A Comparative Analysis of Parallel Disk-Based Methods for Enumerating Implicit Graphs

Eric Robinson
College of Computer Science
Northeastern University
Boston, MA 02115 / USA
tivadar@ccs.neu.edu

Daniel Kunkle
College of Computer Science
Northeastern University
Boston, MA 02115 / USA
kunkle@ccs.neu.edu

Gene Cooperman
College of Computer Science
Northeastern University
Boston, MA 02115 / USA
gene@ccs.neu.edu

ABSTRACT

It is only in the last five years that researchers have begun to use disk-based search techniques on a large scale. The primary examples of its use come from symbolic algebra and from artificial intelligence. In the field of parallel search, disk-based search has been forced on researchers because the historical growth in the amount of RAM per CPU core has now stopped. Indeed, the current trend toward multi-core CPUs now threatens to take us backwards.

This article makes an original contribution to the design of disk-based parallel search algorithms. It presents a survey of disk-based techniques side-by-side, for the first time. This allows researchers to choose from a menu of techniques, and also to create new hybrid algorithms from the building blocks presented here.

Categories and Subject Descriptors: I.1.2 [Symbolic and Algebraic Manipulation]: Algebraic algorithms, I.2.8 [Artificial Intelligence]: Graph and tree search strategies

General Terms: Algorithms

Keywords: parallel, disk-based, search, enumeration, implicit graphs

1. INTRODUCTION

Search techniques are at the heart of many problems in symbolic algebra. As search problems grow, they require both more time and more space. There has been much work in the use of parallelism to enable larger search. However, there has been considerably less work on managing the explosion of search states that occur. In symbolic algebra, this is usually known as *intermediate swell*, and it is a problem that occurs throughout the field.

Examples of large search-type problems with intermediate swell occur in Gröbner bases, in theorem proving, and in applications using heuristic search. In addition, the area of computational group theory provides an especially rich selection of applications. This includes the constructions of permutation group representations from matrix group representations, condensation methods for representation theory, and variations of Sims's method of base and strong generating set for high degree permutations. These methods of computational group theory are often applied to the sporadic simple groups, a natural ladder of challenge problems [2, 4, 5, 6, 7, 13, 18, 21, 22]. (The finite simple groups form the building blocks from which all other finite groups can be constructed.)

By an *implicit graph* we mean the graph of the state space that is produced during search and enumeration problems. Typically, the search begins with a single state and a set of generators. The generators are functions that produce a new state from a known one. Each state is a node of the graph. If a state S_1 is generated from an old state S_2, then we include an edge (S_1, S_2). The state S_2 can be the same as a previously seen state, and so the implicit graph will usually have many cycles. The search continues through repeated application of the generators to new states, until no additional new states are discovered.

One critical aspect in search-type problems is determining, as one arrives at a search state, whether that search state has been seen previously or not. In the case of depth-first search, one backtracks upon seeing an old state, and in the case of breadth-first search, one eliminates previously seen states from the current frontier.

The problem of deciding whether one has previously seen a state is known as *duplicate elimination*. This requires a table of previously seen states. A common technique to implement duplicate elimination is the use of hash tables. However, while hash tables are efficient RAM-based data structures, they are notoriously inefficient as disk-based data structures. Hash tables incur many random accesses, which require one to load random disk blocks, for the purpose of accessing a single search state within that disk block.

This situation has traditionally led researchers to look for clever techniques to compress the table of previously seen states. This allows them to fit larger search spaces purely within RAM. However, the historical growth in RAM per CPU core has now halted. Other approaches must be used if we are to conquer ever larger search applications.

Further, the trend toward multi-core CPUs also potentially limits this approach of compressing tables of previously seen states to fit in RAM. As more CPU cores request more random accesses to a single RAM subsystem, the bandwidth of the RAM subsystem no longer keeps up with the increasing pressure on memory.

Figure 1 demonstrates this issue by illustrating several of the available search techniques, which are described in

PASCO'07, July 27–28, 2007, London, Ontario, Canada.

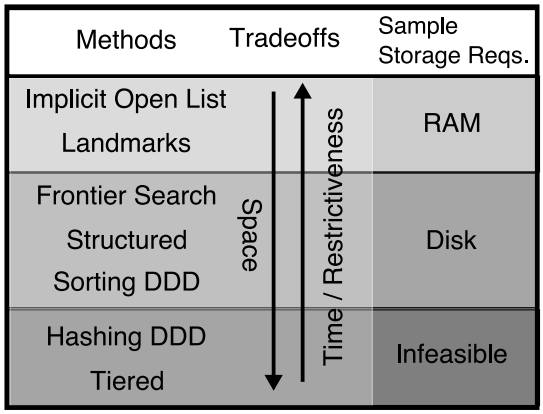

Figure 1: Search and enumeration methods and their tradeoffs

detail in the body. Recall that a *perfect hash function* is a hash function that has no collisions. The search techniques provide a natural time-space tradeoff. (Such time-space tradeoffs are well-known in a variety of algorithms.) The methods near the top typically use less memory, but require additional computation, or have additional constraints (such as certain hash functions, or constraints on the set of generators).

In practice, a given computer has limited RAM and disk space. The shaded regions illustrate the regions of feasibility in pure RAM and in a disk-based algorithm. These regions of feasibility are dependent on the scale of the problem, and the available computing resources. Due to time-space tradeoffs, the use of additional space from disk will often make an algorithm faster (even though disk is slower than RAM).

