LARGE IMPLICIT STATE SPACE ENUMERATION:
OVERCOMING MEMORY AND DISK LIMITATIONS

A Thesis
Presented to the Faculty of the Graduate School
of the College of Computer Science
of Northeastern University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Eric Robinson
May 2008
Contents

1 Introduction 2

2 Implicit State Space Enumeration Background 5
   2.1 Enumeration Definitions 5
   2.2 Enumeration Components 7
   2.3 Node Discovery 7
      2.3.1 Extension to Disk 8
      2.3.2 Parallelization 8
   2.4 Duplicate Detection 8
      2.4.1 State-Based Duplicate Checking 8
      2.4.2 Using a Perfect Hash Function 9
      2.4.3 Using a Lossy Hash Technique 9
      2.4.4 Immediate versus Delayed Duplicate Detection 10

3 Previous Work: Enumeration Techniques 11
   3.1 Breadth-First Search with a Perfect Hash 12
   3.2 Sorting-based DDD 12
   3.3 Hash-based DDD 12
   3.4 Structured Duplicate Detection 13
   3.5 Level Modulo 3 14
   3.6 Implicit Open List 14
   3.7 Landmarks 15
   3.8 Frontier Search 16
   3.9 Tiered Duplicate Detection 17
   3.10 History of Previous Work 18

4 Tiered Duplicate Detection: A New Enumeration Technique 19
   4.1 The Tiered Duplicate Detection Algorithm 19
   4.2 Optimizations 23
   4.3 When to Use Tiered Duplicate Detection 24
5 The Space-Time Search Hierarchy: An Approach to Analysis 25
  5.1 The Space-Time Search Hierarchy ................................................. 26
  5.2 Parameters of Analysis ................................................................. 28
  5.3 Sampling with Replacement ......................................................... 30
  5.4 Level Sizes in Breadth-First Search ............................................. 31
  5.5 Use of Compressed States ......................................................... 32
  5.6 Use of Inverses ................................................................. 33

6 Static Analysis of Enumeration Techniques 33
  6.1 Breadth-First search with a Perfect Hash ...................................... 34
  6.2 Sorting-Based Delayed Duplicate Detection .................................. 35
  6.3 Hash-Based Delayed Duplicate Detection ..................................... 37
  6.4 Structured Duplicate Detection .................................................. 39
  6.5 Level Modulo Three ................................................................. 39
  6.6 Implicit Open List ................................................................. 40
  6.7 Landmarks ................................................................. 41
  6.8 Frontier Search ................................................................. 43
  6.9 Tiered Duplicate Detection ..................................................... 44
  6.10 Sensitivity of Analysis to Parameter Modification .......................... 50
      6.10.1 Effect of Growth in Available Storage ................................ 51
      6.10.2 Variance in the Estimated Search Size ................................ 58

7 General Software for Implicit State Space Enumeration 60
  7.1 Overview of Interfaces ................................................................. 61
  7.2 Automatic Prediction of Parameters ...................................... 63
  7.3 Application Interface ................................................................. 67
  7.4 Architecture Interface ............................................................... 72
  7.5 Algorithm Interface ................................................................. 75
  7.6 The Path Utility ................................................................. 80
  7.7 Implementation ................................................................. 82

8 Enumeration Applications in Group Theory 83
  8.1 Computational Group Theory Background .................................... 83
  8.2 Computational Group Theory Previous Work ................................ 91
8.3 Computational Group Theory Applications ........................................... 93
  8.3.1 The Thompson Group ................................................................. 93
  8.3.2 The Fischer Group $F_{i23}$ ......................................................... 94
  8.3.3 The Baby Monster Group ......................................................... 95
  8.3.4 The Janko Group $J_4$ ............................................................... 97
8.4 Static Analysis of Applications for Computational Group Theory .......... 98
  8.4.1 The Thompson Group ................................................................. 99
  8.4.2 The Fischer Group $F_{i23}$ ......................................................... 100
  8.4.3 The Baby Monster Group ......................................................... 105
  8.4.4 The Janko Group $J_4$ ............................................................... 107

9 Examples from Outside Computational Group Theory .......................... 109
  9.1 Puzzle Search ..................................................................................... 109
    9.1.1 Rubik’s Cube ........................................................................ 110
    9.1.2 Sliding Tile Puzzle .................................................................. 114
  9.2 Formal Verification ............................................................................ 116
    9.2.1 The N-Peterson Algorithm ....................................................... 119
    9.2.2 Summary of Results ................................................................ 122
    9.2.3 Algorithm–based Implementations of the N-Peterson Algorithm 124
    9.2.4 Program–based Implementations of the N-Peterson Algorithm .... 127
    9.2.5 Machine-based Implementations of the N-Peterson Algorithm .... 130

10 Conclusions ....................................................................................... 132

A Appendix ......................................................................................... 141
  A.1 Computational Group Theory Computation Information .................... 141
  A.2 Explicit State Model Checking Computation Information .................. 142
Statement of Thesis

This thesis presents a software implementation that provides an application-independent method for searching and enumerating implicit state spaces too large for distributed memory. When presented with a specific enumeration application and a description of the hardware resources available for the enumeration, the software automatically chooses the fastest enumeration technique. Implicit state spaces are formally defined in Section 2.1 but can be viewed as a compact representation for a large graph using only a small subset of states and a set of functions to generate edges in the graph.

In order to determine the best enumeration technique for an application, analytical formulas are derived in this thesis to predict the time and space requirements for many well-known search and enumeration techniques. These formulas take as input parameters a description of the enumeration to be performed as well as the hardware available for that enumeration. The time and space requirements for the enumeration are then derived by applying the formulas for a given set of parameters. The software offers an application-independent approach for estimating many of these parameters where they are not provided by the user. The size of the search space itself is a prominent example of one of these estimated parameters.

In the process of analyzing the many parallel and oftentimes disk-based techniques for search and enumeration, a natural space-time search hierarchy is uncovered. With this, one can see a trade-off between the amount of storage an enumeration technique requires and the number of computations that technique must perform. A gap was discovered in this hierarchy and a new technique for search and enumeration, tiered duplicate detection, is presented in this thesis to fill that gap.

Tiered duplicate detection is novel in that it requires fewer passes through disk than many competing techniques. It uses an in-core imperfect duplicate detection method, and has an out-of-core method for determining when errors have been made. This allows it to explore multiple depths in the search tree without having to access disk between each to perform duplicate detection as many of the earlier disk-based techniques had done. It was the preferred enumeration technique for two of the computational group theory applications presented in Section 8.3, the baby monster group and Fischer’s group $Fi_{23}$.

With this software implementation, applications in fields such as computational group theory, puzzle search, and implicit state model checking are considered. An in-depth examination of applications in the computational group theory field give evidence that the analysis and formulas derived in this thesis are accurate. For the applications considered in this thesis, these formulas are always accurate to within $\pm 50\%$, and typically the accuracy is within $\pm 25\%$. This is almost always accurate
Introduction

Although CPU clock speed is now stagnating, CPU performance continues to grow with Moore’s law due to the switch to multi-core processors in commodity CPU chips. The time to access memory and disk, however, is improving at a much slower rate. Even with today’s fastest commodity RAM, the run-time of many programs is limited by the time for streaming memory access rather than actual processor time. In programs that utilize disk, this speed discrepancy becomes even more apparent.

If performance of processors continues to grow Moore’s law, this problem can be expected to become much worse. Historically memory bandwidth has grown at a much slower rate than CPU power. In fact, the speed difference between the two has been growing at an exponential rate [76, 53]. This trend can be expected to continue into the future. This is because there are fundamental limitations to memory speed that do not apply to CPU power. These limitations stem mostly from the fact that access to RAM is traditionally off-chip, while CPU computation can and will remain on chip. In addition, the time to load a different row into the buffer inside a RAM chip is limited by the time to precharge, or to raise the voltage in the buffer to the off-chip voltage.

While there are efforts to design processors with integrated RAM or RAM with processing power [49, 65], there is no indication when or even if these efforts will become a reality, and if they do, whether they will be affordable to implement. This is largely due to the architectural differences between RAM and logic chips such as the number of metal layers and power consumption [73].

In addition, on commodity PCs, RAM size is limited to about 4 or 8 gigabytes. Working on larger sets of data requires either: using disk, which does not permit quick random access; working with a distributed algorithm that utilizes a high speed network to connect many CPUs and their memory;
or purchasing a non-commodity specially designed system. History shows that architectures that take advantage of commodity chips will win out in the long run. Traditional algorithms do not consider the complex design space presented by the first two alternatives.

The goal of this thesis is to look at the real-world problem of implicit state space enumeration, also referred to as search and enumeration throughout this thesis. This is a problem that has applications in areas such as formal verification, game tree search, computational group theory, and finite geometries. Traditional methods for storing the larger of these search spaces break modern limits of memory and hard disk space on a single commodity machine, and therefore must consider the complex design space described above.

We take as a testbed problem computations over the sporadic simple groups (described in Section 8.1). This is a challenge problem in the area of computational group theory.

The ability to find a strong generating set for these groups allows us to answer questions about group membership, group order, and allows us to find random group elements. The strong generating set and associated data structure provides a compact representation for the group. Computations on symmetry groups have applications in areas such as biology, quantum physics, and chemistry. These groups can be used to solve for molecular vibrations, etc. The algorithm for producing a strong generating set for a group requires the enumeration of a large implicit state space.

We look at four particular groups. The Thompson group application has generating permutations acting on approximately $1.4 \times 10^8$ points, and has approximately $9.0 \times 10^{16}$ group elements. The baby monster group application has generating permutations acting on approximately $1.4 \times 10^{10}$ points, and has approximately $4.0 \times 10^{33}$ group elements. The Fischer group $Fi_{23}$ application has generating permutations acting on approximately $1.2 \times 10^{10}$ points, and has approximately $4.0 \times 10^{18}$ group elements. The Janko group $J_4$ application has generating permutations acting on approximately $1.3 \times 10^{11}$ points, and has approximately $8.6 \times 10^{19}$ group elements. For instances of this size, the enumeration is too large to be computed and stored using traditional methods. A memory efficient yet still tractable algorithm is required.

We provide this algorithm, *tiered duplicate detection*. Our program time is dominated by memory and disk access rather than CPU time. This implies that using the traditional RAM complexity model, where a single memory access or instruction is assumed to take one clock cycle, to show tiered duplicate detection is the best algorithm for these instances simply will not work. The “big oh” time yielded will not consider the high cost of accessing memory and disk. The solution we provide must use cache, RAM and disk efficiently in order to optimize performance, and the analysis
of it must take all of this into consideration. Details on this algorithm can be found in Section 4.

Next, we show a generalization of tiered duplicate detection to implicit state space enumeration. The generalization is straightforward and only requires minor modifications to our existing algorithm. We outline other implicit state space enumeration techniques and define a natural space-time search hierarchy of these techniques where a trade-off exists between the space required by the algorithm and the time spent processing in the CPU (see Section 5 for more details). We show where tiered duplicate detection falls in this hierarchy, indicating under what conditions it is preferred to the other methods.

With all of these techniques defined, we derive formulas to analytically predict the space (memory and disk) and time (CPU and disk access) that each technique will require on a given computer architecture for a specific problem. This leads to a parametrized run-time where the actual run-time is always within ±50% of the predicted run-time, and typically much tighter, as can be seen for the sample problems in computational group theory. With these estimates for each search technique, it becomes easy to find which technique will work best given a particular problem instance and the computing resources available for use.

We also offer an API for large implicit state space enumeration to ease developers who wish to use the outlined techniques for their enumeration applications. The API has pluggable components representing the application, the search technique, and the underlying computer architecture. With this, it is very easy to remove one component and plug another one in its place. For example, switching search techniques through modifying a single line of code.

Finally, in order to show applicability outside of the realm of computational group theory, we show examples of the implementation of search algorithms using our API and software in the puzzle search and formal verification fields. For each of these fields, we argue the efficacy and usefulness of our software by taking sample problems in the field and applying our analysis to them, then using our software to run those examples. We demonstrate experimental times very close to our predicted run-times. In addition, we examine the existing software in these fields to show how our software compares.

This thesis is divided into 8 sections. Those sections are organized as follows:

- Section 2 provides the necessary background in search and enumeration required to understand the remainder of the thesis;
- Section 3 provides the previous work in search and enumeration, focusing in on specific enumeration techniques;
• Section 4 presents our new algorithm for search and enumeration, tiered duplicate detection;
• Section 5 presents the space-time search hierarchy for search and enumeration techniques, the framework under which each of the enumeration techniques will be evaluated;
• Section 6 provides a static analysis of run-time and space requirements for the techniques presented in Section 3 using the framework presented in Section 5;
• Section 7 describes our API for large search and enumeration and its current software implementation;
• Section 8 describes the applications in computational group theory used to show the accuracy of our analysis;
• Section 9 describes the application of our software and analysis to some other problem areas outside of computational group theory.

2 Implicit State Space Enumeration Background

The background required to understand implicit state space enumeration is given here. First, a definition for the problem is given. After this, the two main components required for any application performing an implicit state space enumeration, node discovery and duplicate detection, are explored.

2.1 Enumeration Definitions

The terms implicit state space and implicit state space enumeration are defined here. In addition, some common terms used when describing implicit state space enumeration both in this section and throughout the rest of the paper are presented here as well. Consider a directed multigraph $S$ represented as the set of nodes $V$ and the set of edges $E$.

• A state $v \in V$ is a node the graph.

• The generators $G = \{ V \to V \}$ are a set of functions that produces a neighbor of a state in the graph $S$, forming an edge $e \in E$ between those states.

• An implicit state space $I$ is a representation of the graph $S$ as the pair $I = (v \subset V, G)$ such that $V$ can be obtained by applying the generating functions in $G$ to $v$ recursively until no new states are reached.
• The root or set of roots of the implicit state space \( I = (v, G) \) is the first element in the pair, the set of states \( v \).

• **Implicit state space enumeration** is the process of computing \( V \) from \( I \). This can typically be done using breadth-first search.

• A pair of generators, \( g, g^{-1} \in G \), are said to be **inverses** of one another if \( \forall a, b \in V, g(a) = b \iff g^{-1}(b) = a \). If each \( g \in G \) has an inverse \( g^{-1} \in G \), \( S \) is an undirected graph.

• A **path**, or **word in the generators** is sequence of generators typically representing a path of edges in \( S \) between two states.

• The **number of generators** \( |G| \) is the size of the set of generators \( G \) for the implicit state space.

• The **branching factor** \( ||G|| \) for the graph \( S \) is, for some depth \( d \) early in a breadth-first traversal of the graph, the ratio between the number of states at depth \( d + 1 \) and the number of states at depth \( d \). This number is well-defined for the applications examined in this thesis because the branching factor between levels is a near-constant early on. The branching factor is always in the range \( 1 < ||G|| \leq |G| \).

• A **generator label** \( l \) is a number in the range \( 0 \leq l < |G| \). Each generator \( g \in G \) has a distinct generator label. Each edge in \( E \) has an edge label corresponding to the generator label for the generator used to produce it.

Typically for an implicit state space \( I = (v, G) \) representing some directed multigraph \( S = (V, E) \), both \( v \) and \( G \) are small sets. This leads to \( I \) being a compact representation for \( S \) when \( V \) is large. We examine **large implicit state space enumeration**, or enumerations where the set of states \( V \) does not fit in memory on the available computing resources, typically the distributed memory of a cluster.

It is also important to know the assumed model of the underlying multigraph being enumerated in order to predict information about the enumeration. For the enumeration applications discussed in this thesis, there is some near-constant out-degree, the number of generators, \( |G| \), as well as some near-constant in-degree, also \( |G| \). While some locality may exist in the graph, it is assumed there is little clustering and that the graph is fully connected. This can be expressed with the following:

\[
\forall L \subset V, \# \{ g \in G, l \in L : g(l) \notin L \} \approx ||G|| \times |L| \times \frac{|V| - |L|}{|V|}
\]
2.2 Enumeration Components

There are two main components to any enumeration technique:

- **Node Discovery** — the discovery of states in the state space by applying the generators to known search space states; and

- **Duplicate Detection** — distinguishing between states that have been seen previously in the search space and those that are new.

Node discovery, described in Section 2.3 is typically easy to implement. It can in most cases be done with a simple queue-based breadth-first search. Queue-based breadth-first search has the nice property that all queue accesses are sequential, so the queue can be implemented efficiently on disk.

Duplicate detection, described in Section 2.4, however, is more difficult to perform. Traditionally, a hash table would be used to detect duplicates. However, it is often the case that no perfect hash function exists that will allow a suitable hash table to fit in distributed memory. Most of the search space enumeration techniques described in this thesis seek to solve this problem efficiently.

2.3 Node Discovery

A typical method of node discovery is breadth-first search using an “open queue” to keep track of the frontier in the search. Algorithm 1 shows the straight-forward implementation of queue-based breadth-first search, or breadth-first search using an open queue, given an initial value, \( v \), and a set of generators, \( G \), which are used to find neighbors of a value.

**Algorithm 1: QueueBasedBFS**

<table>
<thead>
<tr>
<th>Input: ( v, G )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: all the elements in the search, ( final )</td>
</tr>
<tr>
<td>( final \leftarrow ) a set with just ( v );</td>
</tr>
<tr>
<td>( open \leftarrow ) a queue with just ( v );</td>
</tr>
<tr>
<td><strong>while</strong> ( open ) <strong>is not empty</strong> <strong>do</strong></td>
</tr>
<tr>
<td>Dequeue ( cur ) from ( open );</td>
</tr>
<tr>
<td><strong>for</strong> ( g \in G ) <strong>do</strong></td>
</tr>
<tr>
<td><strong>if</strong> ( g(cur) \not\in final ) <strong>then</strong></td>
</tr>
<tr>
<td>Enqueue ( g(cur) ) onto ( open );</td>
</tr>
<tr>
<td>Add ( g(cur) ) to ( final );</td>
</tr>
<tr>
<td><strong>return</strong> ( final )</td>
</tr>
</tbody>
</table>
2.3.1 Extension to Disk

One distinct advantage of Algorithm 1 is that the open queue, open, is accessed in a streaming manner only. Where duplicate detection can be easily performed in memory, such as in most cases where a perfect hash is available, Algorithm 1 can be extended to use disk with no random access and only one read and one write pass through the data on the disk.

2.3.2 Parallelization

Algorithm 1 can also be parallelized simply. Rather than adding a state to the local open queue when it is discovered, that state is passed to the node in the cluster that owns it and stored in the open queue there. Ownership of a state can be determined by looking at the upper bits of the hash index for that state, or the upper bits of the state itself if no hash function exists. In addition, this method also reduces the space required to perform duplicate detection, as a node is only responsible for detecting duplicates amongst the states it owns, and can ignore those it does not.

While this parallelization method does actually create a race condition leading to a non-minimal spanning tree, a synchronization phase can be added after each level’s children in the breadth-first search are generated to retain the breadth-first property if it is desired.

2.4 Duplicate Detection

Here we look at some common techniques for duplicate detection during search. Three different approaches are examined, state-based checking, use of a perfect hash function, and use of lossy hash techniques. In addition, we consider both immediate duplicate removal and delayed duplicate removal.

2.4.1 State-Based Duplicate Checking

One approach to removing duplicates is to store all states seen during the enumeration in some table and simply check that table when a new state is seen to determine if it is a duplicate. As stated previously, the state spaces examined in this thesis are typically too large for distributed memory. For this reason, a sort and scan technique on disk can be used to eliminate duplicates.

Disk-based sorting, also known as external sort, is a well-known method originally developed in the database community for sorting data that is stored on disk [5]. It assumes that access to disk is significantly slower than that to memory and seeks to minimize disk access in addition to
only allowing for streaming access to disk. External sort is used in many of the search techniques examined in this thesis, so it is introduced here.

External sort works by partitioning the data into blocks that will easily fit in memory. These blocks are then loaded into memory, one by one, sorted, and written back out to disk. After this, all of the blocks are scanned simultaneously to obtain a complete ordering of the data. External sort requires a non-constant number of passes to sort the data. Where \( N \) is the size of the data on disk, and \( B \) is the size of the memory buffer available to sort with, external sort makes the following number of passes:

\[
P = 1 + \lceil \log_B(\lceil \frac{N}{B} \rceil) \rceil
\]

Each pass requires scanning the data twice on disk. This leads to an overall access time of:

\[
2 \times N \times P
\]

Because of the typical size of memory on modern machines (an upper bound for the \( B \) variable) compared to the typical size of disk (an upper bound for the \( N \) variable), it is very rare to see a case where \( P \neq 2 \). Therefore, commonly, when external sort is examined, four passes through disk are assumed.

### 2.4.2 Using a Perfect Hash Function

A **hash function** is a function that generates an integer value when given a state. A **perfect hash function** guarantees that each state in the state space generates a unique integer \( h \) in some range \( 0 \leq i \leq |H| \), where \( |H| \) is the hash size. Typically, the space required to store \( i, \log(|H|) \), is much less than the space required to store the state itself. Where this is the case, duplicate detection can be performed using a hash table and storing just 1 bit per entry. The hash index of a state corresponds to its location in the hash table. The bit is set to 1 when a state has been seen previously and is 0 otherwise.

Many cases exist where no perfect hash function exists. In these cases, a lossy hash can often be used. If this is done, one must typically guarantee that there is a suitably low probability of two states hashing to the same value in the enumeration using the lossy hash function.

### 2.4.3 Using a Lossy Hash Technique

A **lossy hash technique** is a method of duplicate detection that does not guarantee that false collisions will not occur. Many techniques for lossy hashing exist. The simplest of these techniques uses a
hash function that does not guarantee the uniqueness of the hash indices produced, or an \textit{imperfect hash function}. Other examples of this include a probabilistic technique known as Bloom filters [6] and another technique known as hash compaction [75]. Many existing techniques can provide a probabilistic guarantee of perfect collision detection even if no perfect hash function with a reasonably small $|H|$ is available.

For many lossy hash techniques, a hash function that sufficiently randomizes the structure of the underlying data when creating the hash index for a state is desired. The applications in this thesis initially chose a naive hash function, one that simply used ‘exclusive or’ to combine the bytes of the state representation. However, due to issues with the hash function not being sufficiently random in many applications, a switch to Jenkins’ hash function [36, 37] was made. This hash function provides good randomization of the underlying structure in the data in addition to a fast time for hash index generation and the ability to seed the hash function randomly.

The purpose of this thesis, however, is not to focus on duplicate detection methods, but rather implicit state space enumeration as a whole. Where these perfect hash function alternatives are applicable to particular enumeration techniques, that fact is mentioned but no additional analysis is provided.

\subsection*{2.4.4 Immediate versus Delayed Duplicate Detection}

\textit{Immediate duplicate detection} is the process of detecting a state as a duplicate before applying any additional generators. While immediate duplicate detection can be beneficial it is not required for implicit state space enumeration. Many enumeration techniques take advantage of this by performing what is known as \textit{delayed duplicate detection}.

\textit{Delayed duplicate detection} is the process of not immediately detecting a duplicate, but guaranteeing that if a state is a duplicate, it will be removed before generators are applied to it. For enumeration, this is sufficient to guarantee that no additional processing is performed in the discovery.

Delayed duplicate detection requires a close connection between the node discovery and duplicate detection components for those techniques that use it. However, it also allows for duplicate detection to be performed on many states at the same time. This sort of “batch processing” can lead to performance gains for those techniques that take advantage of it. These performance gains stem primarily from the ability to stream the states to be processed rather than accessing them randomly.
Table 1: Search and Enumeration Techniques

<table>
<thead>
<tr>
<th>Method</th>
<th>Goal</th>
<th>Duplicate Detection</th>
<th>Restrictions</th>
<th>Refs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS w/ a Perfect Hash</td>
<td>Standard method for search</td>
<td>Immediate</td>
<td>Only works when a memory-resident perfect hash table is available</td>
<td></td>
</tr>
<tr>
<td>Sorting-based DDD</td>
<td>Works for any graph</td>
<td>Delayed</td>
<td>Entire space fits on disk</td>
<td>[5, 40, 64]</td>
</tr>
<tr>
<td>Hash-based DDD</td>
<td>Avoid sorting in DDD</td>
<td>Delayed</td>
<td>Needs perfect hash function</td>
<td>[40, 41]</td>
</tr>
<tr>
<td>Structured DD</td>
<td>Check new states against small subset of previous states in-core</td>
<td>Immediate</td>
<td>Needs definition of subspaces based on graph locality</td>
<td>[78, 79]</td>
</tr>
<tr>
<td>Level mod 3</td>
<td>Compact representation of whole search space by storing each state as a 2-bit value</td>
<td>Immediate</td>
<td>Needs a perfect hash function and all inverse generators as generators</td>
<td>[13]</td>
</tr>
<tr>
<td>Implicit Open List</td>
<td>Compact representation of whole search space by storing each state as a 2-bit value and no open list</td>
<td>Delayed</td>
<td>Needs a perfect invertible hash function</td>
<td>[44]</td>
</tr>
<tr>
<td>Landmarks</td>
<td>Reduce space to fit in memory by saving a fraction of the states in full and the rest compressed as words in the generators</td>
<td>Immediate</td>
<td>Works best with large states and fewer generators</td>
<td>[16, 22, 70, 71]</td>
</tr>
<tr>
<td>Frontier Search</td>
<td>Only check for duplicates at current level</td>
<td>Delayed</td>
<td>Needs all inverse generators as generators</td>
<td>[40, 42]</td>
</tr>
<tr>
<td>Tiered DD</td>
<td>Reduce the number of duplicate detection passes</td>
<td>Immediate and Delayed</td>
<td>Needs (imperfect) hash function in memory</td>
<td>[62, 63]</td>
</tr>
</tbody>
</table>

3 Previous Work: Enumeration Techniques

This section examines the existing techniques for large implicit state space enumeration and looks at the history of work in this area. First, each of the existing techniques analyzed in this thesis are presented. After this, in Section 3.10, a history of previous work is presented. This history shows how these techniques were developed over time.

Table 1 lists each enumeration technique along with certain properties of those techniques, including: whether duplicate detection is done immediately or delayed; restrictions the method places on the set of appropriate problems; and a short description of the technique. These techniques are typically applicable to state spaces with billions to trillions of states, requiring terabytes of storage.
3.1 Breadth-First Search with a Perfect Hash

This technique is the technique described in Section 2.3. It is assumed to utilize a memory-resident perfect hash. This can be implemented using either a perfect hash function, a probabilistic technique, such as one described in Section 2.4.3, or even a hash storing full values if the memory is available.

It is assumed this technique uses a disk-based open queue during the breadth-first search, causing it to make one read and one write pass of each new state discovered during the search.

3.2 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 [64] demonstrated the efficiency of this technique for in-core methods where the search overflows into virtual memory. Korf [40] used an explicitly disk-based version to solve sliding tile puzzle and Towers of Hanoi type problems. Stern and Dill [68] offered a version of this algorithm that maintained an in-core open list. In addition, a variant developed by Yu, Manolios and Lamport [77] used an in-core cache to eliminate some duplicates immediately as well as sorting blocks in memory before writing them to disk, eliminating a read and a write pass for each state.

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 the breadth-first search is completed, the buffer is sorted using standard external sorting methods. 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, a perfect hash function, or graph locality.

3.3 Hash-based DDD

Korf [40, 41] 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. In addition, a similar approach was taken by Bao and Jones [4] in order to reduce the number of states scanned on disk.
The method uses two hash functions, often implemented 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 states will occur within the same file. Then, instead of sorting these buffered states, the portion of the hash corresponding to those states are loaded, they are hashed in, and duplicates are detected in this manner. Once this is done, that portion of the hash is written back to disk, and the search continues with the non-duplicate states.

To avoid repeated scans of the full hash at early depths, typically a standard distributed breadth-first search is performed until that search would no longer fit in memory.

### 3.4 Structured Duplicate Detection

Zhou and Hanson [78] 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 $3 \times 3$ 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 [79] give an automatic method for determining a suitable partitioning, assuming the graph has sufficient locality.

Because this technique’s performance is dependent on the locality of the underlying graph, it will not be considered in the algorithm analysis in the remainder of this paper. More generally applicable techniques are desired here.
3.5 Level Modulo 3

This method was introduced by Cooperman and Finkelstein as a compact representation for Cayley graphs, along with a generalization to Schreier coset graphs [13]. It is similar in ways to breadth-first search with a perfect hash, but uses an even more compact representation for states not in the open list. It requires inverse generators to be used in the enumeration.

The method uses a perfect hash function to associate every state with a single two-bit value. While performing a breadth-first search, rather than remembering states once they have been popped from the open list, this technique uses those two-bit hash entries to distinguish between hash locations corresponding to:

- States whose depths are 0 modulo three
- States whose depths are 1 modulo three
- States whose depths are 2 modulo three
- States that have not been seen

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

While the hash table cannot be used to iterate over the states in the space, it can be used to answer questions about the membership of states, along with acting as a compact representation for the entire graph. Once the breadth-first search has been completed, a shortest path from any given state to the root state can quickly be found in the following way. First, all of the generators are applied to the given state. The first generator that leads one level earlier (modulo three) 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.

3.6 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 [44].

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, two frontier bits are associated with each state. These
two bits will track which states are in the open list. Because the hash function is invertible, the
open list can be reconstructed by scanning the hash table. Two bits are used instead of one because
delayed duplicate detection is used, so the hash must keep track of those states that are

- at the depth currently being processed;
- at the depth currently being generated;
- at a depth before these, or already processed; or
- at a depth later than these, or currently unseen

The algorithm proceeds by scanning through the hash. When a state is seen that must be
processed, its neighbors (represented as hash table locations) are generated and written in bins to
disk as with hash-based delayed duplicate detection. Whenever the bins fill up the available disk,
their contents are merged into the hash table as states that have been generated. Once the scan
through the hash has completed, the processed states become previously seen states, the generated
states become states to process, and another scan is performed. This is done until no additional
states remain to be processed.

To avoid repeated scans of the full hash at early depths, typically a standard distributed breadth-
first search is performed until that search would no longer fit in memory.

The final representation for the search, the hash table, can then be used to produce any state
seen during the search through inversing hashing function. Other information, such as the depth of
a particular state in the search or its depth modulo three, can be stored at the hash entry, producing
a similar end result as that of the level modulo three technique.

3.7 Landmarks

Instead of storing all states in the graph being generated, one can instead store only a small fraction
of states, known as the landmarks, in a technique known as landmark enumeration [16, 22]. 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.
Typically, any landmark ratio can be achieved through the use of the modulo operator.
States are stored in bins according to the landmark that owns them. Ownership is determined though a small breadth-first search in the generators from the state in question to the first landmark seen, where the generators are applied in some predetermined order. The states are stored as the path in that breadth-first search to reach the landmark (in many cases, a more compact representation), with landmarks themselves represented as an empty path.

When a new state is generated, the nearest landmark and the path to that landmark are determined. If there is not yet a bin for that landmark, one is added. The path is then compared to those paths already in the bin for that landmark. If the path is not already present, it is added and the state is a non-duplicate. Otherwise, the path exists already and the state is a duplicate. Duplicate states are ignored, non-duplicates are added to the open list.

This method is especially effective when the state representation is very large, and when there are few generators. In this case, the storage for the path 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 finding the landmark that owns a state. Therefore, this technique is also most applicable in cases where generator application is relatively fast.

3.8 Frontier Search

Korf [40, 42] 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. Others have developed similar techniques to take advantage of the locality of the breadth-first search graph as well even when inverses are not present [59].

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 duplicates only need to be checked for in those three levels, and all other levels can be safely ignored.

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.

