari and Sudborough (1997) give values up to 13 unburned pancakes, and 10 burned pancakes, in another paper focussed on bounds.

Frontier Search with Delayed Duplicate Detection

A previous space-efficient BFS algorithm is breadth-first frontier search (BFFS) (Korf *et al.* 2005). Rather than storing all nodes generated, frontier search only stores the nodes on the boundary of the search. This can be as few as the maximum number of nodes at any depth in the problem space, or the *width* of the problem space. These nodes can also be stored on magnetic disk, using delayed duplicate detection (Korf 2003). The largest complete searches have been done using these techniques, including searches of the Fifteen Puzzle with over 1.3×10^{13} states (Korf & Shultze 2005), and the 22-disc four-peg Towers of Hanoi problem, with over 1.76×10^{13} states (Korf & Felner 2007).

The minimum storage needed by BFFS is the width of the problem space times the storage for a state, typically four bytes. If the graph contains odd-length cycles, such as the Towers of Hanoi or the pancake problems, then up to two levels of the search may have to be stored at once. For problems where the width of the graph is a small fraction of the total number of states, BFFS is the most space-efficient BFS. For example, the size of the problem space of the 2x8 Fifteen Puzzle is 28.5 times its width, while this ratio for the 22-disc four-peg Towers of Hanoi problem is 63.2.

Limitation of Frontier Search For other problems, however, the ratio of the size of the problem space to its width is much smaller. For example, for 15 unburned pancakes, this ratio is about 2.89, while for 12 burned pancakes it is about 2.68. For the 2x2x2 Rubik's Cube, this ratio is 1.95, and for the subspace of the 3x3x3 Rubik's Cube defined by the edge cubies, it is 1.77. If the size of a problem space is less than 16 times its width, it takes less space to store two bits per state, than four bytes times the width of the problem space.

Robinson, Kunkle, and Cooperman

Many of the ideas in this paper were previously published in (Kunkle & Cooperman 2007) and (Robinson, Kunkle, & Cooperman 2007). While our work was done independently, their work came first. For clarity of exposition, however, we describe this work in increasing order of complexity, rather than chronological order of publication, and then characterize in detail the differences between our work and theirs.

TBBFS in Memory

We describe our two-bit breadth-first search in a series of complete, but increasingly complex algorithms. The simplest version runs entirely in memory on a single processor.

Mapping Between States and Array Indices

A key element of TBBFS is a function that maps problem states one-to-one to integer values. Ideally, if every value also correspond to a valid state, this mapping becomes a bijection. For the original pancake problem, we map each permutation of n pancakes to a unique index in the range zero to $n! - 1$. For example, one bijection between permutations of three elements and integers from zero to five is: 0-012, 1-021, 2-102, 3-120, 4-201, 5-210. This mapping is also lexicographic, in the sense that when ordered by their integer indices, the permutations appear in lexicographic order. Korf and Schultze (2005) describe a linear-time implementation of this mapping in both directions.

Next, we define an array whose indices correspond to the range of the mapping function. If the mapping is a bijection, each array element represents a different state of the problem, and the length of the array equals the total number of states. Otherwise, some indices won't represent valid states, and the corresponding array elements constitute wasted space. We assume from here on a bijective mapping.

Storing the Depth of Each State

We first describe a simpler algorithm, which is an in-memory version of Kunkle and Cooperman's (2007) algorithm. Each element of the array represents the depth of the corresponding state, which is the shallowest level at which the state first appears in the BFS. Initially, all elements are set to an "empty" value, such as minus one, except that the element corresponding to the initial state is set to depth zero.

Each iteration scans the array sequentially, expanding all states at a given depth. For each index whose value equals the current search depth, the inverse mapping function is applied to generate the corresponding state. Each legal operator is then applied to the state to generate a child state. For each child state, the forward mapping is applied to generate its index in the array. If the corresponding element is set to the empty value, indicating that it hasn't been generated before, its value is set to a depth one greater than the current search depth. Otherwise, its value is left unchanged. For each iteration, the current depth is increased by one, until there are no more states at the current depth. At that point, the array contains the depth of every reachable state.

The memory required by this algorithm is the log base two of the maximum search depth, times the size of the array. With one byte per entry, searches up to depth 254 are feasible, using 255 for the empty value.

