ABSTRACT

Solving the F/A-18 Mission Computer Virtual Memory Problem

Adam L. Sealey, M.S.

Chairperson: David B. Sturgill, Ph.D.

The F/A-18 has a mission computer that requires physical memory be mapped into a very limited amount of virtual memory space. As requirements for this aircraft have expanded, the mission computer must perform increasingly complex computations without using any additional virtual memory. The elements required by the computations must be assigned physical and logical addresses in a manner that satisfies a variety of constraints imposed by the system. Determining these addresses is an NP-Complete problem, to which the only known way of finding a solution is exponential-time search. We present a formalization and analysis of the problem, along with an analysis of the feasibility of performing search. Additionally, we explore a variety of incomplete search techniques with the goal of producing an acceptable mapping of elements to addresses that satisfies all constraints within a reasonable amount of time.

Solving the F/A-18 Mission Computer Virtual Memory Problem

by

Adam L. Sealey, B.S.

A Thesis

Approved by the Department of Computer Science

Donald L. Gaitros, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science

Approved by the Thesis Committee

David B. Sturgill, Ph.D., Chairperson

Gregory J. Hamerly, Ph.D.

Michael W. Thompson, Ph.D.

Accepted by the Graduate School
August 2009

J. Larry Lyon, Ph.D., Dean

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I first acknowledge God, who has given me the grace to finish this journey. He is my shining light and redeemer.

I would like to thank Dr. David Sturgill for his wisdom, advice, direction, and patience during this long journey. I also acknowledge the support and direction of Dr. Russell Duren and Dr. Michael Thompson who presented the problem and provided necessary details to perform the analysis. Dr. Greg Hamerly has been a great resource for the formal portions of my arguments, as well as general guidance. Additionally, I woud like to thank Mike Hutcheson and Baylor University Academic and Research Computing Services for providing access to the High Performance Computing Cluster which allowed me to run the performance tests in parallel. Finally, I thank my family and friends for their support and encouragement along this journey.

DEDICATION

For my wife, Lisa, who has continually supported me through this process. Her unwavering commitment and stability through the tough times is a testament to her character and devotion.

For her worth is far above jewels.
The heart of her husband trusts in her,
and he will have no lack of gain.
Proverbs 31:10-11

In honor of my grandfather, Gerald Lee Sealey.

CHAPTER ONE

Introduction

The Boeing F/A-18 Hornet is an advanced, multipurpose warplane used by the United States and its allies in a variety of missions around the world. It is a very versatile platform capable of many different types of operations including air-to-air, close air support, maritime strike, in-flight refueling, reconnaissance, and forward air control. These varying mission types require the plane to carry a wide variety of payloads in addition to a very flexible flight control mission computer (Naval Air Systems Command, 2009)(Carder and Guse, 2009).

An integral component of the F/A-18's continued success is an extremely robust mission computer that controls everything from flight plan and objectives to communication and target acquisition. This system was designed to be completely fault tolerant, with redundant hardware and software at every point. If any component of the mission computer fails during a mission, the redundant component takes control in a seamless transition. A significant amount of both time and money have been invested in this hardware, specifically the in CPU and logical components.

As its objectives have become more complex and the mission computer has been required to perform more tasks, upgrades have been made to the mission computer system to expand its functionality and extend the service life of the aircraft. These upgrades have focused on the most cost-effective portions of the computer, specifically the storage system. Since the CPU and logical components are the most complex and expensive portion of the system, they cannot be replaced or upgraded cost-effectively. Unfortunately, this also includes the memory addressing architecture, which is limited to 64 kilobytes of concurrently addressable logical memory. Thus, the system has plenty of physical memory, but is only able to access a relatively small portion of it at any given time, limiting the usefulness of the memory.

The mission computers are based on a time-sliced architecture, where a series of *tasks* are executed in order and then repeated. Each task has 64 kilobytes of virtual memory available for code storage, and another 64 kilobytes of virtual memory available for data storage. Each type of virtual memory is represented by a page table consisting of 64 *slots*. Each slot in the page table is a reference to a one-kilobyte page of physical memory. Upon initialization, the task loads pages into its page table and executes.

Each task requires certain *data elements* for successful execution. These elements must be mapped to memory by being located on a page that is referenced in the page table of the task. The elements range in size from one byte to more than thirty kilobytes, and each element has a potentially unique set of tasks that require it. Elements can only occupy one location in physical and logical memory due to the compiler and verification techniques employed. This restriction makes it extremely important to decide which page to assign each element in order to ensure all required elements are available when needed, without exceeding virtual memory capacity. Since there are only 64 page slots available for each task, most tasks require multiple elements be assigned to the same page. This sharing of pages allows tasks to access all required elements, but it also complicates the problem.

Each change to code on the mission computer defines a new *instance* of the mapping problem. We define a problem instance as all the pertinent information about the problem, including the number of tasks and elements, and requirements between tasks and elements. For example, in a new version of code, a certain task might need access to a data element it previously did not need. This slight change in requirements can completely change the problem and invalidate any solutions that had been found for a previous problem instance. Each problem instance needs to be evaluated separately, since a solution for one problem instance might not work for another, even if the problems are very similar.

When a task initializes, it needs to load pages into its page table in order to execute. A *mapping* is a record of how each page table is constructed, including what pages each element resides on and where the pages are loaded into the page table for each task. Each problem instance has many different possible mappings, each with its own ordering of pages and element assignments. The details of the hardware implementation impose *validation constraints* on where some elements can be placed in the mapping. Some examples are that elements may not overlap in physical memory, pages must be loaded into the same slot for all tasks that require the page, and certain elements cannot share the same page. We consider a mapping to be *valid* if it assigns the elements and pages in such a way that it satisfies all the validation constraints and no page table exceeds the virtual memory capacity. Conversely, we would consider a mapping to be *invalid* if its assignments break any constraints or oversubscribes the virtual memory capacity in any of the page tables. Finding a valid mapping for a problem instance is the ultimate goal of MEM-MAP.

1.1 Mapping Problem Overview

Currently, typical problem instances are quite large, consisting of hundreds of elements being mapped to over 60 tasks. Candidate mappings are difficult to visualize, which makes them hard to validate by hand. A less complicated example helps illustrate some of the concepts of the problem without getting bogged down in details. Figure 1.1 shows the an example problem instance, with Figures 1.2 and 1.3 showing two attempts at mapping the example instance.

This problem instance has four data elements (d_1, d_2, d_3, d_4) that need to be mapped to four tasks (Task 1, Task 2, Task 3, Task 4). For these examples, assume there are only four logical slots available for storing pages, which can each store 1024

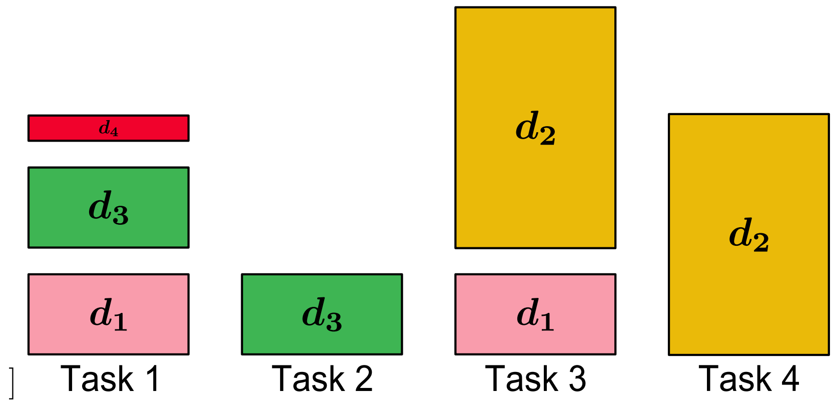


Figure 1.1: Task requirements for an example problem instance. Tasks are represented by columns, and elements are represented by colored blocks and placed into the columns of the tasks that require them. The height of the element block indicates the size of the element.

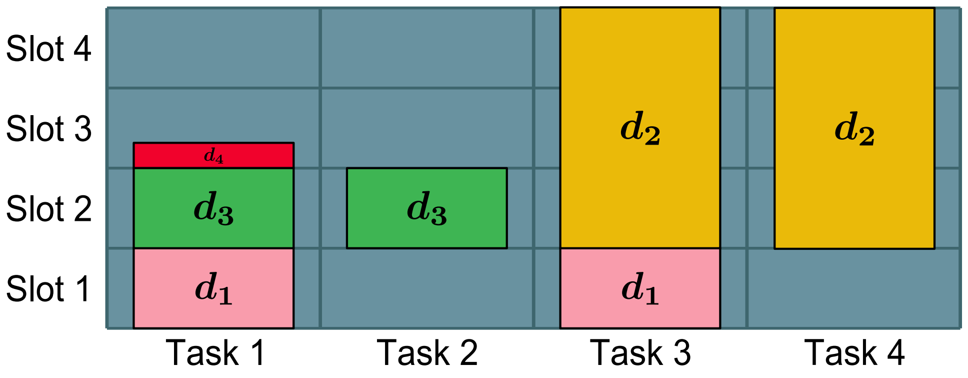


Figure 1.2: The elements for the problem instance are successfully mapped into the page table space. In this example, each task has a four-slot page table to store its required elements.

bytes. These slots are shown as rows in Figures 1.2 and 1.3. Tasks are displayed as vertical columns, with the elements they require being placed in their appropriate column. The height of each element indicates its size in bytes. It is clear that d_1 and d_3 are 1024 bytes in size, since they completely fill one page. The significantly larger d_2 is 3072 bytes in size since it completely fills 3 full pages. Element d_4 is a much smaller element, only requiring 64 bytes, but it still plays an important part in the mapping.

Figure 1.2 is a clever mapping of the elements to the slots. All elements are assigned to slots in the page table in just the right way that allows each task to access its required elements without running off the page table. There are a couple key things to note in this figure. First, the placement of d_2 in Slots 2-4 and d_3 in Slot 2 dictate where they are placed in the other tasks, since elements can only be mapped to one physical and one logical address. This forces Task 2, which only requires element d_3, to place d_3 in Slot 2, even thought Slot 1 is vacant. This vacant space is limited to the one page in Slot 1, since Slot 2 already has a page assigned to it in Task 2.

This limited space is an example of *fragmentation* of the available free space, in which free space is broken up into smaller blocks, reducing or eliminating its ability to store large blocks. A small element with less than 1024 bytes, or a group of small elements totaling in size to less than 1024 bytes, could be mapped into this space, but any larger element would be forced to start after d_3. The fragmentation is a major concern because it limits the effectiveness of the free space. Another important thing to note is that Task 3 has no room to spare; its elements fit perfectly in the table, but a slight modification to their placement could cause some of the elements to fall off the page table, rendering the mapping invalid. There are in fact only two ways for Task 3 to order its elements: either d_1 comes before d_2 in Slot 1, or d_1 comes after d_2 in Slot 4. Assigning d_1 to any other slot would cause Task 3 to overrun the table.