3.9 Tiered Duplicate Detection

Robinson and Cooperman [62] introduced tiered duplicate detection as a method to speed up the enumeration of the Baby Monster sporadic simple group, and more recently applied it to the problem of the Fischer group $Fi_{23}$ [63].

Instead of automatically storing all generated states for delayed duplicate detection, an in-core imperfect or lossy hash function is used as a first pass. If the new state hashes to an empty hash location, it is not a duplicate and can be immediately placed in the open queue for further generation. If, on the other hand, the hash results in a collision, the state may be a duplicate. Because the hash function is imperfect, non-duplicate states may collide. Hash collisions are stored for delayed duplicate detection. This can be done using many approaches, but typically sorting-based delayed duplicate detection is used.

To obtain maximum performance, the tiered duplicate detection performs strictly memory-resident duplicate detection until the open queue is empty. After this, delayed duplicate detection is performed. Any new state seen are added to a new open queue. The process is then repeated until no new state are seen during the delayed duplicate detection.

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 explore multiple levels of the search before accessing disk. With this, fewer passes through disk are performed for duplicate detection. However, shortest paths to the states in the search are not guaranteed. This method is useful for finding the states of a search space producing reasonably short paths to those states, but not for proving the depth of the actual search tree.

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. A more complete description of tiered duplicate detection and when it is useful is presented in Section 4.
3.10 History of Previous Work

Alternative methods of duplicate detection to hash tables and simple state comparison started as early as 1970, when Bloom proposed his Bloom filters [6]. This work has since been elaborated on by many individuals [58, 61]. In addition, other techniques such as hash compaction [25, 67], counting filters [8], Bloomier filters [11], and compact approximators [7] have also been examined.

Regarding implicit state space enumeration itself, one of the early pieces of work in detecting duplicates for state spaces too large to fit in regular memory was carried out in 1994 in the area of model checking by Roscoe [64]. This result was essentially a first implementation of *sort-based delayed duplicate detection*.

Variants of Roscoe’s algorithm were also proposed within the model checking community. Stern and Dill [68] offered a version of the algorithm that maintained an in-core open list. Yu, Manolios, and Lamport [77] offered a version that used memory as a cache to eliminate some duplicates immediately as well as avoid the block sorting phase of external sort. The impact of these results outside of the model checking community was not realized at the time, and it was not until much later that this technique became popular in other fields.

In the meantime, memory-resident approaches were explored for duplicate detection and search. First, in 1992, Cooperman and Finkelstein proposed the *level modulo three* technique to allow for a compact representation of Cayley graphs [13] in computational algebra. Then later in 1996 they proposed a *landmarks* approach for memory-resident duplicate detection in spaces too large to fit in main memory [16, 22]. This became a popular approach in the computational group theory field, and continued to be popular up until very recently, when disk-based algorithms began to be explored.

Finally, in 2003, disk-based techniques for large implicit state space enumeration started becoming popular. This was largely due to new techniques developed for puzzle search. First, in 2003, Korf reintroduced the algorithm initially proposed by Roscoe [38, 39]. He continued to work on the problem of search. A year later he discovered *hash-based delayed duplicate detection* [40], which eliminated the need for the sorting pass of sort-based delayed duplicate detection. The same year Korf also discovered *frontier search* [42], which allowed for a constant number of passes through the states during the delayed duplicate detection. He used this to solve the 4-peg towers of Hanoi problem. Also, in 2004, Zhou and Hansen demonstrated an implementation of *structured duplicate detection* on the sliding tile 15 puzzle. This technique was able to minimize access to disk in graphs with high locality. Finally, in 2005, Bao and Jones [4] proposed a variant of hash-based delayed duplicate detection that sought to reduce the number of states scanned on disk.
The next major advance came in 2006 in the area of computational group theory. Robinson and Cooperman proposed the idea of tiered duplicate detection [62]. This technique used two levels of duplicate detection, much like [77]. An immediate imperfect memory-resident technique which allowed for bad collisions was used, along with a delayed disk-based approach able to correct errors when the memory-resident technique failed. They went on to use this in problems in computational group theory related to forming permutation representations and performing direct condensation [62, 63]. In addition, in 2007, Kunkle and Cooperman proposed the idea of implicit open list search and used it to prove that 26 moves suffice to solve for Rubik’s cube [44].

4 Tiered Duplicate Detection:

A New Hybrid Enumeration Technique

This section presents an approach to implicit state space enumeration that is novel to this thesis. Tiered duplicate detection seeks to reduce the total number of passes through disk during the delayed duplicate detection phase. In order to do this, it performs an in-core imperfect duplicate detection using an imperfect or lossy hash function. This identifies some new states immediately and children of those states can be computed without accessing disk. Delayed duplicate detection must be used on the remaining states. By identifying some new states immediately, tiered duplicate detection can process multiple depths in the search tree before stopping to scan disk.

Inherently, tiered duplicate detection fills in a gap in the space-time search hierarchy described in Section 5. It provides a hybrid technique between sorting- and hash-based delayed duplicate detection and breadth-first search with a perfect hash. The technique is able to make fewer passes through disk for duplicate detection than sorting- and hash-based delayed duplicate detection, but doesn’t require a suitably small perfect hash function as breadth-first search with a perfect hash does.

4.1 The Tiered Duplicate Detection Algorithm

The tiered duplicate detection algorithm uses several variables. Those variables are defined here. How they are used is explained with the individual functions that access them.

- \textit{hash} – This is the memory-resident imperfect or lossy hash table (as defined in Section 2.4.3). Any lossy duplicate detection technique can be used in its place, such as Bloom filters or hash compaction.
• *final* – This is the disk-based sorted set of states seen so far in the search not including those seen during the current pass. When the computation finishes, this structure contains all the states in the state space.

• *collision* – This is a disk-based set of all the hash collisions that have occurred during the current pass’ breadth-first search.

• *new*final – This is a disk-based set of all new states that have been seen during the current pass’ breadth-first search.

• *open* – This is the disk-based open queue for the current pass’ breadth-first search.

It is assumed that the search tree up to some small depth is computed in memory on a single processor. Once this is done, the states seen fall into two categories. Either they have had the generators applied to them, and they are *explored* states, or they have not, and they are *unexplored* states. The *explored* and *unexplored* states can be distributed to the processors that own them. We used the notation \( x = \text{owner}(v) \), where \( v \) is a state and \( x \) is processor id, to indicate that the processor with id \( x \) owns state \( v \). Ownership is typically determined using some hash function over the bits of the state representation. Once the states are distributed, Algorithm 2 shows the top level function used to perform tiered duplicate detection. This section also defines other utility functions. This algorithm is run on each processor.

**Algorithm 2: TieredDuplicateDetection**

<table>
<thead>
<tr>
<th>Input: explored, unexplored</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: final, the distributed sorted disk-based set of all states in the state space</td>
</tr>
<tr>
<td>// The following three variables are globally visible</td>
</tr>
<tr>
<td>// All functions have access to them</td>
</tr>
<tr>
<td>Global hash ← a lossy hash using the available memory;</td>
</tr>
<tr>
<td>Global newfinal ← an empty disk-based set;</td>
</tr>
<tr>
<td>Global collision ← an empty disk-based set;</td>
</tr>
<tr>
<td>(open, final) ← TDDInitialize(explored, unexplored);</td>
</tr>
<tr>
<td>while open is not empty across all processors do</td>
</tr>
<tr>
<td>TDDSearch(open);</td>
</tr>
<tr>
<td>Wait for all processors to finish;</td>
</tr>
<tr>
<td>(open, final) ← TDDDelayedCheck(final);</td>
</tr>
<tr>
<td>Wait for all processors to finish;</td>
</tr>
<tr>
<td>return final</td>
</tr>
</tbody>
</table>

Algorithm 2 alternates between performing a breadth-first search using a memory-resident lossy hash and performing delayed duplicate detection on disk. This is done until no new states are seen
during the delayed duplicate detection phase on any processor, indicating that there are no new states to explore. The first step in Algorithm 2 is to initialize the data structures based on the states seen in \textit{explored} and \textit{unexplored}. This includes initializing the \textit{open} queue for the first round of the breadth-first search. The initialization is shown in Algorithm 3.

\begin{algorithm}
\caption{\textit{TDDInitialize}}
\begin{algorithmic}
\State \textbf{Input}: \textit{explored}, \textit{unexplored}
\State \textbf{Output}: \textit{open}, \textit{final}
\State $final \leftarrow$ an empty sorted disk-based set;
\State $open \leftarrow$ an empty disk-based queue;
\State Sort the states in \textit{explored};
\State Sort the states in \textit{unexplored};
\State $final \leftarrow$ the merge of \textit{unexplored} and \textit{explored};
\For{$v \in final$} Set the location in hash corresponding to $v$;
\EndFor
\For{$v \in \textit{unexplored}$} Enqueue $v$ onto \textit{open};
\EndFor
\State \textbf{return} $(\textit{open}, \textit{final})$
\end{algorithmic}
\end{algorithm}

Algorithm 3 first merges the unexplored states and explored states into a sorted set including all the states seen so far. The \textit{final} set is initialized to be this sorted set. In addition, all the states in the sorted set are entered into the hash as well. Finally, the unexplored states are added to the \textit{open} queue. Once this is done, the queue has been initialized, and the search portion of tiered duplicate detection can begin. Where $G$ is the set of generators, Algorithm 4 shows how to perform the search.

\begin{algorithm}
\caption{\textit{TDDSearch}}
\begin{algorithmic}
\State \textbf{Input}: \textit{open}
\While{$\textit{open}$ is not empty}
\State Dequeue \textit{parent} from \textit{open};
\For{$g \in G$}
\State \textit{child} $\leftarrow g(\textit{parent})$;
\State \textit{check} $\leftarrow$ the result of calling \textit{TDDImmediateCheck}($\textit{child}$) on processor \textit{owner}($\textit{child}$);\If{$\textit{check}$} Enqueue \textit{child} onto \textit{open};\EndIf
\EndFor
\EndWhile
\State \textbf{return} \textit{final}
\end{algorithmic}
\end{algorithm}

Algorithm 4 is the basic algorithm for distributed breadth-first search using a disk-based open queue. The results of the search are assumed to be stored for each state on the remote machine that owns that state. The call to \textit{TDDImmediateCheck} is used both to determine if the state is new, as well as to store it remotely. Algorithm 5 shows how this is done.

Algorithm 5 uses a distributed lossy hash table to determine if a state is new. All results of the hash calls are stored in either \textit{newfinal} or \textit{collision}. When Algorithm 4 finishes, each new state seen that did not initially collide with another state will be stored in \textit{newfinal} and its children will
Algorithm 5: $TDDImmediateCheck$

Input: $v$
Output: $true$ if it can be determined that $v$ has not been seen previously, $false$ otherwise

if $v$ hashes to an empty location in $hash$ then
    Set the location in $hash$ corresponding to $v$;
    Append $v$ to $newfinal$;
    return $true$;
else
    Append $v$ to $collision$;
    return $false$;

have been computed. $Collision$ will contain primarily duplicates. However, some states in $collision$ will be new states to which the generators have not been applied that resulted from an invalid hash collision. In order to remedy this, Algorithm 6 is used.

Algorithm 6: $TDDDelayedCheck$

Input: $final$
Output: $open, final$

$(final, collision) \leftarrow DDDSort(newfinal, final, collision)$;
$(open, final) \leftarrow DDDScan(final, collision)$;
$newfinal \leftarrow$ an empty set;
$collision \leftarrow$ an empty set;
return $(open, final)$;

Algorithm 6 executes in two phases. The sorting phase, $DDDSort$, produces new sorted version of $final$ and $collision$. The scanning phase, $DDDScan$, produces a new version of $final$ and a new $open$ queue to process. Once these are done, $newfinal$ and $collision$ are emptied so the $TDDSearch$ can be started again. Algorithm 7 shows the sorting phase of this algorithm, and Algorithm 8 shows its scanning phase. These algorithms assumes access to an external sort function, $sortedData \leftarrow ExternalSort(data)$, as well as an external merge, $sortedData \leftarrow ExternalMerge(sortedData1, sortedData2)$. Information on external sort can be found in Section 2.4.1. External merge is one component of external sort.

Algorithm 7: $DDDSort$

Input: $newfinal, final, collision$
Output: $final, collision$

$newfinal \leftarrow ExternalSort(newfinal)$;
$final \leftarrow ExternalMerge(newfinal, final)$;
$collision \leftarrow ExternalSort(collision)$;
return $(final, collision)$;
Algorithm 7 sorts the collision and newfinal states. After this is done, it merges the newfinal states into final. This allows Algorithm 8 to use the collision and final sets as if they are disk-based queues and retrieve the entries in sorted order. With this, false collisions can be found.

**Algorithm 8: DDDScan**

<table>
<thead>
<tr>
<th>Input:</th>
<th>final, collision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>open, final</td>
</tr>
</tbody>
</table>

```plaintext
curFinal ← NULL;
prevCol ← NULL;
open ← an empty queue;
newfinal ← a disk-based empty set;
while collision is not empty do
    Dequeue curCol from collision;
    // Get the next final entry that is ≥ curCol
    while final is not empty and (curFinal = NULL or curFinal < curCol) do
        Dequeue curFinal from final;
        if prevCol = NULL or prevCol ≠ curCol then
            if curFinal = NULL or curFinal ≠ curCol then
                // The state is a false collision that has not been seen before
                Enqueue curFinal in open;
                Append curFinal to newfinal;
                prevCol = curCol;
            final ← ExternalMerge(newfinal, final);
    return (open, final)
```

Algorithm 8 scans through the collision and final states to find false collisions. It is assumed dequeue is nondestructive in this case and simply advance a file pointer. This algorithm looks for states in collision that have not been seen previously in either collision or final. These states correspond to false hash collisions. When they are seen, the states are added to newfinal and enqueued in a new open list. After all of the false hash collisions have been found, newfinal and final are merged. Since the states in collision were previously sorted and those states were scanned sequentially, newfinal is already in sorted order for this merge.

### 4.2 Optimizations

There are many optimizations that can be made to the tiered duplicate detection algorithm.

First, one typically does not process a single state at a time during the breadth-first search described in Algorithm 4. Rather, a block of states are loaded from the front of the open queue. This allows for:

- streaming access to disk from the open queue;
• batch requests to generators; and
• streaming network access by calling `TDDImmediateCheck` on multiple states at a time.

In addition to this, a double-buffering technique can be used to allow for generators to be applied to one set of states while another set of states is concurrently written to disk. One can do still better and use the pipeline paradigm from parallel computing. Here, several buffers of states are kept. Buffers are typically allocated to perform the following concurrently:
• reading states from the front of the `open` queue;
• applying generators to states;
• sending states to be checked remotely; and
• writing states to the end of the `open` queue.

The delayed duplicate detection described in Algorithm 6 assumes the `final` data structure is a single disk block kept in fully sorted order. By allowing that data structure to be kept as multiple individually sorted blocks, one can simplify the merge of `newfinal` into `final` by instead appending the sorted blocks of `newfinal` onto `final`. However, if this is done, a priority queue over the sorted blocks in `final` must be used when it is scanned. A similar strategy can be used for `collision` to avoid the merge component of externally sorting this data.

Finally, Algorithm 6 is essentially using sorting-based delayed duplicate detection. One can use a variant of hash-based delayed duplicate detection. This method stores bins of states that fit into main memory. One can predict a total number of bins required based on the search size and available memory. With this, a hash function over the bits of the state representation can be defined that will determine in which bin a state falls. Using this method, the time for sorting is eliminated entirely, and only scans must be performed. However, this technique also requires that each bin retain a write buffer in memory during the breadth-first search component described in Algorithm 4.

### 4.3 When to Use Tiered Duplicate Detection

Tiered duplicate detection is useful because it allows for multiple depths in the search to be performed before accessing disk. Let $|G|$ be the number of generators, $|S|$ be the number of states in the search space, and $|G||S|$ be the branching factor, or ratio between the number of states at subsequent depths in the search. Intuitively, by storing on the order of $|G| \times |S|$ states at any one time on disk rather than the typical $|S|$, tiered duplicate detection is capable of performing fewer passes through disk.
Specifically, the number of passes through disk tiered duplicate detection must perform is on the order of $\log_{HM}(\log_{||G||}(|S|))$ rather than the typical $\log_{||G||}(|S|)$. A hash multiple $HM$, or ratio between the number of bits in the hash and the size of the search, of between 2 and 25 is typically obtainable in our experience. While the reduction in the number of passes performed by tiered duplicate detection is significant, the savings are not as large as they first seem. This is due to the fact that many of the passes through disk avoided are passes early in the search, when few states have been seen.

Because the storage required by tiered duplicate detection scales with the total number of generators $|G|$, this technique is typically preferred when $|G|$ is reasonably small. If $|G| \times |S|$ grows too large, an optimization does exist that allows for early termination of the breadth-first search when the disk is full. However, this leads to a degradation in overall performance. In addition, because the depth of the search increases as the branching factor $||G||$ decreases, tiered duplicate detection typically outperforms other techniques when $||G||$ is reasonably small as well.

Finally, tiered duplicate detection does not guarantee shortest paths due to the fact that it can scan multiple depths of the search tree at a time and may make “mistakes” as it goes that are corrected later. Due to this, when shortest paths are desired or the exact depth of the search tree must be known a different technique is required.

5 The Space-Time Search Hierarchy:
An Approach to the Analysis of Enumeration Techniques

While previous works have examined all of the 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.

A fair and unbiased method for comparing enumeration techniques is required. In order to find this, a “Big Oh” for enumeration needs to be defined. 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 state accesses (typically streamed).
5.1 The Space-Time Search Hierarchy

Techniques for implicit state space enumeration fall naturally into a space-time search hierarchy. At the top of this hierarchy are enumeration techniques where more computations performed but less space is required. At the bottom are techniques where more space is required but fewer computations are performed. This hierarchy is shown in Figure 1(a) for techniques requiring a perfect hash and in Figure 1(b) for those that do not. Because the methods analyzed in Section 3 naturally fall into this hierarchy, a trade-off exists between the space required to perform a particular enumeration and the number of computations that enumeration performs.

This hierarchy has a direct correlation to the memory hierarchy. Given a particular application instance, the space-time search hierarchy can be overlaid with the memory hierarchy, forming a natural partitioning between those algorithms that can remain in-core, those necessarily out of core, and those infeasible given the resources available. One such random partitioning is shown in Figure 2(a) and Figure 2(b).

Traditionally, practitioners have limited themselves to techniques for which the space fits in-core
Figure 2: Space-Time Search Hierarchy overlaid with the memory hierarchy for both techniques requiring a perfect hash and techniques that do not have this requirement.
Table 2: The Classes and Parameters

<table>
<thead>
<tr>
<th>Architectural parameter</th>
<th>name</th>
<th>Application parameter</th>
<th>name</th>
<th>Algorithmic parameter</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk Bandwidth</td>
<td>$BW^D$</td>
<td>Search Size</td>
<td>$</td>
<td>S</td>
<td>$</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>$N$</td>
<td>Number of Generators</td>
<td>$</td>
<td>G</td>
<td>$</td>
</tr>
<tr>
<td>Memory Size</td>
<td>$</td>
<td>M</td>
<td>$</td>
<td>Branching Factor</td>
<td>$</td>
</tr>
<tr>
<td>Disk Size</td>
<td>$</td>
<td>D</td>
<td>$</td>
<td>State Size</td>
<td>$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Edge Bandwidth</td>
<td>$BW^G$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hash Size</td>
<td>$</td>
<td>H</td>
<td>$</td>
</tr>
</tbody>
</table>

(in RAM). This is not always best. The lower methods of Figure 1(a) and Figure 1(b) take less time if performed in RAM. However, a method operating on disk may either:

- require more time due to slower disk speeds; or
- require less time due to the method being inherently faster.

The latter situation was exploited for many of the enumerations in computational group theory discussed in this paper (see Section 8.3).

5.2 Parameters of Analysis

To compute the predicted times for many of these techniques, three different classes of parameters are considered. Architectural parameters are those specific to the cluster on which the computation is being performed. Application 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 most cases, are derived from the algorithm being used as well as the application and architectural parameters. Table 2 shows a listing of all such parameters of each class. Derived parameters are listed in italics.

While some of these parameters are self-explanatory, others are not. Below each parameter is listed and a definition is given. Many of these parameters can be predicted automatically. Later, in the software section, a discussion as to how this can be done is provided (see Section 7.2 for more details).

Search Size The expected number of states to be enumerated, or discovered, in the search space.
Number of Generators The total number of generators used during the enumeration, typically the max out-degree of a node in the graph.

Branching Factor This is the expected ratio early in a breadth-first search between the number of new states at depth $d + 1$ and the number of new states at depth $d$. This ratio is assumed to be a near constant for relatively small values of $d$.

State Size The size of an individual state 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.

Hash Size The number of entries required for a perfect hash table. In the worst case, the hash value for a state is the state itself.

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.

Random Storage The random-access storage required by the algorithm. Typically this is memory.

Streaming Storage The streaming-access storage required by the algorithm. Typically this is disk.

Given these parameters, it is easy to determine if a particular computation is feasible. To do this, the memory and disk requirements must be met.

$$R^R < N \times |M|$$

Memory Requirement, the required random access storage is less than the distributed memory available

$$R^S < N \times |D|$$

Disk Requirement, the required streaming storage is less than the distributed disk available
For feasible algorithms, the total time for the algorithm can be computed:

\[
Time = \frac{X^C}{BW^C \times N} + \frac{X^A}{BW^A \times N}
\]

Time requirement, the time spent applying generators and the time spent accessing disk

It is interesting to note that this time does not take into consideration the time spent passing messages on the network. This is because typically the search techniques only send a newly discovered state across the network a single time. However, it is typically written and read from disk multiple times. Since network and disk speeds are comparable, network is not the bottleneck for these techniques. There are cases where the network time requires consideration, but these are primarily cases that do not use disk or cases when the cluster size grows beyond approximately 100 nodes, as the network is a shared resource but disk is not.

Assuming the application and architectural parameters are fixed (a particular problem and cluster are being examined), the feasibility and run-time for the search given each algorithm can be predicted by swapping the algorithmic parameters. From here, determining the best algorithm for a particular enumeration and cluster becomes an easy matter.

While determining the application and architectural parameters is a straightforward matter, finding the algorithmic parameters is not always as simple. As will be seen, these parameters are typically derived from the architectural parameters, the algorithmic parameters, and the search technique in question, rather than a value based on a fixed equation. In this section each enumeration technique is examined and the value of each parameter for each technique is given. Many of these techniques have tunable components as well, meaning the more memory that is available, the faster they will perform. This is specified by introducing a new “tunable” variable for these techniques.

### 5.3 Sampling with Replacement

Theorem 5.1, which has applications in statistical analysis, polling theory, and other areas, will also be useful for many enumeration techniques.

**Theorem 5.1.** If you sample \( n \) objects randomly with replacement \( m \) times:

- The chance that an object will be seen at least once is \( 1 - (1 - \frac{1}{n})^m \).
- The average number of unique objects seen is \( n \times (1 - (1 - \frac{1}{n})^m) \).

This theorem will be used to determine the size of each level in the breadth-first search. In addition, it can be used to determine how many computations must be performed when selecting
random computations from a set of possible computations of a fixed size. Rather than writing out
the full formula each time it is used, \( \text{sample}(n, m) \) will be used to denote the number of new objects
seen when sampling \( n \) objects with replacement \( m \) times.

5.4 Level Sizes in Breadth-First Search

In analyzing breadth-first discovery, it is important to know how many states will be seen at each
level in the search. This can be defined recursively. Given a level \( i \) and the number of states seen
at the level \( i - 1 \), \( T_{i-1} \), find the number of states that will be seen at level \( i \), \( T_i \). This value can be
predicted by looking at both the branching factor, \( \|G\| \), and the estimated size of the search, \( |S| \).

For the types of graphs described in Section 2.1, the number of states at each subsequent level
in the breadth-first search can be computed. This is done by examining three factors, the number
of edges seen at the current level, the percentage of remaining edges that go to new states, and the
number new states remaining.

The number of edges seen at the current level in the breadth-first search, \( E_i \), is simply the number
of new states seen at the previous level times the branching factor of the problem, or:

\[
E_i = T_{i-1} \times BF
\]

The edge count, the edges going out of states at the previous depth

The percentage of remaining edges that go to new states corresponds to the number of remaining
edges to new states, divided by the number of remaining edges. This can be computed to be the
following:

\[
F_i = \frac{(|S| - \sum_{s=1}^{i-1}(T_s)) \times BF}{(|S| - \sum_{s=2}^{i-1}(T_s)) \times BF}
\]

The percent of edges going to new states, the number of remaining edges to unseen states over the
number of remaining edges

Finally, the number of states remaining is simply the total number of states to be seen minus
the number of states that have been seen previous, or:

\[
|S_r^i| = |S| - \sum_{s=1}^{i-1}(T_s)
\]

The number of states remaining, the number of states in the search minus what’s been seen so far

The number of edges seen is multiplied with the percentage of edges to new states to obtain
the number of new states seen. However, this does not yet deal with edges that lead to the same
unseen state. For this, we use Theorem 5.1. We know how many unseen states remain, these are the objects, and we also know how many edges will go to these states, this is the sampling size. Given this, we apply Theorem 5.1 to get the average number of the remaining states with at least one edge mapping to them:

\[ T_i = \text{sample}(|S'_i|, E_i \times F_i) \]

The number new states seen, sample the remaining states according to the number of edges leading to previously unseen states

The maximum depth of any state in the search, \(|T|\), can be determined by finding the last \(i\) such that \(T_i > 0\).

### 5.5 Use of Compressed States

*Compressed states* are representations for the states seen in the search which are large enough to be unique from one another with a very high probability [62]. Frequently this representation is significantly smaller than than the actual state size, and can be used to reduce the space required by the enumeration.

The use of compressed states moves a technique to a higher location in the space-time search hierarchy. While less storage is used, generators cannot be applied directly to compressed states. In order to apply generators, the compressed state must be expanded into its full form. This is commonly done by storing a path to the state from the root along with the compressed state.

Compressed states change the fundamental storage required for the search space. Each state, rather than taking up \(|V|\) bytes, requires the storage of a compressed form with \(|V_C|\) bytes. This typically has two components, \(|V_C^S|\), and \(|V_C^P|\), where \(|V_C^S|\) is the space required for the compressed form of the state, and \(|V_C^P|\) is space required for a path in the generators from the root to the state, used to obtain the full state when required. These sizes, while they can be expressed in bits (or fractions of a byte), are typically rounded up to the nearest byte. They are shown below.

\[
|V_C^S| = \frac{2 \times \log(|S|)}{8}
\]

The size of the compressed state, space to store a number ranging from 1 to \(|S|^2\)

\[
|V_C^P| = \frac{\log(|G|) \times |T|}{8}
\]

The size of a path, space to store an identifier for each node in the potential path

\[
|V_C| = |V_C^S| + |V_C^P|
\]

The total size, the sum of the two components
Compressed states can be used for all states seen in the search, or only a subset. Typically there is a trade-off between storage and computation required based on this selection. Examples include using compressed states for all states, using compressed states for states not in the frontier, and using compressed states everywhere but in the open list. For each technique described in this section, the most appropriate use of compressed states is considered and analyzed. They are selected in such a way to have as small of a computational overhead as possible while giving as large of a storage reduction as possible.

5.6 Use of Inverses

The inclusion of inverses among the generator moves a technique to a higher location in the space-time search hierarchy. Due to the fact that $|G|$ is typically larger with their inclusion, more generator applications must be performed in the search space discovery. However, a smaller effective space can be considered during the duplicate detection phase. Through the use of inverses, one can guarantee that for a state at some depth in a breadth-first search, duplicates of that state only appear at the same depth as the state that generated it, at the next depth, or at the previous depth.

One can do still better [43]. If all generators mapping to a state from states at the current depth are known, collisions can be avoided that are from the previous depth when that state’s neighbors are generated. This implies that only two levels of depth need to be examined when looking for duplicates rather than three.

While the use of inverses does not reduce the actual size of the resulting state space, it effectively reduces the amount of data examined during the enumeration. This stems from the fact that more generators are applied, producing a larger amount of temporary data, but a smaller amount of space is examined to detect duplicates for the resulting states.

6 Static Analysis of Enumeration Techniques

In this section the techniques presented in Section 3 are examined and parametrized formulas to derive the four variables, $X^A$, $X^C$, $R^R$, and $R^S$ are presented for each. The result are formulas parametrized by the application and architectural parameters. When presented with a specific problem and cluster, the parameters can be plugged in to the equations to obtain the algorithmic parameters, which can then be used to predict the time and feasibility of each technique.

For all of these techniques, some run-time and storage requirement near-constants are ignored.
These include but are not limited to costs associated with

- Storage required for batching network and disk requests,
- Time for memory-resident hash table look-up, and
- Time for moving states around in memory.

For the sake of simplicity, it is assumed that the additional space requirements are trivial in comparison to the space required to explore and store the search space. It is also assumed that the additional time requirements are trivial in comparison to the time required for applying generators while exploring the search space.

### 6.1 Breadth-First search with a Perfect Hash

Assuming a memory-resident perfect hash can be obtained, this method is typically optimal in that it provides a quick representation of the full search space, reading and writing each value only once, and using storage only for the values in the search. The memory and disk requirements for this method are,

\[
\begin{align*}
R^R &= \frac{|H|}{s} \quad \text{The space required for the hash} \\
R^S &= |S| \times |V| \quad \text{The space required for the states}
\end{align*}
\]

The data access and generator components of this method are,

\[
\begin{align*}
X^A &= 2 \times |S| \times |V| \quad \text{A read and write of each state} \\
X^C &= |G| \times |S| \quad \text{A generator to reach each state including duplicates}
\end{align*}
\]

Computing the time for this technique is useful even if a perfect hash is not available. It provides the person analyzing the problem with a close approximation to an “optimal” running time.

**Compressed States** The most effective use of compressed states is for storage of states not in the open list. Using this method, a significant space reduction may be possible.
The space required for the hash

The space to store the open queue for a level

The space to store all previous levels compressed

The space to store compressed and full states

Using compressed states, the only additional computation that must be performed is a rewrite of the states in the open queue to change them to their compressed form once they have been processed. This corresponds to one additional write of the compressed states disk. No additional generator applications need to be performed during the enumeration. However, to produce a state once the enumeration has finished, the path in the generators to that state from the root must be applied.

A read and write of each state and a write of each compressed state

A generator to reach each state including duplicates

Inverses Inverses are of no additional use with this technique. They do not affect its predicted time.

Alternatives to a Perfect Hash This technique can use any of the alternatives to a perfect hash described in Section 2.4.3. This assumes these techniques are capable of providing duplicate detection that can distinguish all of the states with a high probability.