To further motivate the use of disk, note that the aggregate bandwidth of 50 local disks is about 50×50 MB/s = 2.5 GB/s. Thus 50 local disks provide an aggregate bandwidth similar to that of a single extremely large RAM subsystem. Of course, this refers only to streaming access, and not to random access. Much of the body of this paper is concerned with presenting disk-based algorithms that capture this emphasis on streaming access.

2. UNDERLYING TECHNIQUES

All of the methods we analyze here lend themselves to parallel implementations, which are often necessary to effectively address problems of very large scale. Though the algorithm descriptions may not explicitly state the nature of that parallelism, it is usually the case that the parallelism is a natural extension of a few fundamental techniques, particularly parallel breadth-first search and distributed hash tables. We briefly review these techniques, as a foundation for the survey that follows.

2.1 Parallel Breadth-First Search

Parallel breadth-first search requires a master node to perform a piece of the breadth-first search to some shallow depth. Once finished, the current frontier is evenly partitioned amongst the nodes in the cluster and each node generates the children of the values it receives. After this has been done, duplicates in the computed values are removed and the values at the next level in the breadth-first search are generated. This is repeated until all nodes have no new values for which to generate children. At this point, the frontier is empty and the breadth-first search has been completed.

Load balancing can be performed during the computation if a single node gets too large or small of a piece of the current frontier.

2.2 Distributed Hash Tables

A distributed hash table is one in which the hash function is divided into two components. The upper bits of the hash for a value determine the machine on which the value resides, the lower bits determine where in that machine's hash that value occurs. Since passing single values across the network incurs a latency penalty, typically hash checks are batched and performed when there are a sufficient number of values to be checked by the node in question.

Distributed hash tables work particularly well with parallel breadth-first search, as the entire frontier is checked for duplicates at the same time.

3. ALGORITHM SURVEY

Here we present a range of techniques for the search and enumeration of very large implicit graphs. These graphs exceed the size of main memory, and require a more compact representation, the use of disk, or both. Typical graph sizes range from billions to trillions.

Table 1 lists each method, along with certain properties of those methods, including: whether duplicate detection is done immediately or delayed; restrictions the method places on the set of appropriate problems; and, the extent to which states are compressed.

3.1 Landmarks

Instead of storing all states in the graph being generated, one can instead store only a small fraction of states, a method known as *landmarks* [4, 7]. These landmark states are typically chosen using a hash function. For example, to store only one-eighth of the space, only those states that have a hash value where the lower three bits are all zero are saved.

Non-landmark states are stored as a shortest path from that state to the closest landmark state (a more compact representation). The closest landmark state, and a path to it, are typically determined though a small breadth-first search.

When a new state is generated, it is compared to the known states for its closest landmark to determine if it is a duplicate. Duplicate states are ignored, and non-duplicates are saved in the open list.

This method is especially effective when the state representation is very large, and when there are few generators. In this case, a word from any state to its closest landmark will be much smaller than the associated state itself, saving a large amount of space. This comes at the cost of additional generator applications when determining a path to the closest landmark. Therefore, this technique is also most applicable in cases where generator application is relatively fast.

3.2 Storing Level Modulo 3

This method was introduced by Cooperman and Finkelstein as a compact representation for Cayley graphs, along

Table 1: Search and Enumeration Techniques

Method	Goal	Duplicate Detection	Restrictions	Compression	References
Landmarks	Reduce space to fit in memory	Immediate	Works best with large states and fewer generator	Save a fraction of states in full, the rest compressed as words	[4, 7, 21, 22]
Level mod 3	Compact representation of whole search space	Immediate	Needs a perfect hash function	All states as 2-bit value	[3]
Structured DD	Check new states against small subset of previous states in-core	Immediate	Needs definition of subspaces based on graph locality	No compression	[23, 24]
Sorting-based DDD	Works for any graph	Delayed	Entire space fits on disk	No compression	[1, 9, 20]
Hash-based DDD	Avoid sorting in DDD	Delayed	Needs perfect hash function	No compression	[9, 10]
Tiered DD	Reduce the number of duplicate detection passes	Immediate and Delayed	Needs (imperfect) hash function in memory	No compression	[18, 19]
Frontier Search	Only check for duplicates at current level	Delayed	Needs all inverse generators as generators	No compression	[9, 11]
Implicit Open List	Reconstruct open list from hash table instead of saving it explicitly	Delayed	Needs a perfect invertible hash function	All states as 2-bit value	[12]

with a generalization to Schreier coset graphs [3]. It is similar in ways to landmarks, but uses an even more compact representation of states.

The method uses a perfect hash function to associate every state with a single two-bit value. While performing a breadth-first search of the graph, we store for each state its level modulo three. So, the *home* state has a level of zero, its neighbors a value of one, etc. The fourth possible value of the two bits is used to identify states which have not been seen before.

Typically, this hash table fits entirely within memory, and acts as both a method for duplicate detection, as well as a compact representation of the entire graph. Once the breadth-first search has been completed, we can quickly find a shortest path from any given state to the home state in the following way. First, apply all generators to the given state. Find the first generator that leads one level earlier (modulo three). This is the first generator in the word being computed. By iteratively applying this process, a word leading back to the home state can be found.

3.3 Sorting-based DDD

Sorting-based delayed duplicate detection was the first technique developed specifically to allow the use of disk, which has been traditionally avoided because the high latency of disk does not allow for random access patterns.

Roscoe [20] demonstrated the efficiency of this technique for in-core methods where the search overflows into virtual memory. Korf [9] used an explicitly disk-based version to solve sliding tile puzzle and Towers of Hanoi type problems.

Rather than immediately determining if a newly generated state is a duplicate, that state is instead appended to a buffer on disk. Once an entire level of of the breadth-first search is completed, the buffer is sorted, using standard external sorting methods [1]. Once sorted, the new states are compared to all existing states by linearly scanning through the two sorted lists. The non-duplicate states then represent the open list from which the search is continued.