To illustrate some of the more complex issues with the problem, Figure 1.3 shows the same problem instance being mapped with a slight variation. Instead of placing d_4 in Slot 3 after d_3, it is placed at the beginning of Slot 1. This minor change has major consequences for the other elements in the mapping. Since elements cannot overlap in logical space, d_1 is forced to start later in the page, thus spilling over into another page. Since Task 3 is the most constrained task, this modification of the location of d_1 has a cascading effect, forcing d_2 to fall partially off the table. The key

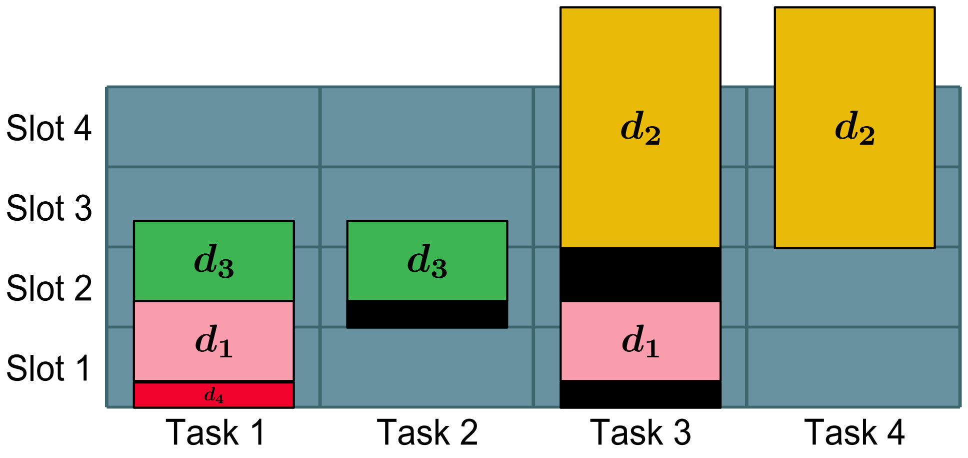


Figure 1.3: An example of an unsuccessful mapping attempt that results from a very minor modification in the ordering of the elements. This forces d_2 to require more space than is available in the page table.

difference in this example is that the page loaded into Slot 2 in Task 2 and 3 now contains elements that are not strictly required by each of the tasks. The beginning of the page holds the end of d_1, which is not a required element for Task 2. Task 3 requires d_1 so it must load the page, but in addition it loads the first part of d_3, which it does not need. We call this space that contains unrequired elements *polluted* space in the task. Pollution occurs when elements with differing task requirements are mapped to the same page. Whenever that page is loaded into a slot by a task, any elements in the page that are not required by the task are occupying space that could otherwise be used by other elements that are required in the task. In this case, pollution could have been reduced by pushing d_3 back to Slot 3, although d_4 would continue to pollute Slot 1. Reducing the pollution does not solve all the problems with the mapping, though, since such a modification still would not enable all the elements to fit. This change would allow d_2 to share the page in Slot 2 with d_1, but it would still not fit in the page table. Pollution is undesirable, but it is also unavoidable. The key to dealing with it is to minimize polluted space on the most constrained tasks.

Realistic problem instances typically include elements similar to d_4 that are not perfectly sized for their pages. Many of the elements are much smaller than a page, allowing multiple elements to be stored on the same page, while other elements are much larger and span multiple pages. This mixture of large and small elements, along with their varying task requirements, makes creating a valid mapping very difficult. Fragmentation can significantly reduce the effectiveness of vacant space, while pollution can quickly reduce the available space for constrained tasks. Subtleties of element positioning can have huge effects on other elements and, eventually, the validity of a mapping. Figures 1.2 and 1.3 contain the same elements, but a seemingly minor change in positioning can result in the mapping becoming invalid by oversubscribing the virtual memory of the page table.

1.2 Automated Validation

Since verifying that the elements are assigned to positions that will satisfy all required tasks is a difficult job, the Department of Defense (DOD) created a set of programs to work with mappings. These tools were originally designed to parse a problem instance and automatically generate a valid mapping. Eventually, the complexity of the problem instances reached the point where the programs were no longer able to create valid mappings. Fortunately, the programs are still able to validate mappings and serve as the official validators for the system. Whenever a change to the mapping is required via a code or hardware change, a human expert must manually inspect the mapping and find a location for the new or changed element. Due to the extremely complex nature of the mappings, this can take months, and only a handful of people in the world are capable of accomplishing this process manually. Once the mapping is hand modified, it is run through the validation programs before being loaded onto the aircrafts' mission computer.

The current method of finding mappings that relies on human assistance is not a scalable solution. The people with the experience necessary to find mappings are approaching retirement, requiring the extensive knowledge necessary to be communicated to new employees. Developing an algorithm and computer implementation that can efficiently find a valid mapping for a problem instance would allow the process to be completely automated, reducing time and cost required for code upgrades.

1.3 Mapping and Computational Complexity

The size and complexity of the problem causes the number of possible mappings, valid or not, to be extremely high for any given problem instance. A reference instance of the problem has 61 tasks and 943 elements, with almost 9000 requirements linking them. Each task then has 64 slots to store pages that can each hold 1024 bytes. Since each element could start at any byte in a page, and that page could be loaded into any slot for any task, there are 4 million possible starting locations for each element: 61 Tasks $\times$ 64 Slots/Task $\times$ 1024 Bytes/Slot $= 3,997,696$ possible starting locations. As shown in the previous example, even slight modifications to the ordering of elements in memory can cause major issues.

Ideally, an algorithm could be designed and implemented that would be able to find a valid mapping for a problem instance, if one exists, in a reasonable amount of time. Formally, a reasonable amount of time is defined as time polynomial in the size of the input. These problems are said to have *polynomial time complexity* (Sipser, 1997).

Unfortunately, this problem is much harder than that. Formal analysis reveals an equivalence to other well known optimization problems in computer science that are very hard, known as $\mathcal{NP}$-Hard problems. These problems are so hard, in fact, that there are no known algorithms that can solve them in polynomial time, assuming NP $\neq \mathcal{P}$. Current algorithms that can solve these problems need time exponential in the size of the problem instance, and are said to have *exponential time complexity* (Sipser, 1997). As the problem size increases, the time required to find a solution quickly exceeds a reasonable amount of time.

1.4 Search-Based Solvers

The complexity associated with completely searching the entire search space motivates the examination of alternate search procedures that can potentially provide solutions in less time. These techniques are referred to as *incomplete search* methods, which only explore a portion of the search space. Incomplete search methods apply heuristics to the problem instance to reduce the size of the search space and focus the search on mappings that are most likely to be valid for the problem instance. These methods can potentially find a valid solution quickly, but they don't systematically search the entire space. This means they can't guarantee that a valid solution does not exist if they don't find one, or that they'll find the solution if one exists. Also, lack of systematicity means that they may search the same group of mappings multiple times, which is wasted effort.

There are a variety of incomplete search methods. This work examines a few of the most applicable incomplete search methods, including variations on hill climbing search, genetic search, and heuristically-guided depth-first search. These incomplete search methods are readily adapted to the problem definition, and provide a good basis for creating a system that is capable of automatically finding solutions to problem instances. A reference implementation is provided that facilitates performance comparisons between each of the approaches. These comparative tests provide insight into the performance qualities of each of the variations, as well as provide direction for further modifications and improvements.

1.5 Overview

The remainder of this document will investigate a variety of approaches to finding a valid mapping for a given problem instance. Chapter 2 explores the existing body of work, including previous attempts at visualizing the problem, as well as standard algorithms and techniques that are applicable to similar problems. Chapter 3 begins by laying the foundation for the work by formally defining the problem, proving that it is $\mathcal{NP}$-Hard, and developing a set of refinements to reduce the size of its search space. This is followed with Chapter 4 offering an explanation of the techniques that are implemented as part of the work, as well as the custom modifications that are made to apply standard algorithms to the problem. Chapter 5 provides the exploratory results of our experiments. Finally, Chapter 6 offers concluding thoughts and suggestions for future work in this space.

CHAPTER TWO

Related Work

MEM-MAP is an interesting problem which has real world applications for the U.S. military forces. Some attempts have been made to apply computer algorithms to solve the problem, but there are no current implementations that are capable of producing a solution in a reasonable amount of time without human assistance. Since MEM-MAP is a $\mathcal{NP}$-Complete problem, some form of search will be required to find a solution. An examination of established search techniques provides the background necessary to design procedures that are tailored to this problem.

2.1 Search

Search problems typically have multiple candidates for solutions but only a limited number actually solve the problem. Finding one of these solutions in the veritable sea of possibilities requires a disciplined approach to examining the candidates. The set of all candidates is called the *search space* of the problem. For MEM-MAP, we define the *Basic Example* to be a search space defined as all possible combined assignments of starting logical addresses to elements. The valid mappings are nodes in the search space that we must identify. There are a variety of ways to explore the search space of the problem, called *search procedures.* Each search procedure has strengths and weaknesses unique to its approach (Luger and Stubblefield, 1998).

With non-trivial search spaces, such as the one for this problem, systematically exploring the space is valuable. A systematic approach guarantees that a solution will be found if one exists, and each candidate will be examined at most once during the search. The first guarantee is important because search is an expensive

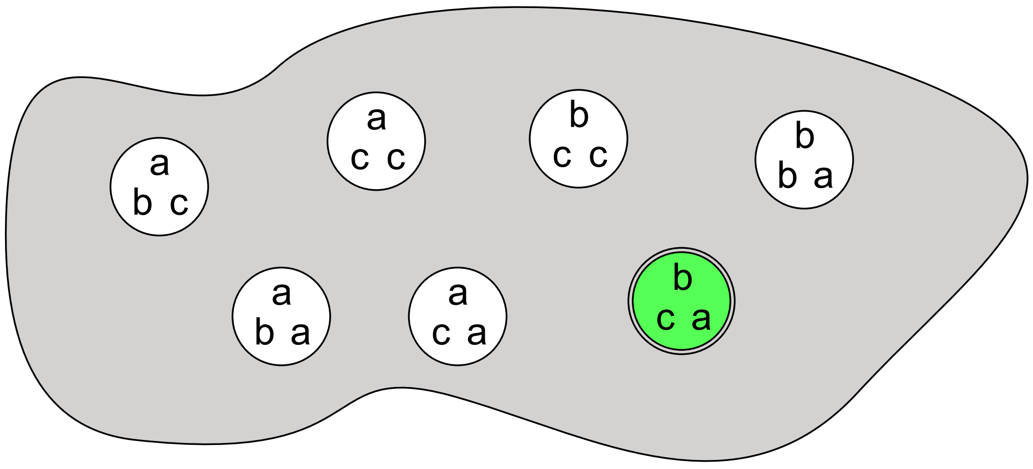


Figure 2.1: The Basic Example search space with no structure imposed upon it. The green node represents a solution.

operation, and it is important that a definitive answer is generated after spending the required time. The second guarantee is a constraint on wasted time, because in a well organized search space, any effort spent re-examining candidates that have already been examined is wasted.

Search relies on the search space having structure, which provides relationships between nodes in the space. These relationships are the paths which the search traverses during execution. Without these relationships, search diverges to randomly sampling the space, with no organization available to guide further examinations. Figure 2.1 shows the Basic Example search space with no structure imposed. Beginning a search with any node in the space does not lead to other nodes in the space.