6.2 Sorting-Based Delayed Duplicate Detection

Sorting-based delayed duplicate detection avoids any additional generator applications, however, it must also scan through disk numerous times in order to find duplicates. The amount of memory used by the method is negligible, however, it must store each state it has discovered along with storing all states at the current depth being discovered, including duplicates.
\[ R^R = 0 \quad \text{A constant amount for buffers} \]
\[ S_i^F = \|G\| \times T_i \quad \text{Size of neighbors of a level} \]
\[ S_i^P = \sum_{s=1}^i (T_s) \quad \text{Number of states up to a level} \]
\[ R^S = |V| \times \max_{i=1}^{\left\lfloor T_i \right\rfloor} (S_i^F + S_i^P) \quad \text{Size of all the states currently stored} \]

The CPU time for sorting-based delayed duplicate detection is easy to predict. It must generate each state seen during the search only once. Therefore it has the exact same CPU time as using the memory resident perfect hash technique. For the disk component, each state must be scanned repeatedly, once for each depth in the search tree beyond its depth, to check for duplicates. In addition, each state must be accessed for the external sorting phase. The access and CPU components of the search are listed below.

\[ A^Q = 2 \times |S| \quad \text{Queue Access, a read and write for each state in the search} \]
\[ A^S = 3 \times \|G\| \times |S| \quad \text{Sort Access, two reads and a write for all states including duplicates} \]
\[ A^D = \sum_{i=1}^{\left\lfloor T_i \right\rfloor} (\sum_{j=1}^{i-1} (T_j)) \quad \text{Detection Access, for each depth, a read of all the previous states} \]
\[ X^A = (A^Q + A^S + A^D) \times |V| \quad \text{The sum of the queuing, sorting, and detection access} \]
\[ X^C = \|G\| \times |S| \quad \text{A generator to reach each state including duplicates} \]

**Compressed States**  The most effective use of compressed states is for states not in the open list. Using this method, a significant space reduction may be possible.

\[ R^R = 0 \quad \text{A constant amount for buffers} \]
\[ R^S = \max_{i=1}^{\left\lfloor T_i \right\rfloor} (S_i^F \times |V| + S_i^P \times |V_C|) \quad \text{Size of all the states currently stored} \]

Using compressed states, the only additional computation that must be performed is a rewrite of the states in the open queue to change them to their compressed form once they have been processed. This corresponds to one additional write of the compressed states disk. No additional generator applications need to be performed during the enumeration. However, to produce a state
once the enumeration has finished, the path in the generators to that state from the root must be applied.

\[ X^A = (A^Q + A^S) \times |V| + (A^D + |S|) \times |V_C| \hspace{1cm} \text{The sum of the sorting and the scanning access plus an additional write of the compressed states no longer in the frontier} \]

\[ X^C = ||G|| \times |S| \hspace{1cm} \text{A generator to reach each state including duplicates} \]

**Inverses**  When each generator has an inverse generator, it is easy to show that duplicates can only occur within two levels of each other in the breadth-first search. They can only appear on the frontier, the current level, or the previous level. This changes the access component of this method by modifying \( A_3 \) in the following manner:

\[ A_D = \sum_{i=1}^{\lfloor T \rfloor} (T_{i-2} + T_{i-1}) \hspace{1cm} \text{Detection Access, for each depth, read the previous states up to depth two} \]

**Optimizations**  In order to avoid some disk accesses, a portion of the breadth-first search that can fit in main memory on the cluster can be performed before switching to the disk-based approach. While this does provide a marginal speedup, it is not significant enough to warrant a separate analysis.

### 6.3 Hash-Based Delayed Duplicate Detection

Hash-based duplicate detection scans a disk-based hash during each delayed duplicate detection phase rather than running through previously seen states. This can reduce the overall time for the computation if the hash requires more space than what is available in memory, but will fit on disk. The amount of space required is the amount of space to store the values seen in the search, the frontier, and the disk-based hash, \( H \).
\[ R^R = 0 \]  
A constant amount for buffers and pieces of the hash

\[ S^F_i = \|G\| \times T_i \]  
Size of neighbors of a level

\[ S^P_i = \sum_{s=1}^{i} (T_s) \]  
Number of states up to a level

\[ R^S = |V| \times \max_{i=1}^{T} (S^F_i + S^P_i) + \left\lceil \frac{|H|}{8} \right\rceil \]  
Size of all the states currently stored and the hash

This technique requires no additional generator applications than does the standard breadth-first search. However, at each depth the new states seen must be added into the disk-based hash rather than requiring a scan of previously seen states.

\[ A^Q = 2 \times |S| \]  
Queue Access, a read and write for each state in the search

\[ A^S = 2 \times \|G\| \times |S| \]  
Scan Access, a read and write for each state including duplicates

\[ A^H = |T| \times \left\lceil \frac{|H|}{8} \right\rceil \]  
Hash Access, for each depth, a read and write of the hash

\[ X^A = (A^Q + A^S) \times |V| + A^H \]  
The sum of the queuing, scanning, and hashing access

\[ X^C = \|G\| \times |S| \]  
A generator to reach each state including duplicates

**Compressed States**  
The most effective use of compressed states is for states not in the open list. Using this method, a significant space reduction may be possible.

\[ R^R = 0 \]  
A constant amount for buffers and pieces of the hash

\[ R^S = \max_{i=1}^{T} (S^F_i \times |V| + S^P_i \times |V_C|) + \left\lceil \frac{|H|}{8} \right\rceil \]  
Size of all the states currently stored and the hash

Using compressed states, the only additional computation that must be performed is a rewrite of the states in the open queue to change them to their compressed form once they have been processed. This corresponds to one additional write of the compressed states disk. No additional generator applications need to be performed during the enumeration. However, to produce a state once the enumeration has finished, the path in the generators to that state from the root must be applied.
\[ X^A = (A^Q + A^S) \times |V| + |S| \times |V_C| + A^H \]

The sum of the hashing and the scanning access plus an additional write of the compressed states no longer in the frontier

\[ X^C = ||G|| \times |S| \]

A generator to reach each state including duplicates

Inverses
Hash-based delayed duplicate detection cannot take advantage of inverses because the hash is not partitioned based on when the states in the search are seen. While it may be possible to devise a hash with this property, it is certainly not trivial and is ignored for the purposes of this analysis.

Optimizations
In order to avoid some disk accesses, a portion of the breadth-first search that can fit in main memory on the cluster can be performed before switching to the disk-based approach. While this does provide a marginal speedup, it is not significant enough to warrant a separate analysis.

6.4 Structured Duplicate Detection
As stated previously, this technique is ignored for the purposes of our analysis as its performance is highly dependent on the underlying structure of the graph and its locality. It is mentioned here for the sake of completeness.

6.5 Level Modulo Three
This technique assumes that there is a memory-resident perfect hash and that inverse generators are used. With this, 2 bits can be used to represent every state in the search not in the open queue. The two bits denote the depth of the state modulo three (initialized to 3 for unseen states). This provides a huge space saving. Where the number of entries in the perfect hash is \(|H|\), the following resources are used.

\[ R^R = \frac{|H| \times 2}{8} \quad \text{Space for the memory resident hash} \]

\[ R^S = \max_{i=1}^{|T|} (T_i \times |V|) \quad \text{Disk to store the open list} \]

This technique performs no additional memory accesses during the search, However, the end representation can only answer queries of group membership and compute a path back to the root
for states in the space.

\[ X^A = 2 \times |S| \times |V| \quad A \text{ read and a write for each state in the open queue} \]

\[ X^C = \|G\| \times |S| \quad \text{Generator applications for the breadth-first search} \]

**Compressed States**  Since states are thrown out after they are removed from the open list, compressed states are generally not of use here.

**Inverses**  This technique must use inverses for correctness. As with inverses in hash-based delayed duplicate detection (Section 6.3), hash locality can garner advantages during the computation, making memory accesses local. This is ignored for the purposes of this analysis.

### 6.6 Implicit Open List

This technique is similar to the two bit trick. Here, however, an invertible hash must also be available. With this, the hash slot of each element can be used to determine the state in that slot directly, without tracing back to the root. With this, it is possible to use a disk-based hash without incurring random access. This allows for a rather significant savings in space. With this technique, only the states reached are recorded. Any additional information desired must be stored elsewhere. Where the number of entries in the perfect is \(|H|\) and the size of the disk buffer allowed is \(|B|\), the following resources are used.

\[ R^R = 0 \quad \text{Space for constant buffers} \]
\[ S^H = \frac{2 \times |H|}{8} \quad \text{Space for the hash} \]
\[ R^S = S^H + |B| \quad \text{Space for the hash and the buffer} \]

Based on the size of the disk buffer, this method has to make passes through the disk-based hash to merge in new states many times at each level. This is accounted for below.
\[ A^I = \left\lceil \frac{\log |H|}{8} \right\rceil \]

The size of a hash table index

\[ A^L_i = \frac{T_i \times \|G\| \times A^I}{|H|} \]

The number of hash passes in a level

\[ A^H = 2 \times S^H \times \sum_{i=1}^{|T|} \left\lceil A^L_i \right\rceil \]

Access to the hash table for merges

\[ A^B = 2 \times S^H \times |T| \]

Access to the hash table for the breadth-first search

\[ A^S = 2 \times \|G\| \times A^I \times |S| \]

Access to the states for merges

\[ X^A = A^H + A^B + A^S \]

Total accesses of three components

\[ X^C = \|G\| \times |S| \]

Total generator applications

Compressed States  Since this method does not store any states in full, compressed states are not applicable.

Inverses  As with inverses in hash-based delayed duplicate detection (Section 6.3), hash locality between neighboring states can garner advantages during the computation, making disk accesses local. This is ignored for the purposes of this analysis.

Optimizations  In order to avoid some disk accesses, a portion of the breadth-first search that can fit in main memory on the cluster can be performed before switching to the disk-based approach. While this does provide a marginal speedup, it is not significant enough to warrant a separate analysis.

6.7 Landmarks

The use of landmarks seeks to avoid disk access all together for the duplicate detection. In order to do this, however, the technique must perform additional generator applications to determine if a state is a duplicate and reproduce states. A landmark ratio, \(1/L\), is selected. This ratio is important because it determines the trade-off between space saved by the computation and additional generator applications performed.

\[ R^P = \log_2 \|G\| (L/2) \times \frac{\log_2 \|G\|}{8} \]

Space for a landmark path

\[ R^R = \frac{|S|}{L} \times |V| + \frac{(L-1) \times |S|}{L} \times R^P \]

Space for landmark and non-landmark storage for all states

\[ R^S = 0 \]

All data is memory resident
The generator time for the landmarks technique is a function of the landmark ratio and the branching factor. Time for both checking duplicates and reproducing states in the fringe must be considered.

\[ X^A = 0 \quad \text{No disk access is required} \]
\[ C^B = \|G\| \times |S| \quad \text{Breadth first search, generator application for each state including duplicates} \]
\[ C^D = \|G\| \times |S| \times \frac{L}{2} \quad \text{Duplicate checking, average distance to a landmark for each state including duplicates} \]
\[ C^R = |S| \times \log_{\|G\|}(\frac{L}{2}) \quad \text{State rebuilding, average path length to the closest landmark for each state in the search space} \]
\[ X^C = C^B + C^D + C^R \quad \text{Total generator time, the sum of the two components} \]

**Compressed States**  While it’s possible to store the landmarks themselves in compressed form, this requires that each landmark state is rebuilt each time a generator path is applied to a state owned by that landmark. This leads to an impractical amount of generator applications and another method is typically preferred instead.

**Inverses**  This method is required to have inverse generators available to rebuild states as it goes. These generators do not need to be part of the search, however. Having them as part of the search provides no additional advantages.

**Optimizations**  The requirement for inverses, as well as the time spent rebuilding states can be avoided by storing the open queue explicitly on disk. This changes the space requirements by adding the states in the open queue to the disk requirements for the algorithm.

\[ R^S = |V| \times \max_{i=1}^{T}(T_i) \quad \text{The space to store the open queue} \]

With this, no duplicates are required and the time spent rebuilding states can be ignored. This adjusts the time spent applying generators as follows.

\[ X^C = C^B + C^D \quad \text{Total generator time, just the time to detect duplicates} \]
6.8 Frontier Search

Frontier search takes advantage of inverses and used operator bits to allow the search to only access the current and previous depth of the search tree in determining duplicates. This can vastly reduce the total amount of data scanned on disk. In order to do this, however, each state needs to be augmented with a set of used operator bits. This changes the effective state size for this technique:

$$|V| = |V| + \lceil \frac{|G|}{8} \rceil$$

The required disk and RAM storage for this technique are given below.

$$R^R = 0 \quad \text{A constant amount for buffers}$$

$$S^F_i = \|G\| \times T_i \quad \text{Size of neighbors of a level}$$

$$S^P_i = \sum_{s=1}^{i} (T_s) \quad \text{Number of states up to a level}$$

$$R^S = |V| \times \max_{i=1}^{T_i} (S^F_i + S^P_i) \quad \text{Size of all the states currently stored}$$

The CPU time for frontier search is easy to predict. It must generate each state seen during the search only once. Therefore it has the exact same CPU time as using the memory resident perfect hash technique. For the disk component, each state must be scanned twice, once when it is generated and once at the next depth to check for duplicates. In addition, each state must be accessed for the external sorting phase. The access and CPU components of the search are listed below.

$$A^Q = 2 \times |S| \quad \text{Queue Access, a read and write for each state in the search}$$

$$A^S = 3 \times \|G\| \times |S| \quad \text{Sort Access, two reads and a write for all states including duplicates}$$

$$A_D = \sum_{i=1}^{T_i} (T_{i-1}) \quad \text{Detection Access, for each depth, read the previous states up to depth one}$$

$$X^A = (A^Q + A^S + A_D) \times |V| \quad \text{The sum of the queuing, sorting, and detection access}$$

$$X^C = \|G\| \times |S| \quad \text{A generator to reach each state including duplicates}$$

**Compressed States** The most effective use of compressed states is for states not in the open list. For frontier search, the size of the compressed states must also be increased to store the frontier
bits. This makes the effective compressed state size:

$$|V_C| = |V_C| + \lceil \frac{|G|}{8} \rceil$$

Using compressed states, a significant space reduction may be possible.

$$R^R = 0 \quad \text{A constant amount for buffers}$$

$$R^S = \max_{i=1}^{T_i} (S^F_i \times |V| + S^P_i \times |V_C|) \quad \text{Size of all the states currently stored}$$

Using compressed states, the only additional computation that must be performed is a rewrite of the states in the open queue to change them to their compressed form once they have been processed. This corresponds to one additional write of the compressed states disk. No additional generator applications need to be performed during the enumeration. However, to produce a state once the enumeration has finished, the path in the generators to that state from the root must be applied.

$$X^A = (A^Q + A^S) \times |V| + (A^D + |S|) \times |V_C| \quad \text{The sum of the sorting and the scanning access plus an additional write of the compressed states no longer in the frontier}$$

$$X^C = ||G|| \times |S| \quad \text{A generator to reach each state including duplicates}$$

**Inverses** This method is required to use inverses in order to guarantee that duplicates will only possibly occur at two depths in the search.

**Optimizations** In order to avoid some disk accesses, a portion of the breadth-first search that can fit in main memory on the cluster can be performed before switching to the disk-based approach. While this does provide a marginal speedup, it is not significant enough to warrant a separate analysis.

### 6.9 Tiered Duplicate Detection

Tiered duplicate detection is capable of exploring multiple levels of the search tree before resorting to using disk to perform delayed duplicate detection. It does this through the use of an imperfect hash function. This effectively changes the number of states at each pass during the enumeration. We break these states down into two components, \( p^{BFS} \) and \( p^{DDD} \), where \( p^{BFS}_i \) and \( p^{DDD}_i \) are the
number of new points seen during pass $i$ of the breadth-first search and delayed duplicate detection respectively.

**Computing Pass Sizes** In order to predict the $P_{BFS}$ and $P_{DDD}$ components, we break down each breadth-first search pass down into its individual levels. At each level $j$ in a pass $i$, a certain number of new states will be seen, $P_{i,j}$. Of these new states, they will be split between those that can be processed immediately in the breadth-first search, $P_{i,j}^{BFS}$, and those that will be discovered during the delayed duplicate detection phase, $P_{i,j}^{DDD}$. To compute these components, we must first compute the number of edges leading into states at depth $j$ in pass $i$, $E_{i,j}$.

\[
P_{i,1}^{BFS} = 1 \text{ The number of states initially, the root}
\]
\[
E_{i,1} = P_{i-1}^{DDD} \times \|G\| \text{ The number of edges to states in the start of a pass, based on the new points seen in the delayed duplicate detection of the previous pass}
\]
\[
E_{i,j} = P_{i-1}^{BFS} \times \|G\| \text{ The number of edges to states beyond the first depth in a pass of the breadth-first search, based on the new points seen at the previous depth}
\]

Just as when predicting the tree structure for a regular breadth-first search, the percentage of remaining edges that go to new values corresponds to the number of remaining edges to new states, divided by the number of remaining edges. This can be computed to be the following:

\[
F_{i}^{L} = \sum_{s=1}^{i-1}(P_{s}^{DDD} + P_{s}^{BFS}) \text{ The number of points processed up to this pass}
\]
\[
F_{i,j}^{N} = \|G\| \times \left( |S| - F_{i}^{L} - \sum_{t=1}^{j-1} P_{t}^{BFS} \right) \text{ Number of new edges remaining, the number of edges remaining minus those that have been seen this pass in the breadth-first search}
\]
\[
F_{i,j}^{R} = \|G\| \times \left( |S| - F_{i}^{L} - \sum_{t=1}^{j-2} P_{t}^{BFS} \right) \text{ Number of edges remaining, the number of edges remaining minus those that have been seen this pass in the breadth-first search before this level}
\]
\[
F_{i,j} = \frac{F_{i,j}^{N}}{F_{i,j}^{R}} \text{ Percent of edges to new states, the fraction of the remaining edges to new states over the remaining edges}
\]

Finally, the number of states remaining is simply the total number of states to be seen minus the number of states that have been seen previous, or:
\[
|S^r_{i,j}| = |S| - P^L_i - \sum_{t=1}^{j-1} (P^{BFS}_t + P^{DDD}_t)
\]
The number of states expected minus the number of states seen in all previous passes as well as the current pass.

The number of edges seen is multiplied with the percentage of edges to new states to obtain the number of edges leading to new states. However, this does not yet deal with edges that lead to the duplicate unseen state. For this, we use Theorem 5.1. We know how many unseen states remain, these are the objects, and we also know how many edges will go to these states, this is the sampling size. Given this, we apply Theorem 5.1 to get the average number of the remaining states with at least one edge mapping to them:

\[
P_{i,j} = \text{sample}(|S^r_{i,j}|, E_{i,j} \times P_{i,j})
\]
The number of edges leading to previously unseen states.

Of these new states seen, some will belong to \(P_i^{BFS}\) and others to \(P_i^{DDD}\). Which go where can be determined by looking at the number of remaining slots in hash, based on the number of states seen during the breadth-first search, the size of the search, \(|S|\), and the hash multiple, \(H\).

\[
H^o_{i,j} = \sum_{s=1}^{i-1} (P^{BFS}_s) - \sum_{t=1}^{j-1} (P^{BFS}_t)
\]
The number of hash entries so far.

\[
H^r_{i,j} = H \times |S| - H^o_{i,j}
\]
The number of hash slots remaining.

We can determine initially how many of the states will hash to locations not already occupied in the hash table, \(H^u_{i,j}\). This is done by looking at the percent of the hash that is unoccupied and the number of new points. The remainder of the points, \(H^c_{i,j}\), must hash into the occupied portion of the hash.

\[
H^u_{i,j} = \frac{H^r_{i,j}}{H \times |S|} \times P_{i,j}
\]
The number of new points whose hash slots were previously empty.

\[
H^c_{i,j} = P_{i,j} - H^u_{i,j}
\]
The number of new points whose hash slots were previously occupied.

Of the states going to the unoccupied locations, we fall back upon Theorem 5.1 to find how many of them hash to empty slots in the hash. This provides us with \(P^{BFS}_{i,j}\). The remaining states, along with \(H^c_{i,j}\), form \(P^{DDD}_{i,j}\).
The number of states hashing to previously empty slots that do not collide with one another

The remainder of the states, those states that collide with previous points or one another

\( P_{BF S}^{i,j} = P_{i,j} - P_{BFS}^{i,j} \)

\( P_{BFS}^{i} \) and \( P_{DDD}^{i} \) are simply the sums of all non-zero \( P_{i,j}^{BFS} \) and \( P_{i,j}^{DDD} \) entries respectively.

Computing Tree Depth Sizes With the pass size information, the tree structure of the search, \( T \), can be determined and used to predict the number of states at each depth in the search for tiered duplicate detection. Here, we use \( T_i \) to denote the number of states seen at each depth in the search during pass \( i \), and \( T_{i,j} \) to denote the number of states at depth \( j \) in the search seen during pass \( i \). In addition, we partition \( T \) into \( T_{BFS} \) and \( T_{DDD} \) to distinguish between values seen during the breadth-first search and those seen during delayed duplicate detection respectively.

It is easy to determine these values for the first pass, \( i = 1 \).

\[ T_{BFS}^{1,i} = P_{BFS}^{1,i} \]

The depths of the breadth-first search states in the first pass, their depth in the breadth-first search for that pass

\[ T_{DDD}^{1,i} = P_{DDD}^{1,i} \]

The depths of the states in the disk-based duplicate detection of the first pass, their depth in the breadth-first search for that pass

From here, the depths in subsequent passes, \( p \) are based on three key pieces of information:

1. the states in the delayed duplicate detection of the previous pass, \( T_{p-1}^{DDD} \);
2. the states at each depth in the delayed duplicate detection of the previous pass, \( T_{p-1,i}^{DDD} \); and
3. the states at each depth in the current pass, either delayed or immediate, \( P_{BFS}^{p,*} \) or \( P_{DDD}^{p,*} \).

With this, the number of states at each depth \( d \) in pass \( p \) can be determined. First, for each depth \( i \) in the previous delayed duplicate detection pass, \( T_{p-1,i}^{DDD} \), the fraction of the total points in that pass at that depth is found by dividing by \( T_{p-1}^{DDD} \). Each depth \( j \) in the current pass, \( BFS_{p,j} \) or \( DDD_{p,j} \), can then be multiplied with each these results to find the total number of states over all pairs of depths at the previous and current pass. By adding the correct pairs of these together, those pairs where the \( i + j = d \), the total number of states seen at depth \( d \) can be determined.
The sum over all depths of the number of breadth-first search states at that depth in the current pass times the percentage of the states in the previous disk-based duplicate detection phase that are at the complimentary depth

The sum over all depths of the number of states seen during the disk-based duplicate detection phase at that depth in the current pass times the percentage of the states in the previous disk-based duplicate detection phase that are at the complimentary depth

**Time and Space Requirements**

Knowing the size of the individual passes, it is now easy to predict the required amount of memory and disk for the search. These sizes are similar to those required for sorting-based delayed duplicate detection, though they need to take the hash array into account. Let \(|P^{DDD}|\) be the last \(i\) such that \(P^{DDD}_i \geq 1\), \(|P^{BFS}|\) be the last \(j\) such that \(P^{BFS}_j \geq 1\), and \(|P|\) be \(\max(|P^{BFS}|, |P^{DDD}|)\).

\[
R^R = H \times |S|/8 \quad \text{A bit for each hash entry}
\]

\[
S^C_i = \|G\| \times (P^{BFS}_i + P^{DDD}_i) \quad \text{Size of neighbors of a level}
\]

\[
S^P_i = \sum_{s=1}^{i} (P^{BFS}_s + P^{DDD}_s) \quad \text{Number of states up to a level}
\]

\[
R^S = |V| \times \max_{i=1}^{\lfloor |P| \rfloor} (S^C_i + S^P_i) \quad \text{Size of all the states currently stored}
\]

Tiered duplicate detection must generate each state seen during the search only once. Therefore it has the exact same generator time as using a memory resident perfect hash or sorting-based delayed duplicate detection. For the disk component, each state must be scanned repeatedly, once for each pass through the disk-based duplicate detection phase, to check for duplicates. In addition, each state must be accessed for the external sort and the breadth-first search. The access and generator components of the search are listed below.
\[ A^Q = 2 \times |S| \quad \text{Queue, a read and write for each state in the search} \]

\[ A^S = 3 \times \|G\| \times |S| \quad \text{Sort, two reads and a write for all states including duplicates} \]

\[ A^D_i = \sum_{t=1}^{i-1}(P_{i,BFS} + P_{i,DDD}) + P_{i,BFS} \quad \text{Detection, for a single pass, a read of all previous states} \]

\[ A^D = \sum_{i=1}^{|P|}(A^D_i) \quad \text{Detection, for all passes, the sum of those passes} \]

\[ X^A = (A^Q + A^S + A^D) \times |V| \quad \text{The sum of the queuing, sorting, and detection access} \]

\[ X^C = \|G\| \times |S| \quad \text{A generator to reach each state including duplicates} \]

**Compressed States** With a large portion of states being discovered during the breadth-first search in the first pass, storing these states in full and the rest in compressed form makes little sense. Instead, with tiered duplicate detection, the final versions of the states and the record of collisions are stored in their compressed forms with paths, but the states in the individual breadth-first search queue on the machines are stored in full. During the disk-based duplicate detection, full states are produced as needed. Much of this computation can be overlapped by not computing the same path from the root twice.

The storage required for tiered duplicate detection given the use of compressed signatures is given below. For this, let \(|P_{i,BFS}|\) be the last \(j\) where \(P_{i,j,BFS} \geq 1\), \(|P_{i,DDD}|\) be the last \(k\) where \(P_{i,k,DDD} \geq 1\), and \(|P_i|\) be \(\max(|P_{i,BFS}|, |P_{i,DDD}|)\).

\[ R^R = H \times |S|/8 \quad \text{The space for the hash table} \]

\[ S_{i,j}^B = \sum_{s=1}^j P_{i,s,BFS} \quad \text{New states seen so far in the breadth-first search of the current pass} \]

\[ S_{i,j}^D = (P_{i,j,DDD}^B + S_{i,j-1,j}^B) \times \|G\| \quad \text{All the states seen in the current pass} \]

\[ S_{i}^P = \sum_{s=1}^{i-1}(P_{s,BFS} + P_{s,DDD}) \times |V_C| \quad \text{Space required by previous passes} \]

\[ S_{i}^C = \max_{j=1}^{|P_i|}(S_{i,j}^D \times |V_C| + P_{i,j,BFS} \times |V|) \quad \text{The space required for the current pass} \]

\[ R^S = \max_{i=1}^{|P_i|}(S_{i}^C + S_{i}^P) \quad \text{The space required for all passes} \]

A generator application must be performed for each state including duplicates seen in the search. However, in addition, the states in \(DDD\) must also be reconstructed. For this, we need to know
something about the current tree structure in the computation. For this, we assume that the values in $T$ are rounded the nearest integer value. In addition, we let $|T_{i,j}^{DDD}|$ be the last $j$ such that $T_{i,j}^{DDD} > 0$ and $|T_{i,j}^{BFS}|$ be the last $j$ such that $T_{i,j}^{BFS} > 0$.

\[
G_{i,j}^P = \frac{\sum_{s=1}^{i-1}(T_{s,j}) + T_{i,j}^{BFS}}{N}
\]

Possible parents during a pass at a particular depth

\[
G_{i,j}^R = \sum_{s=1}^{i-1}(\text{sample}(G_{i,s}^P, \frac{T_{i,j}^{DDD}}{N})) \times N
\]

States rebuilt in a pass for states at a particular depth

\[
G_i^R = \sum_{j=1}^{|T_{i,j}^{DDD}|}(G_{i,j})
\]

States rebuilt during a pass

\[
X^C = \sum_{i=1}^{|T_{i,j}^{DDD}|}(G_i) + \|G\| \times |S|
\]

States rebuilt over all passes and the search

\[
A^Q = 2 \times |S|
\]

Breadth-first search queue access

\[
A^S = 3 \times \|G\| \times |S|
\]

External sort access

\[
A^D = \sum_{s=1}^{|P_{i,j}^{BFS} + P_{i,j}^{DDD}}(P_{i,j}^{BFS} + P_{i,j}^{DDD})
\]

Duplicate detection access

\[
A^R = 4 \times \sum_{i=1}^{|P_{i,j}^{DDD}}(P_{i,j}^{DDD})
\]

Access while rebuilding full states

\[
X^A = A^Q \times |V| + (A^S + A^D + A^R) \times |V_C|
\]

The sum of the four components

**Inverses**  While inverses can be used to guarantee that no more than the two previous levels are accessed during the delayed duplicate detection, it generally preferable to use sorting-based delayed duplicate detection if inverses are present. The levels in tiered duplicate detection correspond to the individual passes, and are therefore much larger and fewer in number. Avoiding scanning passes has very little effect since there are typically very few passes.

**Alternatives to an Imperfect Hash Function**  This technique can use any of the lossy hash techniques described in Section 2.4.3 to replace the imperfect hash function described for this technique. Because hash collisions are checked on disk, these techniques do not need to distinguish all states. Typically, the alternative that results in the fewest false hash collisions is chosen. This increases the number of states seen during the breadth-first search component of the technique, reducing the total amount of disk accessed during the search.

### 6.10 Sensitivity of Analysis to Parameter Modification

Here we look at two different effects on the application and architecture parameters that can compromise the accuracy of the predicted time and space requirements. First, we look at the expected growth in the size of memory and disk over time and show how it effects both the sizes of search
spaces that can be enumerated for each technique as well as how the performance of each technique improves as more resources become available. After this, we consider the effect of variance in the estimated search size on the predicted computation time.

6.10.1 Effect of Growth in Available Storage

As architectures continue to change, so do the resources available. Changing the speed of the CPU, memory, and disk has a straightforward effect on the time required by the algorithms in this section. However, for many of these algorithms, changing the size of the available resources, either the amount of memory or disk, can have an effect both on the size of enumeration that can be performed along with the speed at which a particular enumeration can be performed. This effect is not as straightforward. Here we examine these two factors by looking at some sample applications as available resources grow.

Effect on Feasible Search Size  Consider the fact that the state size, $|V|$, can never be smaller than the log of the search size, $\frac{lg(|S|)}{s}$. To predict the effect of increasing memory and disk capacity on feasible search sizes given the different techniques, we look at three characteristic applications. The first of these being applications where the state size scales with the search size and stays relatively small, $|V| = 16 \times \frac{lg(|S|)}{s}$, the second where the state size scales with the search size and remains relatively typical, $|V| = 80 \times \frac{lg(|S|)}{s}$, and the last where the state size scales with the search size and grows relatively large, $|V| = 800 \times \frac{lg(|S|)}{s}$.

We consider a moderate number of generators, 4, and a branching factor somewhat less than that, 3. We also assume a moderately sized cluster of 30 nodes. Below are listed the relevant application and architectural parameters for each of these applications. The search size, $|S|$, is left out, as that is the parameter we seek to maximize. In addition, information about the speed of the generator applications, $BW^G$, is also left out, as we are only concerned about size and not time of the computation.

<table>
<thead>
<tr>
<th>Small States</th>
<th>Typical States</th>
<th>Large States</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>G</td>
<td>$ = 4</td>
</tr>
<tr>
<td>$|G|$ = 3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$N$ = 30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>$</td>
<td>V</td>
<td>$ = $16 \times \frac{lg(</td>
</tr>
</tbody>
</table>

From this, we can look at each technique to see what search sizes it is capable of solving given some range in the size of memory and disk. We assume there will always be a factor of 10 relative
difference between these sizes. We shall examine search space sizes with memory requirements in the range 4 gigabyte to 32 gigabytes per node, and disk requirements in the range 400 gigabytes to 3.2 terabytes per node.