This method is one of the most general for handling very large graphs. Unlike many other methods, it does not place any restrictions on the problem (such as the need for inverse generators, or graph locality), and does not require a hash function.

3.4 Hash-based DDD

Korf [9, 10] introduced hash-based delayed duplicate detection to avoid the cost of externally sorting states. This savings can be significant, as sorting can often dominate the search time for sorting-based DDD.

The method uses two hash functions, often implemented by by splitting a single perfect hash into its high-order and low-order bits. As new states are generated they are placed into separate files, based on the first hash function. This guarantees that all duplicate nodes will occur within the same file. Then, instead of sorting these buffered states, the existing and new states are compared using the second hash function. Each hash value is associated with a single bit in memory that denotes whether or not that corresponding state has been previously generated. The non-duplicate new states are written back to disk as the open list, and the search continues.

3.5 Structured Duplicate Detection

Zhou and Hanson [23] introduced this method and initially applied it to the problem of finding shortest paths in sliding tile puzzles.

Unlike previous methods developed specifically to utilize disk as the primary means of storage, structure duplicate detection allows duplicates to be identified immediately, instead of delaying this decision.

The method works by exploiting the structure of the graph to localize the detection of duplicates to a small portion of

the previously generated states. For example, consider the problem of the eight-puzzle: a sliding tile puzzle containing the numbers one through eight in a 3×3 grid, with one space blank, where the goal is to restore the natural order of the tiles. While enumerating the graph associated with this puzzle, the application of a generator (i.e. sliding one tile) can only produce a state where the location Of the blank space differs by only one row or one column. Therefore, we need only check for duplicates in those sets of states with the blank in one of these positions. The partitioning of the graph is defined such that the set of states required for duplicate detection fit in main memory.

This method requires that the graph have sufficient locality. Without such locality, it would be impossible to partition the graph in such a way that the set of possible successor states fit in memory. Zhou and Hansen [24] give an automatic method for determining a suitable partitioning, assuming the graph has sufficient locality.

3.6 Tiered Duplicate Detection

Robinson and Cooperman [18] introduced tiered duplicate detection as a method to speedup the enumeration of the Baby Monster sporadic simple group, and more recently applied it to the problem of the Fischer_{23} group [19].

Instead of automatically storing all generated states for delayed duplicate detection, an in-core imperfect hash function is used as a first pass. If the new state hashes to an unseen value, it is not a duplicate and can be immediately placed in the open queue for further generation. If, on the other hand, the hash value results in a collision, the value *may* be a duplicate. Because the hash function is imperfect, non-duplicate states may collied. States producing such a collision are stored for delayed duplicate detection as necessary.

This method speeds the search because it does not stop to do delayed duplicate detection for each level of the breadth-first search. Instead, it can continue until the open list is emptied, which typically results in generating many levels in one pass.

This method is especially applicable to problems where there is no efficient perfect hash, or when such a hash would exceed the size of memory.

3.7 Frontier Search

Korf [9, 11] introduced frontier search as a means for reducing the scope of duplicate detection to only the current level of a breadth-first search, instead of all existing states.

This method requires that the inverses of all generators be generators themselves. With this restriction, newly generated states can occur at one of three levels in a breadth-first search, relative to the parent state: one level previous; the same level; or, one level later. This implies that we need only check for duplicates in those three levels, and we can safely ignore all previous levels.

Frontier search goes one step further, eliminating the need to check against the previous level as well. This is accomplished by storing a set of *used operator bits* for every state. There is one such bit for every generator, where a set bit represents a generator that is known to return to the previous level. These bits are set by marking the inverse of the generator that was used to generate that state. Further, when duplicate states are found, the states are combined into a single state with the union of the used operator bits of the duplicates.

3.8 Implicit Open List

Kunkle and Cooperman introduced this method to make feasible a large breadth-first search for proving an upper bound of 26 for all solutions to Rubik's Cube [12].

In general, all of the search methods above explicitly store the open list, i.e. the newly generated non-duplicate states. For some very large graphs, especially with large branching factors, this open list will exceed available disk space. This technique avoids storing this open list, and allows a breadth-first search that includes levels that do not fit on available disk.

Using a perfect invertible hash function, we associate two *frontier bits* with each state. These two bits will track which states are in the open list. Because the hash function is invertible, we can reconstruct the open list by scanning the table. We require two bits instead of one because we are using delayed duplicate detection, and the open list produced by a single level of the BFS may not fit on disk. The algorithm proceeds iteratively in the following way:

1. Scan through the table of frontier bits, generating new states from states with the first frontier bit set. Store the states on disk, continuing until an entire level is generated, or the disk is full.

2. Read states from disk and eliminate duplicates (usually using another bit from the same hash function). Non-duplicate states have their second frontier bit set.

3. If an entire level has been generated, rotate the frontier bits by assigning the value of the second frontier bit to the first and clearing the second bit. If only a partial level has been generated, return to Step 1 without rotating the frontier bits.

For even greater efficiency, the algorithm can dynamically switch between using an explicit or implicit open list. If the explicit representation of the open list is smaller than the hash table of frontier bits, it is faster to save those states. If the explicit open list becomes larger, it is more efficient to scan the table and reconstruct the open list.

Note that an implicit open list can also be achieved by storing the level in the BFS for each state, and generating children marked with the level. This hash table will also act as a compact representation of the entire graph once the search is complete. However, it requires increasing space with deeper searches, whereas the method presented above uses only two bits, regardless of the structure and scale of the graph.