There are two primary structures used for organizing large search spaces: a search space that only consists of fully defined candidates, and a search space that consists of partially formed candidates. These structures each have advantages and disadvantages, as well as specific search procedures that benefit from particular structure methods. To facilitate describing these structures and relevant search procedures, it is helpful to use the Basic Example which defines how some of the concepts in MEM-MAP can be represented. In Figures 2.1, 2.2, and 2.3, candidates for the Basic Example are represented by circles, with letters inside the circle representing the order that elements occur in the memory read top to bottom, left to right. In addition, a candidate is considered to be a solution if it has assigned a starting logical and physical address to all elements in the problem instance, and all the validation constraints are satisfied. These solution candidates are represented by a green circle. Finally, a partial candidate does not define a starting position for some or all of the elements in the problem. Partial candidates are only found in Figure 2.3.

Applying a structure to the search space facilitates considering candidates both explicitly and implicitly. An *explicit* representation of a candidate is the specific assignment of starting logical addresses to elements, and all the data structures that are associated with the assignment. Unstructured search spaces deal with explicitly represented candidates. Imposing structure on the search space allows candidates to be represented *implicitly* by virtue of their relative location in the search space according to the structure. This allows a search to be defined as explicitly representing a candidate and testing if that candidate is a solution. Only the current candidate needs be explicitly represented at any given time, because the structure provides a path for choosing other candidates to explicitly represent and examine.

2.1.1 Full-Candidate Representation

The first, most straightforward approach for representing the search space is to consider all nodes in the search space as a candidate solution that needs to be investigated. Search methods that work through these spaces need to examine each node and test its validity until a valid solution candidate is found. The strength of this representation is that it only has as many nodes as there are candidates. While the space of all candidates is extremely large and needs to be represented implicitly,

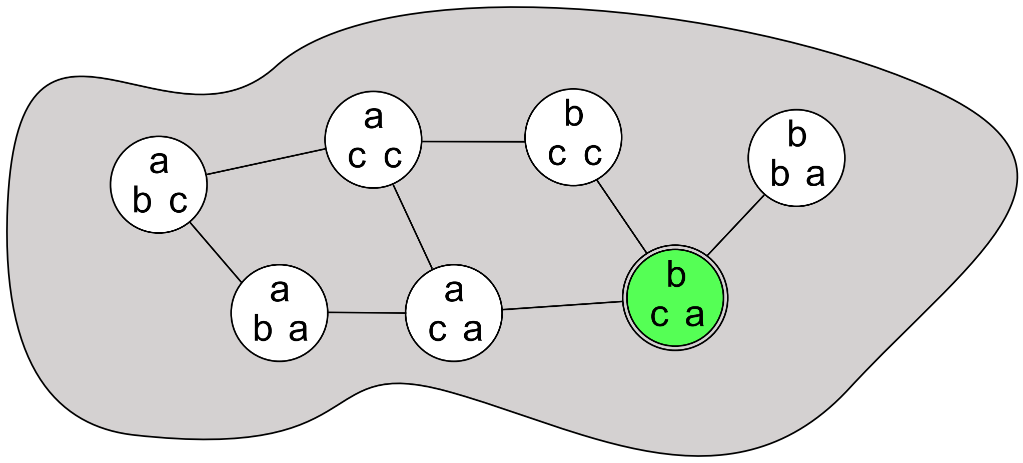


Figure 2.2: The Basic Example search space with a graph structure imposed upon it. This graph structure is one way to facilitate systematic exploration of the search space. Each node in the graph is a fully formed candidate.

search algorithms using this representation can immediately being examining actual candidates.

The organization method used for this representation is a *Graph*, as presented in Figure 2.2. Each candidate in the search space is represented by a node in a large, implicit graph. The neighboring nodes are generated by *expanding* the current node. The expand operation produces a set of neighbors that are connected via an edge to the current node. These neighbors are other candidates that are the result of a minor change to the current candidate. A graph-based search procedure might start at an arbitrary node, test its validity as a solution, then expand the node. It would continue this testing and expanding until a solution is found, hopefully after only a few expansions (Luger and Stubblefield, 1998).

A graph-based approach can easily be applied to the mapping problem. Using the Basic Example, a simple graph-based search procedure could start at a node, check its validity, then generate a neighbor by making a small change to the starting logical address of an element in the original candidate, and continue to test and generate until a valid solution is found. Each neighbor that is generated is part of the *neighbor set* of the node.

The graph-based approach is a simple organization method for the search space. However, it does not have many features built in that facilitate a systematic search. Since the graph will very likely have many loops, ensuring that each node is examined at most once requires using additional data structures to keep track of visited nodes. The large size of the search space renders the maintenance of this complete list infeasible. There are also difficulties associated with ensuring that all possible solution candidates are examined, if the graph were disconnected, for example.

Despite the difficulties associated with performing a systematic search on a graph-based organization, there are other, non-systematic search procedures that perform well in the graph environment. These approaches apply heuristics in an attempt to explore the areas of the search space that are most likely to contain a valid solution. For example, a heuristic-based search may generate all the neighbors for a node and test them, then chose the best neighbor, as determined by the heuristic, for continued exploration.

2.1.2 Partial-Candidate Representation

An alternate representation is to consider the space as a collection of partially constructed candidates, with only a subset of elements assigned to logical addresses. This assignment of an element to a logical address is called a *commitment*. Search methods that use this representation look for a solution by making additional commitments until all elements are assigned a logical address, which is a full candidate.

The organization method used for the partial-candidate representation is a *tree*. Each candidate is represented as a leaf node in a very large tree, with a series of internal nodes forming the path from the root node to the leaves. The structure

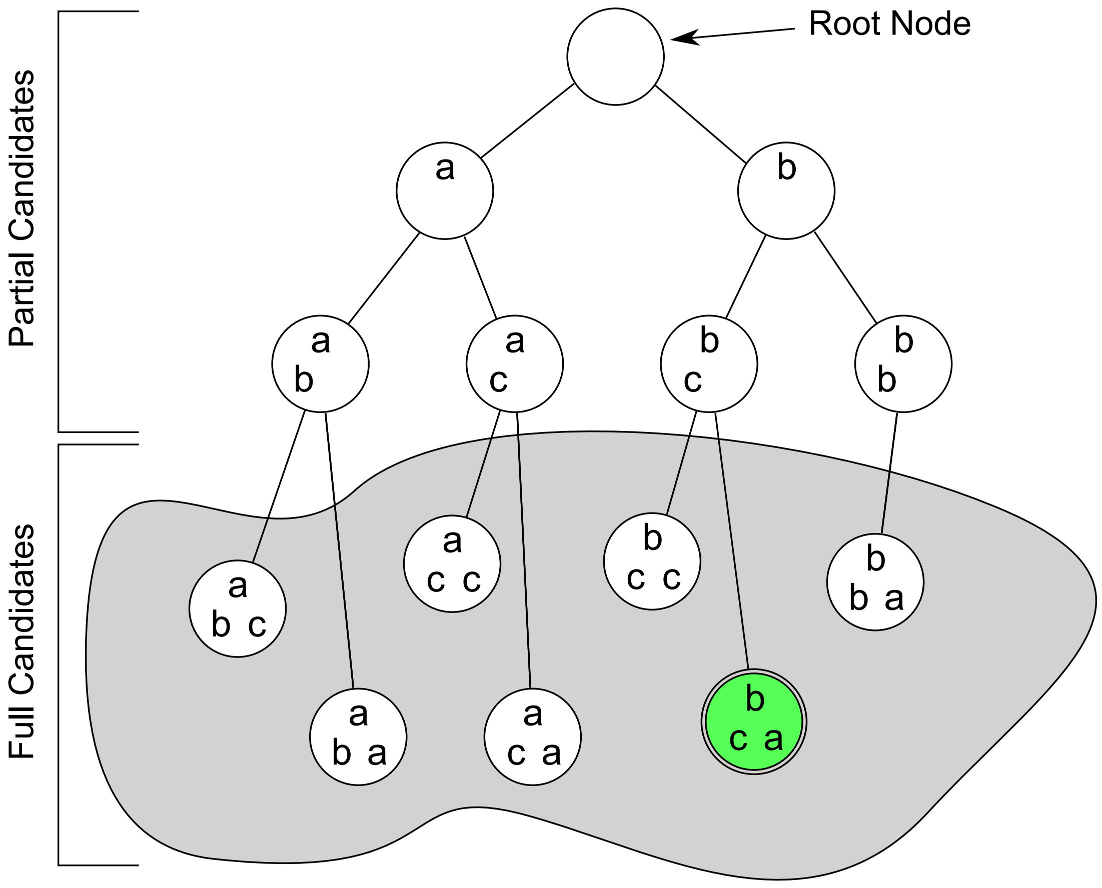


Figure 2.3: The Basic Example search space with a tree structure imposed upon it. The root of the tree is at the top, which has made no commitments. At each level of the tree, more commitments are made until a fully formed candidate is formed, which are the candidates in the search space.

of the tree is usually guided by the principle of *least commitment*, in which nodes are a single step on the path towards constructing a candidate (Weld, 1994). The root node starts by not making any commitments at all. Starting with the root node, each internal node is expanded to create children that represent all possible commitments that can be made in addition to the node's current commitments. These commitments continue to be expanded until all necessary commitments have been made, resulting in a candidate. A tree-based search procedure would start at the root node, expanding and examining all children nodes until it finds a solution candidate, or has explored all candidates in the space (Luger and Stubblefield, 1998).

The mapping problem can be understood as a tree-based search problem. Using the Basic Example, each commitment made in the internal node could be an assignment of an element to a starting logical address. The root node for the tree is a partial candidate that makes no commitments at all. As each internal node adds a commitment to the partial candidate, it assigns a starting logical address to an element that did not previously have a starting logical address. Once all elements have been assigned a starting logical address, the node is a candidate that may be tested for validity.

A tree-based characterization of the search space lends itself well to a systematic search. By definition, a tree does not contain any loops, preventing a search procedure from examining the same candidate multiple times. It also facilitates examining all candidates, because the root node starts with no commitments. While each node has many children nodes, it is clear that there will always be a path from the root to any conceivable candidate. In addition, the concept of least commitment provides a good basis for performing prune operations, which can significantly reduce the size of the effective search space (Cormen et al., 2003).