Below is a table listing what the limitations to search space sizes due to memory and disk are for each technique for the three applications described above. Techniques are ordered based on the maximum size of the state space they are capable of enumerating, from least to greatest. Where a perfect hash is used, it is assumed to be optimal, \(|H| = |S|\), and where inverses are applicable, it is assumed they are already included in the number of generators and branching factor specified above. Finally, (C) indicates the technique is using compressed states.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Memory Limit to States</th>
<th>Disk Limit to States</th>
<th>Bottleneck</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 gigabytes 32 gigabytes</td>
<td>400 gigabytes 3.2 terabytes</td>
<td></td>
</tr>
<tr>
<td>Landmarks ((L = 18))</td>
<td>(2 \times 10^{10}) (2 \times 10^{11})</td>
<td>(\infty) (\infty)</td>
<td>Memory</td>
</tr>
<tr>
<td>Tiered DD ((HM = 1))</td>
<td>(9 \times 10^{11}) (7 \times 10^{12})</td>
<td>(7 \times 10^{10}) (5 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Frontier Search</td>
<td>(\infty) (\infty)</td>
<td>(9 \times 10^{10}) (6 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Hash-Based DDD</td>
<td>(\infty) (\infty)</td>
<td>(9 \times 10^{10}) (6 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Sorting-Based DDD</td>
<td>(\infty) (\infty)</td>
<td>(9 \times 10^{10}) (6 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Frontier Search (C)</td>
<td>(\infty) (\infty)</td>
<td>(1 \times 10^{11}) (1 \times 10^{12})</td>
<td>Disk</td>
</tr>
<tr>
<td>Hash-Based DDD (C)</td>
<td>(\infty) (\infty)</td>
<td>(1 \times 10^{11}) (1 \times 10^{12})</td>
<td>Disk</td>
</tr>
<tr>
<td>Sorting-Based DDD (C)</td>
<td>(\infty) (\infty)</td>
<td>(1 \times 10^{11}) (1 \times 10^{12})</td>
<td>Disk</td>
</tr>
<tr>
<td>BFS w/ Perfect Hash</td>
<td>(9 \times 10^{11}) (7 \times 10^{12})</td>
<td>(1 \times 10^{11}) (1 \times 10^{12})</td>
<td>Disk</td>
</tr>
<tr>
<td>Tiered DD ((HM = 1)) (C)</td>
<td>(9 \times 10^{11}) (7 \times 10^{12})</td>
<td>(3 \times 10^{11}) (1 \times 10^{12})</td>
<td>Disk</td>
</tr>
<tr>
<td>BFS w/ Perfect Hash (C)</td>
<td>(9 \times 10^{11}) (7 \times 10^{12})</td>
<td>(4 \times 10^{11}) (2 \times 10^{12})</td>
<td>Disk</td>
</tr>
<tr>
<td>Level Modulo Three</td>
<td>(4 \times 10^{11}) (3 \times 10^{12})</td>
<td>(4 \times 10^{11}) (3 \times 10^{12})</td>
<td>Both</td>
</tr>
<tr>
<td>Implicit Open List</td>
<td>(\infty) (\infty)</td>
<td>(1 \times 10^{12}) (1 \times 10^{13})</td>
<td>Disk</td>
</tr>
</tbody>
</table>

Here, with small state sizes, we see that the landmarks technique is very inefficient, as storage as a short path offers little advantage. The implicit open list technique is the best of the techniques requiring a perfect hash, as each state is represented with only two bits and no open list is required. Tiered duplicate detection with compressed states is the best of the techniques not requiring a perfect hash. This is primarily due to the fact that duplicates in the disk blocks are able to be stored as compressed states rather than full states. This is possible because these blocks are scanned less
frequently for false collisions.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Memory Limit to States</th>
<th>Disk Limit to States</th>
<th>Bottleneck</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4 gigabytes 32 gigabytes</td>
<td>400 gigabytes 3.2 terabytes</td>
<td></td>
</tr>
<tr>
<td>Tiered DD ((HM = 1))</td>
<td>(9 \times 10^{11}) 7 (\times 10^{12})</td>
<td>(1 \times 10^{10}) (1 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Landmarks ((L = 54))</td>
<td>(1 \times 10^{10}) (1 \times 10^{11})</td>
<td>(\infty) (\infty)</td>
<td>Memory</td>
</tr>
<tr>
<td>Frontier Search</td>
<td>(\infty) (\infty)</td>
<td>(1 \times 10^{10}) (1 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Hash-Based DDD</td>
<td>(\infty) (\infty)</td>
<td>(1 \times 10^{10}) (1 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Sorting-Based DDD</td>
<td>(\infty) (\infty)</td>
<td>(1 \times 10^{10}) (1 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Frontier Search (C)</td>
<td>(\infty) (\infty)</td>
<td>(3 \times 10^{10}) (2 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Hash-Based DDD (C)</td>
<td>(\infty) (\infty)</td>
<td>(3 \times 10^{10}) (2 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Sorting-Based DDD (C)</td>
<td>(\infty) (\infty)</td>
<td>(3 \times 10^{10}) (2 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>BFS w/ Perfect Hash</td>
<td>(9 \times 10^{11}) 7 (\times 10^{12})</td>
<td>(3 \times 10^{10}) (2 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>BFS w/ Perfect Hash (C)</td>
<td>(9 \times 10^{11}) 7 (\times 10^{12})</td>
<td>(9 \times 10^{10}) (6 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Level Modulo Three</td>
<td>(4 \times 10^{11}) (3 \times 10^{12})</td>
<td>(9 \times 10^{10}) (7 \times 10^{11})</td>
<td>Disk</td>
</tr>
<tr>
<td>Tiered DD ((HM = 1)) (C)</td>
<td>(9 \times 10^{11}) 7 (\times 10^{12})</td>
<td>(3 \times 10^{11}) (1 \times 10^{12})</td>
<td>Disk</td>
</tr>
<tr>
<td>Implicit Open List</td>
<td>(\infty) (\infty)</td>
<td>(1 \times 10^{12}) (1 \times 10^{13})</td>
<td>Disk</td>
</tr>
</tbody>
</table>

Here, with typical state sizes, we see similar results. Implicit open list remains the best hash-based approach, and tiered duplicate detection with compressed states the best technique not utilizing a hash. It’s interesting to note that tiered duplicate detection with compressed states actually surpasses breadth-first search with a perfect hash here. This is due to the fact that the largest open queue size in the breadth-first search is typically much bigger than the largest open queue for tiered duplicate detection with compressed states, and only states in the open queue for both approaches are stored in full. It’s important to note that while implicit open list appears to surpass the other techniques by a large margin, it’s not entirely a fair comparison. If a perfect invertible hash exists of size \(|S|\), one can represent states as their hash index in that hash rather than the full state value.
Here, with large state sizes, implicit open list is still the best technique using a hash and tiered duplicate detection with compressed states still the best technique not requiring one. Landmarks has worked its way up to being a feasibly good technique, as the large state sizes allow for it to be used effectively to reduce the storage. Once again, it’s important to note that while implicit open list appears to surpass the other techniques by a large margin, it’s not entirely a fair comparison. If a perfect invertible hash exists of size $|S|$, one can represent states as their hash index in that hash rather than the full state value.

From all of these tables, a complete ordering of techniques exist that is independent of the size of disk and memory available (as long factor of ten between them remains constant). This shows that the ordering of these techniques in the space-time search hierarchy remains constant with respect to relative space required by the computations as storage grows.

**Effect on Predicted Time** Consider the fact that for most disk-based techniques, each time a state is generated it must be scanned multiple times during the duplicate detection phase. To predict the effect of increasing memory and disk capacity on predicted search times given the different techniques, we look at two characteristic applications. We first fix a state size, $|V| = 100$, a search
size, \(|S| = 30 \times 10^9\), a disk bandwidth, \(BW^D = 10 \times 10^6\), and an average number of passes through states on disk, \(P = 2.5\). The first application of these being one where the generator time is fast compared to the time to scan a state \(P\) times on disk, \(BW^G = 2.5 \times \frac{BW^D}{P|V|}\), and the second where the generator time is slow compared to the time to scan a state \(P\) times on disk, \(BW^G = \frac{1}{4} \times \frac{BW^D}{P|V|}\).

As in the previous section, we consider a moderate number of generators, 4, and a branching factor somewhat less than that, 3, and a moderately sized cluster of 30 nodes. Below are listed the relevant application and architectural parameters for each of these applications.

<table>
<thead>
<tr>
<th>Fast Generation</th>
<th>Slow Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>S</td>
</tr>
<tr>
<td>(</td>
<td>G</td>
</tr>
<tr>
<td>(</td>
<td>G</td>
</tr>
<tr>
<td>(N)</td>
<td>30</td>
</tr>
<tr>
<td>(</td>
<td>V</td>
</tr>
<tr>
<td>(BW^D)</td>
<td>(10 \times 10^6)</td>
</tr>
<tr>
<td>(BW^G)</td>
<td>100000</td>
</tr>
</tbody>
</table>

From this, we can look at each technique to see how quickly it is capable of enumerating the state space given some range in the size of memory and disk. We assume there will always be a factor of 10 relative difference between these sizes. We shall examine search space sizes with memory requirements in the range 4 gigabyte to 32 gigabytes per node, and disk requirements in the range 400 gigabytes to 3.2 terabytes per node.

The table below lists, given increasing amounts of memory and disk for applications with relatively fast generators, the time each technique requires in addition to the percent of that time devoted to the application of the generators. Where a perfect hash is used, it is assumed to be optimal, \(|H| = |S|\), and where inverses are applicable, it is assumed they are already included in the number of generators and branching factor specified above. Finally, \((C)\) indicates the technique is using compressed states and \((I)\) that inverses are present. The techniques are sorted based on the percent of the time spent applying generators.
### Performance for Application with Fast Generator Speed

<table>
<thead>
<tr>
<th>Memory:</th>
<th>4 GB</th>
<th>8 GB</th>
<th>16 GB</th>
<th>32 GB</th>
<th>% Gen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk:</td>
<td>400 GB</td>
<td>800 GB</td>
<td>1.6 TB</td>
<td>3.2 TB</td>
<td>Time</td>
</tr>
<tr>
<td>Landmarks</td>
<td>6.08 days</td>
<td>3.01 days</td>
<td>1.79 days</td>
<td>1.16 days</td>
<td>100%</td>
</tr>
<tr>
<td>Implicit Open List</td>
<td>0.39 days</td>
<td>0.38 days</td>
<td>0.38 days</td>
<td>0.38 days</td>
<td>88 – 98%</td>
</tr>
<tr>
<td>BFS w/ Perfect Hash</td>
<td>0.56 days</td>
<td>0.56 days</td>
<td>0.56 days</td>
<td>0.56 days</td>
<td>60%</td>
</tr>
<tr>
<td>Level Modulo Three</td>
<td>0.56 days</td>
<td>0.56 days</td>
<td>0.56 days</td>
<td>0.56 days</td>
<td>60%</td>
</tr>
<tr>
<td>Tiered DD (C)</td>
<td>0.77 days</td>
<td>0.76 days</td>
<td>0.75 days</td>
<td>0.73 days</td>
<td>48%</td>
</tr>
<tr>
<td>Hash-Based DDD</td>
<td>1.28 days</td>
<td>1.28 days</td>
<td>1.28 days</td>
<td>1.28 days</td>
<td>27%</td>
</tr>
<tr>
<td>Sorting-Based DDD (I) (C)</td>
<td>1.67 days</td>
<td>1.67 days</td>
<td>1.67 days</td>
<td>1.67 days</td>
<td>21%</td>
</tr>
<tr>
<td>Sorting-Based DDD (C)</td>
<td>1.77 days</td>
<td>1.77 days</td>
<td>1.77 days</td>
<td>1.77 days</td>
<td>20%</td>
</tr>
<tr>
<td>Sorting-Based DDD (I)</td>
<td>1.97 days</td>
<td>1.97 days</td>
<td>1.97 days</td>
<td>1.97 days</td>
<td>18%</td>
</tr>
<tr>
<td>Tiered DD</td>
<td>2.07 days</td>
<td>2.07 days</td>
<td>2.06 days</td>
<td>1.96 days</td>
<td>17 – 18%</td>
</tr>
<tr>
<td>Frontier Search</td>
<td>0.40 days</td>
<td>0.40 days</td>
<td>0.40 days</td>
<td>0.40 days</td>
<td>17%</td>
</tr>
<tr>
<td>Sorting-Based DDD</td>
<td>2.82 days</td>
<td>2.82 days</td>
<td>2.82 days</td>
<td>2.82 days</td>
<td>12%</td>
</tr>
</tbody>
</table>

In addition, these results can also be seen in the graph in Figure 3. Notice that the landmarks technique continues to improve until it is the best technique not requiring a perfect hash outside of tiered duplicate detection and frontier search. Frontier search is the best technique not requiring a perfect hash, though it does require inverses, and implicit open list is the best technique requiring one.

This table lists, given increasing amounts of memory and disk for applications with relatively slow generators, the time each technique requires in addition to the percent of that time devoted to the application of the generators. Where a perfect hash is used, it is assumed to be optimal, $|H| = |S|$, and where inverses are applicable, it is assumed they are already included in the number of generators and branching factor specified above. Finally, (C) indicates the technique is using compressed states and (I) that inverses are present. The techniques are sorted based on the percent of the time spent applying generators.
Figure 3: The time for an application with fast state generation for each technique as memory and disk storage grow
### Performance for Application with Slow Generator Speed

<table>
<thead>
<tr>
<th>Memory:</th>
<th>Disk:</th>
<th>Landmarks</th>
<th>Tiered DD (C)</th>
<th>Implicit Open List</th>
<th>BFS w/ Perfect Hash</th>
<th>Level Modulo Three</th>
<th>Hash-Based DDD</th>
<th>Sorting-Based DDD (I) (C)</th>
<th>Sorting-Based DDD (C)</th>
<th>Sorting-Based DDD (I)</th>
<th>Frontier Search</th>
<th>Tiered DD</th>
<th>Sorting-Based DDD</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 GB</td>
<td>400 GB</td>
<td>2.03 months</td>
<td>4.06 days</td>
<td>3.52 days   3.70 days</td>
<td>3.70 days</td>
<td>3.70 days</td>
<td>4.40 days</td>
<td>4.79 days</td>
<td>4.89 days</td>
<td>5.09 days</td>
<td>5.11 days</td>
<td>5.20 days</td>
<td>5.94 days</td>
</tr>
<tr>
<td>8 GB</td>
<td>800 GB</td>
<td>1.00 months</td>
<td>3.98 days</td>
<td>3.50 days   3.70 days</td>
<td>3.70 days</td>
<td>3.70 days</td>
<td>4.40 days</td>
<td>4.79 days</td>
<td>4.89 days</td>
<td>5.09 days</td>
<td>5.11 days</td>
<td>5.20 days</td>
<td>5.94 days</td>
</tr>
<tr>
<td>16 GB</td>
<td>1.6 TB</td>
<td>17.94 days</td>
<td>3.93 days</td>
<td>3.50 days   3.70 days</td>
<td>3.70 days</td>
<td>3.70 days</td>
<td>4.40 days</td>
<td>4.79 days</td>
<td>4.89 days</td>
<td>5.09 days</td>
<td>5.11 days</td>
<td>5.20 days</td>
<td>5.94 days</td>
</tr>
<tr>
<td>32 GB</td>
<td>3.2 TB</td>
<td>11.57 days</td>
<td>3.88 days</td>
<td>3.50 days   3.70 days</td>
<td>3.70 days</td>
<td>3.70 days</td>
<td>4.40 days</td>
<td>4.79 days</td>
<td>4.89 days</td>
<td>5.09 days</td>
<td>5.11 days</td>
<td>5.20 days</td>
<td>5.94 days</td>
</tr>
<tr>
<td>Time</td>
<td>% Gen</td>
<td>100%</td>
<td>99%</td>
<td>99%</td>
<td>94%</td>
<td>94%</td>
<td>79%</td>
<td>72%</td>
<td>71%</td>
<td>68%</td>
<td>68%</td>
<td>67%</td>
<td>58%</td>
</tr>
</tbody>
</table>

In addition, these results can also be seen in the graph in Figure 4. The landmarks technique is so slow it is not even presented in the given range. Tiered duplicate detection with compressed states is the best technique not requiring a perfect hash and implicit open list is the best technique requiring one.

From the above tables and graphs, we see the effect of increasing the available storage on computation time. Some techniques, such as implicit open list and tiered duplicate detection can obtain performance gains from this increase. However, these gains are typically marginal. It is more interesting to look at each technique to see which component of the enumeration dominates the time for the enumeration. Those techniques dominated by CPU time will see the largest performance increase with faster and more processors going into future. Those dominated by disk time will see the largest performance increase with faster and more disks going into the future.

#### 6.10.2 Variance in the Estimated Search Size

In order to show that errors in estimates of the search size do not significantly impact run-times, we examine many different application examples and look at transitioning between techniques as the search size grows. The two figures below show two applications plotted with their search sizes increasing and how that effects the time for the computation. The point at which the technique
Figure 4: The time for an application with slow state generation for each technique as memory and disk storage grow.
used in enumeration was switched is shown. Figure 5(a) looks at a case when no perfect hash is available. Figure 5(b) looks at the case when a perfect hash is available.

What we see is that these transitions are very smooth for techniques that do not have access to a perfect hash, and reasonably smooth for those that do. The jumps for techniques requiring a perfect hash occur mainly when that technique is forced out of core. This is a common trend, and while more applications could be analyzed, the results would be similar. The reasonably gradual increase is largely due to the fact that the space-time search hierarchy exists, allowing one to select a new technique that requires only slightly more computation than the previous technique required.

7 General Software for Implicit State Space Enumeration

We define three “experts” when examining search and enumeration problems. There are the:

- *Application expert*, who knows details about the specific state space to be enumerated; the
- *Architecture expert*, who knows about the computer architecture, or cluster, that will be used to perform that enumeration; and the
- *Algorithm expert*, who knows how to program a particular enumeration technique.

Each of these individuals may develop their component of the application independently, using only the API as an intermediary to relay information between them. This allows for different
problems to be run on different systems (clusters) using different search algorithms almost completely transparently.

Here an API for large implicit state space enumerations is defined. Each expert is responsible for writing their own component, and can utilize the available functions and data from existing components. With this, the goal is to allow the end user (the application expert) to input his or her algorithm parameters into the API and have the software:

- Determine the run-times for each of the implemented search techniques;
- Select the fastest technique and enumerate the search space; and
- Allow user access to the enumerated space independently of the technique selected.

In addition, it should also be easy for an algorithm expert to add a new search technique to the existing suite of techniques by obeying the API defined in this thesis.

### 7.1 Overview of Interfaces

We use the term *interface* in the sense of a Java interface. An interface is a specification for the functionality of a component of the computation. An instance of the interface is an implementation of that functionality. An interface is defined for each expert listed previously in this section. Figure 6 shows an overview of the relationships between these interfaces and the application using them.

An example of how these interfaces can be used to find the depths of solutions in the 15-puzzle is shown below. The actual implementations of the interfaces themselves for this application are shown in the appropriate sections (Section 7.3, Section 7.4, and Section 7.5). The algorithm interface instance used is one that implements a memory-resident hash table storing full states. It is meant for exposition only, and will not work on large enumeration applications, as defined in Section 2.1. It is assumed that a C++ style syntax is used, where & is used for pass-by-reference, and * indicates a pointer value.

```c++
// An example main function for an application using the API
int main( int argc, char **argv ) {
    // Solve the 15-puzzle, predicting depths
    ApplicationInterface *p = new Sliding15Puzzle();
    // Use my account on a local cluster with MPI
    ArchitectureInterface *a = new TeraClusterMPI( &argc, &argv, "myusername" );
    // Search with a memory-resident hash that resolves collisions
    AlgorithmInterface *s = new MemoryHashSearch();

    // Verify the search can be run and report the findings
    if( !s->Run( p, a, s->RunPredict ) ) {
        cout << "Not enough resources to run the search!" << endl;
        return -1;
    }
}
```
Figure 6: The relationships between the three interfaces and the application using them

```c
// Query the results
BYTE state[p->StateSize] = /* Initialize a state to ask about depth */;
BYTE result[p->StateSize];
s->Query( s->QueryHasState, state, result );
if( result[0] ) {
    s->Query( s->QueryGetState, state, result );
    cout << "The depth of the state is at most " << result[16] << endl;
} else cout << "That state is not present in the search" << endl;
return 0;
}
```

Listed below is a table with the basic data and functionality of each interface. These interfaces are defined in more detail in the following sections. In addition to these interfaces, a generic path utility is also defined since both the application expert and algorithm expert must use a representation of
a path in the search.

<table>
<thead>
<tr>
<th>Application Interface</th>
<th>Architecture Interface</th>
<th>Algorithm Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>int SearchSize</td>
<td>int MemorySize</td>
<td>bool Queries[256]</td>
</tr>
<tr>
<td>int StateSize</td>
<td>int DiskSize</td>
<td>int TimeComp</td>
</tr>
<tr>
<td>int NumGenerators</td>
<td>int NumMachines</td>
<td>int TimeAccess</td>
</tr>
<tr>
<td>void getRoot(...)</td>
<td>void getCommInfo(...)</td>
<td>int MemoryReq</td>
</tr>
<tr>
<td>void applyGenToState(...)</td>
<td>void sendMessage(...)</td>
<td>int DiskReq</td>
</tr>
<tr>
<td>int HashBits</td>
<td>void waitMessage(...)</td>
<td>bool run(...)</td>
</tr>
<tr>
<td>bool HashInvertible</td>
<td>void recvMessage(...)</td>
<td>bool query(...)</td>
</tr>
<tr>
<td>int Inverses</td>
<td>string CommonDir</td>
<td></td>
</tr>
<tr>
<td>float BranchingFactor</td>
<td>string TempDir</td>
<td></td>
</tr>
<tr>
<td>float KeySize</td>
<td>string FinalDir</td>
<td></td>
</tr>
<tr>
<td>void getHash(...)</td>
<td>int NumFiles</td>
<td></td>
</tr>
<tr>
<td>void inverseGenToState(...)</td>
<td>int NetworkPacketSize</td>
<td></td>
</tr>
<tr>
<td>void applyGensToStates(...)</td>
<td>int DiskPacketSize</td>
<td></td>
</tr>
<tr>
<td>void inverseGensToStates(...)</td>
<td>void startComm(...)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>void endComm(...)</td>
<td></td>
</tr>
</tbody>
</table>

Note that variables and functions in boldface are mandatory, while those in italics are optional and have some default definition or behavior if not specified. The next section, Section 7.2, contains a discussion as to how the variables can be estimated automatically.

The following three sections, Section 7.3, Section 7.4, and Section 7.5, provide information on the application interface, architecture interface, and algorithm interface respectively. This includes the default values and behaviors for optional variables and functions.

### 7.2 Automatic Prediction of Parameters

For many applications, the architectural and application parameters may not be known ahead of time. While it is possible to run many of the search techniques without knowing these ahead of time, it makes selecting the best technique and predicting run-times difficult. Here we discuss methods for predicting unknown parameters for the search application or the architecture. Each of the variables for these interfaces are examined here.

**SearchSize** Unlike many of the other parameters the user may not know this parameter ahead of time. The parameter is also not trivial to predict. Some applications may know the size of the state space.
space in advance. For those that don’t, many application-specific techniques exist for estimating it. In addition, some general, and typically weaker, methods exist as well. These approaches are discussed later in the section.

**StateSize**  This parameter can not be predicted and must be provided. It is intrinsic to applying the generators successfully during the search.

**NumGenerators**  This parameter can not be predicted and must be provided. It is intrinsic to knowing the size of steps in a search path as well as determining what calls to the generators are valid.

**HashBits**  While a very conservative hash function can be obtained, where the state key itself is the hash value, this is very rarely optimal. Typically, a better bound on the number of bits required for a perfect hash function can not be predicted automatically.

**HashInvertible**  If the conservative hash function, where the state key itself is the hash value, was used, it is trivially invertible. Outside of this situation, it would be very difficult to predict the inverse hash function, and therefore to predict whether or not this variable can be set to true.

**Inverses**  One can easily perform a small portion of the search, and then determine if there are inverse generators for all of the edges seen so far and what those inverse generators are. This would allow one to “estimate” that inverses existed and were included in the search. However, typically this is not the case, and can be dangerous, as there is no guarantee that these are actual inverses if they are used. Instead, this variable is typically specified by the user.

**BranchingFactor**  This parameter can be easily predicted by performing a small portion of the search on a single machine. Typically, if this is done to a deep enough depth, any locality in the graph will be observed. The branching factor can be determined by dividing the number of states at depth $d + 1$ by the number of states at depth $d$, where the depth selected, $d$, is small compared to the maximum depth of the search.

**KeySize**  This parameter, much like the state size, can not be predicted and must be provided. It is intrinsic to performing duplicate detection for generated states.
MemorySize This parameter can typically be obtained by a call togetrlimit. However, this typically gives an upper bound. Where resource limits want to be controlled to be within some reasonable range, this parameter should be specified by the user.

DiskSize This parameter can typically be obtained by a call tostatfs. However, this typically gives an upper bound. Where resource limits want to be controlled to be within some reasonable range, this parameter should be specified by the user.

NumMachines Once the communication package is initialized, this parameter can typically be obtained directly through the architecture interface. If unknown, communication can be initialized prior to making run-time predictions in order to obtain it.

CommonDir When left blank, the common directory is assumed to be the directory with the executable. Typically, though, this parameter is specified by the user.

TempDir When left blank, the temporary directory is assumed to be the directory with the executable. Typically, though, this parameter is specified by the user.

FinalDir When left blank, the final directory is assumed to be the directory with the executable. Typically, though, this parameter is specified by the user.

NumFiles This parameter can typically be obtained by a call togetrlimit. Typically the parameter obtained works well for the computation.

NetworkPacketSize This parameter can be estimated once the communication package is initialized. By performing sends and receives across the network of various sizes, it is easy to observe when a peak in performance is obtained.

DiskPacketSize This parameter can be estimated once the directories for the application are known. By performing reads and writes to disk of various sizes, it is easy to observe when a peak in performance is obtained.

Estimating the Size of the State Space

Here we look at two different approaches to estimating the size of the state space. Application-specific approaches generally only work for certain varieties of applications, or when certain conditions hold.
General techniques work in more or all cases, but typically are not as efficient, in that they take longer to form an estimation or the estimation obtained is not as accurate.

**Application-Specific Approaches**  Here we consider one application specific technique for predicting search sizes in the case of explicit state model checking for concurrent processes. Here, the problem sizes typically scale in the number of processors running concurrently. To estimate the search size, enumerations can be performed with fewer processors, and the scaling factor of adding an additional processor can be obtained. This approach was used to predict search sizes for Peterson’s mutual exclusion algorithm found in Section 9.2 and yielded accurate predictions.

**General Approaches**  We present three general approaches to predicting the size of the implicit state space. These require varying levels of knowledge of the underlying state space. Many other approaches exist outside of just these. However, it is only our goal to show that estimating the search size independent of the search application is feasible.

The first approach requires running the enumeration using a memory-resident Bloom filter. Once this has been done, the resulting Bloom filter can be analyzed to determine the theoretical size of the state space. The enumeration can then be run again with this estimate using any technique desired. Details of this approach can be found in [25].

The second approach looks at random walks in the graph and uses the length of these random walks to estimate the graph size. For example, in cases where no locality exists, an average random walk of length of \( t \) before reaching a state that has been seen previously implies a graph size \( O(t^2) \). This allows one to predict the size of the state space in the case of random graphs after only \( O(\sqrt{|S|}) \) generator applications, where \(|S|\) is the size of the state space. More information on random walks and their relation to the size of the underlying state space can be found in [2].

The final approach uses an “iterative deepening” technique. The search size can be assumed to be some small size, and then, when found to be larger during the search, that estimate can be doubled. Eventually an estimate large enough to fit the entire search will be obtained. Since the time for these algorithms relative to the search size increases at best linearly, if the exact search size is used in the final approximation, no more than double the time for that search will be used in all approximations. However, one can largely overestimate the search size during this process, up to approximately double the actual size. To avoid this, analysis of the underlying search tree can be performed after each approximation to give a better estimate on the approximate size of the search for the subsequent attempt.
7.3 Application Interface

An example of an instance of the application interface for the sliding tile 15 puzzle is shown below.

This is the instance of the interface used in the example at the beginning of this section.

```cpp
// An example application interface, the 15-puzzle (Sliding Tile) storing depths
class Sliding15Puzzle: public ApplicationInterface {
private:
    // The direction to move the space in
    enum { LEFT = 0, RIGHT, UP, DOWN };

    // Finds the row and column of the blank space (0)
    int findSpaceRow( BYTE *state ) {...}
    int findSpaceCol( BYTE *state ) {...}

    // Finds the square value at a particular row and column
    int getSquare( BYTE *state, int row, int col ) {...}

    // Sets the square value at a particular row and column
    int setSquare( BYTE *state, int row, int col, int val ) {...}

public:
    Sliding15Puzzle( ) {
        // Initialize the branching factor, the number of possible moves
        NumGenerators = 4;

        // Set the state size, 4 bits for each square, plus the depth, max depth of 256
        StateSize = 16 + 1;
        KeySize = 16;
    }

    void applyTargetGen( const BYTE *state, const UINT32 genLabel, BYTE *child, BOOL *success,
                         const SEARCH_PATH path = NULL, const UINT32 pathSize = 0 ) {
        // Copy the state to the child
        memcpy( child, state, StateSize );

        // Find the row and column with the space, assume failure, and increment the depth
        int spaceRow = findSpaceRow( child );
        int spaceCol = findSpaceCol( child );
        *success = false;
        child[16]++;

        switch( genLabel ) {
            case UP:
                if( spaceRow==0 ) break;
                *success = true;
                // Swap the space with the value above it
                setSquare( child, spaceRow, spaceCol, getSquare( child, spaceRow-1, spaceCol ) );
                setSquare( child, spaceRow-1, spaceCol, 0 );
                break;
            case DOWN:
                if( spaceRow==4 ) break;
                *success = true;
                // Swap the space with the value below it
                setSquare( child, spaceRow, spaceCol, getSquare( child, spaceRow+1, spaceCol ) );
                setSquare( child, spaceRow+1, spaceCol, 0 );
                break;
            case LEFT:
                if( spaceCol==0 ) break;
                *success = true;
                // Swap the space with the value to its left
                setSquare( child, spaceRow, spaceCol, getSquare( child, spaceRow, spaceCol-1 ) );
                setSquare( child, spaceRow, spaceCol-1, 0 );
                break;
            case RIGHT:
                if( spaceCol==4 ) break;
                *success = true;
                // Swap the space with the value to its right
```
Here the interface used by the application expert is defined in more detail. This interface is used to specify how values in the search space are produced. It also gives the general information about the graph to be generated. The application interface specifies both variables and functions. Each of these are defined separately below.

**Variables**  Below is a list of the variables specified by the application interface. Where those parameters are optional, the default value is listed in parentheses next to the variable name.

**SearchSize**  – The estimated number of states that will be enumerated during the search. A poor estimate will yield worse performance and performance predictions.

**StateSize**  – The size of a state in bytes. The total storage required by the search is typically on the order of $\text{SearchSize} \times \text{StateSize}$ bytes.

**KeySize** $(\text{StateSize})$  – The size in bytes of the unique identifier for the state (assumed to be at the beginning of the state data). All extra information with the state is assumed to irrelevant to the enumeration.

**NumGenerators**  – The total number of generators used during the enumeration, or an upper bound on the out-degree of a state. The maximum generator label encountered is never more than $\text{NumGenerators}$.

**BranchingFactor** $(\text{NumGenerators})$  – The estimated number of non-duplicate, valid children a state early in the search will have. This number can take into account generators that are not applicable to states along with locality in the graph. A poor estimate will yield worse performance and performance predictions.
HashBits \((\text{KeySize} \times 8)\) – The number of bits required to be produced by a perfect hash function for the states of the search space, or the ceiling of the log base two of the maximum value produced by the perfect hash function. The default assumes the key value is the hash value for a state.

HashInvertible \((\text{StateSize} \times 8 == \text{HashBits})\) – A boolean value that indicates whether or not an invertible hash is available for the enumeration. If the hash value and the state are identical, then the default for this parameter is true. This can be determined by comparing \text{StateSize} and \text{HashBits}.