4. THEORETICAL ANALYSIS

While previous works have examined all of the search space enumeration techniques presented in Section 3, they fail to provide a comparison between these techniques. In cases where a comparison can be found, this comparison is rarely fair. Generally these comparisons do not look at the complete space of techniques or problem instances. This leads to one technique appearing to be superior, when in reality, it may just be the case that not all methods have been examined, or that the technique is only superior for a certain class of problems.

We seek a fair and uniform method for comparing search space enumeration techniques. In order to find this, we define a "Big Oh" for search space enumeration. The techniques examined generally fall into one of two categories. Either the technique tries to reduce the space used to fit in aggregate RAM (or even disk), increasing the number of generator applications in the process, or the technique uses disk for the additional storage requirements, forcing out-of-core value accesses (typically streamed). Given this, the two key elements in any search technique are the time spent streaming disk, along with the time spent applying generators.

To compute these times, three different classes of parameters are considered. Architectural parameters are those specific to the cluster on which the computation is being performed. Search space parameters are those specific to the enumeration being performed (though they may have some dependence on the architecture as well). Finally, algorithmic parameters are those specific to algorithm used to perform the enumeration. The algorithmic parameters, in many cases, are derived from parameters of the other two classes. Table 2 shows a listing of all such parameters of each class.

While some of these parameters are self-explanatory, others require an explanation. The list below defines each parameter in the context of search space enumeration:

Search Size The expected number of values to be enumerated, or found, in the search space.

Branching Factor The average out-degree of a node in the graph, typically the number of generators.

Value Size The size of an individual value in the search in bytes.

Edge Bandwidth The number of edges that can be generated, or generator applications that can be performed in a second.

Disk Bandwidth The number of bytes that can be streamed from disk in a second.

Number of Nodes The number of machines in the cluster, or number of machines performing the enumeration.

Memory Size The size of available memory on an individual machine in the cluster.

Disk Size The size of available disk on an individual machine in the cluster.

Generator Apps The number of generator applications performed during the life of the algorithm.

Data Access The amount of data streamed during the life of the algorithm.

Memory Required The memory required by the algorithm per machine.

Storage Required The total amount of storage required by the algorithm. In most cases, this can be either memory or disk.

Once the values of these parameters are known, the overall runtime for an individual enumeration can be predicted, with accuracy sufficient to compare methods.

$$Time = \frac{X_G}{BW_G \times N} + \frac{X_A}{BW_D \times N}$$

Assuming the search space and architectural parameters are fixed (you're looking to solve a particular problem given a particular cluster), the runtime for the search given each algorithm can be predicted by swapping parameters. From here, determining the best algorithm for a particular search space and cluster becomes an easy matter.

While determining the search space and architectural parameters is a straightforward matter, finding the algorithmic parameters is not always as simple. As mentioned before, these parameters are typically derived, rather than constant. Here we look at each enumeration technique and solve for the value of each parameter for each technique. Many of these methods have tunable components as well, meaning the more memory that is available, the faster they will perform. This is specified per method and a new variable is typically used to denote the tunable component.

4.1 Level Sizes in Breadth-First Search

For many of these techniques, it is important to know the number of values at each depth in the breadth first search. If a random graph is assumed, one in which each node goes to BF other random nodes, the size of each level can be defined recursively as follows,

$$\begin{aligned} L_1 &= 1 \\ L_i &= BF \times L_{i-1} \times \frac{|S| - \sum_{j=1}^{i-1} L_j - \frac{L_i}{2}}{|S|} \\ &= 2 \times BF \times L_{i-1} \times \frac{N - \sum_{j=1}^{i-1} L_j}{2 \times |S| + BF \times L_{i-1}} \end{aligned}$$

While at first glance, it looks as if this equation will approach $|S|$ values in all levels, this is not the case. This is because the assumptions we made about a random graph are not true of search spaces in general. While it is the case that the out degrees behave as they would in a random graph, no guarantee is made that each node has an in degree of at least one. This leads to the above algorithm terminating ($L_i < 1$) when the connected components of the graph are found. This is easily remedied by artificially raising $|S|$ until the algorithm terminates with the correct number of points. In addition, a more exact (and more complex) formulation is being worked on for this algorithm, but it is not complete at the current time.

4.2 Implicit Open List

As described above, the implicit open list uses only two bits to both determine if an element in the search has been seen previously as well as determine what is on the frontier. It requires a perfect invertible hash.

4.2.1 Storage Required

The implicit open list technique utilizes just 2 bits per element, assuming a perfectly dense hash is available. It may also use a near-perfectly dense hash at the cost of more storage. Where a perfectly dense hash is available, the basic implicit open list technique requires

$$R = \frac{2}{8} \times |S|.$$

If the space is to be stored using level modulo three technique, then the representation for the values and the open list requires

$$R = \frac{3}{8} \times |S|.$$

If the space is to be stored using the level technique, then the representation for the values and the open list requires

$$R = \frac{log_{BF}(|S|)}{8} \times |S|.$$

Table 2: The Classes and Parameters

Search Space		Architectural		Algorithmic	
parameter	name	parameter	name	parameter	name
Search Size	$\|S\|$	Disk Bandwidth	BW_D	Generator Apps	X_G
Branching Factor	BF	Number of Nodes	N	Data Access	X_A
Value Size	$\|V\|$	Memory Size	$\|M\|$	Memory Required	R_M
Edge Bandwidth	BW_G	Disk Size	$\|D\|$	Storage Required	R

The remaining space is typically used as a value buffer in this technique. This technique can be performed either in memory or on disk, depending on the size of your search. For the remainder of the analysis, it is assumed that the basic implicit open list technique is used. The other values can be easily substituted if needed.