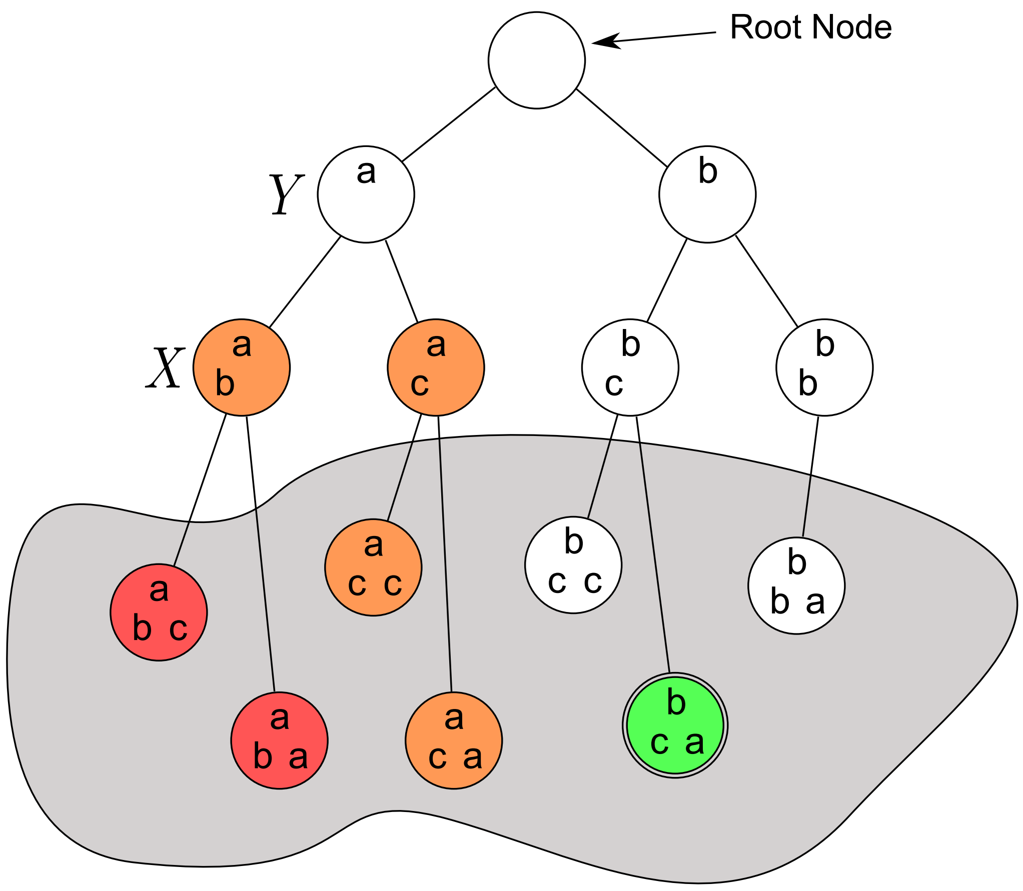


Figure 2.4: The effects of two different pruning choices are shown via the red and orange nodes. If node X already violates one of the solution requirements, the red nodes are not examined. If it were determined that node Y violates one of the solution requirements, neither the orange nor the red nodes are examined. Simply moving the pruning up one level in the tree has a significant effect on the number of nodes pruned.

2.1.3 Pruning

One of the key advantages of using a tree-based method for organizing the search space is that it enables the search procedure to *prune* the search space as it goes. Pruning is the act of eliminating a portion of the search space that is known to contain no solutions. Figure 2.4 shows how pruning can be applied at various locations in a tree to reduce the number of nodes that are examined. Notice that pruning higher in the tree results in a more significant reduction in the size of the examined portion of the search space.

In a partial candidate representation, if an assignment of a starting logical address to an element in an internal node causes any subsequent assignments of logical addresses to elements to result in an invalid candidate, then none of the children nodes or leaves need be examined. This could happen if the assignment of the logical address results in a physical address being assigned such that elements overlap in physical memory, for example.

A *pruning function* applies logic that identifies which commitments ultimately result in invalid children. These functions can be very powerful in speeding up search, but they can be difficult to create. The goal for the pruning function is to be able to prune as early as possible in the tree. This maximizes the reduction in search space of each successful prune. The problem is that there typically is not enough information at the early nodes in the tree to know if the decision made in the node was a poor one. Thus we are left with pruning functions that operate in the middle to lower nodes of the tree, and are less effective (Reed, 1993).

2.1.4 Systematic Search Methods

Regardless of the representation chosen, there are a variety of approaches to examining the search space in a systematic manner. The most common approaches are broadly classified as *Breadth-First Search (BFS)* and *Depth-First Search (DFS)*. When the tree-based method is used, these approaches start at the root of the tree and move down. In a graph-based method, an arbitrary starting point is selected, and the search proceeds from there (Cormen et al., 2003).

2.1.4.1 *Breadth-First search* Breadth-First Search (BFS) examines the search space in a level-by-level fashion, where all nodes in a level must be inspected before any nodes in the next level. It relies on a First-In-First-Out (FIFO) queue for maintaining the set of generated nodes. It maintains a queue which contains nodes that need to be examined, called the *open queue*. It also maintains a list of nodes that have already been examined, called the *closed list*. BFS initializes by pushing the starting node onto the open queue. It then proceeds by popping a node from the open queue and examining it. If the node is a valid solution, then the search is complete. Otherwise, it expands the node to generate all of the its neighbors and places the ones that aren't in the closed list or already open into the open queue. Once the node has been expanded, it is placed into the closed list, and the process continues (Luger and Stubblefield, 1998).

BFS can be very useful with tree-based organizations because the tree inherently prevents loops, which eliminates the need to keep and reference the closed list. However, BFS is most useful for instances where the goal is to identify the solution that is closest to the starting node. This can permit guarantees that the first solution found is, in some way, the smallest or simplest.

2.1.4.2 *Depth-First search* Like the Breadth-First Search, Depth-First Search (DFS) explores the space by maintaining a list of open nodes and successively expanding one of them. Instead of managing this list in queue order, it is maintained in Last-In-First-Out(LIFO) stack order. DFS initializes in similar manner to BFS by pushing the starting node on the open-stack. The search then proceeds by popping a node from the open stack and examining it. If the node is a solution, then search is done. Otherwise, it expands the node, pushing all the neighbors of the node that have not already been examined onto the open stack. Once the node has been expanded, it is added to the closed list.

The minor change from a FIFO queue to a LIFO stack has major impact on the manner in which nodes are inspected. Instead of moving level by level like BFS, DFS, as the name implies, searches as deep as possible in the search space for a solution before backtracking and searching deep elsewhere. This approach is very useful for trees, and particularly useful for our method of storing data in the tree. Since it dives as deep as possible in its search for a solution, it quickly reaches the leaves of the tree, where it can evaluate the validity of fully formed candidates. Additionally, DFS only maintains state for its current path from the root node, giving it a polynomial memory requirement, as opposed to BFS's exponential memory requirement.

2.1.4.3 *Best-First search* BFS and DFS each have their own strengths and weaknesses. Combining these two approaches into a blended approach can draw on the strengths and reduce the impact of the weaknesses. *Best First Search* is a systematic search approach that blends the two. It utilizes a *heuristic* to dive depth-first into areas of the tree that seem most promising.

A heuristic is a formalized set of rules that guide the exploration of the search space in a direction that is most likely to lead to a solution. Heuristics are typically based on previous experience that indicates which direction has resulted in solutions before. Humans use heuristics, or rules of thumb, every day to guide their decisions, in situations ranging from choosing the fastest route to the grocery store to choosing a move in chess that brings them one step closer to checkmate (Luger and Stubblefield, 1998).

The quality of the heuristic is critical to the success of best-first search. A good heuristic will lead the search to a solution quickly, while a poor heuristic could lead the search into a portion of the search space that does not contain any solutions. The limitation of heuristics is that they can only make an informed guess of the best next step given the current information. Even the best guess at a decision point can lead the search into a poor area of the search space from a given location. This cannot be eliminated by better heuristics or more efficient search algorithms (Garey and Johnson, 1973).

Best-first search, similar to the breadth-first search, maintains a list of open and closed nodes, except the open nodes are stored in a priority queue which uses the heuristic to determine order. Nodes are expanded, and all their neighbors that aren't in the closed list are added to the open priority queue. The search then continues its search by expanding the first node in the open priority queue, which the heuristic identifies as closest to a solution (Luger and Stubblefield, 1998).

Applying best-first search to MEM-MAP is simple in concept, but difficult in practice, for similar reasons to difficulties encountered using breadth-first search. The typical problem instance is so large that maintaining a full list of closed nodes is not feasible. One modification that helps with the memory size is to limit the size of the closed list. This reduces the effectiveness of the search because it can result in the search examining the same portion of the space multiple times (Luger and Stubblefield, 1998).

2.2 Incomplete Search

Systematic search is the obvious method of finding a solution in the search space if it can complete in a reasonable amount of time. It not only guarantees that a solution will be found if one exists, but it can also conclusively determine when a solution does not exist. Since a typical problem instance for MEM-MAP has hundreds of elements, each with thousands of potential starting logical addresses, the search space is extremely large. Even application of the most sophisticated pruning functions may not allow exhaustive exploration of the search space in a reasonable amount of time.

Incomplete search techniques are one way to focus the effort on the most interesting portions of the search space. These search techniques do not give any guarantee of exploring all candidates. Instead, they apply heuristics to guide the search towards areas of the search space that seem more likely to contain a solution.

There are a variety of incomplete search techniques available, each with its own approach to exploring the search space and identifying promising regions. We anticipate the need to establish a set of tools based on these techniques that will allow us to test the application of the techniques.

Since incomplete search does not have any systematicity requirements, the full-candidate, graph-based representation of the search space is a good characterization. Each node in this representation contains a fully formed candidate, allowing the incomplete search to quickly begin evaluating actual candidates without having to move through the partial candidates. The elimination of the systematicity requirement also eliminates the need to maintain the open and closed lists for the graph. One advantage of applying heuristics is that they generally prevent loops in the search. Since they continually try to move the search towards better candidates, searches that utilize them typically will not visit the same portion of the search space twice.

One thing that all incomplete search methods have in common is the need compare how close different candidates are to a solution. The *Fitness score* is a quantitative score that indicates how close invalid candidates are to a solution (Thede, 2004). There are a variety of fitness scoring systems available. For our application, a simple fitness scoring system might return how many elements *overlap*, where two or more elements that are required by the same task and occupy the same logical address location. In this case, the lower the score the better. Even though the fitness score is calculated on invalid candidates, it can identify those which are closer to a solution than others.

2.2.1 Hill Climbing

Hill Climbing is a basic greedy optimization algorithm that attempts to find a solution by always choosing the best option in the short term. It starts by expanding the current node to generate all of its neighbors. Each neighbor is evaluated to determine its fitness. The neighbor with the best score is designated the current node for future expansion while all other neighbors and the current node are discarded. The search halts when it reaches a state that has a better fitness score than any of its neighbors, a *local optimum*, as indicated in Figure 2.5. Hill climbing searches

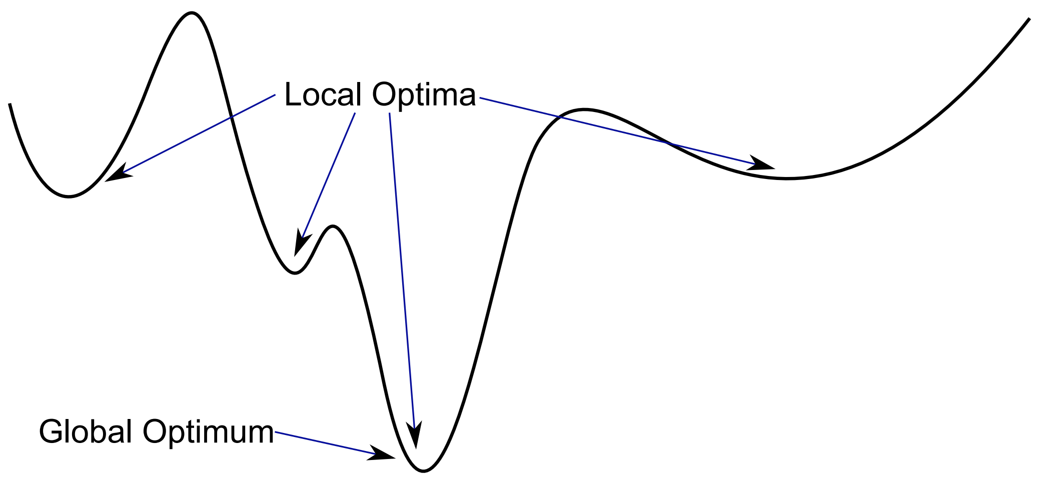


Figure 2.5: An example showing local and global optima. Some search techniques risk getting caught in local optima, and are unable to locate the global optimum. Escaping local optima requires accepting temporarily worse scores with the goal of finding the globally best score.

can be either maximizing or minimized in the fitness function, despite the naming convention. In this case, lower fitness scores are better, so the hill climbing search works to minimize the fitness score. Thus, the local and global optima are present at local and global minimas in the figure. Since it does not keep any history of where it has been or other alternative paths, there is no backtracking available when it reaches a local optimum (Luger and Stubblefield, 1998).