Storing Only Two Bits per State

In fact, we can reduce the memory required to two bits per state, or four states per byte, regardless of the maximum search depth. The four values encoded by the two bits indicate whether the corresponding state is "old", meaning its depth is less than the current depth, "current", meaning its depth equals the current depth, "next", meaning its depth is one greater than the current depth, or "new", meaning its depth is more than one greater than the current depth.

The two-bit array is initialized to all "new" values, except that the element corresponding to the initial state is set to "current". In each iteration of the search, all states whose corresponding array values are set to "current" are expanded, as described above. For each child state whose corresponding array value is "new", its value is set to "next", and all other array values are left unchanged.

At the end of each iteration, those elements marked "current" must be changed to "old", and those elements marked "next" must be changed to "current", in preparation for the next iteration. The simplest way to do this is to scan the entire array again. At the end of this conversion scan, another iteration begins, until all elements are marked "old".

Eliminating the Conversion Scan

The conversion scan can be eliminated in at least two ways.

One is to interpret the array values differently depending on their position relative to the index of the parent state currently being expanded. In particular, when changing the value of a child marked "new", if the child index is greater than the parent index, meaning it will be read in the current iteration, it is marked "next". Alternatively, if the child index is less than the parent index, meaning it will not be read until the next iteration, it is marked "current". In addition, after a parent state is expanded, its value is changed from "current" to "old". Finally, each state marked "next" is changed to "current" as it is encountered during the expansion scan, to be expanded in the next iteration. This performs the conversion during the expansion scan.

A simpler solution is that child nodes marked "new" are changed to "next" regardless of their location, and parent nodes marked "current" are changed to "old" as they are expanded. At the end of an iteration, the value that indicated "next" in the previous iteration is interpreted as "current" in the next iteration. The value that indicated "current" in the previous iteration will no long appear in the array, and is used to signify "next" in the next iteration. The interpretation of these two values is swapped between each iteration.

Time and Space Complexity

Using either of these mechanisms to avoid the conversion scan results in an algorithm whose asymptotic time complexity is the length of the array times the number of iterations, which is the maximum search depth. In practice, however, the running time is proportional to the number of nodes generated, since this cost dominates the time to scan the array. The space complexity is two bits times the length of the array. If the mapping between states and indices is a bijection, the length of the array is the total number of states.

Parallel TBBFS in Memory

Most modern computers have multiple processors, or multiple cores, or both. Thus, parallel processing is required to get the best performance from these machines.

One way to parallelize TBBFS in memory is for different threads to expand different parts of the array in each iteration. For example, with two threads, one expands the first half of the array while the other expands the second half.

Consider, however, what happens if the same child marked "new" is simultaneously generated by two threads. Each thread will try to change its value to "next". Regardless of the order in which the two reads and writes occur, however, the end result will be correct. If one thread completes its read and write before the other, the second thread will read a value of "next" and will leave it unchanged. Alternatively, if both reads occur before either write, then each thread will read a value of "new", and will write a value of "next". Thus, no synchronization is required in this case.

A problem occurs if two threads try to write different child bits in the same byte, however. Since we allocate two bits per state, four states are packed in each byte. If one thread tries to change one pair of bits in a byte, and another thread tries to change a different pair of bits in the same byte, and both reads occur before either write, then only the last write will take effect, and only one pair of bits will change, since a single byte is read and written as a unit.

One solution is to have a mutual exclusion lock for each byte of the array. This is prohibitively expensive, however, since the overhead of locking and unlocking each byte will dwarf the cost of changing the value of the byte.

An effective solution is to use a full byte in memory for each state in the parallel algorithm, increasing the memory required by a factor of four. When storing the array on disk however, as we discuss next, we only store two bits per state.

TBBFS on Disk

Storing the entire two-bit array in memory limits the size of problems that can be solved with TBBFS. For example, the original 14 pancake problem would require $14!/4$ bytes, which is over 20 gigabytes. Eleven burned pancakes would require $11! \cdot 2^{11}/4$ bytes, which is over 19 gigabytes.

The solution to this problem is to store the array on magnetic disk instead. Recently, disk capacities have increased dramatically, with correspondingly large drops in the cost per byte. One terabyte disks are now available for about \$300. The two-bit array for 15 unburned pancakes only occupies $15!/4$ or 327 gigabytes, while the array for 12 burned pancakes occupies $12! \cdot 2^{12}/4$ or 490 gigabytes.[2]

Magnetic disks must be accessed sequentially, however, since accessing a random byte on disk can take up to 10 milliseconds. While the expansion scan of TBBFS is sequential, the child updates can appear to be randomly distributed.