4.2.2 Data Access

Assuming the hash is disk-based, there are three components to the data access portion of the implicit open list method. The first component is the base reads/writes required for a breadth-first search,

$$X'_A = \overbrace{2}^{\text{read/write}} \times \overbrace{|S| \times BF \times |V|}^{\text{data generated}}.$$

The second component is the scans through the hash table at each level in the BFS to find elements on the frontier,

$$X''_A = \overbrace{\frac{2}{8} \times |S|}^{\text{size of hash}} \times \overbrace{log_{BF}(|S|)}^{\text{search depth}}.$$

The final component is the merging of the full values with the hash when there is no longer any room to store those values on disk,

$$X'''_A = \overbrace{2}^{\text{read/write}} \times \overbrace{\frac{2}{8} \times |S|}^{\text{size of hash}} \times \overbrace{\frac{|S| \times BF \times |V|}{N \times |D| - \frac{2}{8} \times |S|}}^{\text{number of scans}}.$$

The number of scans is computed by determining what portion of the data generated in the search will fit on disk at one time.

The total data access for the implicit open list technique is therefore

$$X_A = X'_A + X''_A + X'''_A.$$

4.2.3 Generator Apps

This method requires the base number of generator applications,

$$X_G = \overbrace{BF \times |S|}^{\text{BFS component}}$$

4.3 Landmarks

The landmarks technique does not require a perfect hash, but it does require that the inverses of the generators are present (though they need not be generators themselves).

4.3.1 Storage Required

The amount of memory used is tunable. With increased memory usage, more landmarks can be stored, resulting in fewer generator applications. Where the landmark ratio is LM, the amount of memory required is

$$R_M = (\overbrace{LM \times |V|}^{\text{landmarks}} + \overbrace{(1 - LM) \times log_{BF}(L))}^{\text{non-landmarks}} \times |S|.$$

The landmarks technique can be performed on disk as well as in memory, however, the landmarks themselves must always fit in memory and there must be reasonably few of them.

4.3.2 Data Access

If performed on disk, this technique requires accessing the non-landmarks repeatedly, once for each subsequent level in the breadth-first search tree. This implies

$$X'_A = \overbrace{\sum_{c=1}^{|L|}((|L| - c) \times L_c)}^{\text{value accesses}} \times \overbrace{log_{BF}(L)}^{\text{non-LM size}}.$$

In addition, during the standard BFS, all values, even duplicates are written to and read from disk to be checked as duplicates, but only non-duplicates are written back and read later to be processed. This component requires

$$\begin{aligned} X''_A = \; & (2 \times \log_{BF}(L) \times BF \times |S|) + && \text{(sorting)} \\ & (2 \times \log_{BF}(L) \times BF \times |S|) + && \text{(duplicate detection)} \\ & (\log_{BF}(L) \times |S|). && \text{(BFS write)} \end{aligned}$$

This leads to a total disk access component of

$$X_A = X'_A + X''_A.$$

4.3.3 Generator Apps

At each edge, the node generated by that edge must be traced to the closest landmark to determine if it is a duplicate. For each new node, the path must be followed in reverse to reproduce that value later as well when its children are to be computed. The number of generator applications for this technique is

$$X_G = \overbrace{(|S| \times BF) \times LM^{-1}}^{\text{duplicate detection}} + \overbrace{|S| \times LM^{-1}}^{\text{expansion}}.$$

4.4 Sorting-Based DDD

When all the values are able to be stored on disk, even if no perfect hash is available, sorting-based delayed duplicate detection can enumerate the entire space without performing any additional generator applications.

4.4.1 Storage Required

This method, while capable of being performed in main memory, was intended for disk. The amount of memory per

machine required is relatively small. There are two components to consider, the buffer of frontier values to be sent to each machine along with the memory required in the external sort.

For the frontier values, the size selected must be large enough to prevent a bottleneck in message passing due to the latency of sending small packets. Typically, a reasonable message size is around $100KB$. More generally, where messages must be at least $|M|$ bytes to avoid latency,

$$R'_M = N \times |M|.$$

Though external sorting can operate with any amount of memory, it performs best when each file to be merged fits fully into memory. Typically an individual machine can have 100-200 files open at once. More generally, where the total number of files that can be open at a single time on a machine is limited to F,

$$R''_M = \frac{|S| \times |V|}{N \times F}.$$

Since these two memory components are used at different times during the enumeration and are not persistent, they can overlap,

$$R_M = \max(R'_M, R''_M).$$

In terms of disk usage, all of the values in the search space must fit on disk. This requires a disk usage of

$$R' = |S| \times |V|.$$

In addition, the largest level in the breadth-first search (including duplicates) must also fit on disk at the same time. This requires a disk usage of

$$R'' = \max(L) \times BF \times |V|.$$

These two terms combine to form the total disk usage for the method,

$$R = R' + R''.$$

4.4.2 Data Access

This technique requires accessing the values discovered in the search repeatedly, once for each subsequent level in the breadth-first search tree. This implies

$$X'_A = \overbrace{\sum_{c=1}^{|L|} ((|L| - c) \times L_c)}^{\text{value accesses}} \times |V|.$$

In addition, during the standard BFS, all values, even duplicates are written to and read from disk to be checked as duplicates, but only non-duplicates are written back and read later to be processed. This component requires

$$X''_A = \overbrace{2 \times |V| \times BF \times |S|}^{\text{sorting}} + \overbrace{2 \times |V| \times BF \times |S|}^{\text{duplicate detection}} + \overbrace{|V| \times |S|}^{\text{BFS write}}.$$

This leads to a total disk access component of

$$X_A = X'_A + X''_A.$$

4.4.3 Generator Apps

This method applies only one generator per edge in the graph. The number of generator applications is

$$X_G = |S| \times BF.$$

4.5 Hashing-Based DDD

Hashing-based delayed duplicate detection operates in the same manner as sorting-based DDD. However, it reduces the number of passes through disk, but relies on a reasonably compact hash in order to do this. Pieces of hash must be brought into memory as well during the computation.