Hill climbing is clearly an incomplete search technique. Using a heuristic to choose the next best neighbor without providing backtracking does not allow for a systematic search of the space, since other candidates that appear less fit could start a path to a solution that would not be examined. The value of hill climbing is the simplicity of the approach. Since no data is maintained from one decision to the next, there is no need to maintain open and closed lists. It is also highly sensitive to the heuristic, since it does not allow for any alternative decisions. We use this straightforward approach as the basis for other incomplete searches.

Hill climbing is the most simple application of a heuristic to MEM-MAP. In order to improve its effectiveness and increase the portion of the search space that it examines, some minor modifications are necessary. If simple hill climbing is directly applied to the problem, consistently using the same root for a starting point, it would always find the same local optimum. An effective modification to hill climbing that addresses this problem is to restart the hill climbing procedure with a randomly generated starting point whenever it reaches a local optimum that is not a solution. This modification is called *random restart*, or shotgun hill climbing (Ghannadian and Shonkwiler, 1996).

An alternative option to the random restart modification is the *persistent* modification, which modifies the fitness score to take into account additional information about the candidate. This provides more information to score the candidate, and reduces the number of local optima encountered. However, this more complex fitness function is more difficult to compute, increasing the time spent at each stage of the search. Even though it extends the length of each attempt and reduces the number of local optima in the space, it is still not guaranteed to find a valid solution. Both of these modification need to be examined, but a hybrid approach is likely the best application of hill climbing (Luger and Stubblefield, 1998).

2.2.2 Genetic Search

Hill climbing and best-first search are good applications of heuristics to search, but they are not the only useful applications of a heuristic to the search. One limitation of these approaches is that, given a starting point, they will always follow the same path through the search space based on the heuristic. Randomization must be added to the searches to allow the techniques to explore a larger portion of the search space.

Genetic search is another tool in the set of incomplete search techniques. It creates a population of candidate solutions for the problem, then allows them to evolve over the course of many generations. In the context of search, evolution is the process of creating a generation of candidates, mutating them, then selecting the best candidates from the mutated ones to constitute the next generation. As new candidates are formed and examined through the evolution process, the search space is explored. Natural selection helps to move the search toward more promising candidates by utilizing the fitness score to prefer mutated candidates that are better than others in the new generation. In order to prevent all the candidates in the new generation from being worse than the current generation, a limited number of candidates are allowed to survive from one generation to the next, if they have comparable fitness scores (Thede, 2004).

Genetic search can be applied to MEM-MAP. It is an incomplete search technique that uses a graph-based representation of the search space. A straightforward application might use a simple mutation function that applies a single mutation to the parent node. This single mutation could consist of swapping the starting logical addresses of two randomly selected elements in the candidate. The resulting mutated candidate would be identical to the parent, with the exception of the swapped elements.

If the neighbor relation is based upon swapping the starting address of two candidates, then the mutated child candidate would be a neighbor of the parent in the search space graph. From a graph perspective, the mutation can be considered to be a random selection of one of the parent node's neighbors for expansion and potential inclusion into the next generation. More complex mutation functions can be achieved by chaining a random number of mutations together, with each mutant randomly selecting a neighbor for the next mutation in the chain.

A simple mutation function effectively makes genetic search a randomized, incomplete version of depth-first search. Even though it is not guaranteed to select the best neighbor at each expansion, it collects a subset of the neighbors, and prioritizes

```
#@PHASE VARICORE 140000 0
VARICORE,   Q2IMXB,  0000335,D, CORERW ,            ,0,   140000,2,            ,0,
     00000000,3,M00,MIO,M01,M02,M03,M04,M05,M06,M07,M10,M11,M12,M13,M14,M20,M21,M22,
     M23,M24,M25,M26,M40,M41,M42,M43,M44,M45,M50,M51,S00,S01,S02,S03,S04,S05,S06,S07
     ,S10,S11,S12,S13,S14,S15,S16,S17,S20,S21,S30,S31,S32,S33,MRM,SRM
VARICORE,   Q2IMXE,  0000015,D, CORERW ,  Q2IMXB,2,            ,0,            ,0,
     ,0,M00,MIO,M01,M02,M03,M04,M05,M06,M07,M10,M11,M12,M13,M14,M20,M21,M22,M23,M24,
     M25,M26,M40,M41,M42,M43,M44,M45,M50,M51,S00,S01,S02,S03,S04,S05,S06,S07,S10,S11
     ,S12,S13,S14,S15,S16,S17,S20,S21,S30,S31,S32,S33
```

Figure 2.6: Excerpt from an input file in Map2 format which shows two elements, Q2IMXB and Q2IMXE, along with their information and tasks which require them.

the better candidates via the fitness function. This will tend to focus the search on better candidates and hopefully find a solution.

2.3 Previous Work

The DOD has been working on this problem since it became apparent that upgrades were necessary to keep the F/A-18 in service and competitive with other warplanes worldwide. They developed a series of programs with the goal of automating the search for solutions. *Map1* (Map, 2007a), the first stage in the process, gathers the data sources for the problem and condenses them into a standard format of the problem instance for entry into *Map2*. Map2 takes the standard format and applies a variety of tools in an attempt to assign an address to every element in a manner that satisfies all constraints. The ability to check the satisfaction of constraints also makes Map2 the canonical validator of any solution (Map, 2007b). Once Map2 has generated and/or validated a solution, it passes the standard format to *Map3* to generate the sysgnctl file for loading on the warplane.

The Map2 file format serves as the official description of the problem instance. The format contains information about the problem instance, as well as recommended address specifications. Each field is separated by a comma for easy parsing. The fields in the file are specified in Table 2.1. Figure 2.6 shows a portion of a file in Map2 format.

Table 2.1: Specification for the Map2 file format. Fields are separated by commas for easy parsing. The total number of columns is determined by the number of task requirements for the given element.

Field#	**Contents**	**Description**
1	Phase-Name	Suggested name of the phase
2	Element-Name	8 or fewer characters. Used to identify the element
3	Size of Element	Octal number. Represents the number of 8-bit data words required
4	Data/Code Type	Data (D) or Code(C)
5	Type	8 or fewer characters. Name of the memory type for the element
6	Concat Relationship	Specifies that this element should immediately follow another specified element. Must match an element name specified in the file
7	Concat-Rel Strength	Strength of 0-3
8	Virtual-Addr Relationship	Octal location in the 64K memory map
9	Virtual-Addr Strength	Strength of 0-3
10	Vir/Phys Offset	Value that, when added to the virtual address, must equal the physical address
11	Vir/Phys Strength	Strength of 0-3
12	Physical-Loc-Rel	Octal physical location. If it starts with "+",understood to be offset from virtual relationship
13	Physical-Loc-Rel Str	Strength of 0-3
N	Task Requirement	List of tasks that require the element

An important piece of information for each element is its *memory type*, which is the kind of memory that is needed to store the element. The possible memory types are shown in Table 2.2.

The specific compilation techniques employed by the mission computer require careful consideration be given to the memory types of the elements stored on them. Pages must only contain elements with *compatible* memory types. Each memory type is clearly compatible with itself, but there is also an expanded concept of compatibility, in which elements with the same Read(R), Read-Write(RW), Read-

Table 2.2: Memory Types

Generic Type	Memory Type
Read-Only	MEEPR
Read-Only	SEEPR
Read-Only	CORER
Read-Write	MRAMRW
Read-Write	SRAMRW
Read-Write	CORERW
Read-Write-Execute	MRAMRWE
Read-Write-Execute	SRAMRWE
MEEP	MEEP
SEEP	SEEP

Write-Execute(RWE) permission are considered compatible. This reduces the effective number of memory types, and makes the problem easier to solve.

As the complexity of the system has increased, Map2 has ceased to be capable of generating valid solutions. The set of tools it utilizes to search for solutions is simply too limited. It still maintains the validation capabilities, and its role in the process has changed to that of the *de facto* validator. In order to find solutions, it is now necessary for a human to inspect the standard format and modify it to fit all the elements while satisfying the constraints. Once the human has modified the file in a way that solves the problem, they pass the solution to Map2 for validation before continuing the process.

Since Map2 now serves only as a validation tool and humans are the only method for finding solutions, work has concentrated on developing tools that assist the humans in visualizing the mapping problem. Baylor's electrical engineering department, in a partnership with DOD, is assisting in development of these tools. Jeremy Stevenson, a graduate electrical engineering student, developed the first

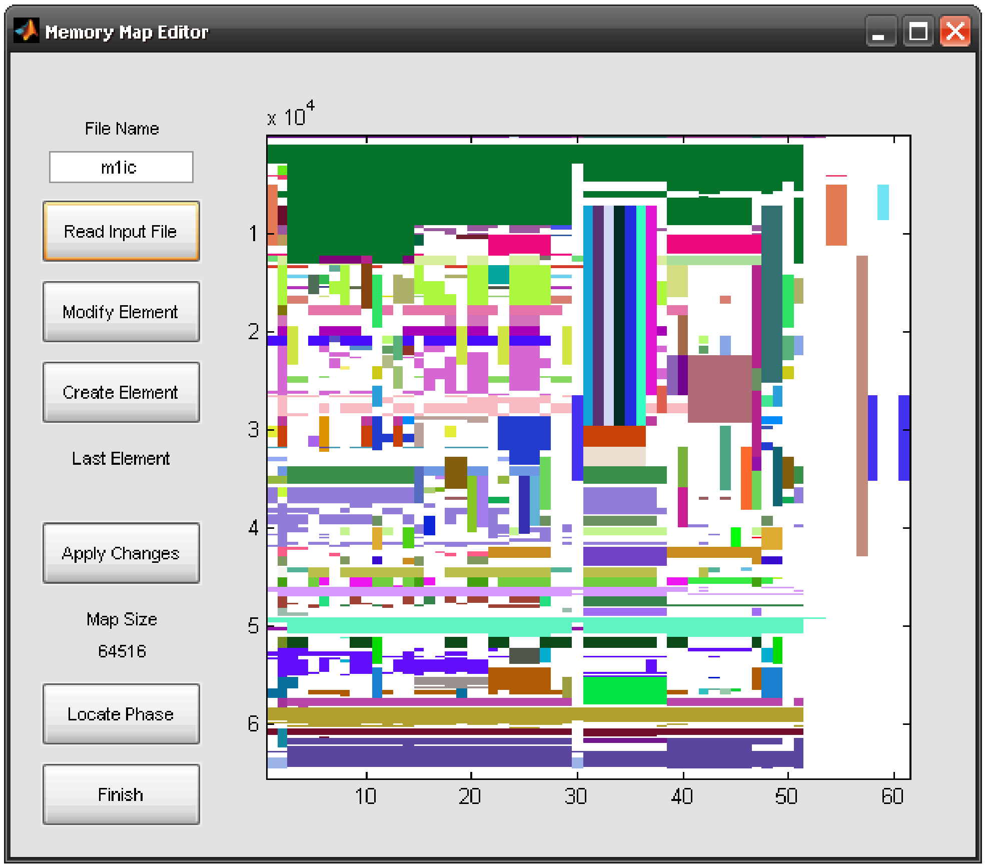


Figure 2.7: Output of the Memory Map Editor implemented by Stevenson. Different elements are colored with different colors, and size of the blocks indicate the element size in bytes. The editor provides the ability to parse a problem file, and perform basic modifications.

automated visualization tool in MatLab called Memory Map Editor (Stevenson, 2007). This program applies some basic reorganization and validation strategies in an effort to come as close to a solution as possible. Once these strategies are applied, the program presents the user with a basic editing interface for further modifications.