We divide the two-bit array into contiguous segments, each of which is small enough to fit into memory. During an iteration, each segment is loaded into memory in turn, and expanded. Any updates to a child in the current segment resident in memory are made immediately. Next, we consider how to update children that are not in memory.

Logging Potential Updates on Disk

Child states do not need to have their values updated until they are expanded in the next iteration. Thus, we store the indices of all generated children that are not resident in memory, until their corresponding segments are loaded into memory, and any necessary updates can be made to the two-bit array. In principle, any updates to the segment in memory do not have to be made until the next iteration either, but in practice we make all updates to the resident segment immediately, to save storage for the non-resident updates.

The list of updates are also stored on disk, with updates to different segments stored in different files. When an array segment is loaded into memory, its update file is read, the updates are made to the segment, and the file is deleted.

Another advantage of separate update files for each segment is that it reduces the space needed for the updates. For a problem such as 13 unburned pancakes or 11 burned pancakes, an index to each state requires more than 32 bits. Storing these updates as double precision integers requires eight bytes. If we limit the size of an array segment to one gigabyte, at two bits per entry a segment can contain at most 2^{32} different entries, and any offset within a segment can be specified with an unsigned 32-bit integer. We only store the offsets within the segments in the update files, and store the identities of the segment as part of the update file names.

[2]A gigabyte of memory is 2^{30} bytes, while a gigabyte of disk space is 10^9 bytes.

Outstanding Updates at the End of an Iteration

In the simplest case, at the end of each iteration, any remaining updates are committed to the two-bit array by loading each array segment into memory, reading its update file, making the updates to the array, writing the array segment back to disk, and deleting the update file. This requires reading most segments into memory twice per iteration, once to expand them, and once to commit any outstanding updates to them at the end of the iteration.

A more I/O efficient algorithm labels each update file with its corresponding iteration. At the end of each iteration, we don't commit outstanding updates to the array immediately, but instead begin the next iteration. When a new segment is read into memory, any corresponding update file from the previous iteration is read, those updates are made to the array segment, and then any update files from the current iteration are read, and those updates are made to the array segment, before expanding the nodes in that segment.

What to Do When the Disk Fills Up

So far we have assumed sufficient disk space to wait until an array segment is loaded into memory for expansion before reading, committing, and deleting its update files. On a large problem, however, this algorithm may fill up the disk(s).

Our solution to this problem is to choose a disk occupancy level, such as 95%. Once that level is reached at the end of expanding a segment, we load another segment into memory, read its update file(s), commit the updates, write the segment back to disk, and delete the update file(s). This continues with another array segment until the disk occupancy drops below the chosen threshold, at which point the next array segment to be expanded is loaded into memory.

When the disk reaches its target occupancy level, we update the segment with the most updates on disk, since this reduces the disk occupancy the most. We expand array segments in numerical order for simplicity, although they could be expanded in any order, such as always expanding next the one with the most updates on disk. Expanding the array segments in the same order in every iteration approximates this, since the array segment which has been out of memory the longest is likely to have the most updates stored on disk.

With sufficient disk capacity, we can wait until an array segment is to be expanded before reading it into memory and updating it, but this algorithm works even with no more disk space than that needed for the two-bit array. In that case, all updates would be stored in memory. This could be very inefficient, however, since it may require loading and updating many array segments between segment expansions.

Using Operator Locality to Minimize Disk I/O

For multiple reasons of efficiency, we want to minimize the amount of disk I/O. Writing and reading update files takes time. In addition, by minimizing updates on disk, we reduce the amount of disk space used, which allows the program to run longer or even complete without filling up the disk. Once the disk is full, the algorithm incurs additional overhead to read segments into memory just to commit their updates to disk and delete the corresponding update files.

The primary way to minimize updates on disk is to maximize the number of updates that can be made to the array segment resident in memory. This is done by designing the mapping function between states and indices so that the indices of child nodes will be close to those of their parents.