Inverses (InversesNone) – An enumerated type that describes the availability of inverse generators. The enumerated values are: InversesNone implying no inverses are available are available; InversesAvailable implying that the inverses are available but cannot be used in the enumeration; and finally InversesIncluded implying that the inverses are available and included among the standard generators for the enumeration.

Functions – Below is a list of the functions (C++ methods) defined by the problem interface. Where the function is optional, the default behavior of that function is specified.

getRoot

Input

\text{BYTE *} \text{root} \text{ (Output)}

The memory location to write the roots to.

Description: This function loads the root state for the problem into the memory addressed by root.

getHashIndex

Input

\text{BYTE *} \text{index} \text{ (Input or Output)}

The memory location for the hash value.

\text{BYTE *} \text{state} \text{ (Output or Input)}

The memory location for the state.

\text{bool} \text{ inverse} \text{ (Default: false)}

Whether or not an inverse is being requested.
**Description:** If `inverse` is false, this function obtains the hash index for `state` and writes it to `index`. Otherwise, this function finds the state associated with `index` and writes it to `state`. This function may only be called with `inverse` set to true if the application instance has `HashInvertible` set to true. The default behavior of this function is to return the state or hash index itself.

**applyGenToState**

**Input**

- `const BYTE *state`
  - The state to apply the generator to.
- `const UINT32 label`
  - The generator to apply.
- `BYTE *child` (Output)
  - The memory location to write the child state to.
- `bool *success` (Output)
  - Whether the generator is valid for the state.
- `const PATH *path` (Default: NULL)
  - The path in generators used to reach the state.
- `const UINT32 pathSize` (Default: 0)
  - The size in bytes of the path.

**Description:** This function applies the generator specified by `label` to `state`, writing the state generated to the memory pointed to by `child`, and setting `success` to true. If a child for that state and generator pair does not exist or if it corresponds to a state already seen along `path`, then the content of the memory at `child` is unspecified and the function sets `success` to false.

**inverseGenToState**

**Input**

- `const BYTE *state`
  - The state to apply the generator to.
- `const UINT32 label`
  - The generator to apply.
- `BYTE *child` (Output)
  - The memory location to write the child state to.
- `bool *success` (Output)
  - Whether the generator is valid for the state.
Description: When given a generator specified by its label, this function applies the inverse of that generator to state, writing the state generated to the memory pointed to by child, and setting success to true. If an inverse for that state and generator pair does not exist, then the content of the memory at child is unspecified and the function sets success to false. This function must be defined if Inverses ≠ InversesNone.

applyGensToStates

Input

const BYTE *states
The states to apply the generators to.

const UINT32 count
The number of states to which to apply the generators.

BYTE *children (Output)
An array to write the child states to.

bool *success (Output)
An array to write the success of each generator application to.

const PATH *path (Default: NULL)
An array of paths in generators used to reach the states.

const UINT32 pathSize (Default: 0)
The size in bytes of a single path in the array.

Description: This function applies each generator to states, writing the states generated to the memory pointed to by children, and writing true to the corresponding locations in success. If a child for a particular state and generator pair does not exist, or if it corresponds to a state already seen along the path, then the content of the memory at the corresponding location in children is unspecified and the corresponding location in success is set to false. It is assumed that the results are ordered by state first, then by generator label. The default behavior for this function is to iteratively call applyGenToState.

inverseGensToStates

Input

const BYTE *states
The states to apply the generators to.

const UINT32 count
The number of states to which to apply the generators.

BYTE *children (Output)
An array to write the child states to.

\textit{bool \*success} (Output)

An array to write the success of each generator application to.

\textit{Description:} This function applies each generator’s inverse to \textit{states}, writing the states generated to the memory pointed to by \textit{children}, and writing true to the corresponding locations in \textit{success}. If an inverse for a particular state and generator pair does not exist then the content of the memory at the corresponding location in \textit{children} is unspecified and the corresponding location in \textit{success} is set to false. It is assumed that the results are ordered by state first, then by generator label. The default behavior for this function is to iteratively call \textit{inverseGenToState}.

7.4 Architecture Interface

An example of an instance of the architecture interface for a local cluster using MPI is shown below.

This is the instance of the interface used in the example at the beginning of this section. The details of how communication is performed are hidden in the parent class, \textit{MPIArchitecture}, as including them here would require too much space. The \textit{MPIArchitecture} class provides the appropriate wrappers for the MPI communication functions.

\begin{verbatim}
// An example architecture interface, MPI on a local cluster
class TeraClusterMPI : public MPIArchitecture {
public:
    TeraClusterMPI( int *argc, char ***argv, char *user ) : MPIArchitecture( argc, argv ) { 
        MemorySize = 1000000000ll; // 1 gigabyte of memory per machine
        DiskSize = 400000000000ll; // 400 gigabytes of disk per machine
        NumMachines = 30; // 30 computers in the cluster
        sprintf( CommonDir, "/home/%s/TreeBuilder/", user );
        sprintf( TempDir, "/state/partition1/home/%s/TBS/tmp/", user );
        sprintf( FinalDir, "/state/partition1/home/%s/TBS/final/", user );
    }
};
\end{verbatim}

Here the interface used by the architecture expert is defined in more detail. This interface is used to define how nodes of the cluster pass messages between one another. It also specifies where to store the results of the computation on disk. The architecture interface specifies both variables and functions. Each of these are defined separately below.

\textbf{Variables} Below is a list of the variables specified by the architecture interface. Where those parameters are optional, the default value is listed in parentheses next to the variable name.

\textbf{MemorySize} – The size of available memory in bytes on each node in the computation.

\textbf{DiskSize} – The size of available disk in bytes on each node in the computation.
NumMachines – The number of machines that will be performing the enumeration.

NumFiles (computed using rlimit) – The maximum number of files that can be opened simultaneously by a single process.

NetworkPacketSize (1 Megabyte) – The minimum size of a message required to obtain efficient streaming performance from the network.

DiskPacketSize (100 Kilobytes) – The minimum size of a message required to obtain efficient streaming performance from the disk.

CommonDir (directory of executable) – The directory where the common information for the enumeration will be stored (e.g. list of nodes in the computation and the log files for each node). This directory is assumed to be shared (for example, via NFS). If it’s not, all common data must be distributed before the computation is performed.

TempDir (directory of executable) – The directory in which to put the temp folder, where temporary information will be held locally on each machine. The temp folder is deleted after the enumeration has been performed.

FinalDir (directory of executable) – The directory in which to put the final folder, where the enumeration results will be held locally on each machine.

Functions Below is a list of the functions (C++ methods) defined by the architecture interface. Where the function is optional, the default behavior of that function is specified.

startComm
Description: This function initializes the communication between nodes.

getCommInfo
Input

int &myNumber (Output)
The communication number for the current node.

int &maxNumber (Output)
The number of nodes participating in the computation, typically NumMachines.

Description: This interface uses the MPI concept of rank, where in a computation with max computing nodes, each node is assumed to have a distinct rank n, where 0 ≤ n < max. getCommInfo retrieves the number (n value) for the current node along with the total number of machines in the computation (or max).
endComm
Description: This function closes the communication. It should leave the communication package in the same state as it was before the initial call to startComm.

sendMessage
Input
   const void *message
   The memory location of the message.
   const int length
   The length of the message.
   const int machine
   The destination for the message.
   const int label
   The message label to use, for messages of different types.
Description: This function sends a message of a certain length with a particular label to the target machine. The label must be a non-negative value.

waitMessage
Input
   int &length (Output)
   The length of the message.
   int &machine (Input and Output)
   The machine that sent the message.
   int &label (Input and Output)
   The label of the message.
Description: This function waits for a message from the requested machine with the requested label. It reports the length of the message, the originating machine, and the message label. The requested machine may be −1, indicating any machine. The requested label may be −1, indicating any label.

recvMessage
Input
   void *message (Output)
   The memory location of the message.
   int length (Input and Output)
   The length of the message.
   int machine (Input and Output)
The machine that sent the message.

\[ \text{int label (Input and Output)} \]

The label of the message.

**Description:** This function receives a message into the *message* buffer of a particular *length* from the requested *machine* with the requested *label*. It is assumed that the message is already present, as determined by a prior call to *waitMessage*.

### 7.5 Algorithm Interface

An example of an instance of the algorithm interface for a distributed breadth-first search (where the search is *not* synchronized at each depth) using a memory-resident hash for the end-storage is shown below. This is the instance of the interface used in the example at the beginning of this section. As can be seen, instances of this interface are typically the longest and most difficult to implement. By allowing the re-use of these instances for different applications, programming time is saved.

```cpp
// An example algorithm interface, distributed BFS with a memory-based hash table
class MemoryHashSearch : public AlgorithmInterface {
private:
    // The interfaces, static so the threads can access them
    static ApplicationInterface *prob;
    static ArchitectureInterface *arch;
    static AlgorithmInterface *search;
    // The hash table storing the results
    Hash hash;
    // Information about the machine label and max label
    static int myRank, maxRank;
    // The size of a block to check across the network
    static int blockSize;
    // The message labels
    enum{ MESSAGE_CHECK, MESSAGE_CHECKED, MESSAGE_FIRST_BLOCK };
    // Tell everyone our processor is done
    void syncCall() {...}
    // Wait for all the processors to be done
    void syncWait() {...}
    // Compute the first block and send pieces of it to each computer
    void computeAndDistributeFirstBlock() {...}
public:
    // The constructor, initialize the queries that can be run
    MemoryHashSearch() {
        memset( Queries, '\0', 256 );
        Queries[QueryHasStateChanged] = true;
        Queries[QueryGetStateChanged] = true;
    }
    // The destructor, kill the threads, end the comm, and free the memory
    ~MemoryHashSearch() {...}
    // The run method, predicts the time and starts the computation
    bool run( ApplicationInterface *prob; ArchitectureInterface *arch, int mode ) {
        // No way to resume this, all memory resident
        if( mode == RunRestart ) return false;
        // Find the time and space requirements
```
TimeComp = prob->BranchingFactor * prob->SearchSize / prob->genSpeed;
TimeAccess = 0;
MemoryReq = prob->SearchSize * (2 * prob->StateSize + sizeof(BYTE *));
DiskReq = 0;

// If we're in predict mode, we're done, otherwise, try to run
if( mode == RunPredict )
    return MemoryReq <= arch->MemorySize * arch->numMachines;
if( MemoryReq > arch->MemorySize * arch->numMachines ) return false;

// Find the block size to use, enough to get streaming network performance
blockSize = arch->NumMachines * arch->NetworkPacketSize / prob->StateSize;

// Initialize interfaces and communications and start the hash checking thread
this->search = this; this->prob = prob; this->arch = arch;
arch->startComm( );
arch->getCommInfo( myRank, maxRank );
pthread_t checkMan;
if( pthread_create( &checkMan, NULL, runChecker, NULL )!=0 ) assert( 0 );

// Compute and distribute the first block, and start processing
if( myRank==0 ) computeAndDistributeFirstBlock( );

// Wait for everything to finish, and start the query thread
syncWait( );

// Thread to discover new states
static void *runDiscovery( void *trash ) {
    int length, machine, label;
    Queue<BYTE> open
    BYTE block[blockSize * prob->StateSize], result[blockSize];
    BYTE *end = block + blockSize * prob->StateSize;

    // Receive the first block, report we're done and return if it's empty
    arch->waitMessage( length, machine = 0, label = MESSAGE_FIRST_BLOCK );
    BYTE *temp = malloc( length );
arch->recvMessage( temp, length, machine, label );
if( length==0 ) {
    syncCall();
    return;
}

// Initialize the head and the queue
for( int count = 0; count < prob->StateSize; count++ ) {
    head[count] = temp[count];
}
for( int count = prob->StateSize; count < length; count++ ) {
    open.push_back( &temp[count] );
}
free( temp );

// While the open queue has data, or the head still needs to be processed
for( int gen = 0; !open.empty() || gen < prob->NumGenerators; ) {
    // Find the next block and sort it
    int length = search->computeBlock( block, end, head, open, gen );
    sortBlockByStateOwner( block, length );

    // Distribute it among the nodes for checking and receive results
    BYTE *startData = block, endData;
    for( int comp = 0; comp < maxRank; comp++ ) {
        for( endData = startData;
            endData < end && stateOwner( endData ) == comp;
            endData += prob->StateSize )
            if( endData > startData ) {
                int messLength, messLabel;
                arch->sendMessage( startData, endData - startData, comp, MESSAGE_CHECK );
                arch->waitMessage( messLength, comp, messLabel = MESSAGE_CHECKED );
                arch->recvMessage( result + (startData - block) / prob->StateSize, messLength, comp, messLabel );
            }
    }

    // Remove the duplicates and update the head if necessary
    length = stripDuplicates( block, result, length );
    for( int count = 0; count < length * prob->StateSize; length++ ) {
        open.push_back( block[count] );
    }
    if( gen == prob->NumGenerators && !open.empty() ) {
        for( int count = 0; count < prob->StateSize; count++ ) {
            head[count] = open.pop_front();
        }
    }
}

// Report we're done
syncCall();

// Thread to check if states exist
static void *runChecker( void *trash ) {
    // Declare the local variables
    int length, machine, label;
    vector <BYTE> buffer;
    // Initialize the hash with the key size and value size
    search->hash.init( prob->KeySize, prob->StateSize );

    // Receive for blocks to check
    while(1) {
        // Wait for a check message, resize the buffer appropriately, and receive
        arch->waitMessage( length, machine = -1, label = MESSAGE_CHECK );
        buffer.resize( length );
        arch->recvMessage( &(buffer[0]), length, machine, label );

        // Check each state, adding it to the hash if possible, send back the results
        for( int count = 0; count < buffer.length(); count += prob->StateSize ) {
            if( search->hash.add( &buffer[count] ) ) buffer[count/prob->StateSize] = 1;
        }
    }
}
else buffer[count/prob->StateSize] = 0;
arch->sendMessage( &(buffer[0]), length/prob->StateSize, machine, MESSAGE_CHECKED );
}
}

static void *runQuery( void *trash ) {
  // Declare the local variables
  int length, machine, label, query;
  BYTE result[prob->StateSize];

  while( 1 ) {
    // Wait for a query message and get the query and the state
    arch->waitMessage( length, machine = -1, label = MESSAGE_QUERY );
    arch->recvMessage( &query, length, machine, label );
    arch->waitMessage( length, machine, label );
    arch->recvMessage( result, length, machine, label );

    // Process the query
    switch( query ) {
      case QueryHasState:
        result[0] = search->hash.has( result );
        arch->sendMessage( result, 1, machine, MESSAGE_QUERIED );
        break;
      case QueryGetState:
        assert( search->hash.get( result, result ) );
        arch->sendMessage( result, prob->StateSize, machine, MESSAGE_QUERIED );
        break;
      }
  }
};

Here the interface used by the algorithm expert is defined in more detail. This interface is used to determine the time and resources required by the search. It also runs the search and provides the application expert with functions to query the search space. The algorithm interface specifies both variables and functions. Each of these are defined separately below.

Queries   The algorithm instance is allowed to implement any number of queries. The first 256 of these queries (numbered 0-255) are reserved for queries defined in the full API documentation, some examples of these are shown in the architecture instance code above (QueryHasState and QueryGetState). Queries above number 255 are specific to the algorithm interface. These queries may only be called once the search has been performed.

Variables   Below is a list of the variables specified by the algorithm interface. Where those parameters are optional, the default value is listed in parentheses next to the variable name.

Queries   – An array of booleans of size 256 indicating which queries this search technique is capable of performing once the search is complete.

TimeComp   – An output variable set when the run function is called. This is the number of seconds
required per machine to perform the generator applications in the search.

**TimeAccess** – An output variable set when the run function is called. This is the number of seconds required per machine to perform the disk access in the search.

**MemoryReq** – An output variable set when the run function is called. This is the number of bytes of memory per machine required by the search.

**DiskReq** – An output variable set when the run function is called. This is the number of bytes of disk per machine required by the search.

**Functions** Below is a list of the functions (C++ methods) defined by the algorithm interface. Where the function is optional, the default behavior of that function is specified.

**run**

**Input**

```cpp
custom ApplicationInterface *probInt
const ArchitectureInterface *archInt
const int runMode
```

The problem on which to run the enumeration.

The architecture on which the enumeration is performed.

The mode to run in, either RunRestart, RunEnumerate, or RunPredict.

**Output**

```cpp
bool success
```

Whether or not the search can be/was performed.

*Description:* This function runs the enumeration. If `runMode` is RunEnumerate, the enumeration is performed, and resumed from a checkpoint if possible. If `runMode` is RunRestart, the enumeration is performed from the start, even if a checkpoint exists. If `runMode` is RunPredict, the information for the enumeration is generated without running the enumeration itself. This function returns true if the enumeration has been performed successfully or could be performed successfully (in the case of RunPredict).

**query**

**Input**

```cpp
const int query
```

The query to be performed.
The data for the query.

`BYTE *result` (Output)

The location at which to store the query result.

**Output**

`bool success`

Whether or not the query was performed successfully.

*Description:* This function performs the requested *query* given the provided query *data*, and stores the result in the space designated by *result*. This function returns true if the query was performed successfully.

### 7.6 The Path Utility

It is important to be able to retrieve the necessary information about the values in the enumeration, namely which generators lead to those states seen during the enumeration. To this end, a path utility is defined to allow both the application expert and the algorithm expert, who typically both create private copies of this utility, to deal with strings of generators in a logical way. Assuming the algorithm expert is satisfied with the compactness of the representation and its efficiency, this utility has already been defined and works over all algorithm interface implementations. The functionality for this utility is listed below. It is assumed that *Step* corresponds to a generator label, denoting the generator that was applied to reach the next state in the *Path*.

**Initialize**

**Input**

`const int numGens`

The number of edge generators used in the problem.

*Description:* This function initializes the instance of the *PathInterface* when provided with the total number of generators, *numGens*.

**getPathSize**

**Input**

`const int max`

The maximum length (number of steps) that the path can be.

**Output**

`int bytes`

The number of bytes required for the path.
Description: This function returns the total number of bytes required to store a path of some max length. This number can then be used to determine how much memory to allocate for either a single path or an array of paths.

**createPaths**

Input

Path *paths

A pointer to the location at which to store the paths.

const int size

The maximum path size, obtained by a call to getPathSize.

int count

The total number of paths to create.

Description: When provided with a sufficient memory buffer, this function produces count paths of some maximum size. It is assumed the memory to store these paths is preallocated by the user, and is available at the location pointed to by paths. Initially the lengths of all the created paths are set to zero.

**getStep**

Input

const Path *path

The path for which to get the particular step.

const int step

The number of the step to get.

Output

Step label

The label of the generator for that step in the path.

Description: This function gets the generator label of the specified step in the path.

**setStep**

Input

Path *path

The path in which to set the particular step.

const int step

The number of the step to set.

const Step label
The label of the generator for that step in the path.

**Description:** This function sets the generator *label* of the specified *step* in the *path*.

### getLength

**Input**

```c
const Path *path
```

The path for which to determine the length.

**Output**

```c
int length
```

The length of the path, or the number of steps in the path.

**Description:** This function gets the *length* of the *path*, the total number of steps in the path.

### setLength

**Input**

```c
Path *path
const int length
```

The path for which to set the length.

**Description:** This function sets the *length* of the *path*, the total number of steps in the path.

## 7.7 Implementation

Currently the *path* utility has been created and should be suitable for most algorithm and application instances. In addition to the required functionality it also provides an append operation for adding a step to a path as well as a comparison function, which compares paths based on tree order (left-most first).

Three instances of *Algorithm Interfaces* for tiered duplicate detection have been defined. These include the use of full values and paths, the use of full values only, and the use of compressed values (which require storing paths).

A generic instance of the *Architecture Interface* with the use of MPI (that requires only an MPI subset that includes the primary point-to-point functions) has also been defined. For each computer cluster that our lab currently has access to, we have created a new instance of this computer architecture that specifies *MemorySize, DiskSize, NumMachines, CommonDir, TempDir*, and *FinalDir*. 
Finally, instances of the Application Interface for all the applications discussed in Section 8.3 have been created. These interfaces can be initiated using either standard vector-matrix multiplication or a technique called greasing, which can speed up vector-matrix multiplication by storing pre-computed multiplication tables for some applications. Beyond this, the next section (Section 9) discusses the implementation of Application Interfaces in two other application areas, puzzle search and formal verification.

8 Experimental Evidence of the Validity of the Enumeration Analysis:

Enumeration Applications in Group Theory

This section provides the experimental evidence that the analysis provided in Section 5 is valid by examining enumeration applications in computational group theory. The parameters for each of these applications are given and a predicted run-time is computed for all applicable enumeration techniques. The enumeration is then run, and an experimental time is provided. What is seen is that the predicted run-times are always accurate to within ±50%, and typically the accuracy is within ±25%.

8.1 Computational Group Theory Background

The 26 sporadic simple groups form a natural ladder of challenge problems. The finite simple groups are the building blocks from which all finite groups can be built. In a famous mathematical effort over 20 years, a project on the classification of finite simple groups showed that all groups can be built from several parametrized families of finite simple groups, along with the 26 sporadic simple groups [3, 30]. Proofs of the existence of the 26 sporadic simple groups required their construction — often on a computer by ad hoc methods. There is current interest in a uniform computer construction using group membership techniques.

For one type of group, small base groups, many useful group membership techniques exist. Fortunately, sporadic simple groups are an example of small base groups. This gives us access to small base group membership techniques when working on sporadic simple groups. Such techniques have previously only been demonstrated for in-core computation (in main memory). One goal of this thesis is to develop new algorithms for out-of-core computation.
In order to understand the problem of computing with small base groups, we must first define what a base of a group is. In addition, we must also look at existing group membership techniques for small base groups.

This section is meant to serve as a brief overview of the field of group theory, with a focus on small-base groups. For a more detailed discussion, a book in group theory such as [9, 10, 23] is recommended.

**Group**  
A *group* is a set $G$ under some operation $\otimes$ with the following axioms:

- **Associativity** $\forall a, b, c \in G, (a \otimes b) \otimes c = a \otimes (b \otimes c)$
- **Identity Element** $\exists e \in G, \forall a \in G a \otimes e = e \otimes a = a$
- **Inverse Element** $\forall a \in G, \exists a^{-1} \in G, a \otimes a^{-1} = a^{-1} \otimes a = e$
- **Closure** $\forall a, b \in G, b \otimes a, a \otimes b \in G$

Typically $e$ is used to represent the identity element of $G$. A subgroup of $G$, $H \leq G$, is a subset of the elements of $G$ that form a group under the $G$’s group operation, $\otimes$. A proper subgroup of $G$, $H < G$ is a subgroup where $G \neq H$.

**Permutation Groups**  
Here we lay out the basic definitions for a permutation group.

- A *permutation* $P$ of degree $N$ is a rearrangement of the ordered list containing the numbers $1 \ldots N$.

- The *permutation domain* of degree $N$ is the set of all possible permutations of degree $N$. There are $N!$ such permutations.

- *Permutation multiplication* of two permutations of degree $N$, $P[Q]$, is the rearrangement of the numbers in the permutation $Q$ according to the order specified by $P$. The result is another permutation of degree $N$.

- A *permutation group* is a group in some permutation domain where the group operation $\otimes$ is permutation multiplication.
**Base of a Group** For a permutation group in the permutation domain of degree \(N\), a *base of the group* is a set of points in \(N\) that uniquely distinguish each permutation in the group. More formally, for \(i \in \{1 \ldots N\}\) and \(g \in G\), let \(i^g\) denote the action of the permutation \(g\) on the point \(i\). The set \(B \subset \{1 \ldots N\}\) is a base of the group if

\[
\forall g, h \in G, (\forall i \in B, i^g = i^h) \iff g = h.
\]

The constant \(m = |B|\) is known as the base size of the group.

**Cosets** A coset is defined for two groups \(H \leq G\). Where \(g \in G\), \(Hg = \{hg : h \in H\}\) is a right coset of \(H\) in \(G\). All cosets in this work will be right cosets. We use \(G/H\) to represent the set of all right cosets of \(H\) in \(G\). One useful property of cosets is that each element in \(G\) belongs to one and only one coset of \(G/H\), therefore \(G/H\) is a partitioning of all the elements of \(G\). We shall see that enumeration of all of the cosets of the form \(G/H\) for some \(H\) is a critical step for all permutation algorithms on \(G\).

**Point Stabilizer Subgroup Chain** Though concise, the previous definition of the base of a group is commonly replaced by one using a *point stabilizer subgroup chain*. Point stabilizer subgroup chains lend themselves more easily to a recursive algorithm for creating a group representation that can easily answer questions of group membership and generate random group elements. For ease of exposition, we will always assume the special case where base \(B\) is of the form \(B = \{1, 2, \ldots, m\}\).

Define the *point stabilizer subgroup*

\[
G^{(i)} = \{g : g \in G, \forall j < i, j^g = j\},
\]

sometimes called “\(G\) move \(i\)”, where only for \(j \geq i\) is it possible that \(g \in G^{(i)}\) and \(j^g \neq j\). Note that this yields a *point stabilizer subgroup chain*:

\[
G = G^{(1)} \geq G^{(2)} \geq \cdots \geq G^{(m)} \geq G^{(m+1)} = \cdots = G^{(n)} = \{e\}.
\]

**Transversals** Each point stabilizer subgroup can be broken up into cosets. The cosets \(G^{(i)}/G^{(i+1)}\) provide a partitioning of \(G^{(i)}\). One can quickly determine which coset a new element of \(G^{(i+1)}\) is in. Note that \(\forall g \in G, h \in G^{(i+1)}, i^{hg} = i^g\) (where \(i^{hg} = (i^h)^g\)). So, for any \(h \in G^{(i)}g\), \(i^h\) is a unique identifier, of \(G^{(i)}g\), and \(h\) can be used as a representative of \(G^{(i)}g\). In other words,

\[
\forall h_1, h_2 \in G^{(i)}, \quad i^{h_1} = i^{h_2} \iff G^{(i)}h_1 = G^{(i)}h_2.
\]
A transversal of $G^{(i)}/G^{(i+1)}$, $T^{(i)}$ is defined as a set of representatives of the cosets of $G^{(i+1)}$ in $G^{(i)}$, hence $T^{(i)} \in G^{(i)}$. So,

$$|T^{(i)}| = |G^{(i)}/G^{(i+1)}|.$$ 

Further, a transversal $T^{(i)}$ satisfies

$$\forall j \in \Omega, \quad i^g = j \iff \exists t, T^{(i)} \cap G^{(i+1)}g = \{t\} \text{ with } i^g = i^t.$$ 

A group representation using transversals can be used to store any group and answer questions of group membership as well as generating random elements of the group. A small base groups is a group where $\exists B$ s.t. $m \ll n$. With small base groups, using transversals leads to a compact representation.

**Example of Transversals** To illustrate the above definitions, let $G$ be the group corresponding to the symmetries of a pentagon, where the edges are labeled consecutively from 1 to 5. We shall use an array representation for permutations, where $[k_1, k_2, \ldots, k_n]$ denotes the permutation $g$ such that $1^g = k_1, 2^g = k_2, \ldots, n^g = k_n$. Once two adjacent vertices are chosen, the rest of the vertices are locked in place, meaning $G$ has a base size of 2.

This can also be seen by looking at the point stabilizer subgroup chain:

$$G^{(1)} = \{[1, 2, 3, 4, 5], [1, 5, 4, 3, 2], \ldots, [5, 4, 3, 2, 1]\} = G$$
$$G^{(2)} = \{[1, 2, 3, 4, 5], [1, 5, 4, 3, 2]\}$$
$$G^{(3)} = G^{(4)} = G^{(5)} = \{[1, 2, 3, 4, 5]\} = \{e\}$$

From this we can choose our coset representatives for the transversals of $G$:

$$T^{(1)} = \{[1, 2, 3, 4, 5], [2, 3, 4, 5, 1], [3, 4, 5, 1, 2], [4, 5, 1, 2, 3], [5, 1, 2, 3, 4]\}$$
$$T^{(2)} = \{[1, 2, 3, 4, 5], [1, 5, 4, 3, 2]\}$$
$$T^{(3)} = T^{(4)} = T^{(5)} = \{[1, 2, 3, 4, 5]\} = \{e\}$$

**Generation of Random Elements** Let $\text{random}(G)$ generate a random element in $G$. Let $T_x^{(i)}$ denote the coset representative $g$ where $t^{G^{(i+1)}g} = x$. Algorithm 9 generates a random element in $G(i)$ given $T^{(i-1)}, \ldots, T^{(1)}$.

The loop invariant arises because at the end of each pass through the loop, we have multiplied $res \in G^{(x-1)}$ by $(T_i^{(x)})^{-1}$, where $x^{res} = t$ and $t^{G^{(i+1)}-1} = x$, and stored the result back in $res$. This means $res^x = x$, or $x$ is “fixed”, at the end of the loop.
Algorithm 9: randomElement

Input: $i, T^{(i-1)}, \ldots, T^{(1)}$
Output: random element in $G^{(i)}$

$res = \text{random}(G);$ 
for $x = 1 \ldots i - 1$ do
  $t = x^{res};$
  $res = res \cdot (T^{(x)}_t)^{-1};$
// Invariant: $res \in G^{(x)}$
return $res$

Membership Test  Modifying Algorithm 9 slightly gives us Algorithm 10. This algorithm decides if permutation $h$ is in $G$, where the base of $G$ is $m$. At the end of the loop, we have a mapping between $h$ and $res$ consisting of the inverses of the transversal elements applied in the loop. If $res = e$, then we have a mapping from $e$, a group member, to $h$. If $res \neq e$, then $res$ has the same signatures as $e$ and cannot be in the group, this implies $h$ also cannot be in the group. This algorithm is also known as the Schreier-Sims group membership algorithm.

Algorithm 10: isMember

Input: $h, T^{(m-1)}, \ldots, T^{(1)}$
Output: true if $h \in G$

$res = h;$
for $x = 1 \ldots m - 1$ do
  $t = x^{res};$
  if $T^{(x)}_t$ does not exist then return false;
  $res = res \cdot (T^{(x)}_t)^{-1};$
return $res = e$;

Application of Algorithms  Given a random element in $G^{(1)} = G, g = [3,2,1,5,4]$ for example, a random element in $G^{(2)}$ can be computed:

$$[3, 2, 1, 5, 4] \cdot (T^{(1)}_3)^{-1} = [3, 2, 1, 5, 4] \cdot [3, 4, 5, 1, 2]^{-1} = [3, 2, 1, 5, 4] \cdot [4, 5, 1, 2, 3] = [1, 5, 4, 3, 2]$$

Given the same random element, the above calculation can be extended to find a random element
in \( G^{(3)} \):

\[
[1, 5, 4, 3, 2] \cdot (T^{(2)}_s)^{-1} = [1, 5, 4, 3, 2] \cdot [1, 5, 4, 3, 2]^{-1}
\]
\[
= [1, 5, 4, 3, 2] \cdot [1, 5, 4, 3, 2]
\]
\[
= [1, 2, 3, 4, 5]
\]

All “random” elements found in \( G^{(3)} \) will be the identity since the group we are working with has a base of 3.

The above derivation also proves \( g \in G \) by providing a mapping, \( g = (T^{(1)}_s)^{-1} \cdot (T^{(2)}_s)^{-1} \), from \( e \) to \( g \).

**Schreier Trees** Algorithm 11 discovers a transversal given all previous transversals. It takes advantage of the time spent to discover a single random element in \( G^{(i)} \) by applying it to all the known elements of \( T^{(i)} \).

A randomized version of the deterministic algorithm is used because the deterministic version produces many Schreier generators \((n \times \text{size of the generating set})\). This makes its implementation difficult and impractical for reasons of time and space.