One of the strategies that Stevenson developed is the concept of *Fall Through Gravity*. This approach attempts to condense the elements in the mapping by pushing elements as early in the address space as possible. Elements continue to be reordered in an effort to reduce the height of as many elements as possible (Stevenson et al., 2006).

Stevenson's editor is a first attempt at visualizing the problem, but the automated techniques applied are not capable of finding solutions even for known solvable problems in a reasonable amount of time. The user still must do significant reordering in order to find a valid solution.

In an attempt to reduce the amount of user interaction required, Ray Holder re-implemented the algorithms from Stevenson's work into a Java-based visualization application. Holder's implementation provides a visualization interface for humans to use, but it continues to have difficulty providing an efficient, automated search for a solution. The compiled Java sped up the computation, but the algorithms that are used were not efficient enough to find a solution in a reasonable amount of time.

Both the Stevenson and Holder implementations attempted to find a solution by applying developed techniques that were not based on a complete search. The developed techniques can best be described as informal attempts that incorporated ideas found in gradient search, multistage gradient search, genetic search and simulated annealing search algorithms. A more formal analysis of the problem and systematic application of standard methods is needed to explore the possibilities of developing an automatic mapping generator.

CHAPTER THREE

Formal Complexity and Search Characterization

We have shown that there are a variety of approaches to finding a solution for MEM-MAP. The motivation to explore incomplete searches is based on the premise that MEM-MAP is an $\mathcal{NP}$-Complete problem, which prevents complete search from producing a solution in a reasonable amount of time using known algorithms. We describe a formal specification for the problem to support this claim, in addition to proving that compressed mappings and oracles are sufficient for exploring the space of candidates.

3.1 The MEM-MAP Problem

A more formal definition of MEM-MAP will allow a proof of $\mathcal{NP}$-Completeness and will permit various equivalence-preserving transformations of the problem. In formalizing the problem, we abstract away some of the details specific to the problem.

3.1.1 MEM-MAP Problem Instance

A MEM-MAP problem instance defines a set of tasks, data elements and a requirement relation between them. It consists of:

(1) A set of tasks, T.

(2) A set of data elements, D.

(3) A set of memory types, Y.

(4) A set of pairs, R, that describes which data elements are needed by each task. If element $d \in D$ is required by task $t \in T$, then $\langle t, d \rangle \in R$.

In addition to these core components of the problem, there are also some key constraints to the problem that directly affect potential solutions. These include:

(1) Data elements may differ in size. Each data element $d \in D$ occupies $s(d) \in Z^+$ contiguous bytes of memory.

(2) Each data element $d \in D$ has an associated memory type, represented as $y(d) \in Y$.

(3) Physical memory is organized into pages, each P bytes in length.

(4) Each task employs a page table with Q slots, permitting access to Q pages of physical memory.

(5) Each data element $d \in D$ has a memory type denoted as $y(d)$.

3.1.2 MEM-MAP Mapping Solution

A solution to a MEM-MAP is called a *mapping*. It specifies a physical memory location for each data element and page table contents for each task. A mapping M consists of:

(1) An assignment of a starting address in physical memory, $M(d) \in \mathbb{Z}$, for each data element $d \in D$. Any data element d is understood to occupy $s(d)$ contiguous bytes of physical memory starting at $M(d)$.

(2) For each task, identify Q physical pages to be mapped into the logical address space. For every task $t \in T$ and every logical page $q \in \{0, \ldots, Q-1\}$, the function $M(t,q) \in \mathbb{Z}$ defines a starting physical address for the page. Since the page table must reference physical memory pages, $M(t,q)$ must be evenly divisible by P. Also, since no physical memory page may be mapped into two different page table slots, $M(t,q_1) = M(t,q_2)$ only if $q_1 = q_2$.

A valid solution to a MEM-MAP problem instance yields a logical address space for each task that makes available every required data element. It must also satisfy certain validity constraints.

3.1.3 MEM-MAP Mapping Validity Constraints

The various requirements for a valid MEM-MAP solution are simplified by making reference to the physical address space and the logical address space of each task. For task $t \in T$ and data element $d \in D$, we define a logical address $l(t,d) \in \mathbb{Z} \bigcup \{\perp\}$, where $\perp$ represents an undefined logical address. If d starts in a page that is mapped into the page table of t, then $l(t,d) \in \mathbb{Z}$. More specifically, if there is a logical page $q \in \{0, \ldots, Q-1\}$ such that $M(t,q) \leq M(d) < M(t,q) + P$, then $l(t,d) = qP + (M(d) \mod P)$. If there is no such page, then $l(t,d) = \perp$.

The logical address of an element has some limitations. Because of the compilation techniques employed, the logical address for all data items must be the same across all tasks. For any data element $d \in D$ and any two tasks $t_1 \in T$ and $t_2 \in T$, if $t_1 \neq t_2$, then one of the following holds:

(1) $l(t_1, d) = \perp$

(2) $l(t_2, d) = \perp$

(3) $l(t_1, d) = l(t_2, d)$

This consistency of logical addresses leads to a single parameter version of the logical address notation $l(d) = l(t,d)$ for some $t \in T$ for which $l(t,d) \notin \perp$. This single-parameter version is useful for discussing the logical address of an element without making a commitment to the set of tasks for which the element is required. It is understood that the $l(d)$ is only meaningful for tasks that have defined $l(t,d)$.

Using this definition of the logical and physical address spaces, the requirements can be effectively and concisely described. A mapping must satisfy all of these constraints to be considered valid.

(1) *Element Visibility Constraint:* The logical address space for each task must include every byte of every required element. If $\langle t, d \rangle \in R$, then all pages containing d must be contiguously mapped into the page table for task t. If the element spans multiple pages, extra care needs to be taken to ensure all

pages that contain the element are available. Some definitions are necessary to verify that all necessary pages are required.

(a) Address of the first physical page that contains d is defined as
$\text{first}_d = \left\lfloor \frac{M(d)}{P} \right\rfloor \cdot P$

(b) Number of pages on which d occurs is defined as
$\text{count}_d = \left\lfloor \frac{M(d)+s(d)}{P} \right\rfloor - \left\lfloor \frac{M(d)}{P} \right\rfloor + 1$

Each of the count_d pages that contain d must be accessible to t. Since the pages must be mapped contiguously, they must be loaded into neighboring slots in the page table. First, there must exist some $q_1 \in \{0, \ldots, Q-1\}$ such that $M(t, q_1) = \text{first}_d$, which satisfies the visibility of the first page that the element occupies. The slot q_1 must occur early enough in the page table to allow the rest of the pages to be mapped contiguously and still fit in the page table, where $q_1 + \text{count}_d - 1 < Q$. If $\text{count}_d > 1$, then all subsequent pages must be mapped in order. That is, $\forall i \in \{2, \ldots, \text{count}_d\}$ it is the case that $M(t, q_i) = M(t, q_{i-1}) + P$.

(2) *Mutual Exclusion Constraint:* The mapping must organize data elements so that they do not overlap in physical memory. In particular, for any two data elements, $d_1 \in D$ and $d_2 \in D$, if $d_1 \neq d_2$, then either $M(d_1) + s(d_1) \leq M(d_2)$ or $M(d_2) + s(d_2) \leq M(d_1)$.

(3) *Type Homogeneity Constraint:* Each page of physical memory must contain only data elements of a single memory type, or multiple memory types which are compatible. The compatible memory types are statically defined, so we still use equality of y to indicate that two elements either have the same memory type, or a compatible memory type. If two data elements $d_1 \in D$ and $d_2 \in D$ occupy the same page, they must have a compatible memory type. More precisely, either $y(d_1) = y(d_2)$ or $\{M(d_1), \ldots, M(d_1) + s(d_1) - 1\} \cap \{M(d_2), \ldots, M(d_2) + s(d_2) - 1\} = \emptyset$.

(4) *Ordering Constraint:* The order in which elements are placed in a page is important. If multiple elements are stored in the same page, they must be ordered such that every element is required by at least one task that requires an element earlier in the page. The first element that occurs in a page is not affected by this requirement, because there are no preceding elements to check. Formally, this can be described as $\forall t \in T, \forall q \in Q$, there exists $D' \subset D$ such that $\forall d \in D', M(t,q) \leq M(d) < M(t,q) + P$ or $M(t,q) \leq M(d) + s(d) < M(t,q) + P$. This criterion requires that $\forall d \in D'$, either d is the first element on the page, such that $\nexists d' \in D'$ where $l(t,d') < l(t,d)$, or d is required by at least one task that requires a preceding element, $\forall d' \in D'$ where $d' \neq d$ and $l(t,d') < l(t,d)$, $\exists t \in T$ such that $\langle t,d \rangle \in R$ and $\langle t,d' \rangle \in R$.

3.2 $\mathcal{NP}$-Completeness Proof

We use the formal definition of MEM-MAP as the language for proving that it is $\mathcal{NP}$-Complete. We begin by showing that $MEM - MAP \in$NP, followed by showing that MEM-MAP is $\mathcal{NP}$-Hard. Demonstration of these two characteristics form the proof that MEM-MAP is $\mathcal{NP}$-Complete.

Theorem 3.2.1. *MEM-MAP∈NP*

Proof. It is clear that MEM-MAP $\in \mathcal{NP}$, since a nondeterministic algorithm could simply guess the assignment of data elements D to physical addresses, and then guess the assignment of physical addresses to the slots in the page table. The nondeterministic algorithm would then need to verify the solution was valid by performing a polynomial-time check to ensure all requirements of a valid solution are satisfied. □

Theorem 3.2.2. *MEM-MAP is $\mathcal{NP}$-Hard*

Proof. In order to show $\mathcal{NP}$-Hardness, we reduce GRAPH K-COLORABILITY (Garey and Johnson, 1973), a known $\mathcal{NP}$-Complete problem, to MEM-MAP in polynomial time. The specification of GRAPH K-COLORABILITY we will use is:

(1) A graph G is denoted by an ordered pair $\langle V, E\rangle$ where V is the set of vertices, and E is the set of edges.

(2) There are K integer colors available for coloring, and the available colors are numbered $\{0, \ldots, K-1\}$.

(3) A coloring C is an assignment of a color from $\{0, \ldots, K-1\}$ to each $v \in V$ where $C(v)$ is the color of v.