4.5.1 Storage Required

This method was also intended for disk. On top of the memory required for frontier values, the amount of memory per machine depends on the number of elements in the perfect hash, $|H|$.

For the frontier values, the same equation used in sorting-based delayed duplicate detection is used for hashing-based delayed duplicate detection,

$$R'_M = N \times MS.$$

For the hashing component, H will denote the hash with $|H|$ 1 bit entries. In order to perform well, the individual blocks of entries for each hash block must be large enough to obtain disk streaming rather than random access. We will assume that each block must be at least $|B|$ bytes before being written to a file. If this is the case, the amount of memory required by the hash is

$$R''_M = \frac{|B| \times |H|}{8 \times |M|}.$$

These two components are both used during the generation of neighbors and not the duplicate detection phase. Therefore the total memory usage is

$$R_M = R'_M, + R''_M.$$

Hashing-based delayed duplicate detection requires the same amount of disk as sorting-based delayed duplicate detection.

4.5.2 Data Access

Just as in sorting-based delayed duplicate detection, hashing-based delayed duplicate detection also requires repeatedly accessing previously discovered elements.

$$X'_A = \overbrace{\sum_{c=1}^{|L|} ((|L| - c) \times L_c)}^{\text{value accesses}} \times |V|.$$

Hashing-based delayed duplicate detection also requires reading and writing for duplicate detection, however, it eliminates the sorting component. This requires

$$X''_A = \overbrace{2 \times |V| \times BF \times |S|}^{\text{duplicate detection}} + \overbrace{|V| \times |S|}^{\text{BFS write}}.$$

This leads to a total disk access component of

$$X_A = X'_A + X''_A.$$

4.5.3 Generator Apps

This technique applies only one generator per edge in the graph. The number of generator applications is

$$X_G = |S| \times BF.$$

4.6 Structured DDD

Structured delayed duplicate detection is not examined here. Its performance depends on the structure and locality of the search space. In addition, it does not lend itself to being easily parallelized. While it is of use when only a single machine is available and the search space has a lot of locality, it is not a general enough method for our purposes.

4.7 Frontier Search

Frontier search takes advantage of inverse generators to guarantee that all duplicates seen will occur in at most the two previous iterations in the breadth-first search.

4.7.1 Storage Required

This method requires the exact same amount of memory as either sorting-based or hashing-based delayed duplicate detection, depending on which underlying method is used.

Rather than storing all levels in the breadth-first search, this method requires storage of at most the previous level, the current level, and the current frontier. This leads to a total storage component of

$$\max_{j=1}^{|L|-2} \left(L_j + L_{j+1} + BF \times L_{j+2} \right).$$

4.7.2 Data Access

This method reduces the number of passes through previous elements to a constant (3). This leads to this portion of the access component being

$$X'_A = 3 \times |S| \times |V| - 2 \times L_i - L_{i-1}.$$

The access component for sorting/hashing is dependent on the method used in frontier search. It uses the same component as either sorting-based or hashing-based delayed duplicate detection. This leads to a total access component of

$$X_A = X'_A + X''_A.$$

4.7.3 Generator Apps

While this method performs the standard number of generator applications, it is important to note that this number may be higher due to the forced use of generator inverses increasing the branching factor. The number of generator applications for this method is

$$X_G = BF \times |S|.$$

4.8 Tiered Duplicate Detection

This technique uses delayed duplicate detection with a form of immediate duplicate detection that can produce false positives, indicating things are duplicates when they are not. By doing this, it discovers more points at each level in the search, requiring fewer passes through disk. However, more memory is used in the computation. Rather than the standard equations for levels presented above, this method using the following formulas, where HM is the hash multiple used,

$$La_i = \left(|S| - \sum_{j=1}^{i-1}(La_j) - \sum_{j=1}^{i-1}(Lb_j) \right) \times \text{(unseen points)}$$
$$\left(\frac{|S| \times HM - \sum_{j=1}^{i-1}(La_j) - \frac{La_i}{2}}{|S| \times HM} \right) \text{(hash emptiness)}$$

$$Lb_0 = 1$$

$$Lb_i = (La_i \times (BF - 1) + Lb_{i-1}) \times \text{(collisions in } La_i\text{)}$$
$$\left(\frac{|S| - \sum_{j=1}^{i-1}(La_j + Lb_j) - La_i - \frac{Lb_i}{2}}{N} \right) \text{(unseen percent)}$$

Once again, here the $|S|$ used in the computation of Lb must be augmented to compensate for disconnected nodes. The exact formulas for this are currently being worked on.