Consider, for example, the unburned pancake problem. A state is a permutation of the pancakes. Some operators, such as flipping the top two pancakes, only make a small change to the state. We would like a function that maps the indices of these two states to nearby locations. This is accomplished by mapping permutations to integers in a particular lexicographic order. If the pancakes are listed in order from the bottom of the stack to the top, then flipping the top two pancakes changes the value of the index by only one. Note that if our lexicographic index ordered the pancakes from top to bottom instead, we would lose this locality.

In our implementation of the unburned pancake problem, we use array segments with 12! elements each. Thus, flipping 12 or fewer pancakes generates a child state whose index is in the same segment as its parent, and hence can be updated in memory without writing to disk. For the 15-pancake problem, for example, this reduces the number of updates per expansion written to disk from 15 to three. For the burned pancake problem, a state is a permutation of the pancakes, plus an additional bit for each pancake to indicate which side is burned. The two-bit array is divided into segments that contain the permutations of the top nine pancakes and all the "burned bits". Thus, flips of nine or fewer pancakes generate children in the same segment as their parent.

Parallel TBBFS on Disk

For simplicity, we described a serial version of TBBFS on disk above. We now consider two different parallel versions. One is based on the parallel in-memory algorithm described above, expanding a single segment in parallel, and the other expands multiple segments in parallel at the same time.

Parallelizing the Expansion of Individual Segments

The main problem with the parallel version of TBBFS described above is simultaneous updates by multiple threads to different pairs of bits in the same byte. To solve this problem, we store two bits per state on disk, but expand each array segment to one byte per state in memory when it is loaded, and recompress it to two bits per state before writing it back to disk. In addition to saving disk space, this saves time relative to storing a full byte per state on disk, since the reduced I/O time more than compensates for the time spent compressing and uncompressing the array segments.

We divide each array segment by the number of parallel threads. Since all disk I/O is buffered in memory, each parallel thread maintains its own update file buffers, to avoid synchronizing parallel writes to common buffers. The parallel threads are synchronized after reading and expanding each segment into memory, after committing all updates from the previous iteration, after committing all updates from the current iteration, at the end of node expansion, and after compressing and writing each segment to disk. The reason is that while each thread only expands states in its part of the segment, it generates updates to the entire segment.

The drawbacks of this approach are that the multiple synchronizations per segment invariably result in idle processor time, and that all threads perform disk I/O at the same time, leaving the processors idle during that time. To minimize this I/O wait time, we use three separate disks, thereby increasing the total disk bandwidth, and three parallel threads in most cases. Each thread keeps its portion of each array segment and its own update files on its own disk, eliminating contention between threads for the same disk.

Expanding Multiple Segments in Parallel

An alternative way to parallelize this algorithm is for separate threads to update and expand separate array segments. A queue of segments awaiting expansion is maintained. As each thread finishes its previous segment, it grabs the next segment from the queue. It then reads the segment into memory, updates it from any existing update files on disk, expands the nodes in the segment, and writes it back to disk.

There are several advantages to this scheme. One is that the segments don't have to be expanded to a full byte per entry, since only one thread updates a given segment in a single iteration. Furthermore, since the different threads are not synchronized, when one thread is blocked by disk I/O, other threads expanding or updating other segments can run.

The main disadvantage of this approach is that multiple array segments being processed by different threads must be in memory simultaneously. This means that the size of individual segments must be smaller, compared to having one segment in memory at a time. As a result, the number of updates that must be written to disk instead of being committed in memory will be larger, increasing the amount of disk I/O, and decreasing the time before the disk fills up.

Comparing the Two Parallel Algorithms

It is not clear which of these two parallel algorithms will perform better, as there is a complex set of tradeoffs, involving operator locality, the amount of memory available, the number of processors or cores, the amount of disk space, and the relative cost of computation vs. disk I/O. Ultimately this has to be resolved empirically.

One can also combine the two approaches by partitioning the threads into several groups. Each group will parallelize the expansion of a single segment, while multiple groups expand different segments in parallel.

In general, with either parallelization or their combination, it is most efficient to run more threads than processors or cores. The reason is that at any given time, one or more threads will be blocked by disk I/O, and we want to be able to run at least as many threads as we have processors or cores. The best number of threads is determined empirically.

Interruptible TBBFS

By using only two bits per state and large disks, we can solve problems that take weeks to run. At this time scale, the strategy of restarting a computation from scratch in the event of a power glitch or other system crash doesn't work if the mean time between failure is less than the total running time.