Though Algorithm 11 is randomized, only a small number of passes, \( p \), are required to obtain a high level of confidence that the entire transversal has been discovered.

Consider the transversal \( T \), where the \( |T| \) is the size of \( T \). At any point in Algorithm 11 when \( c \) transversal elements have been discovered, the probability of seeing no new transversal elements in a pass is only \((c/|T|)^c\). Where \( c \) ranges from \( 1 \ldots |T| - 1 \), this equation has a maximum at \( c = |T| - 1 \). This implies a probability of seeing no new points of at most \(((|T| - 1)/|T|)^{|T| - 1}\). Even with \( |T| = 2 \), this probability is at worst 0.5. As \( |T| \) grows, this probability approaches the number \( e^{-1} = 0.3678 \).
This allows for a reasonably small number of passes in which no new elements are seen to obtain a high degree of confidence that there are no new elements to be seen, or that \( c = |T| \).

Though this algorithm is efficient, it performs a permutation multiplication as well as stores a full permutation for each element of the transversal. However, both of these requirements can be eliminated by switching the representation for the transversal from a set to a Schreier tree.

A Schreier tree is a tree in which nodes correspond to transversal elements and edges to applications of random elements. In the actual representation of the Schreier tree for \( T^{(i)} \), each nodes stores only the image of \( i \) for its transversal element, and each edge stores only a reference to the random permutation to which it corresponds. More formally, where \( r \) is the root of the Schreier tree:

\[
\forall n \in T^{(i)} \text{ where } r \overset{a_1}{\Rightarrow} n_1 \ldots \overset{a_k}{\Rightarrow} n_k = n, n = (r \times a_1 \times \ldots \times a_k)^{i}.
\]

The advantage of using a Schreier tree is that each element of the transversal requires storing only an integer as well as a reference to the random permutation that got us there instead of a full permutation. The disadvantage is that one does not have access to the actual transversal element (permutation) when used later. However, for our purposes, the operation performed on transversal elements by the computation is to determine where they map individual points, this can be done easily by performing a trace of the transversal element, as shown in Algorithm 12, where the initial generator is \( e \).

**Algorithm 12: tracePoint**

- **Input:** origin, \( T^{(i)}_x \), generator
- **Output:** origin\( T^{(i)}_x \), the mapping of the origin point through \( T^{(i)}_x \)
- if \( T^{(i)}_x = T^{(i)}_1 \) then return \( \text{gen} \);
- \( \text{origin} = \text{tracePoint}(\text{origin}, \text{parent}(T^{(i)}_x), \text{parentGenerator}(T^{(i)}_x)) \);
- return \( \text{origin} \text{generator} \);

Performing a trace has time proportional to the depth of the Schreier tree, whereas if the actual permutation was stored, the computation could be done in a single operation.

**Shallow Schreier Trees**  Shallow Schreier trees were developed by Cooperman and Finkelstein [15, 14]. The primary result on shallow Schreier tree follows.

**Theorem 1** (Theorem 3.5 (paraphrased)[14]). For \( \delta \geq 1 \), let

\[
d = \lfloor 20\delta \log_2 |G^{(i)} / G^{(i+1)}| \rfloor.
\]

Then \( d \) random group elements suffice for a Monte Carlo algorithm
to build a new Schreier tree for $G^{(i)}/G^{(i+1)}$ of depth $d$. The probability of error is less than $|G^{(i)}/G^{(i+1)}|^{-\delta}$.

Intuitively, the theorem says that for a transversal of size $T$, $20 \log_2 T$ random elements suffice to build a Schreier tree. Further, the tree will have depth at most $20 \log_2 T$. The probability of failing to construct the tree within the stated bounds is less than $1/T$. The proof of the theorem uses a very conservative algorithm and in practice, the constant of 20 is often too high.

**Generating Sufficiently Random Elements** Typically random elements are generated by using a random sequence in the generators, or the base permutations capable of generating the entire group from the identity. There is some concern over the “quality” of these random elements when applying them to a Schreier tree. For example, if looking for a random element in $G^{(2)}$ and our random element in $h \in G$, where $1^h = i = T^{(1)}_1$, then the random element in $G^{(2)}$ will be $hh^{-1} = e$, and application throughout the tree will not yield any new points. While this can be easily fixed through the use of longer random elements, shorter random elements are preferred as they result in fewer permutation accesses during a trace.

**Matrix Groups** Though the above algorithms deal with permutation groups, there is also a rich background using these algorithms on matrix groups over finite fields.

We can view a matrix group as a permutation group. Each matrix can be viewed as a permutation of vectors of a finite dimension over a finite field in the corresponding vector space. Here, a vector represents a point, and a matrix an element of the group. Instead of actions of permutations on points, vector-matrix multiplication is used for the action of matrices on vectors. The notation of $v^M$ is changed to $vM$. Solutions in either space are interchangeable because methods for converting between the two representations exist.

**Condensation** Condensation is a critical technique in the Modular Atlas project [35]. The Modular Atlas project does for modular representations of mathematical groups what the Atlas project does for ordinary representations [12, 35] and a more complete description of condensation is provided here.

The orbits of a group contain useful information about the group itself in many cases. However, even after an orbit has been discovered, it is sometimes impractical to analyze it due to its large size. Condensation, or specifically direct condensation, is the process of reducing an orbit with millions or even billions of vectors to an incidence matrix with dimension in the thousands. This representation
retains enough of the original structure of the orbit to allow for useful analysis, while condensing the representation to be more manageable for that analysis.

We consider a matrix group $G$ and look at the orbit of some vector $v$ in that group, $O = v^G$. Typically, $O$ will have millions or even billions of vectors. Consider $K \leq G$. $O$ can be expressed as a disjoint union of $K$-orbits $S = \{S_1, \ldots, S_k\}$. Typically $k$ will be on the order of thousands.

The set $S$ is still just as large as the original orbit $O$, however, so instead, we consider a condensation element $c \in G$. Applying $c$ to the elements of $e \in s$, for some $s \in S$, produces $e \times c \in s'$, for some $s' \in S$. The mappings between these $K$-orbits provide useful information about the structure of the group itself. In order to make this information manageable, instead of storing each mapping, an incidence, or orbit counting, matrix for $c$, $C^c$, is formed of dimension $k \times k$. This matrix stores:

$$\forall m, n \in \{1, \ldots, k\}, C^c_{m,n} = |\{e : e \in S_m, e \times c \in S_n\}|$$

This new data structure provides a compact representation of the space, requiring storage for the matrix $C^c$, proportional to $k \times k \times \log_2(|O|)$ rather than space for the orbit $O$, proportional to $|O| \times |v|$.

Typically, direct condensation proceeds in three phases:

1. Orbit expansion, discovery of $O$;
2. Suborbit partitioning, discovery of $S$; and
3. Orbit counting, discovery of $C$

### 8.2 Computational Group Theory Previous Work

The need for analysis of very high degree groups has a long history. Sims presented a specialized construction of the Baby Monster in 1980 [66]. In 1998, in a tour de force, a specialized construction of the Monster was produced by Linton, Parker, Walsh and Wilson [47]. This was later elaborated on by Holmes and Wilson [34]. However, in each of those cases, specialized techniques were needed due to the limitations of CPU, RAM and disk.

Cooperman et al. [16, 17] produced a permutation representation of degree 9,606,125 for Lyons’ group acting on a conjugacy class of subgroups of order three. The representation was found using the matrix representation of Wilson [74] as a starting point. The representation was verified in a Monte Carlo manner by computing its order through ad hoc methods. Gollan then began his work on a revised existence proof of Lyons’ group [26, 27, 28]. As one part of that work, he deterministically
verified the order of the permutation representation through the “double coset trick”, an independent rediscovery of an unpublished Verify algorithm of Sims.

Later, a coset enumeration of Lyons’ group yielded a permutation action on 8,835,156 points, based on Sims’ original unpublished presentation. The coset enumeration was executed in two different ways. It was carried out as a parallel enumeration by Cooperman and Havas [18] (described therein as part of the future work). It was also demonstrated as a sequential coset enumeration by Havas and Sims [32]. That presentation was verified as producing Lyons’ group by Gollan and Havas [29].

Later work produced large permutation representations for Thompson’s group acting on 143,127,000 and for Janko’s group $J_4$ acting on 173,067,389 points. A permutation representation was implicit in the condensation computation for Thompson’s group of Cooperman et al. [19, 22]. Weller [72] carried out a direct computation at approximately the same time. Havas et al. [33] produced a presentation for Thompson’s group, and also a permutation representation thereof through coset enumeration. Weller [70, 71] did the same for Janko’s group $J_4$, using some of the hashing techniques of [16, 17] and the double coset trick of [26, 28]. That work was used in a revised existence proof for Janko’s group, $J_4$ [20].

Finally, the matrix recognition project [45, 46] expects to reduce certain matrix group recognition problems to the base case of the simple groups, which then require other methods for analysis. The methods of this paper provide a useful alternative in this setting, since they allow the well-developed computational methods for permutation groups to be applied. Note, for example, that any group with a representation in $GL(30, 2)$ (the group of matrices of dimension 30 over the finite field with two elements) has a permutation representation on at most $2^{30}$, or approximately one billion points.

Condensation was originally formalized in [69] to construct irreducible representations of a group and to analyze existing representations. Its theoretical underpinnings as particular Schur functors are described in [31]. Since its invention it has been used in a number of settings. For more details we refer the reader, e.g. to the overviews in [54, 55]. In particular the direct condense technique examined in this thesis has raised some interest in recent years: after its invention, it has been implemented as parallelized versions in [22, 48], has been extended using subgroup structures in [55, 56, 57], and a different disk-based version has been described in [63]. Separately, the landmark-based condensation technique was introduced for matrix representations in [17].
8.3 Computational Group Theory Applications

While the use of Schreier trees largely reduces the space required to store transversals for small base groups, for very large groups, even this space reduction is not enough. In many cases, the first transversal, also known as an orbit, does not fit easily in memory. In this section we look at applications that required discovering a large first transversal. Four such applications are considered.

These applications include:

<table>
<thead>
<tr>
<th>Group Name</th>
<th>Orbit Size</th>
<th>Section</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thompson Group</td>
<td>$1.4 \times 10^8$</td>
<td>8.3.1</td>
<td>Formed Schreier tree group representation</td>
</tr>
<tr>
<td>Baby Monster</td>
<td>$1.4 \times 10^{10}$</td>
<td>8.3.3</td>
<td>Orbit enumeration only</td>
</tr>
<tr>
<td>Fischer, $F_{23}$</td>
<td>$1.2 \times 10^{10}$</td>
<td>8.3.2</td>
<td>Condensation using landmark-based approach</td>
</tr>
<tr>
<td>Janko, $J_4$</td>
<td>$1.3 \times 10^{11}$</td>
<td>8.3.4</td>
<td>Condensation using disk-based approach</td>
</tr>
</tbody>
</table>

Typically, rather than performing a search over random elements for the initial orbit enumeration, a breadth-first search over the generators can be used instead. This was done in all cases in order to speed up the computation. The table below shows, for each of these applications, the total size of the state space, the technique from Section 3 that was used to enumerate it. The predicted time for that technique using the formulas derived in Section 6, the actual time observed during experimentation, and the percent difference between the two.

<table>
<thead>
<tr>
<th>Group Name</th>
<th>Space Size</th>
<th>Enumeration Technique</th>
<th>Predicted Time</th>
<th>Actual Time</th>
<th>% Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thompson Group</td>
<td>3 GB</td>
<td>Perfect Hash BFS</td>
<td>12.7 minutes</td>
<td>14.5 minutes</td>
<td>+13.9%</td>
</tr>
<tr>
<td>Baby Monster</td>
<td>1 TB</td>
<td>Tiered DD</td>
<td>13.4 hours</td>
<td>17.2 hours</td>
<td>+28.9%</td>
</tr>
<tr>
<td>Fischer, $F_{23}$</td>
<td>7 TB</td>
<td>Tiered DD</td>
<td>7.8 days</td>
<td>6.0 days</td>
<td>−22.9%</td>
</tr>
<tr>
<td>Janko, $J_4$</td>
<td>8 TB</td>
<td>Sorting-Based DDD</td>
<td>13.2 days</td>
<td>12.1 days</td>
<td>−9.2%</td>
</tr>
</tbody>
</table>

As can be seen, all results fall well within an error range of ±50% and are typically much closer to ±25%.

8.3.1 The Thompson Group

The Thompson group is a small base group of size $9.0 \times 10^{16}$ with generating permutations that act on $1.4 \times 10^8$ points. Assuming the use of 4 byte integers, this means that each permutation requires approximately 500 MB of storage. Even ignoring the constant factor of 20, the first Schreier tree will use a minimum of $\log(1.4 \times 10^8) = 38$ random permutations. This means the tree will use 19 GB of memory. This is not practical even on commodity shared memory machines. While disk could be
used, access to this will necessarily be random (traces access random permutation locations), and therefore very slow.

Fortunately, this storage of random permutations can be avoided. Instead of storing full permutations for the random elements, the string of generators that produced these permutations will be stored instead. This leads to more permutation accesses during a trace, but also requires only the generators and their inverses to be stored. With this, the entire computation uses only 2 GB of memory and is feasible on today’s commodity machine.

The largest and most time consuming portion of the computation is the discovery of the first Schreier tree. For the Thompson group, a perfect hash is available that fits in memory, making duplicate detection a simple matter. In addition to a being fast, the breadth-first search extends naturally to disk. Only the hash table, which is accessed in a random manner, must be kept in memory. Discovery of the first Schreier tree required only 15 minutes and 2 gigabytes of both memory and disk on a single machine.

Once the first Schreier tree for the Thompson Group has been computed, the remaining Schreier trees, which are considerably smaller, can be computed. Their computation requires additional work during the traces of elements through the first Schreier tree as well. The computation of the remaining Schreier trees required only 20 minutes and virtually no memory or disk when compared to the generation of the first Schreier tree.

### 8.3.2 The Fischer Group $Fi_{23}$

The Fischer group $Fi_{23}$ is a small base group of size $4.0 \times 10^{18}$, with two generating permutations that act on $1.2 \times 10^{10}$ points. As with the Baby Monster Group (see Section 8.3.3), storing even the generators as permutations would be impractical. Matrix generators of dimension 782 over $GF(2)$ are used instead.

After “greasing”, which requires an additional 25 MB of storage per generator, a vector-matrix multiplication for the Fischer group $Fi_{23}$ can be performed in $2.0 \times 10^{-5}$s. This leads to a predicted time of 4.2 hours to compute the first orbit on a 56 node cluster. As with the Baby Monster, because determining if a vector is a duplicate remains a hard problem, tiered duplicate detection is used to obtain an actual time very close to the predicted one.

Though somewhat smaller than the Baby Monster, this computation is novel in that additional post-processing of the data was performed after the enumeration to arrive at the desired solution. Once the enumeration had been performed, a condensation, which partitioned the orbit into
6486 suborbits and then computed the mappings between those suborbits, was performed. This was done by a *landmark-based condensation technique*.

To discover the suborbits, a fraction of the search-space that fit in distributed memory were set as the *landmarks*. It was assumed that there existed at least one such landmark for each suborbit. By scanning through the results of the enumeration on disk and expanding unseen landmarks into their full suborbits, the suborbits themselves can be identified. When a suborbit is seen, some canonical element is determined (decided simply by sorting the elements) to represent that suborbit. In addition, the landmarks in that suborbit are marked as “seen” so that suborbit is not recomputed later. The predicted time for this partitioning, just as for the breadth-first search, is 4.2 hours on a 56 node cluster, as a breadth-first search must be performed over each suborbit, these suborbits combine to form entire search space.

Once all the suborbits were found, and their associated landmarks were identified, the condensation matrix was computed. This was performed by recomputing each suborbit, one at a time. The condensation element was then applied to each state in that suborbit. For non-landmark states, the resulting suborbit was discovered by finding the closest landmark state and determining its suborbit. The predicted time for this counting requires both the expanding of each suborbit, 4.2 hours on a 56 node cluster, and a trace to the closest landmark. This trace has time proportional to the ratio of landmarks chosen, 7 for this problem. This resulted in a predicted time of 11.5 hours on a 56 node cluster for the computation.

### 8.3.3 The Baby Monster Group

The Baby Monster group is a small base group of size $4.0 \times 10^{33}$, with generating permutations that act on $1.4 \times 10^{10}$ points. For this group, storing just the two generators as permutations would require 70 GB each. Since these permutations will be accessed randomly, storing them on disk is not practical because random disk access would make the run-time infeasible.

Fortunately, a matrix representation for the group can be used in place of a permutation representation. The matrices for the Baby Monster are of dimension 4370 over $GF(2)$. Each matrix generator requires only approximately 2.5 MB of memory. This can easily fit in RAM on a typical commodity machine. This space reduction comes at a cost, however. Each node in the Schreier tree now must store a vector of length 4370 over $GF(2)$ rather than an individual integer value. This means the first Schreier tree will require 7.6 TB of space. While this is too large for an individual commodity machine, a small cluster of 32 commodity machines and their disks can be used.
Use of a cluster also makes the time requirements for the computation of the first Schreier tree feasible. Following an edge in the Schreier tree will require a vector-matrix multiplication. There is a fast method for this multiplication known as “greasing” that involves pre-computation of a multiplication table for the matrix to be multiplied. Given an additional 128 MB of storage per matrix generator, the time to do a vector-matrix multiplication can be reduced by a factor of 8 from approximately $1.13 \times 10^{-3}$ seconds to approximately $3 \times 10^{-4}$ seconds. This leads to a discovery of the $14 \times 10^9$ nodes in the first Schreier tree in approximately 20 days. An additional 20 days is also spent exploring vector-matrix multiplications that lead to duplicate values. This implies a total of approximately 40 days. Given a small cluster of 32 machines, this time is cut down to a much more reasonable time of just over a day.

A simple queue-based parallel breadth-first search can be performed. The one problem that remains is how to determine if a vector has already been seen. With a permutation representation for a group, the answer is obvious. A perfect hash function is available and, when distributed, a hash table for this requires only one bit per element and can easily fit in memory. This is not true for the matrix representation, however. Here, the best unique hash available is far too large for memory, even distributed memory.

An approach that allows for unresolved hash collisions is chosen. When a collision occurs, the vector is discarded from the breadth-first search and its children are not computed. However, these collisions are recorded on disk. Later, the vectors can be sorted using external sort and scanned through sequentially on disk to identify values that were discarded because of invalid hash collisions. By performing multiple passes, the first Schreier tree is discovered in a reasonable amount of time and space. This technique, tiered duplicate detection, is described earlier in Section 3.

Since the results for the first Schreier tree are stored on disk, generating random elements in the transversal requires randomly looking up an element on disk. In order to use the resulting data structure efficiently, rather than generating random elements one at a time, large batches may be generated simultaneously. Assuming all the random elements required can be generated at once, a single pass through the disk-based structure can be used to find the required information from the first Schreier tree.

While a full Schreier tree generation has not yet been performed, using batch generation of random elements and the method described for Schreier Tree generation for Thompson group in Section 8.3.1 can be used to easily complete the task, even on a single machine. The resulting data structure will be a distributed disk-based representation of the Schreier trees for the Baby Monster.
Once again, for efficiency, the data structure should be queried through batching membership queries.

### 8.3.4 The Janko Group $J_4$

The Janko group $J_4$ is a small base group of size $8.6 \times 10^{19}$, with two generating permutations that act on $1.3 \times 10^{11}$ points. As with the the previous two groups (see Section 8.3.3 and 8.3.2), storing even the generators as permutations would be impractical. Matrix generators of dimension $112$ over $GF(2)$ are used instead. Rather than a single vector, a 4-dimensional subspace is required.

Four vector-matrix multiplications and a normalization of the resulting vector for the Janko group $J_4$ can be performed in $4.5 \times 10^{-5}$s. This leads to a predicted time of 4.5 days to compute the first orbit on a 30 node cluster. Once again, because determining if a vector is a duplicate remains a hard problem, tiered duplicate detection is used here as well to obtain an actual time very close to the predicted one.

As with the Fischer group $F_{i23}$, a condensation which partitioned the orbit into 7173 suborbits and then computed the mappings between those suborbits was performed. This was done by a *disk-based condensation technique*.

To discover the suborbits, a fraction of the search-space that fit in distributed memory were set as the *landmarks*. Here it is not a requirement that there exist at least one landmark in each suborbit. However, the same approach as with the Fischer group $F_{i23}$ (see Section 8.3.2) was used to enumerate the suborbits. During the enumeration, in addition to storing the landmarks in memory, pairs containing each state and its suborbit identifier were also stored on disk. To find suborbits which may have been missed, these pairs along with the original orbit were sorted using external sort, and the missing states, those states in the orbit that do not have a matching in the suborbits, were determined.

From here, the process can be repeated selecting a much higher landmark ratio (based on the size of the missed states, not the whole orbit), and the missed states can be scanned to detect suborbits using that new landmark ratio. This can be repeated until there are no remaining missed states. Each pass, only the new missed states of the orbit and the new suborbit states must be examined, making the time to scan disk minimal. Using this approach, the predicted time spent enumerating the suborbits was 6.8 days on a 30 node cluster, with an additional disk time of 1.9 days on a 30 node cluster, or a total time of 8.7 days on a 30 node cluster.

Finally, to compute the condensation matrix, the original orbits were enumerated one at a time, the condensation element was then applied to each element in that orbit, and a pair containing the
resulting suborbit and the initial state was written to disk. By sorting these pairs using external sort and then performing a matching of the pairs seen during the suborbit partitioning and those seen here, the suborbit before the application of the condensation element and the resulting suborbit can be determined. This can was done for each element and the condensation matrix was computed. This required time for the enumeration of the suborbits and the application of the condensation element, 9.0 days on a 30 node cluster, as well as time to scan disk and perform the matchings, 0.5 days on a 30 node cluster. The total predicted time was 9.5 days on a 30 node cluster.

Using a landmark-based condensation technique, the best landmark ratio possible given the available memory would have been 1/40. This method was chosen over the landmark method because, using the landmark method:

- Not all of the suborbits would have been seen during the suborbit discovery phase; and
- The computation of the condensation matrix would have required 54.4 days on a 30 node cluster.

### 8.4 Static Analysis of Applications for Computational Group Theory

Here the algorithm analysis presented in Section 5 is applied to the search applications presented in Section 8.3 through the use of software developed to predict the parameters $R^S$, $R^R$, $X^C$, and $X^A$. These parameters are then used to determine the effectiveness of each search algorithm to the application presented. Actual observed run-times are given where available.

The constant buffer sizes mentioned in many of the algorithms are included in the analysis of these sample problems. Also, an optimization mentioned in Section 5 is included when computing the run-times. This optimization is the reduction in the number of passes through the hash for the implicit open list technique and the hash-based delayed duplicate detection technique. This optimization was found to have a reasonably significant impact on the run-times of the algorithms.

Where tunable parameters were available for the different techniques, these parameters were computed based on those used during experimentation, or where no experimentation was performed, they were computed to utilize as much available space as possible in order to reduce the overall run-time. In cases where no experimentation was available, these parameters were computed automatically by the software used for the analysis.

For applications where no reasonably small perfect hash table is available, those techniques requiring a perfect hash table are ignored. However, the information for breadth-first search with a perfect hash is still given to provide a basis for comparison. For applications where the size of
the search exceeds the number of bits available in distributed memory, tiered duplicate detection is ignored.

8.4.1 The Thompson Group

A Schreier tree of size 143127000 was computed for the Thompson group in order to form a Schreier tree representation for the group. More details for the application can be found in Section 8.3.1 and in [21]. An analysis of the algorithms on the local cluster is given. The application parameters for the Thompson group are given here. Where inverses are used, $|G_I|$ is used to represent the number of generators, and $\|G_I\|$ the average branching factor.

\[
\begin{align*}
|S| &= 143127000 & \text{143 million points in the space} \\
|G| &= 2 & \text{2 permutation generators} \\
\|G\| &= 2 & \text{An average branching factor of 2} \\
|G_I| &= 4 & \text{4 permutation generators} \\
\|G_I\| &= 3 & \text{An average branching factor of 3} \\
|V| &= 20 & \text{Space for an integer from 0 to 143127000 as well as a path to the element from the root} \\
BW^G &= 1500000 & \text{1.5 million permutation lookups per second} \\
|H| &= 143127000 & \text{An optimal perfect hash}
\end{align*}
\]

The Thompson Group on a Single Shared Memory Machine  We ran the enumeration for the Thompson Group on a single shared memory machine. The information for that machine is shown below.

\[
\begin{align*}
BW^D &= 10000000 & \text{10 megabytes per second disk bandwidth} \\
N &= 1 & \text{A single machine} \\
|M| &= 12000000000 & \text{12 gigabytes of memory} \\
|D| &= 10000000000 & \text{10 gigabytes of disk}
\end{align*}
\]

Predicted Resource Use for Applicable Techniques  Since this technique can easily perform in-memory duplicate detection, only breadth-first search with a perfect hash is examined.
Experimental Results

To discover the first Schreier tree of the Thompson group, a breadth-first search with a perfect hash was used. Inverses were not used. We observed the following time. As can be seen, the experimental time very closely matched the predicted time.

<table>
<thead>
<tr>
<th></th>
<th>Predicted Time</th>
<th>Experimental Time</th>
<th>Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>12.72 minutes</td>
<td>14.5 minutes</td>
<td>+13.9%</td>
</tr>
</tbody>
</table>

Post-search computation Upon completion, the remainder of the Schreier Trees were built. Because of the large amount of memory available, the results from the enumeration were loaded directly into memory, allowing the remainder of the computation to be performed in only 20 minutes and require no disk access.

8.4.2 The Fischer Group $F_{i23}$

An orbit of size $11739046176$ was computed for the Fischer group, $F_{i23}$, in order to perform a condensation for the group. More details for the application can be found in Section 8.3.2 and in [63]. Computations on two clusters were performed, and an analysis of the algorithms on those clusters is given. The application parameters for the Fischer group, $F_{i23}$ are given here. Where inverses are used, $|G_i|$ is used to represent the number of generators, and $\|G_i\|$ the average branching factor.
\[ |S| = 11739046176 \quad \text{11 billion vectors in the space} \]
\[ |G| = 2 \quad \text{2 matrix generators} \]
\[ \|G\| = 1.7 \quad \text{An average branching factor of 1.7} \]
\[ |G_I| = 4 \quad \text{4 matrix generators with inverses} \]
\[ \|G_I\| = 2.7 \quad \text{An average branching factor of 2.7 with inverses} \]
\[ |V| = 100 \quad \text{Space for a vector of dimension 782 over GF(2)} \]
\[ BW^G = 30000 \quad \text{30 thousand vector-matrix multiplies per second} \]
\[ |H| = \infty \quad \text{No perfect hash available} \]

**The Fischer Group \( F_{i23} \) on the Opportunity Cluster**  
Opportunity is a local cluster at Northeastern University with 60 nodes. We ran the enumeration and the post-processing condensation on 56 of these machines. The information for the cluster is shown below.

\[ BW^D = 10000000 \quad \text{10 megabytes per second disk bandwidth} \]
\[ N = 56 \quad \text{A 56 node cluster} \]
\[ |M| = 2000000000 \quad \text{2 gigabytes of memory per machine} \]
\[ |D| = 20000000000 \quad \text{20 gigabytes of disk per machine} \]

**Predicted Resource Use for Applicable Techniques on the Opportunity Cluster**  
With the architectural and application parameters, the required disk and memory for each of the search techniques can be predicted. We show the predicted time and space for the memory-resident perfect hash technique to offer a basis of comparison for the remainder of the algorithms.
<table>
<thead>
<tr>
<th>$R^R/N$</th>
<th>$R^S/N$</th>
<th>compressed inverses</th>
<th>compressed inverses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>Hash</td>
<td>20.96 GB</td>
<td>5.13 GB</td>
<td>20.96 GB</td>
</tr>
<tr>
<td>Landmarks $L = 12$</td>
<td></td>
<td>1.84 GB</td>
<td></td>
</tr>
<tr>
<td>Sorting</td>
<td>280.00 MB</td>
<td>1.23 GB</td>
<td>280.00 MB</td>
</tr>
<tr>
<td>DDD</td>
<td>22.54 GB</td>
<td>7.39 GB</td>
<td>32.77 GB</td>
</tr>
<tr>
<td>Tiered DD $HM = 1.87$</td>
<td>329.00 MB</td>
<td>1.84 GB</td>
<td>329.00 MB</td>
</tr>
<tr>
<td>Frontier</td>
<td>280.00 MB</td>
<td>1.17 GB</td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td>33.10 GB</td>
<td>19.66 GB</td>
<td></td>
</tr>
</tbody>
</table>

In addition, the run-times for these algorithms can also be determined.

<table>
<thead>
<tr>
<th>$X^C/N$</th>
<th>$X^A/N$</th>
<th>compressed inverses</th>
<th>compressed inverses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>3.30 hours</td>
<td>3.30 hours</td>
<td>5.24 hours</td>
</tr>
<tr>
<td>Hash</td>
<td>1.16 hours</td>
<td>1.25 hours</td>
<td>1.16 hours</td>
</tr>
<tr>
<td>Landmarks $L = 12$</td>
<td></td>
<td>1.69 days</td>
<td></td>
</tr>
<tr>
<td>Sorting</td>
<td>3.30 hours</td>
<td>1.25 hours</td>
<td>5.24 hours</td>
</tr>
<tr>
<td>DDD</td>
<td>17.41 hours</td>
<td>6.21 hours</td>
<td>7.63 hours</td>
</tr>
<tr>
<td>Tiered DD $HM = 1.87$</td>
<td>3.30 hours</td>
<td>10.72 hours</td>
<td>5.24 hours</td>
</tr>
<tr>
<td>Frontier</td>
<td>10.88 hours</td>
<td>2.64 hours</td>
<td>5.24 hours</td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td>5.24 hours</td>
<td></td>
</tr>
</tbody>
</table>

**Experimental Results on the Opportunity Cluster** To enumerate the orbit of the Fischer Group, $Fi_{23}$, tiered duplicate detection was used with a hash multiple of 1.87. Inverses were not used but compressed states were. We observed the following time. As can be seen, the experimental time matches reasonably closely with the predicted time. The pass information for the tiered duplicate detection algorithm in this computation can be found in Appendix A.1.
The Fischer Group $Fi_{23}$ on the Teraccluster Cluster  

Teraccluster is a local cluster at Northeastern University with 32 nodes. We ran the enumeration and the post-processing condensation on 30 of these machines. The information for the cluster is shown below. The run on this cluster provides an interesting comparison to our run on the opportunity cluster, as there is the same amount of memory available per machine, twenty times as much disk available, and only half the number of machines available.

$BW^D = 10000000$  
$N = 30$  
$|M| = 2000000000$  
$|D| = 450000000000$

$10$ megabytes per second disk bandwidth  
A $30$ node cluster  
$2$ gigabytes of memory per machine  
$450$ gigabytes of disk per machine

Predicted Resource Use for Applicable Techniques on the Teraccluster Cluster  

With the architectural and application parameters, the required disk and memory for each of the search techniques can be predicted. We show the predicted time and space for the memory-resident perfect hash technique to offer a basis of comparison for the remainder of the algorithms.