(4) The coloring C is *valid* only when, for every u and $v \in V$, if $(u, v) \in E$, then $C(u) \neq C(v)$

(5) A graph is considered K-*colorable* if there exists a valid coloring C for G using K colors.

3.2.1 Construction

In order to show that MEM-MAP is $\mathcal{NP}$-Hard, we define a construction method to create an instance of MEM-MAP from an instance of GRAPH K-COLORABILITY. Let an instance of GRAPH K-COLORABILITY be denoted by the graph G with K colors available for coloring. We construct a MEM-MAP problem instance I to represent G such that I has a mapping if and only if G is K-colorable.

We first define the construction of the MEM-MAP instance, defining all required terms. Once the construction has been defined, we ensure that a valid mapping for I will exist if and only if G is K-colorable. First, we must ensure that the existence of a valid mapping for I results in a K-colorable graph. Second, if G is K-colorable, then the constructed I will have a valid mapping.

The construction works by creating a data element in I for each vertex in G. The color of the vertices will be represented by the logical address of the corresponding elements, so there will be only K logical addresses possible. Each edge in G has a corresponding task in I, and the data elements in I corresponding to the vertices connected by the edge are specified as required by the task. This requirement prevents the elements from being located in the same logical address, and thus from their corresponding vertices from having the same color.

Additionally, a task is created for each isolated vertex. Since it is not connected to any other vertex, a single required data element is created for the vertex in its task. This single requirement allows the element to occupy any slot in the page table for its task.

First, the constructed I will have K single-byte page slots in each task's page table, where $Q = K$ and $P = 1$. Since pages are only one byte long, only one element can be stored on each page. This allows only one memory type to be required, $Y = \{\text{MEEPR}\}$. Then, for each vertex $v_i \in V$, create a corresponding element $d_i \in D$ that has $m(d_i) =$MEEPR. Set $s(d_i) = 1$, which forces each element to only occupy a single page, and each page may contain at most one element. Finally, for each edge $(v_i, v_j) \in E$, create a new task $t_{i,j} \in T$ along with a requirement for the element associated with each vertex connected by the edge: $\langle t_{i,j}, d_i \rangle, \langle t_{i,j}, d_j \rangle \in R$.

This construction of a MEM-MAP problem instance for a given instance of GRAPH K-COLORABILITY can be completed in time polynomial to the size of the graph. Each vertex has a single corresponding element and each edge has a corresponding task. Each edge also has two requirements present in I. Since K is bounded by $|V|$, where each vertex would be assigned its own unique color, there will be at most V slots in the page table.

3.2.2 Validation

All that remains to be shown is that G is K-colorable if and only if I has a mapping.

Lemma 3.2.3. *If I has a mapping, the original graph G is K-colorable*

Let I have a valid mapping M. Using M, a valid K-coloring C for G can be obtained. First, let $K = Q$. Each $v_i \in V$ will be colored by the logical address of the corresponding d_i such that $C(v_i) = l(d_i)$. To be valid, coloring C must only assign one color to each vertex. The construction of C from M guarantees this because of the mapping validity constraints. One of the requirements on the logical address is that the data element must have the same logical address in all tasks which require the element. This ensures that there is no confusion about which color to assign to the element.

The validity requirement for GRAPH K-COLORABILITY states that vertices that are connected by an edge may not have the same color. The construction prevents this assignment from occurring. Suppose two vertices, v_i and v_j, connected by edge (i, j) were colored the same color, creating an invalid coloring. This coloring could only occur if the corresponding elements d_i and d_j were assigned the same logical address in M such that $l(d_i) = l(d_j)$. Since the construction creates a requirement for task $t_{i,j}$ for both of the elements, they must both be available in its page table, preventing them from occupying the same logical address. Recall that the construction of MEM-MAP dictated that $P = 1, s(d_i) = 1$, and $s(d_j) = 1$, ensuring that each page table slot can only contain one element. Since both of the elements are required by the same task and only one element may occupy a slot, a valid mapping must assign them different logical addresses, so $l(d_i) \neq l(d_j)$.

Lemma 3.2.4. *If G is K-colorable, then I has a valid mapping*

Any valid K-coloring, C, for G can be easily converted into a valid mapping M for I. The construction begins with assigning each d_i to a physical address, $M(d_i)$. Since $s(d_i) = 1$, a trivial and valid physical address assignment is $\forall d_i \in D$, let $M(d_i) = i$.

The page table that is built from this conversion will be very sparse: tasks will reference one or two pages. These references are built according the coloring. In order to ensure that each slot in each page table has a page reference defined, we create pseudo-pages which contain pseudo-elements, which are referenced in the otherwise unoccupied page table slots.

Since $K = Q$, each slot in logical memory corresponds directly to a color in the coloring. For each task $t \in T$, and all that that task's requirements $\langle t, d_i \rangle \in R$, element d_i must be mapped into the page table of t in order for it to have access to the necessary data. The coloring C provides information about which slot should contain d_i. Since the set of colors and set of pages table slots are the same, slot $C(v_i)$ in task t should point to the physical memory of d_i, which can be formally specified as $M(t, C(v_i)) = M(d_i)$.

In addition to assigning physical addresses to necessary elements, we also create Q pseudo-elements to fill in page tables for tasks requiring fewer than Q elements. These pseudo-elements will occupy the physical addresses $|D|, \ldots, |D| + Q - 1$. We then use these pseudo-elements to fill the undefined slots in the page table, $\forall t \in T$, $\forall q \in \{0, \ldots, Q - 1\}$ where $M(t, q) = \perp$, assign $M(t, q) = |D| + q$. This ensures that no slot has an undefined page reference; it either references an actual physical page, or one of the pseudo-pages that contain pseudo-elements.

The construction of the mapping M from the coloring C ensures that no element is mapped into more than one logical address. Suppose an element d were mapped into different slots for two tasks such that $M(t_1, x) = M(d), M(t_2, y) = M(d)$, and $x \neq y$, causing an invalid mapping. The construction of M from C defines the correspondence from virtual to physical address space such that $M(t, C(v_i)) = M(d_i)$. The invalid mapping could only result if $M(t_1, C(v_i)) = M(t_2, C(v_i))$ and $C(v_i) \neq C(v_i)$. Since the coloring function $C(d)$ is deterministic, such an invalid mapping could not occur.

To verify that the constructed mapping M is valid, we must ensure each validity constraint is satisfied.

(1) *Element Visibility Constraint:* Since $s(d) = 1$ and $P = 1$, we are guaranteed that each element fits onto a single page, requiring us only to ensure that the starting logical address of the required data element in each requirement is available in logical space in the necessary task. For all $\langle t, d \rangle \in R$, there exists a $q \in Q$ such that $M(t, q) = M(d)$. The second step of the construction of M explicitly handles each requirement in T, ensuring that a specific slot, $M(t, C(d))$ contains the physical page for the element, $M(d)$.

(2) *Mutual Exclusion Constraint:* Each element is assigned its own page in physical memory, $M(d_i) = i$. Our assertion that $s(d) = 1$ prevents elements from overlapping using this assignment. Since we ensure they don't start at the same logical address, we also ensure they don't overlap one another.

(3) *Type Homogeneity Constraint:* Each page only has enough space for one element, since $s(d) = 1$ and $P = 1$. Since elements can only have one memory type, the memory types contained on each page will be homogeneous.

(4) *Ordering Constraint:* Each element starts its own page, which satisfies the first criteria for the Ordering Constraint.

□

Theorem 3.2.5. *MEM-MAP is $\mathcal{NP}$-Complete*

Proof. A problem is $\mathcal{NP}$-Complete if it is shown to be in the class $\mathcal{NP}$ and is also $\mathcal{NP}$-Hard. Theorem 3.2.1 shows that MEM-MAP is in the class $\mathcal{NP}$. Theorem 3.2.2 provides a polynomial-time reduction from GRAPH K-COLORABILITY, a known $\mathcal{NP}$-Complete problem, to MEM-MAP, which indicates that MEM-MAP is $\mathcal{NP}$-Hard. Given these two theorems, we conclude that MEM-MAP is $\mathcal{NP}$-Complete. □

3.3 Enumerating Mappings via Element Sequences

The only known way to conclusively find a solution $\mathcal{NP}$-Complete-type problems is exponential-time exhaustive search, which requires a way to enumerate each node in the search space. Incomplete search techniques also require this enumeration capability. We now move away from the examination of $\mathcal{NP}$-Completeness to discuss search space representation, specifically using element sequences to provide enumeration capabilities.

So far, we have considered a candidate to be an assignment of starting logical addresses to all elements and placements into physical pages. This simple definition is sufficient for illustrating basic concepts, but it entails a very large search space that is not feasible to search in a reasonable amount of time. The assignment of specific physical and logical addresses to elements facilitates examination and validation, but it also requires search to make commitments that may be unnecessarily strict.

An example helps illustrate the strict commitment to specific logical addresses of the candidate. Figure 3.1 shows three simple candidates that could be examined during a search of the candidate search space. In this example, $l(d_1) \leq l(d_3) \leq l(d_2)$, and they all have incompatible memory types which prevents them from sharing pages. d_1 has been assigned a starting logical address, and the current choice revolves around the commitment for d_3. These three situations are only a small subset of the large number of commitments that would need to be evaluated for d_3 at this point in the search. Even a very minor change in position for d_3 leads to a different candidate in the search space. For each situation, d_2 is placed in the earliest logical address possible, taking into account the placement of d_3.

The issue with this commitment to specific logical addresses is that it is very difficult to know which of the commitments, if any, will lead to solutions. Here, the very fact that d_3 is being placed between d_1 and d_2 is the reason that d_2 cannot fit into the page table. Figure 3.2 shows that changing the order such that d_3 comes