To insulate our system from power glitches, we use an uninterruptible power supply (UPS). A UPS sits between the computer and the power outlet, and provides battery power to the machine in the event of a temporary power failure.

Our system crashed with no diagnostic messages three times while running a complete BFS of the 12 burned pancake problem. To deal with this, we retreated to a simpler version of TBBFS that can be resumed from the last completed iteration. In particular, each iteration scans each array segment twice. The first scan expands the current nodes, but doesn't change their values, and is done on all segments before any of the second scans. At the end of the expansion scan, all outstanding updates are written to disk. The second scan converts the "current" states to "old" states, and the "next" states to "current" states, as well as committing any updates on disk. In the event of a crash during the expansion scan, the expansion scan can be restarted from the beginning. In the event of a crash during the conversion scan, the program can resume by updating and converting those segments that haven't been written to disk yet.

Robinson, Kunkle, and Cooperman's Work

As mentioned above, many of these ideas appear in (Kunkle & Cooperman 2007) and (Robinson, Kunkle, & Cooperman 2007). Here we identify their contributions, and compare them with ours. The main result of (Kunkle & Cooperman 2007) is that any scrambled Rubik's Cube can be solved in at most 26 moves. As part of their analysis, they did a complete BFS of a subspace of the cube, known as a coset.

Algorithm 1

"Algorithm 1" in (Kunkle & Cooperman 2007), is a parallel disk-based BFS that stores the depth of each state in an array. In particular, they introduced the technique of using an array with one element per state, which they call an "implicit open list". They used four bits per entry. Their algorithm proceeds in at least two phases per iteration. In the expansion phase, all nodes at the current search depth are expanded, and all their children are written to disk, in separate files for each segment. No updates are made in memory, and hence the segments are not written to disk in the expansion phase. Then, in the merge phase, each segment is read into memory, its update file is read, the updates are made in memory, and the segment is written to disk. If the disk fills up during the expansion phase, then a complete merge phase is run, updating every segment with any updates on disk, and deleting all update files. At that point, the expansion phase continues until it completes without filling the disk, followed by a final merge phase phase in that iteration. Different threads expand or merge different segments in parallel.

Their Two-Bit Breadth-First Search

Robinson, Kunkle and Cooperman (2007) claim that the amount of space per state can be reduced to two bits, rather than the log of the maximum search depth, and propose such an algorithm on page 81. They use the first bit of each entry to indicate that a state is at the current depth, and use the second bit to indicate that the state is at the next depth. Initially, all bits are reset to zero, except that the first bit of the initial state is set to one. In the expansion phase of an iteration, all states with their first bit set are expanded, and all their children are written to disk. During the subsequent merge phase, those child states that are not duplicates have their second bits set. While it is not clear from their description, the only way to detect a duplicate child at this point would be if one of its bits were already set. Finally, if an entire level was completed during the expansion phase, the value of the first bit of each pair is set to the value of its second bit, and the second bit is cleared, changing states at the next depth to the current depth in preparation for the next iteration.

Unfortunately, this algorithm doesn't work. While it marks states at the current depth and the next depth, it doesn't distinguish states that have never been generated from states already expanded at previous depths. When a given state has its first bit set, it will be expanded in the current iteration. Even if it is also generated in the current iteration, its second bit will not be set, because its first bit is set, indicating that it's a duplicate state. At the end of the current iteration, the first bit is set to the value of the second bit, leaving both bits zero. If the operators of the problem apply in both directions, the same state will be regenerated in the next iteration from each of its children, and its second bit will be set. At the end of this iteration, its first bit will be set to the value of its second bit, setting it to one. Thus, it will be reexpanded in the following iteration. As a result, this algorithm won't terminate. No implementation of this algorithm is reported, and we believe this is the reason that this error was not detected.

Differences Between Our Work and Theirs

The main differences between our work and that of Robinson, Kunkle, and Cooperman fall into three general areas.

First, we provide a correct two-bit BFS algorithm, supported by implementations on several different problems.