<table>
<thead>
<tr>
<th></th>
<th>compressed</th>
<th>compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>inverses</td>
<td>inverses</td>
</tr>
<tr>
<td>Perfect Hash</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$R^R/N$</td>
<td>$39.13$ GB</td>
<td>$9.50$ GB</td>
</tr>
<tr>
<td></td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td></td>
<td>$39.13$ GB</td>
<td>$14.04$ GB</td>
</tr>
<tr>
<td>Landmarks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L = 23$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorting DDD</td>
<td>$75.00$ MB</td>
<td>$330.00$ MB</td>
</tr>
<tr>
<td>$HM = 16.35$</td>
<td>$41.94$ GB</td>
<td>$13.89$ GB</td>
</tr>
<tr>
<td></td>
<td>$75.00$ MB</td>
<td>$330.00$ MB</td>
</tr>
<tr>
<td>Tiered DD</td>
<td>$874.72$ MB</td>
<td>$1.28$ GB</td>
</tr>
<tr>
<td></td>
<td>$63.95$ GB</td>
<td>$47.85$ GB</td>
</tr>
<tr>
<td></td>
<td>$874.72$ MB</td>
<td>$1.28$ GB</td>
</tr>
<tr>
<td>Frontier Search</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$75.00$ MB</td>
<td>$314.06$ MB</td>
</tr>
<tr>
<td></td>
<td>$61.79$ GB</td>
<td>$36.71$ GB</td>
</tr>
</tbody>
</table>
Experimental Results on the Teracluster Cluster  To enumerate the orbit of the Fischer Group, $Fi_{23}$, Tiered Duplicate Detection was used with a hash multiple of 16.35. Inverses and compressed states were not used. We observed the following time. As can be seen, the experimental time matches reasonably closely with the predicted time. The pass information for the tiered duplicate detection algorithm in this computation can be found in Appendix A.1.

<table>
<thead>
<tr>
<th>Method</th>
<th>Predicted Time</th>
<th>Experimental Time</th>
<th>Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20.36 hours</td>
<td>26.66 hours</td>
<td>+30.9%</td>
</tr>
</tbody>
</table>

Post-search computation  As discussed in Section 8.3.2, this computation was performed on the Opportunity cluster. The post-search computation was a condensation using a landmark-based condensation strategy. The times obtained are shown below.

<table>
<thead>
<tr>
<th>Method</th>
<th>Predicted Time</th>
<th>Experimental Time</th>
<th>Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suborbit Partitioning</td>
<td>4.2 hours</td>
<td>4.0 hours</td>
<td>-4.7%</td>
</tr>
<tr>
<td>Orbit Counting</td>
<td>11.5 hours</td>
<td>20.0 hours</td>
<td>+73.9%</td>
</tr>
<tr>
<td>Total</td>
<td>15.7 hours</td>
<td>24.0 hours</td>
<td>+52.8%</td>
</tr>
</tbody>
</table>
8.4.3 The Baby Monster Group

An orbit of size 13571955000 was computed for the Baby Monster group. To date, no additional post-processing has been performed. More details for the application can be found in Section 8.3.3 and in [62]. An analysis of the algorithms on the local cluster is given. The application parameters for the Baby Monster group are given here. Where inverses are used, $|G_I|$ is used to represent the number of generators, and $\|G_I\|$ the average branching factor.

The application parameters for the Baby Monster group are presented here.

\[
\begin{align*}
|S| &= 13571955000 & \text{13 billion vectors in the space} \\
|G| &= 2 & \text{2 matrix generators} \\
\|G\| &= 1.7 & \text{An average branching factor of 1.7} \\
|G_I| &= 4 & \text{4 matrix generators} \\
\|G_I\| &= 2.7 & \text{An average branching factor of 2.7} \\
|V| &= 548 & \text{Space for a vector of dimension 4370 over GF(2)} \\
BW^G &= 1500 & \text{15 hundred vector-matrix multiplies per second} \\
|H| &= \infty & \text{No perfect hash available}
\end{align*}
\]

**The Baby Monster on the Teracuster Cluster** Teracuster is a local cluster at Northeastern University with 32 nodes. We ran the enumeration on 32 of these machines. The information for the cluster is shown below.

\[
\begin{align*}
BW^D &= 1000000 & \text{10 megabytes per second disk bandwidth} \\
N &= 30 & \text{A 32 node cluster} \\
|M| &= 2000000000 & \text{2 gigabytes of memory per machine} \\
|D| &= 450000000000 & \text{450 gigabytes of disk per machine}
\end{align*}
\]

**Predicted Resource Use for Applicable Techniques** With this information, the required disk and memory for each of the search techniques can be predicted.
Experimental Results To enumerate the orbit of the Baby Monster, tiered duplicate detection was used with a hash multiple of 16.5. Inverses were not used, but compressed states were used. We observed the following time. As can be seen, the experimental time very closely matched the predicted time. The pass information for the tiered duplicate detection algorithm in this computation can be found in Appendix A.1.
<table>
<thead>
<tr>
<th>Predicted Time</th>
<th>Experimental Time</th>
<th>Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.84 days</td>
<td>6.05 days</td>
<td>-22.9%</td>
</tr>
</tbody>
</table>

**Post-search computation**  No additional computation has been performed from the initial data to date. In the future, the data could be used to form a Schreier tree representation for the group, or to find permutation generators for the group, which do not exist currently [12].

### 8.4.4 The Janko Group $J_4$

An orbit of size 131358148251 was computed for the Janko group, $J_4$, in order to perform a condensation for the group. More details for the application can be found in Section 8.3.4, the work is currently unpublished. An analysis of the algorithm on the clusters is given. The application parameters for the Janko group, $J_4$ are given here. Where inverses are used, $|G_I|$ is used to represent the number of generators, and $\|G_i\|$ the average branching factor.

\[
\begin{align*}
|S| &= 131358148251 & & 131 \text{ billion vectors in the space} \\
|G| &= 2 & & 2 \text{ matrix generators} \\
\|G\| &= 1.7 & & \text{An average branching factor of 1.7} \\
|G_I| &= 4 & & 4 \text{ matrix generators with inverses} \\
\|G_I\| &= 2.7 & & \text{An average branching factor of 2.7 with inverses} \\
|V| &= 64 & & \text{Space for a 4-subspace of vectors of dimension 112 over GF(2)} \\
BW^G &= 30000 & & 30 \text{ thousand 4-space multiplications and normalizations per second} \\
|H| &= \infty & & \text{No perfect hash available}
\end{align*}
\]

**The Janko Group $J_4$ on the Teraccluster Cluster**  Teraccluster is a local cluster at Northeastern University with 32 nodes. We ran the enumeration and the post-processing condensation on 30 of these machines. The information for the cluster is shown below.
\[ BW^D = 10000000 \quad \text{10 megabytes per second disk bandwidth} \]
\[ N = 30 \quad \text{A 30 node cluster} \]
\[ |M| = 2000000000 \quad \text{2 gigabytes of memory per machine} \]
\[ |D| = 450000000000 \quad \text{450 gigabytes of disk per machine} \]

**Predicted Resource Use for Applicable Techniques**  With the architectural and application parameters, the required disk and memory for each of the search techniques can be predicted.

<table>
<thead>
<tr>
<th>Technique</th>
<th>( R^R/N )</th>
<th>( R^S/N )</th>
<th>( X^C/N )</th>
<th>( X^A/N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>Hash</td>
<td>262.72 GB</td>
<td>81.02 GB</td>
<td>262.72 GB</td>
<td>114.42 GB</td>
</tr>
<tr>
<td>Landmarks</td>
<td></td>
<td>9.74 GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L = 39</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorting</td>
<td>256.56 MB</td>
<td>256.56 MB</td>
<td>256.56 MB</td>
<td>256.56 MB</td>
</tr>
<tr>
<td>DDD</td>
<td>291.07 GB</td>
<td>108.25 GB</td>
<td>528.09 GB</td>
<td>269.32 GB</td>
</tr>
<tr>
<td>Tiered DD</td>
<td>2.00 GB</td>
<td>2.00 GB</td>
<td>2.00 GB</td>
<td>2.00 GB</td>
</tr>
<tr>
<td>HM = 3.35</td>
<td>446.46 GB</td>
<td>338.77 GB</td>
<td>910.04 GB</td>
<td>426.99 GB</td>
</tr>
<tr>
<td>Frontier</td>
<td></td>
<td>260.57 MB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td>275.97 GB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In addition, the run-times for these algorithms can also be determined.

<table>
<thead>
<tr>
<th>Technique</th>
<th>( R^R/N )</th>
<th>( R^S/N )</th>
<th>( X^C/N )</th>
<th>( X^A/N )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>3.17 days</td>
<td>2.69 days</td>
<td>6.33 days</td>
<td>4.28 days</td>
</tr>
<tr>
<td>Hash</td>
<td>14.60 hours</td>
<td>16.42 hours</td>
<td>14.60 hours</td>
<td>16.53 hours</td>
</tr>
<tr>
<td>Landmarks</td>
<td></td>
<td>4.49 months</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L = 39</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorting</td>
<td>3.17 days</td>
<td>2.69 days</td>
<td>6.33 days</td>
<td>4.28 days</td>
</tr>
<tr>
<td>DDD</td>
<td>10.10 days</td>
<td>4.15 days</td>
<td>5.17 days</td>
<td>3.39 days</td>
</tr>
<tr>
<td>Tiered DD</td>
<td>3.17 days</td>
<td>11.88 days</td>
<td>6.33 days</td>
<td>8.45 days</td>
</tr>
<tr>
<td>HM = 3.35</td>
<td>5.12 days</td>
<td>1.84 days</td>
<td>6.04 days</td>
<td>1.97 days</td>
</tr>
<tr>
<td>Frontier</td>
<td></td>
<td>6.33 days</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td>5.25 days</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Experimental Results  We ran the Janko Group, $J_4$, with sorting-based delayed duplicate detection. Inverses and compressed states were not used in both cases. We observed the following time.

<table>
<thead>
<tr>
<th>Predicted</th>
<th>Experimental</th>
<th>Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>13.27 days</td>
<td>12.05 days</td>
<td>-9.2%</td>
</tr>
</tbody>
</table>

Post-search computation  As discussed in Section 8.3.4, this computation was performed on the teracluster cluster. The post-search computation was a condensation using a disk-based condensation strategy. The predicted and experimental times for the suborbit partitioning and the orbit counting are shown below.

<table>
<thead>
<tr>
<th>Suborbit Partitioning</th>
<th>Predicted Time</th>
<th>Experimental Time</th>
<th>Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.7 days</td>
<td>7.82 days</td>
<td>-10.1%</td>
<td></td>
</tr>
<tr>
<td>Orbit Counting</td>
<td>9.5 days</td>
<td>9.48 days</td>
<td>-0.2%</td>
</tr>
<tr>
<td>Total</td>
<td>18.2 days</td>
<td>17.30 days</td>
<td>-5.2%</td>
</tr>
</tbody>
</table>

This work is currently unpublished due to the fact that the proper condensation subgroup has not yet been provided. Once this subgroup is determined, condensation matrices that yield the information desired can be constructed. The computation over the desired condensation subgroup is expected to perform similarly to the computation presented here.

9 Examples from Outside Computational Group Theory

In order to show the usefulness of our API and software, we examine two other areas outside of computational group theory where this framework is applicable. We look at puzzle search, where our software can be used to solve problems such as variations of the Rubik’s cube puzzle and the sliding tile 15 puzzle. In addition, we look at formal verification, where our software can be used to solve problems involving implicit state model checking for algorithmic, programming, and hardware models, such as Peterson’s mutual exclusion algorithm implemented in these three models.

9.1 Puzzle Search

Puzzle search typically involves finding the reachable states and paths to those states from some “home” state in a puzzle. Once this is done, for any starting state, a path to the “home” state
can be found, assuming one exists, by inverting the path recorded to that state. In many of these cases, shortest path solutions are desired. This requires a breadth-first search. Here two examples are considered: a Rubik’s cube example, through which an existing search application was used to show that 26 moves suffice to solve Rubik’s cube [44]; and a sliding tile puzzle example, which can be used to find solution paths for states in this problem.

We examine the puzzle search problems because of the interest in the Artificial Intelligence community of these types of problems. The techniques used to solve for large puzzle problems such as the sliding tile and Rubik’s cube puzzle can typically be extended and used to explore game trees, oftentimes with pruning. This leads to better endgame heuristics, and a better overall computer opponent in games such as chess, checkers, go, etc.

For the Rubik’s Cube example, while our API is capable of representing the existing computation sufficiently, the current software implementation has no algorithm interface instance capable of running the computation feasibly on any of the available clusters. Fundamentally, this is because the implicit open list search technique has not yet been implemented by our software. This technique adds two key features required to successfully compute over the Rubik’s cube problem:

- Access to a technique capable of using a hash table stored on disk; and
- Access to a technique capable of performing incremental merges of the frontier into the known states.

For the sliding tile 15 puzzle, our API is capable of representing the search problem and we have an efficient software implementation of this application using our API. We are not currently capable of discovering the state space due to insufficient disk space to store all the states. Fundamentally, this is because our sorting-based delayed duplicate detection does not take advantage of the use of inverses. This modification would add one key feature required to improve performance on the sliding tile puzzle:

- The ability to detect duplicates at every level by only looking at the previous two levels.

With this, states seen before these required levels could be discarded, reducing the overall space required.

### 9.1.1 Rubik’s Cube

The work described here on the Rubik’s cube problem was done by Daniel Kunkle and Gene Cooperman in 2007 to show that 26 moves suffice to solve the cube [44]. The 3 by 3 by 3 Rubik’s cube forms
a group with $4.3 \times 10^{19}$ elements. While this search space was far too large to explore, a subgroup of size 663,552 was considered. This is easily managed and solved for optimally in memory on a single machine. After this is done, the coset space for this subgroup in the full Rubik’s Cube group can be considered. There are $6.5 \times 10^{13}$ such cosets. This number can be reduced through symmetry reduction to $1.3 \times 10^{12}$ cosets. While exploration of this space is not feasible in main memory, a distributed disk-based algorithm can be used.

The application parameters for the Rubik’s Cube enumeration are presented here.

| $|S|$ | $= 1357981544340$ | 1.3 trillion integer 4-spaces in the space |
| $|G|$ | $= 18$ | 18 generators that permute a 4-space |
| $\|G\|$ | $= 13.5$ | An average branching factor of 13.5 |
| $|G_I|$ | $= 18$ | 18 generators that permute a 4-space |
| $\|G_I\|$ | $= 13.5$ | An average branching factor of 13.5 |
| $|V|$ | $= 8$ | Space for four 2-byte integers |
| $BW^G$ | $= 5000000$ | 5 million function applications per second |
| $|H|$ | $= 1471074877440$ | A perfect nearly fully-occupied hash |

The Rubik’s Cube Group on Datastar Datastar is a storage intensive cluster available through Teragrid. The enumeration and the post-processing to reduce the overall search depth was performed on 128 processors. The information for the cluster is shown below. While Datastar uses a Storage Area Network (SAN), a disk rate of 10 megabytes per second is assume for each processor, which was the disk rate observed when all processors were writing data simultaneously.

| $BW^D$ | $= 10000000$ | 10 megabytes per second disk bandwidth |
| $N$ | $= 128$ | 128 processors in the cluster available |
| $|M|$ | $= 2000000000$ | 2 gigabytes of memory per processor |
| $|D|$ | $= 55000000000$ | 55 gigabytes of disk per machine |

Predicted Resource Use for Applicable Techniques With the architectural and application parameters, the required disk and memory for each of the search techniques can be predicted.
In addition, the run-times for these algorithms can also be determined.
Experimental Results  
Dan Kunkle and Gene Cooperman used search with an implicit open list, using a 6 terabyte disk buffer, here we compare our predicted time to the experimental results they observed [44].

\[
\begin{array}{c|c|c|c}
\text{Predicted} & \text{Experimental} & \text{Percent} \\
\text{Time} & \text{Time} & \text{Difference} \\
2.49 \text{ days} & 2.62 \text{ days} & +5.2\% \\
\end{array}
\]

Once again, as described in the beginning of this section, our software is not currently capable of running this algorithm due to the fact that we do not currently have the implicit open list instance of the algorithm interface implemented. It’s interesting to note, however, that with access to a cluster with twice as much disk space (15 TB overall instead of 7 TB), this enumeration could be run using breadth-first search with a perfect hash in approximately a quarter of the time the implicit open list technique required.
9.1.2 Sliding Tile Puzzle

Zhou and Hansen [78] provide the fastest current method for searching for solutions to the 4 by 4 sliding tile puzzle (15 puzzle). Their work was built off of Korf’s earlier work on this problem [38, 42]. Zhou and Hansen found solutions through the use of structured duplicate detection. Korf, on the other hand, simply used frontier search to solve the puzzle. The size of the search for the 15 puzzle is $\frac{16!}{2} = 10.4 \times 10^{12}$. With symmetry reductions, this can be reduced by a factor of 8 to $1.3 \times 10^{12}$. We offer an application interface implementation of this problem to show that our software is also capable of solving this puzzle.

$$\begin{align*}
|S| & = 1307674368000 & 1.3 \text{ trillion } 4 \times 4 \text{ matrices in the space} \\
|G| & = 4 & 4 \text{ generators that permute a } 4 \times 4 \text{ matrix} \\
\|G\| & = 2 & \text{An average branching factor of 2} \\
|G_I| & = 4 & 4 \text{ generators that permute a } 4 \times 4 \text{ matrix} \\
\|G_I\| & = 2 & \text{An average branching factor of 2} \\
|V| & = 8 & \text{Space for sixteen } 4\text{-bit integers} \\
BW^G & = 150000 & 150 \text{ thousand permutations of matrices per second} \\
|H| & = \infty & \text{No perfect hash implemented currently}
\end{align*}$$

The 15 puzzle on Teracluster  Teracluster is a local cluster at Northeastern University with 32 nodes. We consider running enumeration on all of these machines. The information for the cluster is shown below.

$$\begin{align*}
BW^D & = 10000000 & 10 \text{ megabytes per second disk bandwidth} \\
N & = 32 & 30 \text{ machines in the cluster available} \\
|M| & = 4000000000 & 4 \text{ gigabytes of memory per machine} \\
|D| & = 450000000000 & 450 \text{ gigabytes of disk per machine}
\end{align*}$$

Predicted Resource Use for Applicable Techniques  With the architectural and application parameters, the required disk and memory for each of the search techniques can be predicted.
Experimental Results

As described in the beginning of this section, our software is not currently capable of running this algorithm. However, an encoding of it with and without symmetry reduction exists using our software. Currently we do not have the cluster resources to execute this problem. It’s interesting to note that given a cluster with twice as much disk as is currently available, the 15 puzzle under symmetry reduction could be enumerated.

Limitations of the Current Software

As mentioned at the start of this section, our software is currently capable of implementing the sliding tile 15-puzzle, but does not take advantage of the use of inverse generators. With this, a 25% speedup over the current predicted CPU time could be obtained.


9.2 Formal Verification

Formal verification seeks to prove properties about hardware or software systems by modeling them and analyzing their potential execution paths on an actual system. By doing this, one can both:

1. Detect bugs: find execution states where an invalid program configuration exists; and

2. Prove invariants: show that for all possible states, some aspect of the program configuration holds.

Specifically, we examine explicit state model checking, where verification is performed through enumerating and examining potential program states, rather than constructing proofs about the system. For this, our software implemented with our API is an ideal method to perform task 2 above, as it can use disk to enumerate more potential execution paths than would be possible strictly in RAM. Our software can also be used to find solutions to task 1 above. However, since finding all potential bugs is not typically desired, our software will spend extra work dynamically verifying all states rather than allowing for some states to be missed.

We consider only model checking problems involving concurrency or non-determinism. Problems with these characteristics lead to a non-linear search, and the ability to parallelize the computation in a non-trivial manner.

Existing Explicit State Model Checking Software Two of the more popular systems for software verification are Murphi [24] and SPIN [1]. Here, we choose to compare to the Murphi verification system. This system is chosen because Murphi uses an underlying state-based model, as opposed to a program-based model, for its verifications and this most closely matches our software and API. This makes it much easier to implement solutions that already exist in Murphi using our software and draw a fair comparison.

It is important to note that we don’t seek to compete with existing model checkers in terms of the actual speed of verification. Our goals is to show an explicit state verification system could be built on top of our API that would allow for verifications that had not previously been feasible. To this end, our goals are two-fold:

1. To show feasibility and ease of implementation of model checking problems with our software using our API; and

2. To find verification problems that are too large for distributed RAM, that can be verified using distributed disk.
Automatic Program Translation While both of the model-checking programs discussed above have translation software to convert between the modeling language and the actual code that runs, we have to hand-code our solutions. One can imagine defining a modeling language and building another layer on top of our software to do this automatically. While this is possible, our goal is to show feasibility and relative ease of implementation. This is a project for future work.

Symmetry Reduction Murphi [24], as described above, takes advantage of symmetry reduction to reduce the total number of states seen during the search. Symmetry reduction identifies certain states as symmetrically the same. When two states in the same symmetry class are seen, they are identified as the same state by reducing both to some canonical element, which is unique in the symmetry class.

Our current software is capable of dealing with symmetry reduction and, in fact, does in the case of the 15 puzzle described in Section 9.1.2. We do not consider this further in this work.

Partial Order Reduction SPIN [1], as described above, takes advantage of partial order reduction to reduce the total number of states seen during the search. Partial order reduction is able to identify certain execution paths that when explored are identical to other execution paths. Once this is done, it is then possible to trim these paths to avoid whole branches of the search tree. To do this, it must have knowledge of at least part of the execution path used to reach the current state.

Because knowledge of the previous states along the execution path are required, implementing partial order reduction using our software is not trivial. However, by using a representation for a state that also stores a path (such as one using compressed states, or simply one storing the path only), the full execution path to a particular state can be built. While this will require additional time, it can be used to guarantee that the number of states seen using our software for a particular application agrees with the number seen with SPIN. Allowing us to scale our problem sizes similarly.

Dealing with Large States Many of the verification problems discussed here have very large state sizes. Taking advantage of disk, which is typically one hundred times larger than memory, is of no advantage if one is storing full states on disk that are one thousand times larger than a hash index stored in memory. For this case, however, compressed states can be of use. One can do one of the following:

- *Store compressed states and paths:* The final representation will be capable of reproducing any state in the search.
• *Store full states for the open list, compressed states only on disk:* The final representation will be capable of recognizing states that are in the search, when queried, but will not be able to generate those states.

Since typically for verification problems, it is not desirable to generate states after the search, the second technique will likely be preferred by most.

**Motivation for Disk: Comparison to Bloom Filters** An obvious question is whether the use of disk-based techniques are of benefit in general as compared to an immediate duplicate detection through Bloom filters. Consider the following, for duplicate detection for a search of size between $1 \times 10^{10}$ and $1 \times 10^{12}$:

- Using a Bloom filter will require between 53 and 63 bits per state
- Using compressed states will require between 74 and 87 bits per state

Knowing this, one can look at a typical cluster, one with 30 machines, and 4 gigabytes of RAM per machine, along with 250 gigabytes of disk per machine. From here, the maximum size of search possible for each of these approaches can be determined:

- Using a memory-based Bloom filter between $1.5 \times 10^{10}$ and $1.8 \times 10^{10}$ states can be explored without collisions with high probability.
- Using disk-based compressed states between $68.9 \times 10^{10}$ and $81.0 \times 10^{10}$ states can be explored without collisions with high probability.

This is not a perfect comparison. Using compressed states typically requires storing a path along with the compressed state as well, which may double the effective size of the compressed state. Search techniques that use disk with compressed states also typically require temporarily storing some duplicate states. However, typically problem sizes can scale up at least 10-fold when one extends to disk rather than strictly using memory.

**Unknown Search Sizes** For many verification problems, the search size is not known in advance. While it may be possible to get an estimate of the search size prior to executing the search, it turns out that this is not necessary to perform many of the search techniques. Obviously, not knowing the size of the search makes it impossible to predict run-times for the search techniques, but it does not make the search any more difficult to perform for techniques such as sorting-based delayed duplicate detection, tiered duplicate detection, and frontier search.
9.2.1 The N-Peterson Algorithm

In the late 1970’s, Peterson proposed a software implementation for the N processor mutual exclusion problem [60]. We refer to this as the \(N\)-Peterson algorithm. This problem is a common test problem for explicit state software verification tools, as one can ramp the size of the search by increasing \(N\), the number of processors seeking to access the critical section.

An example of pseudo-code that implements Peterson’s mutual exclusion algorithm is shown below:

```
Algorithm 13: Peterson

Input: \(N\)
for each process \(rank \in \{1, \ldots, N\}\) in parallel do
  // Flags is which round each process has requested the critical section in
  Global integer flags[N];
  // Turn is which processor will go in each round
  Global integer turn[N];
  while process \(rank\) wants to enter the critical section do
    // \(k\) is the current round, each round select a processor
    for \(k \in \{1, \ldots, N\}\) do
      // Indicate that we want to use the critical section this round
      flags[rank] = k;
      // Tell everyone that it’s our turn this round
      // The last one to set the turn for this round wins
      turn[k] = rank;
      // Check with all the other processors
    ;
    for \(j \in \{1, \ldots, N\}\) do
      if \(j \neq rank\) then
        // Unit there are processors with requests this round or later
        // Or until it’s our turn for this round, wait
        while flags[j] \(\geq k\) and turn[k] == myRank do
          // Perform the critical section
          // Indicate that we’re back in the first round
          flags[rank] = 0;
```

We look at this problem as a sample of a verification problem. In order to show applicability of our software at a variety of different levels of detail. We illustrate a verification of this problem under three different models:

1. an algorithm model (described in Section 9.2.3), where the states correspond to phases in the N-Peterson algorithm;

2. a program model (described in Section 9.2.4), where the states correspond to the transitions
between instructions in a program; and

3. a machine model (described in Section 9.2.5), where the states correspond to transitions between the states of a quad-core machine with a three-stage pipeline.

**The Architecture**

We choose to model the three-stage pipeline machine with 8 bit registers and a bare-bones instruction set described in [50]. We choose this model because using BAT, the symbolic model checker described in [50, 51], an equivalence between the instruction set architecture (ISA) model (a model at the program level) and the machine architecture (MA) model (a model at the machine level) can be shown. This implies that any properties we show about the ISA model also apply to the MA model [52].

In addition, a quad-core architecture is assumed, making the search non-linear (any processor may take a step at any time). We assume a simple shared memory model for this machine, where memory is shared among the processors, but data and instruction registers are not. While a more complex model involving a cache could be used, for ease of exposition, we choose not to implement this.

The number of cores, size and number of the data registers and instruction registers, size of the main memory, and the initial contents of all memory are all easily modified with our implementation. In the examples in this section, we use the following configuration for our machine:

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Instruction Registers</td>
<td>28 bits</td>
<td>256</td>
</tr>
<tr>
<td>Data Registers</td>
<td>8 bits</td>
<td>256</td>
</tr>
<tr>
<td>Shared Memory</td>
<td>8 bits</td>
<td>256</td>
</tr>
</tbody>
</table>

**The Instruction Set**

We consider an encoding of the N-Peterson algorithm in an assembly level language involving eight instructions. An instruction consists of the operation code along with up to three numbers, $C$, $A$, and $B$, typically corresponding to the registers to operate on. The instruction list and a description of the instructions follow.

- **0 - ADD RC RA RB**: Adds the contents of registers $RA$ and $RB$, and stores the result in register $RC$.

- **1 - SUB RC RA RB**: Subtracts the contents of registers $RB$ from $RA$, and stores the result in register $RC$. 
• **2 - AND RC RA RB**: Performs a bit-wise and of the contents of registers $RA$ and $RB$, and stores the result in register $RC$.

• **3 - LOAD RC MA**: Loads the value at address $MA$ in memory into register $RC$.

• **4 - LOADI RC RA**: Loads the value at the memory address specified by register $RA$ into register $RC$.

• **5 - STORE RA RB**: Stores the contents of register $RB$ at the address in memory specified by the contents of register $RA$.

• **6 - BEZ RA RB**: Offsets the PC by the amount specified by the contents of register $RB$ if the contents of register $RA$ is zero.

• **7 - JUMP RA**: Jumps to the instruction specified by the contents of register $RA$.

The N-Peterson Algorithm  The variables for the N-Peterson Algorithm were set up as follows in the data registers and memory on a quad-core machine. Any register or memory address not specified is assumed to be zero.

<table>
<thead>
<tr>
<th>Type</th>
<th>Number(s)</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>0–30</td>
<td>The numbers 0–30</td>
</tr>
<tr>
<td>Register</td>
<td>31</td>
<td>128 (0x80, Negative)</td>
</tr>
<tr>
<td>Register</td>
<td>32</td>
<td>Temporary Variable</td>
</tr>
<tr>
<td>Register</td>
<td>33</td>
<td>247 (-9)</td>
</tr>
<tr>
<td>Register</td>
<td>41</td>
<td>The number of the core (rank, 0-3, different on each core)</td>
</tr>
<tr>
<td>Register</td>
<td>42</td>
<td>The number of cores ($N$, 4)</td>
</tr>
<tr>
<td>Register</td>
<td>43</td>
<td>The $K$ variable</td>
</tr>
<tr>
<td>Register</td>
<td>44</td>
<td>The $J$ variable</td>
</tr>
<tr>
<td>Register</td>
<td>51</td>
<td>Memory location of the flags array (0)</td>
</tr>
<tr>
<td>Register</td>
<td>52</td>
<td>Memory location of the turn array (10)</td>
</tr>
<tr>
<td>Memory</td>
<td>0–3</td>
<td>The flags array</td>
</tr>
<tr>
<td>Memory</td>
<td>10–13</td>
<td>The turn array</td>
</tr>
</tbody>
</table>

With this, the N-Peterson algorithm can be implemented on a quad-core machine using the following instruction set. This instruction set is independent of the core it is stored on.

```c
// while( 1 ) {
0:       ADD 43 0 0
1:       SUB 32 43 42
```
2: BEZ 32 24
// for( k=0; k < maxRank; k++ ) {
3: ADD 32 51 41
4: STORE 32 43
// flags [myRank] = k;
5: ADD 32 52 43
6: STORE 32 41
// turn [k] = myRank;
7: ADD 44 0 0
8: SUB 32 44 42
9: BEZ 32 15
// for( j=0; j < maxRank; j++ ) {
10: SUB 32 44 41
11: BEZ 32 11
// if( j!=myRank ) {
// First compare
12: ADD 32 44 51
13: LOADI 32 32
14: SUB 32 32 43
15: AND 32 31 32
16: BEZ 32 2
17: JUMP 22
// Second compare
18: ADD 32 43 52
19: LOADI 32 32
20: SUB 32 32 41
21: BEZ 32 33
// while ( flags[j] >= k and turn [k]==myRank );
// }
22: ADD 44 44 1
23: JUMP 8
// }
24: ADD 43 1 43
25: JUMP 1
// }
26: ADD 32 51 41
27: STORE 32 0
// flags [myRank] = 0;
// }
28: JUMP 0

9.2.2 Summary of Results

A brief summary of our computations over this problem under these models is shown here. The table below shows the search space statistics for many problem sizes. The branching factors presented in the table were estimated for each computation based on the branching factor seen early in the search. However, they seem to remain between 1.0 and 1.5, independently of the number of generators, which is given after the branching factor in parenthesis.
In addition, the table below shows the computation statistics for each of these problems. Tiered duplicate detection was used in all cases, and compressed states were used for the program and machine models.
### Algorithm Model

<table>
<thead>
<tr>
<th>Version</th>
<th>Time</th>
<th>Memory</th>
<th>Disk</th>
<th>Passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-Peterson</td>
<td>&lt; 1 minute</td>
<td>&lt; 1 GB</td>
<td>&lt; 1 GB</td>
<td>0</td>
</tr>
<tr>
<td>3-Peterson</td>
<td>&lt; 1 minute</td>
<td>&lt; 1 GB</td>
<td>&lt; 1 GB</td>
<td>0</td>
</tr>
<tr>
<td>4-Peterson</td>
<td>&lt; 1 minute</td>
<td>&lt; 1 GB</td>
<td>&lt; 1 GB</td>
<td>0</td>
</tr>
<tr>
<td>5-Peterson</td>
<td>&lt; 1 minute</td>
<td>&lt; 1 GB</td>
<td>&lt; 1 GB</td>
<td>0</td>
</tr>
<tr>
<td>6-Peterson</td>
<td>2 minutes</td>
<td>4 GB</td>
<td>&lt; 1 GB</td>
<td>2</td>
</tr>
<tr>
<td>7-Peterson</td>
<td>14 minutes</td>
<td>4 GB</td>
<td>&lt; 1 GB</td>
<td>3</td>
</tr>
<tr>
<td>8-Peterson</td>
<td>13 hours</td>
<td>4 GB</td>
<td>135.49 GB</td>
<td>6</td>
</tr>
</tbody>
</table>

### Program Model

<table>
<thead>
<tr>
<th>Version</th>
<th>Time</th>
<th>Memory</th>
<th>Disk</th>
<th>Passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-Peterson</td>
<td>&lt; 1 minute</td>
<td>&lt; 1 GB</td>
<td>&lt; 1 GB</td>
<td>0</td>
</tr>
<tr>
<td>3-Peterson</td>
<td>6 minutes</td>
<td>4 GB</td>
<td>&lt; 1 GB</td>
<td>2</td>
</tr>
<tr>
<td>4-Peterson</td>
<td>19 hours</td>
<td>4 GB</td>
<td>214.54 GB</td>
<td>5</td>
</tr>
</tbody>
</table>

### Machine Model

<table>
<thead>
<tr>
<th>Version</th>
<th>Time</th>
<th>Memory</th>
<th>Disk</th>
<th>Passes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-Peterson</td>
<td>&lt; 1 minute</td>
<td>&lt; 1 GB</td>
<td>&lt; 1 GB</td>
<td>0</td>
</tr>
<tr>
<td>3-Peterson</td>
<td>12 minutes</td>
<td>4 GB</td>
<td>5.1 GB</td>
<td>2</td>
</tr>
</tbody>
</table>

Pass information for those computations not completed on a single machine (in pass 0) can be found in Appendix A.2. In addition, the following sections provide more details for the largest of each of these computations under each of the above models: algorithm; program; and machine.