4.8.1 Storage Requirement

This technique can use either sorting-based or hashing-based duplicate detection. It uses the memory required by those techniques. On top of this, an imperfect hash must be stored in memory. The size of this hash is a tunable parameter, $HM \geq 1$. Given this parameter, the memory usage for the hash can be determined,

$$R'''_M = \frac{HM \times |S|}{8 \times N}.$$

This leads to a total memory usage of

$$R_M = R'_M + R''_M + R'''_M.$$

In terms of disk, this method must also store all values in full on disk. This requires a disk usage of

$$R' = |S| \times |V|.$$

In addition, the largest frontier must be stored, including duplicates. However, frontiers are computed differently in this approach. The amount of data requires is

$$R'' = \max_{j=0}^{|La|}(La_j \times (BF - 1) \times Lb_{j-1}).$$

This leads to a total disk usage of

$$R = R' + R''.$$

4.8.2 Disk Access

This technique reduces the number and size of each level in the search, the levels generated in La are not counted toward the total in this approach. However, the points there do need to be scanned each pass.

$$X'_A = \overbrace{\sum_{c=1}^{|La|} ((|La| - c) \times Lb_c + (|La| - c + 1) \times La_c)}^{\text{value accesses}} \times |V|.$$

This method requires a duplicate detection access component as well. This component is the same as that required by either hashing-based or sorting-based delayed duplicate detection, depending on which is used. This leads to a total disk access component of This leads to a total disk access component of

$$X_A = X'_A + X''_A.$$

4.8.3 Generator Apps

This method applies only one generator per edge in the graph. The number of generator applications is

$$X_G = |S| \times BF.$$

5. PROBLEM SURVEY

Here we give three examples of problems that the authors have applied parallel disk-based search and enumeration techniques to. These include: enumerating the Baby Monster group; enumerating the Fischer_{23} group; and proving an upper bound of 26 on solutions to Rubik's Cube.

5.1 Point Stabilizer Subgroup Chains

For many very large mathematical groups, it is infeasible to work with the group as a whole. For these groups a well-known divide and conquer algorithm exists that defines the group in terms of a point stabilizer subgroup chain. This algorithm works by computing a chain of subgroups, starting with the full group, and finishing with the identity. Typically, the largest portion of this chain of subgroups is the first subgroup along the chain. The computation of this first subgroup, however, can be performed using search space enumeration.

5.2 Baby Monster Enumeration

The Baby Monster is a mathematical group with 4.1×10^{33} elements. For the Baby Monster, the first subgroup in the point stabilizer subgroup chain must contain at least 1.4×10^{10} elements. Given that the size of an individual element in the smallest representation is around 550 bytes, the storage for this subgroup requires around 8 terabytes of space. Up until recently, it was thought that dealing with this large amount of data was infeasible. Now, with many disk-based techniques available, this initial subgroup has been discovered.

The enumeration of the first subgroup [18] in the point stabilizer subgroup chain was performed using Tiered Duplicate Detection with a hash multiple of 2. It was done on a cluster of 32 nodes with just 8 terabytes of aggregate disk (250 gigabytes per node) and 16 gigabytes of aggregate RAM (500 megabytes per node). Because the size of the search was approaching disk limits, a compression technique was used that reduced the size of an element from 550 bytes to around 30 bytes. This technique required the recomputation of values during the delayed duplicate detection phase, resulting in an additional $\sum_{i=1}^{|Lb|}(L_b) \times log_{BF}(|S|)$ generator applications. The branching factor for the Baby Monster problem was around 1.75. Tiered Duplicate Detection was used primarily because generator applications were performed through vector-matrix multiplications over very large matrices, and were for that reason very slow. The generator bandwidth for this problem was around 1250 generators per second.

With an assumed disk bandwidth of 50 megabytes per second, this computation was bound by the time to compute generators. Given the cluster information along with the problem size and branching factor, the time spent computing generators for the algorithm is 20 days. The actual computation time was 14 days, the decrease in time was the result of optimizing out some of the generator applications during the recomputation of values.

5.3 Fischer$_{23}$ Enumeration

Fischer$_{23}$ is another mathematical group, this time with 8.6×10^{19} elements. As in the Baby Monster Enumeration, a divide and conquer algorithm was used. The first subgroup in the point stabilizer subgroup chain, in this case, has 1.2×10^{10} elements. However, the elements in this case were only 100 bytes, and the generator bandwidth was around 2000 generators per second. Here, a cluster of 32 nodes with 640 gigabytes of aggregate disk (20 gigabytes per node) and 32 gigabytes of aggregate RAM (1 gigabyte per node) was used.

Once again, because the entire search space, which would typically take up 1.2 terabytes, was so close to fitting on disk, a compression technique was used on the values to allow them to take on average only 20-30 bytes. While this technique required some additional computation time, it allowed the entire search space to be stored on disk. This allowed for the enumeration of the first subgroup [19] to be performed using Tiered Duplicate Detection as well. The branching factor was again around 1.75.

With an assumed disk bandwidth of 50 megabytes per second, this computation was bound by the time to compute generators. Given the cluster information along with the problem size and branching factor, the time spent computing generators for the algorithm is 6 days. The actual computation time was 2 days, after optimizing out some of the generator applications.

5.4 Rubik's Cube

Proving bounds on the number of moves that suffice to solve any state of Rubik's Cube is a long standing challenge problem [8]. Cooperman and Finkelstein used the method "Level Mod 3" to show that the number of moves needed to solve Rubik's $2 \times 2 \times 2$ is 11 [3].

For the full Rubik's cube, previous work has proved a lower bound of 20 [17] and an upper bound of 27 [16, 14, 15]. Using some of the techniques presented here, Kunkle and Cooperman proved an improved upper bound of 26 [12]. All bounds are in the traditional face-turn metric. (A half turn and quarter turn each count as a single move.)

Rubik's Cube has approximately 4.3×10^{19} possible states. The most common approach for proving an upper bound on number of moves sufficing to solve any of these states is to define a chain of subgroups for the group corresponding to Rubik's Cube. Then, an upper bound for each of these subproblems is achieved by performing a breadth-first search for each of the subgroups, and corresponding cosets. These upper bounds can then be combined to produce an overall upper bound.