Next, our algorithm is designed to minimize the amount of disk I/O and the amount of disk space used, in several ways. The first is to make any updates to the array segment resident in memory immediately, whereas Kunkle and Cooperman (2007) write the index of every child node to disk. For this reason, we design our mapping functions to maximize the number of child nodes that occur in the same segment as their parent. Second, rather than having separate expansion and merge phases in each iteration, we make any updates to a segment immediately before expanding it. Third, in the uninterruptible version, we don't perform all updates at the end of an iteration, but save pending updates until a segment is read in before being expanded in the next iteration. Fourth, when the disk fills up, we only update enough segments to reclaim enough disk space to continue expanding, rather that making all outstanding updates. Fifth, in the non-interruptible version, we eliminate the scan to convert next states to current states by swapping the interpretation of these two values after every iteration.

We also present a method of parallelizing the expansion of an individual segment, by temporarily expanding the segment to one byte per entry in memory.

Experimental Results

We ran experiments on the original and burned pancake problems, and Rubik's Cube. We implemented several different BFS algorithms, both to compare their performance, and to verify our results. For a complete BFS, the results consist of the number of unique states at each depth. The best evidence for the correctness of these results is that we got identical values from several very different algorithms. In addition, the sum of the numbers of nodes at each depth always equaled the total number of states in the problem.

All our experiments were run on an IBM Intellistation A Pro workstation, with dual AMD two-gigahertz Opteron processors and two gigabytes of memory, running Centos Linux. We used three Hitachi one-terabyte internal SATA-II disk drives. Our code was written in C.

Pancake Problem Experiments

Breadth-First Frontier Search with DDD First, we tested breadth-first frontier search (BFFS) with delayed duplicate detection. The largest problems we could search exhaustively with this method were 14 unburned and 11 burned pancakes, each of which are one larger than the previous state of the art. The problem-space for 14 unburned pancakes contains $14!$ states. This problem took 43.4 hours to run using six threads, and used a maximum of 449 gigabytes of disk space at four bytes per node. The problem space for 11 burned pancakes contains $11! \cdot 2^{11}$ states. This problem took 33 hours to run using five threads, and used a maximum of 395 gigabytes of disk space. If the maximum disk storage grows linearly with the size of the problem space, then 15 unburned pancakes would require about 6.74 terabytes, and 11 burned pancakes would need about 8.47 terabytes.

Algorithm 1 We also implemented Kunkle and Cooperman's Algorithm 1 on 14 unburned and 11 burned pancakes. We allocated one byte of memory per state, since the maximum depths of these problems are 16 and 19 moves, respectively. For the unburned problem, each segment occupied $12!$ bytes, or 457 megabytes of memory. For the 11 burned pancake problem, each segment occupied $9! \cdot 2^{11}$ bytes, or 709 megabytes. In both cases we had sufficient disk space so the disks did not fill up. On the 13 unburned and 10 burned pancake problems, we found that three parallel threads gave the best performance with only a small difference between three and two or four threads. For 14 unburned pancakes with three threads, the algorithm took 52.7 hours, compared to 43.4 hours for BFFS. For 11 burned pancakes, we only had enough memory to run two parallel threads, and it took 49.5 hours, compared to 33 hours for BFFS.

Two-Bit Breadth-First Search We also implemented our TBBFS, on both unburned and burned pancakes.

TBBFS on 14 Unburned and 11 Burned Pancakes Using five threads expanding multiple segments in parallel, the non-interruptible version of TBBFS took 23.22 hours to run a complete BFS of the 14 unburned pancake problem, compared to 43.4 hours for BFFS, and 52.7 hours for Kunkle and Cooperman's Algorithm 1. It used about 22 gigabytes to store the two-bit array, and a maximum of 141 gigabtyes to store updates on disk, for a total of 163 gigabytes, compared to 449 gigabytes for breadth-first frontier search, and 1664 gigabytes for Algorithm 1.

This same version of TBBFS took 15.63 hours to run a complete BFS of the 11 burned pancake problem, compared to 33 hours for BFFS, and 49.5 hours for Algorithm 1. It used about 20 gigabytes to store the two-bit array, and a maximum of 145 gigabtyes to store updates on disk, for a total of 165 gigabytes, compared to 368 gigabytes for breadth-first frontier search, and 1396 gigabytes for Algorithm 1.

Note that BFFS requires the full amount of disk space given above, whereas both TBBFS and Kunkle and Cooperman's Algorithm 1 only require enough disk space for the array. Given less space than cited above, they will run slower, since they will fill up the disk and have to interrupt expansion in order to update segments to reclaim disk space.

The main reason that TBBFS is faster than both BFFS and Algorithm 1 is that it generates significantly less disk I/O.