### 9.2.3 Algorithm–based Implementations of the N-Peterson Algorithm

Here we examine an algorithm-based verification of the N-Peterson algorithm described above. We take \( N = 8 \) for our example. This implementation has five states a single processor can be in. These states correspond to:

1. Initialization of the outer loop;
2. Setting of the flag variable;
3. Setting of the turn variable;
4. Execution of the inner loop and entrance into the critical section; and
5. Execution of the critical section and exiting the critical section.
In addition, it must keep track of four local variables per processor:

1. A flag variable;
2. A turn variable;
3. The outer loop counter for that processor; and
4. The program counter for that processor.

**Application Parameters**  While the search size of the 8-Peterson problem without symmetry reduction was not known prior to performing the search, it was estimated as $11.0 \times 10^9$ based on the sizes of the searches with fewer processors. It turns out the exact number of states seen during the search was $15.4 \times 10^9$. The observed branching factor was also not known prior to performing the search but was easily estimated to be 1.5 by looking at the branching factor early in the search. This leads to the following parameters for the 8-Peterson problem:

$$|S| = 15426300028$$ 15 billion states with 4 integer arrays of size 8

$$|G| = 8$$ 8 generators that take a step with a single process

$$|G_I| = none$$ Inverses unavailable

$$|V| = 32$$ Space for 4 integer arrays of size 8

$$BW^G = 4.57 \times 10^5$$ 457 thousand processor steps taken per second

$$|H| = \infty$$ No perfect hash available

**The 8-Peterson Algorithm on Teracluster**  Teracluster is a local cluster at Northeastern University with 32 nodes. We ran the enumeration on 30 of these machines. The information for the cluster is shown below:

$$BW^D = 10000000$$ 10 megabytes per second disk bandwidth

$$N = 30$$ 30 machines in the cluster available

$$|M| = 4000000000$$ 4 gigabytes of memory per machine (8 gigabytes available)

$$|D| = 450000000000$$ 450 gigabytes of disk per machine
Predicted Resource Use for Applicable Techniques  With the architectural and application parameters, the required disk and memory for each of the search techniques can be predicted.

<table>
<thead>
<tr>
<th>$R^R/N$</th>
<th>$R^S/N$</th>
<th>compressed</th>
<th>inverses</th>
<th>compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Hash</td>
<td>15.43 GB</td>
<td>13.02 GB</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Sorting</td>
<td>80.00 MB</td>
<td>88.89 MB</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>DDD</td>
<td>25.73 GB</td>
<td>22.42 GB</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Tiered DD</td>
<td>2.00 GB</td>
<td>2.00 GB</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>$HM = 30.61$</td>
<td>120.60 GB</td>
<td>135.49 GB</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

In addition, the run-times for these algorithms can also be determined.

<table>
<thead>
<tr>
<th>$X^C/N$</th>
<th>$X^A/N$</th>
<th>compressed</th>
<th>inverses</th>
<th>compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>2.34 hours</td>
<td>2.34 hours</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Hash</td>
<td>51.42 minutes</td>
<td>1.22 hours</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Sorting</td>
<td>2.34 hours</td>
<td>2.34 hours</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>DDD</td>
<td>1.00 days</td>
<td>22.32 hour</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Tiered DD</td>
<td>2.34 hours</td>
<td>2.46 hours</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>$HM = 30.61$</td>
<td>13.70 hours</td>
<td>13.31 hours</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Experimental Results  To show that all possible program configurations for the Peterson algorithm with 8 processors are valid, search using tiered duplicate detection was used with a hash multiple of $HM = 30.61$. Compressed states were not used. We observed the following time.

<table>
<thead>
<tr>
<th>Predicted Time</th>
<th>Experimental Time</th>
<th>Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>16.04 hours</td>
<td>12.95 hours</td>
<td>-19.2%</td>
</tr>
</tbody>
</table>

Development Time and Performance  Coding this example in our software (to work for any value of N) based on the identical Murphi model required only 100 lines of code along with 1 hour of programmer time and 1 hour of debugging time. This illustrates the ease of implementing model checking problems with our software using our API, a goal discussed above.

While symmetry reduction was not performed in this case, that feature could have easily been added to our software. To run this example using a distributed version of Murphi working out of
memory exclusively would have required 3.7 gigabytes of memory per machine for duplicate detection alone using a Bloom filter with a 5% chance of a false collision. This would have used more memory than we had access to for just the duplicate detection for this example, illustrating that verification problems exist that are too large for distributed RAM, the second goal described above.

Finally, though not critical in our analysis, we do also provide some performance comparison with Murphi. It’s important to note that Murphi is known to be slower than SPIN. Taking this into consideration, the rate at which new states could be produced and “hashed” (presumably into a Bloom filter) with Murphi on a single machine was 800,000 states per second. Our software is capable of producing 457,000 states per second per machine. Our times also include the use of a distributed hash table.

In addition, if we consider the the overall rate of discovery (including the delayed duplicate detection phases), our rate of discovery drops to 88,000 states per second per machine. This is a discovery rate ten times slower than the memory-resident, single node discovery rate Murphi is capable of achieving.

### 9.2.4 Program–based Implementations of the N-Peterson Algorithm

In the Section 9.2.3, an implementation of the N-Peterson algorithm that was built around the algorithm itself was examined. While code was provided, the model did not follow the code line-for-line and instead was broken into components based on the high-level algorithm. Here we look at another implementations of the N-Peterson algorithm, one based on the instruction set specification for an assembly-level encoding of the algorithm.

**The 4-Peterson Algorithm using the ISA Model**

Here the information for the 4-Peterson algorithm on a quad-core machine under the ISA model is presented. Each program state corresponds to a unique configuration of the machine, including a program counter for each core. We choose to round the size of the instruction registers up by 4 bits for the operation code in order to have even byte boundaries. The size of a state, given this configuration, is:

\[
|V| = 4 \times (4 \times 256 + 1 \times 256 + 1) + 1 \times 256
\]

\[
= 5380 \text{ bytes}
\]

While the search size of the 4-Peterson problem was not known prior to performing the search, it was estimated as \(1.2 \times 10^9\) based on the sizes of the searches with fewer processors. It turns out the exact number of states seen during the search was very close to this. The observed branching
factor was also not known prior to performing the search but was easily discovered to be 1.14 by looking at the branching factor early in the search. This lead to the following parameters for the 4-Peterson problem:

\[
\begin{align*}
|S| &= 1237248824 & & 1.2 \text{ billion machine states} \\
|G| &= 4 & & 4 \text{ generators that take a step on a single core} \\
\|G\| &= 1.14 & & \text{An average branching factor of } 1.14 \text{ (approx)} \\
|G_I| &= \text{none} & & \text{Inverses unavailable} \\
\|G_I\| &= \text{none} & & \text{Inverses unavailable} \\
|V| &= 5380 & & \text{Space to store the full content of the machine} \\
BW^G &= 3.1 \times 10^3 & & 3 \text{ thousand processor steps taken per second} \\
|H| &= \infty & & \text{No perfect hash available}
\end{align*}
\]

**The 4-Peterson Algorithm using the ISA Model on the Terachuster Cluster** Terachuster is a local cluster at Northeastern University with 32 nodes. We ran the enumeration on 30 of these machines. The information for the cluster is shown below.

\[
\begin{align*}
BW^D &= 10000000 & & 10 \text{ megabytes per second disk bandwidth} \\
N &= 30 & & 30 \text{ machines in the cluster available} \\
|M| &= 4000000000 & & 4 \text{ gigabytes of memory per machine (8 gigabytes available)} \\
|D| &= 450000000000 & & 450 \text{ gigabytes of disk per machine}
\end{align*}
\]

**Predicted Resource Use for Applicable Techniques** With the architectural and application parameters, the required disk and memory for each of the search techniques can be predicted.

<table>
<thead>
<tr>
<th></th>
<th>(R^R/N)</th>
<th>(R^S/N)</th>
<th>(R^R/N) compressed inverses</th>
<th>(R^S/N) compressed inverses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect Hash</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Sorting DDD</td>
<td>203.14 MB</td>
<td>532.03 MB</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Tiered DD</td>
<td>3.01 GB</td>
<td>5.16 GB</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>(HM = 581.31)</td>
<td>829.80 GB</td>
<td>214.54 GB</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>
In addition, the run-times for these algorithms can also be determined.

<table>
<thead>
<tr>
<th>$X^C/N$</th>
<th>$X^A/N$</th>
<th>compressed inverses</th>
<th>compressed inverses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>13.92 hours</td>
<td>13.92 hours</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Hash</td>
<td>11.56 hours</td>
<td>11.60 hours</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Sorting</td>
<td>13.92 hours</td>
<td>13.92 hours</td>
<td>Not applicable</td>
</tr>
<tr>
<td>DDD</td>
<td>23.25 days</td>
<td>3.51 days</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Tiered DD</td>
<td>13.92 hours</td>
<td>14.22 hours</td>
<td>Not applicable</td>
</tr>
<tr>
<td>$HM = 581.31$</td>
<td>4.33 days</td>
<td>12.30 hours</td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

**Experimental Results** To show that all possible program configurations for the Peterson algorithm with 4 processors are valid, search using tiered duplicate detection was used with a hash multiple of $HM = 581.31$. Compressed states were used. We observed the following time.

<table>
<thead>
<tr>
<th>Predicted Time</th>
<th>Experimental Time</th>
<th>Percent Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.52 hours</td>
<td>19.5 hours</td>
<td>-26.4%</td>
</tr>
</tbody>
</table>

**Development Time and Performance** Coding this example in our software (to work for any instruction set) required only 200 lines of code along with 2 hour of programmer time and 1 day of debugging time. This, we believe, illustrates the ease of implementing model checking problems with our software using our API, a goal discussed above.

A point of interest was that our initial instruction set encoding of the N-Peterson problem had a couple of bugs. When run, our program, in one case, did not allow for each machine to enter the critical section, and in another, allowed two machines to enter the critical section at the same time. Our verification tool implemented with our software quickly spotted these bugs in the program and allowed us to analyze and resolve them.

While symmetry reduction would not be trivial to add, in this case, due to the fact that the implicit role of the rank of a machine in the algorithm is not necessarily obvious for the instruction set architecture model, it could be done with a bit more effort. Beyond this, one very quick optimization would be to recognize that the only important program states in determining the behavior of a multi-core machine involve those that correspond to loads and stores. With this, one can optimize out intermediate program states corresponding to operations local to a single core. This would provide a significant reduction in the size of the search space for this problem, by at least a factor of two.
For this application, we have no comparison to Murphi, because no implementation of this instruction set language exists currently in Murphi. However, we do know that we achieve a state generation rate of $3.1 \times 10^3$ states per second per machine when including the time for a distributed hash table. The overall algorithm, including the delayed duplicate detection phase, yields a discovery rate of 587.5 states per second per machine. While this is a significant reduction in the speed of discovery, to approximately 20%, it is not prohibitive in terms of feasibility.

9.2.5 Machine-based Implementations of the N-Peterson Algorithm

In the Section 9.2.5, an implementation of the N-Peterson algorithm that was built around an assembly level encoding was examined. While a machine architecture was provided, the model did not consider how the machine itself executed the individual instructions. Here we look at another implementations of the N-Peterson algorithm, one based on the machine behavior for that encoding on a quad-core machine with a three-stage pipeline.

The 4-Peterson Algorithm using the MA Model  While the BAT tool [50, 51] can be used to show an equivalence between the ISA model described above and the MA model described here, we show an explicit state model checker for the MA model as well.

Here the information for the 4-Peterson algorithm on a quad-core machine with a three stage pipeline under the MA model is presented. Each program state corresponds to a unique configuration of the machine, including a pipeline for each core. We choose to round the size of the instruction registers up by 4 bits for the operation code in order to have even byte boundaries. The pipeline requires 82 bits to store on each core, which we round up to 12 bytes. The size of a state, given this configuration, is:

$$|V| = 4 \times (4 \times 256 + 1 \times 256 + 12) + 1 \times 256$$

$$= 5424 \text{ bytes}$$

While the search size of the 4-Peterson problem was not known prior to performing the search, it was estimated as $44.5 \times 10^9$ based on the sizes of the searches with a smaller value of $N$. The computation is currently being run. The observed branching factor was also not known prior to performing the search but was easily discovered to be 1.14 by looking at the branching factor early in the search. This lead to the following parameters for the 4-Peterson problem:
\[|S| = 44500000000\] 44 billion machine states
\[|G| = 4\] 4 generators that take a step on a single core
\[\|G\| = 1.10\] An average branching factor of 1.10 (approx)
\[|G_I| = \text{none}\] Inverses unavailable
\[|V| = 5424\] Space to store the full content of the machine
\[BW^G = 11.0 \times 10^3\] 11 thousand processor steps taken per second
\[|H| = \infty\] No perfect hash available

The 4-Peterson Algorithm using the MA Model on the Teracluster Cluster

Teracluster is a local cluster at Northeastern University with 32 nodes. We are running the enumeration on 30 of these machines. The information for the cluster is shown below.

\[BW^D = 10000000\] 10 megabytes per second disk bandwidth
\[N = 30\] 30 machines in the cluster available
\[|M| = 4000000000\] 4 gigabytes of memory per machine (8 gigabytes available
\[|D| = 450000000000\] 450 gigabytes of disk per machine

Predicted Resource Use for Applicable Techniques

With the architectural and application parameters, the required disk and memory for each of the search techniques can be predicted.

<table>
<thead>
<tr>
<th>(R^R/N)</th>
<th>(R^S/N)</th>
<th>compressed</th>
<th>inverses</th>
<th>compressed</th>
<th>inverses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>(\infty)</td>
<td>(\infty)</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td>Hash</td>
<td>7.54 TB</td>
<td>257.87 GB</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td>Sorting</td>
<td>7.37 GB</td>
<td>7.37 GB</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td>DDD</td>
<td>7.54 TB</td>
<td>899.71 GB</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td>Tiered DD</td>
<td>10.18 GB</td>
<td>3.00 GB</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td></td>
</tr>
<tr>
<td>HM = 16.17</td>
<td>25.37 TB</td>
<td>353.34 GB</td>
<td>Not applicable</td>
<td>Not applicable</td>
<td></td>
</tr>
</tbody>
</table>

In addition, the run-times for these algorithms can also be determined.
Experimental Results  To show that all possible program configurations for the Peterson algorithm with 4 processors are valid, search using tiered duplicate detection was used with a hash multiple of $HM = 16.17$. Compressed states were used. We have not yet obtained the full experimental results of this run.

Development Time and Performance  Coding this example in our software (to work for any instruction set program) required only 400 lines of code along with 5 hour of programmer time and 5 hours of debugging time. The time was reduced for the MA model due to time already spent developing and debugging the ISA model. This, we believe, illustrates the ease of implementing model checking problems with our software using our API, a goal discussed above.

As with the ISA model, all the optimizations, including symmetry reduction and combining completely local computations, could be performed here as well. This would provide a significant reduction in the size of the search space for this problem, by at least a factor of two.

10 Conclusions

We provide the most comprehensive examination of very large implicit state space enumeration to date. This includes looking at all of the following:

- Surveying existing techniques for large implicit state space enumeration;
- Proposing a new hybrid technique, tiered duplicate detection, that uses both immediate and delayed duplicate detection;
- Introducing a framework under which to analyze the run-time and storage requirements for all these techniques;
• Providing formulas to predict the run-times and storage requirements of all these techniques using this framework;

• Applying these formulas to a set of sample problems in computational group theory and showing the agreement with the experimental time;

• Developing an API and software for large search that allows users to easily plug in their search applications; and

• Suggesting application areas outside of computational group theory where the use of these techniques for very large search can be of use.

By examining large implicit state space enumeration, we encounter a problem where the traditional analysis method, the random access memory complexity model, falls short. We provide an analysis of enumeration techniques that is conscious of both the time spent in the CPU and the storage space required for the technique in question. In doing this, we form a natural hierarchy of techniques where there exists a trade-off between time and space.

This space-time search hierarchy can be overlaid with the memory hierarchy for a particular problem instance to determine which search techniques will be in-core, out-of-core, or simply impossible given the resources available. For the Baby Monster enumeration a technique we developed, tiered duplicate detection, although out-of-core, is still preferred to the other methods. This demonstrates a problem where out-of-core processing is preferred to in-core because of a reduced number of computations. From this, we hope to define a class of problems such as this where considering the use disk in potential solutions may lead to the development of better techniques to solve these problems.

To do this, we analyze very large search taking into account both the CPU time and disk access time. There exists a large number of computations where the bottleneck is due to CPU time, along with a large number where that bottleneck is due to disk access time. Our analysis must account for the size and speed of both RAM and disk as well as the processor speed. From this, we hope define a large class of problems where a performance analysis that is RAM- and disk-conscious can yield better solutions.
References


REFERENCES


A Appendix

A.1 Computational Group Theory Computation Information

Tiered duplicate detection performs multiple passes. Each pass has two phases: the breadth-first search phase (BFS); and the delayed duplicate detection phase (DDD). This section provides pass and computation information for the computational group theory applications described in this thesis.

Pass 0 is the breadth-first search on the master node that begins the computation. All of these computations were performed on a 30 node cluster, except for the computation of Fischer’s group $Fi_{23}$ on the Opportunity cluster, which is a 56 node cluster. The Fischer group $Fi_{23}$ computation and the baby monster group computation used compressed states.

<table>
<thead>
<tr>
<th>Pass</th>
<th>BFS states</th>
<th>BFS time</th>
<th>DDD states</th>
<th>DDD time</th>
<th>Pass states</th>
<th>Pass time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$22.5 \times 10^3$</td>
<td>2.8 min</td>
<td>$1.3 \times 10^9$</td>
<td>4.0 hour</td>
<td>$22.5 \times 10^3$</td>
<td>2.8 min</td>
</tr>
<tr>
<td>1</td>
<td>$6.7 \times 10^9$</td>
<td>1.7 hour</td>
<td>$917.8 \times 10^6$</td>
<td>3.1 hour</td>
<td>$8.0 \times 10^9$</td>
<td>5.7 hour</td>
</tr>
<tr>
<td>2</td>
<td>$1.7 \times 10^9$</td>
<td>1.1 hour</td>
<td>$335.0 \times 10^6$</td>
<td>1.7 hour</td>
<td>$2.8 \times 10^9$</td>
<td>4.2 hour</td>
</tr>
<tr>
<td>3</td>
<td>$506.4 \times 10^6$</td>
<td>1.2 hour</td>
<td>$58.0 \times 10^6$</td>
<td>35 min</td>
<td>$841.4 \times 10^6$</td>
<td>2.9 hour</td>
</tr>
<tr>
<td>4</td>
<td>$83.1 \times 10^6$</td>
<td>43 min</td>
<td>$4.0 \times 10^6$</td>
<td>21 min</td>
<td>$141.1 \times 10^6$</td>
<td>1.3 hour</td>
</tr>
<tr>
<td>5</td>
<td>$5.7 \times 10^6$</td>
<td>30 min</td>
<td>$97.1 \times 10^3$</td>
<td>19 min</td>
<td>$9.7 \times 10^6$</td>
<td>51 min</td>
</tr>
<tr>
<td>6</td>
<td>$144.3 \times 10^3$</td>
<td>19 min</td>
<td>$1.2 \times 10^3$</td>
<td>19 min</td>
<td>$241.4 \times 10^3$</td>
<td>38 min</td>
</tr>
<tr>
<td>7</td>
<td>$1.2 \times 10^3$</td>
<td>9 min</td>
<td>$41$</td>
<td>13 min</td>
<td>$2.4 \times 10^3$</td>
<td>28 min</td>
</tr>
<tr>
<td>8</td>
<td>63</td>
<td>16 min</td>
<td>$13$</td>
<td>13 min</td>
<td>104</td>
<td>29 min</td>
</tr>
<tr>
<td>9</td>
<td>34</td>
<td>16 min</td>
<td>$47$</td>
<td>29 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>16 min</td>
<td>4</td>
<td>27 min</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total $9.0 \times 10^9$ 6.5 hour $2.6 \times 10^9$ 10.8 hour $11.7 \times 10^9$ 17.3 hour
### Fischer's Group $F_{i23}$: Teraccluster

<table>
<thead>
<tr>
<th>Pass</th>
<th>BFS states</th>
<th>BFS time</th>
<th>DDD states</th>
<th>DDD time</th>
<th>Pass states</th>
<th>Pass time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>632</td>
<td>4 sec</td>
<td></td>
<td></td>
<td>632</td>
<td>4 sec</td>
</tr>
<tr>
<td>1</td>
<td>$11.3 \times 10^9$</td>
<td>5.5 hour</td>
<td>$347.3 \times 10^6$</td>
<td>14.0 hour</td>
<td>$11.6 \times 10^9$</td>
<td>19.5 hour</td>
</tr>
<tr>
<td>2</td>
<td>$37.1 \times 10^6$</td>
<td>13 min</td>
<td>$2.3 \times 10^6$</td>
<td>2.9 hour</td>
<td>$39.4 \times 10^6$</td>
<td>3.1 hour</td>
</tr>
<tr>
<td>3</td>
<td>$21.0 \times 10^3$</td>
<td>6 sec</td>
<td>$1.3 \times 10^3$</td>
<td>2.0 hour</td>
<td>$22.3 \times 10^3$</td>
<td>2.0 hour</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>0 sec</td>
<td>0</td>
<td>2.0 hour</td>
<td>5</td>
<td>2.0 hour</td>
</tr>
<tr>
<td>Total</td>
<td>$11.3 \times 10^9$</td>
<td>5.7 hour</td>
<td>$349.6 \times 10^6$</td>
<td>20.9 hour</td>
<td>$11.7 \times 10^9$</td>
<td>26.6 hour</td>
</tr>
</tbody>
</table>

### Baby Monster Group

<table>
<thead>
<tr>
<th>Pass</th>
<th>BFS states</th>
<th>BFS time</th>
<th>DDD states</th>
<th>DDD time</th>
<th>Pass states</th>
<th>Pass time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$63.4 \times 10^3$</td>
<td>31 sec</td>
<td></td>
<td></td>
<td>$63.4 \times 10^3$</td>
<td>31 sec</td>
</tr>
<tr>
<td>1</td>
<td>$13.1 \times 10^9$</td>
<td>4.5 day</td>
<td>$400.3 \times 10^6$</td>
<td>1.4 day</td>
<td>$11.6 \times 10^9$</td>
<td>19.5 hour</td>
</tr>
<tr>
<td>2</td>
<td>$41.5 \times 10^6$</td>
<td>3.5 hour</td>
<td>$2.6 \times 10^6$</td>
<td>1.6 hour</td>
<td>$39.4 \times 10^6$</td>
<td>3.1 hour</td>
</tr>
<tr>
<td>3</td>
<td>$21.6 \times 10^3$</td>
<td>1.2 min</td>
<td>$1.4 \times 10^3$</td>
<td>1.3 hour</td>
<td>$22.3 \times 10^3$</td>
<td>2.0 hour</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>1 sec</td>
<td>0</td>
<td>1.3 hour</td>
<td>5</td>
<td>2.0 hour</td>
</tr>
<tr>
<td>Total</td>
<td>$13.2 \times 10^9$</td>
<td>4.7 day</td>
<td>$402.9 \times 10^6$</td>
<td>1.4 day</td>
<td>$13.5 \times 10^9$</td>
<td>6.1 day</td>
</tr>
</tbody>
</table>

### A.2 Explicit State Model Checking Computation Information

Tiered duplicate detection performs multiple passes. Each pass has two phases: the breadth-first search phase (BFS); and the delayed duplicate detection phase (DDD). This section provides pass and computation information for the explicit state model checking of the N-Peterson application. Pass 0 is the breadth-first search on the master node that begins the computation. All of these computations were performed on a 30 node cluster.
<table>
<thead>
<tr>
<th>Pass</th>
<th>BFS states</th>
<th>BFS time</th>
<th>DDD states</th>
<th>DDD time</th>
<th>Pass states</th>
<th>Pass time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11.9 × 10^6</td>
<td>1.25 min</td>
<td>11.9 × 10^6</td>
<td>1.25 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>499.4 × 10^6</td>
<td>4 min</td>
<td>234.3 × 10^3</td>
<td>5.25 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>753</td>
<td>30 sec</td>
<td>1</td>
<td>11 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1 sec</td>
<td>0</td>
<td>0 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>511.3 × 10^6</td>
<td>5.75 min</td>
<td>234.3 × 10^3</td>
<td>5.5 min</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pass</th>
<th>BFS states</th>
<th>BFS time</th>
<th>DDD states</th>
<th>DDD time</th>
<th>Pass states</th>
<th>Pass time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>500.0 × 10^3</td>
<td>2.16 min</td>
<td>500.0 × 10^3</td>
<td>2.16 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>15.4 × 10^9</td>
<td>2.42 hours</td>
<td>258.5 × 10^6</td>
<td>7.5 hours</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>325.1 × 10^3</td>
<td>2.75 min</td>
<td>11.2 × 10^3</td>
<td>50 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12.6 × 10^3</td>
<td>10 sec</td>
<td>431</td>
<td>46 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1.2 × 10^3</td>
<td>5 sec</td>
<td>51</td>
<td>38 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>162</td>
<td>3 sec</td>
<td>2</td>
<td>35 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>2 sec</td>
<td>0</td>
<td>4 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>15.5 × 10^9</td>
<td>2.5 hours</td>
<td>258.6 × 10^6</td>
<td>10.4 hours</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pass</th>
<th>BFS states</th>
<th>BFS time</th>
<th>DDD states</th>
<th>DDD time</th>
<th>Pass states</th>
<th>Pass time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>72.7 × 10^3</td>
<td>3.5 min</td>
<td>72.7 × 10^3</td>
<td>3.5 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2.0 × 10^6</td>
<td>2.3 min</td>
<td>2.0 × 10^6</td>
<td>2.3 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0 sec</td>
<td>0</td>
<td>1 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2.0 × 10^6</td>
<td>5.8 min</td>
<td>4</td>
<td>2 sec</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithm Model: 7-Peterson
HM: 1094, BFS block: 333 MB, DDD block: 1 GB

Algorithm Model: 8-Peterson
HM: 30, BFS block: 16 MB, DDD block: 1 GB

Instruction Set Architecture Model: 3-Peterson
HM: 347798, BFS block: 333 MB, DDD block: 1 GB
### Instruction Set Architecture Model: 4-Peterson

HM: 581, BFS block: 333 MB, DDD block: 1 GB

<table>
<thead>
<tr>
<th>Pass</th>
<th>BFS states</th>
<th>BFS time</th>
<th>DDD states</th>
<th>DDD time</th>
<th>Pass states</th>
<th>Pass time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>61.9 \times 10^3</td>
<td>5.85 min</td>
<td></td>
<td></td>
<td>61.9 \times 10^3</td>
<td>5.85 min</td>
</tr>
<tr>
<td>1</td>
<td>1.2 \times 10^6</td>
<td>16.13 hours</td>
<td>1.1 \times 10^6</td>
<td>1.9 hours</td>
<td>1.2 \times 10^6</td>
<td>17.03 hours</td>
</tr>
<tr>
<td>2</td>
<td>2.3 \times 10^6</td>
<td>4.9 min</td>
<td>3.9 \times 10^3</td>
<td>22.8 min</td>
<td>2.3 \times 10^6</td>
<td>27.7 min</td>
</tr>
<tr>
<td>3</td>
<td>11.2 \times 10^3</td>
<td>30 sec</td>
<td>18</td>
<td>22.5 min</td>
<td>11.2 \times 10^3</td>
<td>23 min</td>
</tr>
<tr>
<td>4</td>
<td>183</td>
<td>7 sec</td>
<td>2</td>
<td>21 min</td>
<td>185</td>
<td>21.1 min</td>
</tr>
<tr>
<td>5</td>
<td>34</td>
<td>2 sec</td>
<td>0</td>
<td>15.5 min</td>
<td>34</td>
<td>15.5 min</td>
</tr>
<tr>
<td>Total</td>
<td>1.2 \times 10^9</td>
<td>16.3 hours</td>
<td>1.1 \times 10^6</td>
<td>3.2 hours</td>
<td>1.2 \times 10^9</td>
<td>19.5 hours</td>
</tr>
</tbody>
</table>

### Machine Architecture Model: 3-Peterson

HM: 27007, BFS block: 333 MB, DDD block: 1 GB

<table>
<thead>
<tr>
<th>Pass</th>
<th>BFS states</th>
<th>BFS time</th>
<th>DDD states</th>
<th>DDD time</th>
<th>Pass states</th>
<th>Pass time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>80.6 \times 10^3</td>
<td>3.13 min</td>
<td></td>
<td></td>
<td>80.6 \times 10^3</td>
<td>3.13 min</td>
</tr>
<tr>
<td>1</td>
<td>2.6 \times 10^6</td>
<td>7.5 min</td>
<td>507</td>
<td>1.25 min</td>
<td>2.6 \times 10^6</td>
<td>8.75 min</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0 sec</td>
<td>0</td>
<td>1 sec</td>
<td>0</td>
<td>1 sec</td>
</tr>
<tr>
<td>Total</td>
<td>2.6 \times 10^6</td>
<td>10.63 min</td>
<td>507</td>
<td>1.25 min</td>
<td>2.6 \times 10^6</td>
<td>11.88 min</td>
</tr>
</tbody>
</table>