In previous work, subgroup chains have been chosen to minimize the size of the largest breath-first search that must be performed. For example, the work that proved an upper bound of 27 was based on a single subgroup of size near the square-root the size of the entire group. This resulted in two breadth-first searches on graphs with approximately 6.6×10^9 states each, which can be done within the limits of RAM on a cluster.

This recent work instead chose a relatively small subgroup, generated by using only the "squared" generators, which turn a face of the cube by 180 degrees. This subgroups is of size $663,552$, yielding over 65 trillion cosets. Using the 48 symmetries of the cube, this was reduced to approximately 1.5 trillion states to search over.

Using a perfect invertible hash function, a breadth-first search was performed using hash-based delayed duplicate detection and an implicit open list. The states were stored as four-byte values, yielding over 100 terabytes of data, with the disk usage at any given time being at most 7 terabytes. Generator application was highly optimized, to make efficient use of CPU cache, and was performed up to 10 million times per second. Given this, the computation was bound by I/O bandwidths. The entire computation used sixteen nodes with eight processors each (128 CPUs) and took approximately 65 hours total, or over 8000 CPU hours. It was shown that the diameter of the graph is 16.

6. CONCLUSION

We have presented a comparative analysis of methods for search and enumeration of very large implicit graphs. While earlier methods typically focused on using a more compact representation, allowing the search to fit within available RAM, newer techniques tend to focus on the efficient use of disk. This shift has arisen because the amount of available RAM per CPU has plateaued. With the advent of multi-core CPUs, this trend can be expected to continue, in fact, the amount of RAM per CPU may even decrease.

With an increase in the number and variety of enumeration algorithms comes a need for a framework to analyze their relative efficiency. The framework we provide here allows for this kind of analysis, taking into account the nature of the problem, the target computer architecture, and the algorithms themselves. With this framework, we can accomplish several related tasks:

- Determine which method, or combination of methods, is the best for a given problem and computer architecture.
- Predict the runtime of a method for a given problem and computer architecture.
- Determine what classes of problems are best approached with a given method and available computing resources.
- Guide the development of new methods, with respect to some target problems or computer architectures.

7. REFERENCES

[1] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM Trans. Database Syst.*, 8(2):255–265, 1983.

[2] J.H. Conway, R.T. Curtis, S.P. Norton, R.A. Parker, and R.A. Wilson. *Atlas of finite groups*. Clarendon Press, Oxford, 1985.

[3] G. Cooperman and L. Finkelstein. New methods for using Cayley graphs in interconnection networks. *Discrete Applied Mathematics*, 37/38:95–118, 1992. (special issue on Interconnection Networks).

[4] G. Cooperman, L. Finkelstein, M. Tselman, and B. York. Constructing permutation representations for matrix groups. *J. Symbolic Comput.*, 1997.

[5] G. Cooperman, W. Lempken, G. Michler, and M. Weller. A new existence proof of Janko's simple group j_4. In *Progress In Mathematics*, volume 173, pages 161–175. Birkhauser, 1999.

[6] G. Cooperman and E. Robinson. Memory-based and disk-based algorithms for very high degree permutation groups. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '03)*, pages 66–73. ACM Press, 2004.

[7] G. Cooperman and M. Tselman. New sequential and parallel algorithms for generating high dimension Hecke algebras using the condensation technique. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '96)*, pages 155–160. ACM Press, 1996.

[8] A. H. Frey Jr. and D. Singmaster. *Handbook of Cubik Math*. Enslow Publishers, 1982.

[9] R. E. Korf. Best-first frontier search with delayed duplicate detection. In *AAAI*, pages 650–657, 2004.

[10] R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *AAAI*, pages 1380–1385, 2005.

[11] R. E. Korf, W. Zhang, I. Thayer, and H. Hohwald. Frontier search. *J. ACM*, 52(5):715–748, 2005.

[12] D. Kunkle and G. Cooperman. Twenty-six moves suffice for Rubik's cube. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '07)*. ACM Press, 2007.

[13] F. Lübeck and M. Neunhöffer. Enumerating large orbits and direct condensation. *Experiment. Math*, 10:197–206, 2001.

[14] S. Radu. Solving Rubik's cube in 28 face turns. `http://cubezzz.homelinux.org/drupal/?q=node/view/37`, 2005.

[15] S. Radu. Rubik can be solved in 27f. `http://cubezzz.homelinux.org/drupal/?q=node/view/53`, 2006.

[16] M. Reid. New upper bounds. `http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/michael_reid__new_upper_bounds.html`, 1995.

[17] M. Reid. Superflip requires 20 face turns. `http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/michael_reid__superflip_requires_20_face_turns.html`, 1995.

[18] E. Robinson and G. Cooperman. A parallel architecture for disk-based computing over the Baby Monster and other large finite simple groups. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '06)*, pages 298–305. ACM Press, 2006.

[19] E. Robinson, G. Cooperman, and J. Müller. A disk-based parallel implementation for direct condensation of large permutation modules. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '07)*. ACM Press, 2007.

[20] A. W. Roscoe. Model-checking CSP. *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378, 1994.

[21] M. Weller. Construction of large permutation representations for matrix groups. In W. Jäger E. Krause, editor, *High Performance Computing in Science and Engineering '98*, pages 430–. Springer, 1999.

[22] M. Weller. Construction of large permutation representations for matrix groups ii. *Applicable Algebra in Engineering, Communication and Computing*, 11:463–488, 2001.

[23] R. Zhou and E. A. Hansen. Structured duplicate detection in external-memory graph search. In *AAAI*, pages 683–689, 2004.

[24] R. Zhou and E. A. Hansen. Domain-independent structured duplicate detection. In *AAAI*, 2006.