Using three threads to parallelize the expansion of individual segments, TBBFS took 25.73 hours on 14 unburned pancakes, compared to 23.22 hours expanding multiple segments in parallel. For 11 burned pancakes, the corresponding times were 15.37 hours and 15.63 hours. Thus, these two methods perform comparably with three to five threads.

TBBFS on 15 Unburned and 12 Burned Pancakes We also ran a complete TBBFS on 15 unburned and 12 burned pancakes. The 15 unburned pancake problem contains $15!$ states. With two threads parallelizing the expansion of individual segments, it took 41.67 days to run. The two-bit array occupied about 327 gigabytes, and the rest of our three terabytes were filled with update files.

The 12 burned pancake problem contains $12! \cdot 2^{12}$ states. With three threads parallelizing the expansion of individual segments, it took 39.58 days to run. The two-bit array occupied about 490 gigabytes, with the rest of the disks filled with update files.

The 15 unburned pancake problem was run with the non-interruptible version of TBBFS. The successful run on the 12 burned pancake problem was made with the interruptible version described above.

Table 1 summarizes all the known results for the pancake problems. The first column gives the number of pancakes, the second column the diameter of the corresponding unburned pancake problem, and the third column the diameter of the burned pancake problem. The data below the lines in the second and third columns were previously unknown.

A Surprising Anomaly It has been conjectured by Cohen and Blum (1992) that the most difficult stack of burned pancakes to solve is correctly sorted, but with all the burned sides face up. We verified this conjecture for 11 and 12 burned pancakes, but discovered the following anomaly. For 6 through 10, and also 12 burned pancakes, this hardest state is the only state at the maximum distance from the goal state. For 11 burned pancakes, however, there are 36 different states at the maximum distance from the goal state. We have no explanation for this anomaly.

Pancakes	Unburned	Burned
1	0	1
2	1	4
3	3	6
4	4	8
5	5	10
6	7	12
7	8	14
8	9	15
9	10	17
10	11	18
11	13	19
12	14	21
13	15	
14	16	
15	17	

Table 1: Problem-Space Diameters for Pancake Problems

Rubik's Cube Experiments

The standard 3x3x3 Rubik's Cube contains 4.3252×10^{19} states, and is too large to search exhaustively. Both we and Kunkle and Cooperman, however, have done complete breadth-first searches of large subspaces of this problem.

Two-Bit Breadth-First Search The 3x3x3 Rubik's Cube consists of eight corner cubies, with three faces each, and 12 edge cubies, with two faces each. If we ignore the corner cubies, and consider only the edge cubies, the resulting subspace contains $12! \cdot 2^{11} \approx 9.81 \times 10^{11}$ states. The edge cubies can be permuted arbitrarily, and can be in either of two different orientations. The orientation of one cubie is determined by the positions and orientations of the other 11.

The operators of this problem have significantly less locality than those in the pancake problems. We define each array segment by the three cubies occupying edges that intersect at a single point. Each such segment contains about 743 million elements. The branching factor is 18 moves, representing 90, 180, and 270 degree twists of each of six different faces. Since the three edge cubies that define a segment are only affected by moves of three faces, moves of the other three faces generate children in the same segment as their parent. Thus, half the moves generate child nodes in the same segment as their parent, and the other half generate children in non-resident array segments.

We performed a complete breadth-first search of this problem space with the interruptible version of TBBFS, using three threads to expand individual segments in parallel. It took about 35.125 days, using all three terabytes of our disk space. The number of unique nodes at each depth is shown in Table 2. These values were first determined by Tomas Rokicki, using a number of symmetries of the cube.

Kunkle and Cooperman's Rubik's Cube Search Kunkle and Cooperman applied their Algorithm 1 to a complete BFS of a similarly-sized subspace of Rubik's Cube, with 1.358×10^{12} states, allocating four bits per state. Their algorithm ran for 63 hours on a cluster with 128 processors, 256 gigabytes of memory, and seven terabytes of disk space.

Depth	Unique States
0	1
1	18
2	243
3	3240
4	42807
5	555866
6	7070103
7	87801812
8	1050559626
9	11588911021
10	110409721989
11	552734197682
12	304786076626
13	330335518
14	248

Table 2: Rubik's Cube Subspace Defined by Edge Cubies

Comparing These Two Searches These two searches are different, and were run on very different machines, but a comparison may nevertheless be made. The basic unit of work is a node generation. TBBFS expanded 9.81×10^{11} states with a branching factor of 18 moves, for a total of 1.7658×10^{13} node generations. Kunkle and Cooperman's algorithm expanded 1.358×10^{12} nodes with a branching factor of 12 moves, since 180 degree twists don't affect the state in their search, yielding 1.63×10^{13} node generations.

If we divide total node generations by the product of elapsed time and number of CPUs, we get an overall rate of $561,334$ node generations per CPU second for their search, compared to $2,909,238$ for ours. By this measure, our search ran over five times faster. Counting only computation time, their implementation generates 5 to 10 million nodes per CPU second on their machine, while ours generates about 10.6 million nodes per CPU second on our machine. Our algorithm generates only half as many updates on disk, however, and consumes half as much disk space for the array. Thus, some of the difference in overall node generation rate may be due to higher communication and I/O costs between nodes in a cluster and an external file server, compared to a workstation with internal disks.

Conclusions

We present a correct breadth-first search algorithm, which we call two-bit BFS (TBBFS), that requires only two bits of storage per problem state. We also show how to minimize the disk I/O required by this algorithm, and how to parallelize the expansion of individual array segments. TBBFS is most appropriate for complete breadth-first searches, used for example to determine the radius or diameter of a problem space, or to compute a pattern database heuristic. For problem spaces where the ratio of the total number of states to the maximum width of the problem space is relatively small, such as the pancake problems or Rubik's Cube for example, TBBFS requires less disk space than breadth-first frontier search, or a very similar algorithm by Kunkle and Cooperman that stores the depth of each state in the search.

We implemented TBBFS on the burned and unburned pancake problems, and determined for the first time the diameter of these problems for 14 and 15 unburned pancakes, and 11 and 12 burned pancakes. While breadth-first frontier search with delayed duplicated detection is able to solve the 14 unburned and 11 burned pancake problems, it takes longer than TBBFS, uses more disk space, and is not able to solve the larger problems with our resources. Kunkle and Cooperman's algorithm takes two to three times longer to solve these problems on our machine. Finally, we performed a complete breadth-first search of the subspace of Rubik's Cube defined by the edge cubies. Our algorithm generates less than half the disk I/O of Kunkle and Cooperman's Rubik's Cube search, uses less disk space, and runs faster.

Acknowledgments

This research was supported by NSF Grant IIS-0713178. Thanks to Satish Gupta and IBM for providing the machine these experiments were run on, and to Yuval Tamir, Eddie Kohler, and Peter Schultze for answering systems questions.

References

Cohen, D., and Blum, M. 1992. Improved bounds for sorting pancakes under a conjecture. Manuscript, Computer Science Division, University of California, Berkeley.

Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Dweighter, H. 1975. Elementary problem e2569. *American Mathematical Monthly* 82(10):1010.

Felner, A.; Korf, R.; Meshulam, R.; and Holte, R. 2007. Compressing pattern databases. *JAIR* 30:213–247.

Garey, M.; Johnson, D.; and Lin, S. 1977. Comments on elementary problem e2569. *American Mathematical Monthly* 84(4):296.

Gates, W., and Papadimitriou, C. 1979. Bounds for sorting by prefix reversal. *Discrete Mathematics* 27:47–57.

Heydari, M., and Sudborough, I. 1997. On the diameter of the pancake network. *Journal of Algorithms* 25:67–94.

Korf, R., and Felner, A. 2007. Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem. In *IJCAI-07*, 2334–2329.

Korf, R., and Shultze, P. 2005. Large-scale, parallel breadth-first search. In *AAAI-05*, 1380–1385.

Korf, R.; Zhang, W.; Thayer, I.; and Hohwald, H. 2005. Frontier search. *J.A.C.M.* 52(5):715–748.

Korf, R. 2003. Delayed duplicate detection: Extended abstract. In *IJCAI-03*, 1539–1541.

Kunkle, D., and Cooperman, G. 2007. Twenty-six moves suffice for rubik's cube. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISAC-07)*, 235–242.

Robinson, E.; Kunkle, D.; and Cooperman, G. 2007. A comparative analysis of parallel disk-based methods for enumerating implicit graphs. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation (PASCO-07)*, 78–87.