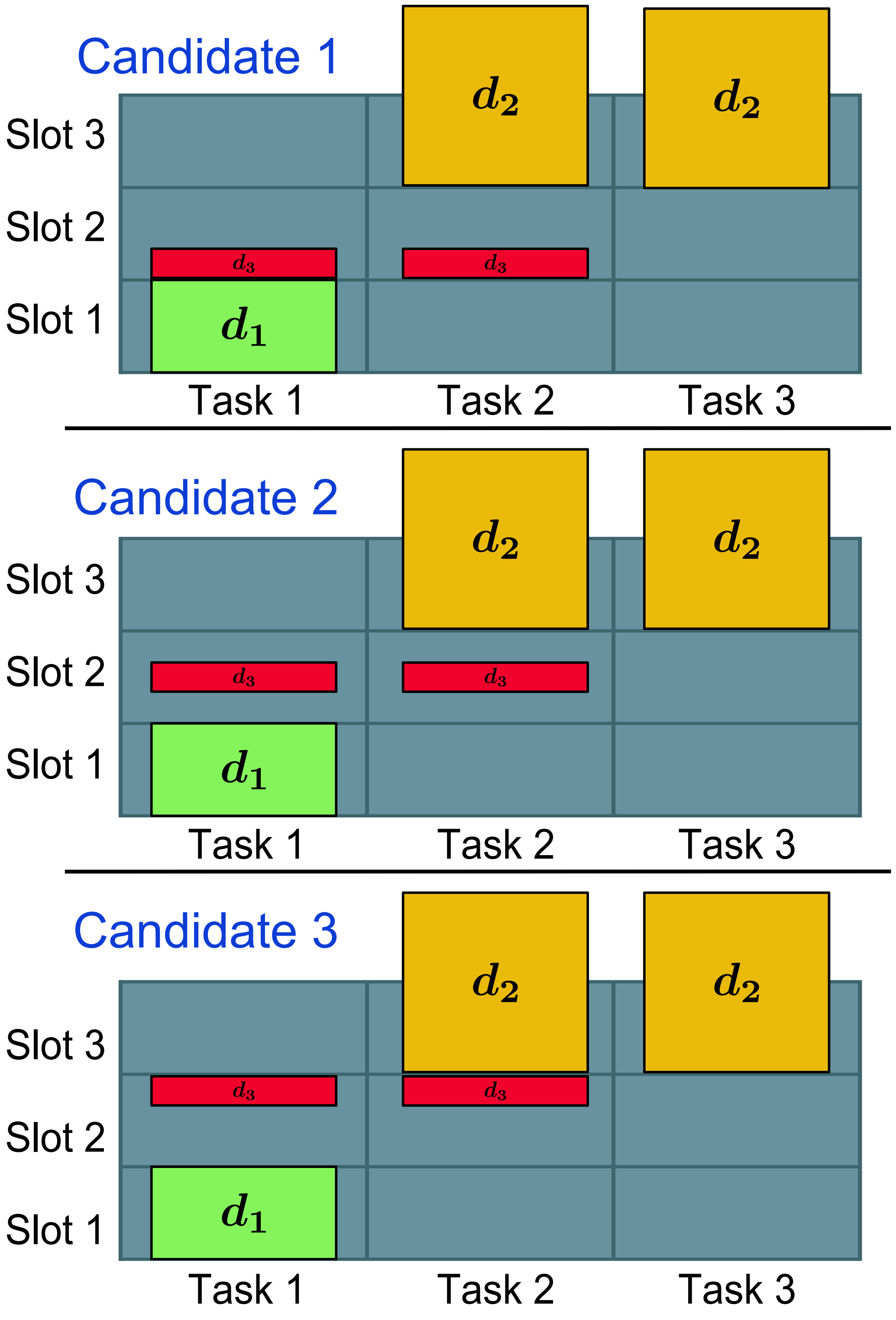


Figure 3.1: If the elements d_1, d_2, and d_3 are ordered such that d_3 is between d_1 and d_2, the candidate cannot be valid. The small size of d_3 results in many candidates that all exhibit the same ordering problem.

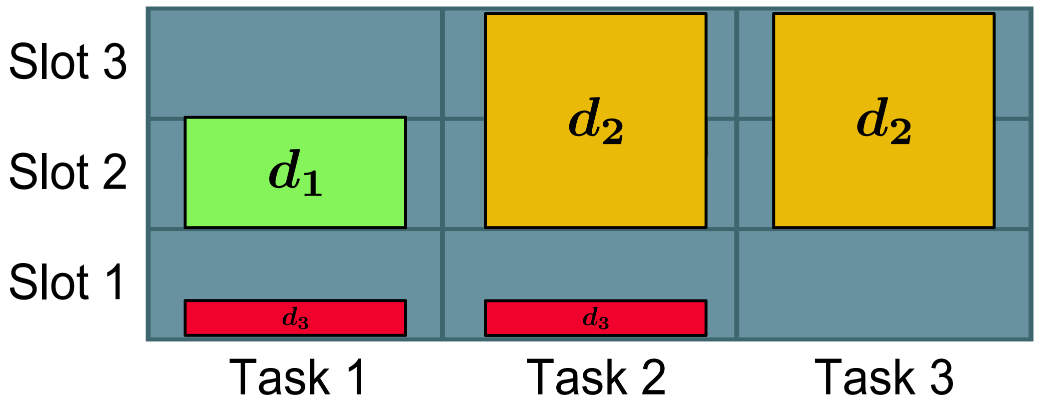


Figure 3.2: Placing d_3 before d_1 allows all elements to fit. The order of elements is an important characteristic of a candidate.

before d_1 results in a valid mapping. Under this naïve characterization of the search space, much effort could be wasted searching the variety of ways that d_3 could be tweaked between d_1 and d_2 instead of getting to the actual solution.

Since the space of all candidates is so large, it is critical to find a less strict, smaller search space that is equivalent in some way. The goal is to explore the space of possible candidates by searching this smaller space that captures some of the essential qualities of a candidate.

3.3.1 Equivalence Class and Canonical Candidates

One way to develop a substitute search space is to find a special candidate which represents all three candidates in Figure 3.1. This representative candidate could then represent all similar cases that place d_3 between d_1 and d_2. The space of all representative candidates would be significantly smaller than the space of all candidates. We use *equivalence classes* to identify this smaller search space. An equivalence class is a group of candidates that have similar characteristics. This allows many states from the naïve search space to be folded into fewer equivalence classes, resulting in a smaller search space.

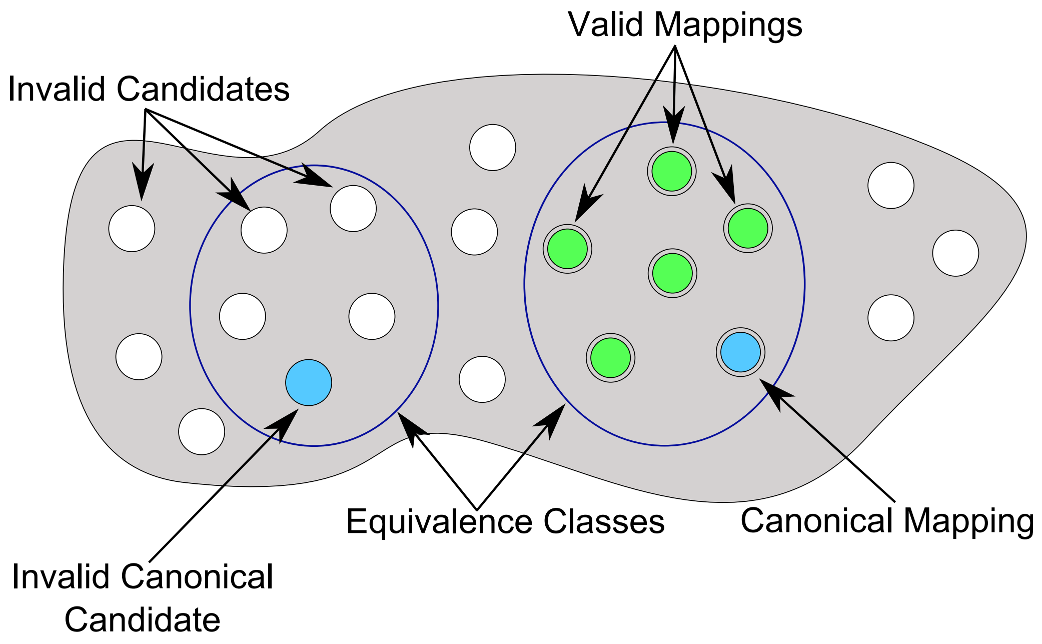


Figure 3.3: The space of candidates can contain invalid candidates, as well as candidates that are valid mappings. Each valid mapping has a canonical form that can represent multiple valid mappings.

The equivalence class space can be examined by choosing a representative candidate from each class. This representative candidate is called a *canonical candidate* for its equivalence class. The canonical candidate for an equivalence class is *equivalent* to the candidates in the class under some equivalence relation. In this case, two elements are considered to be equivalent if they compress to the same compressed element. This compressed candidate is considered the canonical candidate for the equivalence class. The set of all canonical candidates forms a smaller search space that is analogous to the original candidate search space, ensuring that if a valid mapping exists in the space of candidates, a valid canonical mapping will also exist in the space of canonical mappings. Figure 3.3 illustrates a portion of a search space, including equivalence classes and their representative, canonical candidates. Note that we do not require that all canonical candidates be valid, they simply represent an equivalence class of candidates. However, we critically depend on the requirement

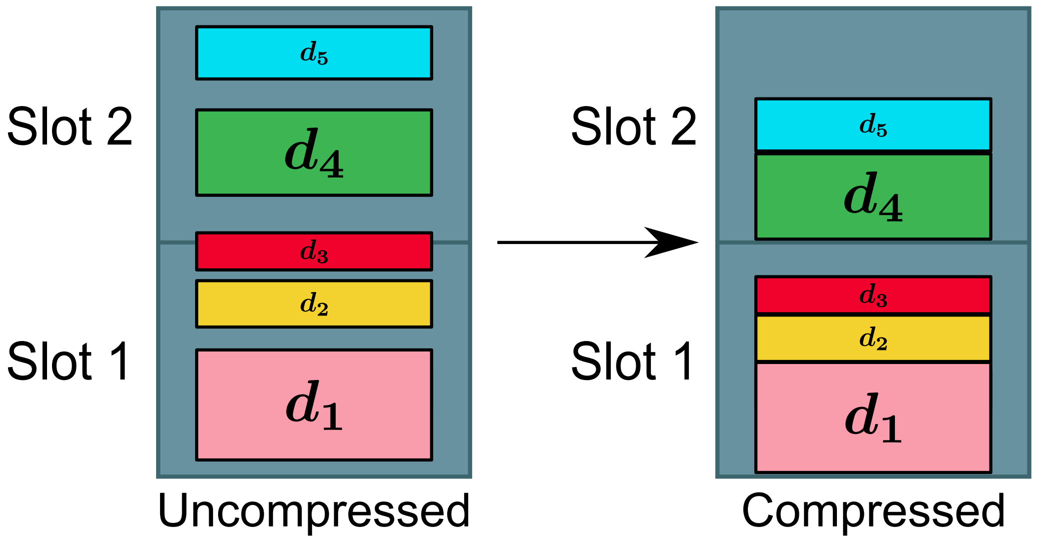


Figure 3.4: All elements in a compressed mapping must either start on a page boundary or immediately follow another element in a page.

that a canonical candidate be valid when any candidate in its equivalence class is valid.

We define our canonical candidate by eliminating unnecessary space between elements in the logical address space. We call this type of candidate a *compressed candidate*. This compression of space represents a reduced commitment with respect to the logical address assignments, allowing a compressed candidate to represent all the uncompressed candidates in its equivalence class. A valid compressed mapping is a valid mapping that satisfies an additional requirement to the standard validity requirements. The extra requirement is that every element must either immediately follow another element, or start on page boundary such that it cannot start on the preceding page due to a validity constraint. Formally, this can be described as $\forall d \in D, t \in T$, either:

- d is not referenced in t and thus its logical mapping is undefined, $l(t, d) = \perp$
- d immediately follows another element: $\exists d' \in D$ such that $d' \neq d$ and $l(t, d') + s(d') = l(t, d)$

- d starts on a page boundary, $l(d) \bmod P = 0$, and cannot be placed onto a page in the preceding slot. We define $q = \left\lfloor \frac{l(d)}{P} \right\rfloor$ to be the current slot for d. Using this definition, d can be placed onto a preceding page if one of the following cases occur:

  (1) *Empty preceding page*: $\forall t \in T$, either $l(t, d) = \perp$, or $M(t, q-1) = \perp$

  (2) *Single compatible preceding page*: $\exists p$ such that $\forall t \in T$, either $l(t, d) = \perp$, or $M(t, q-1) = \{p, \perp\}$. This p must have some specific qualities, mostly revolving around the set of of elements that occur on p. Let $D_p = \left\{ d' \in D \mid \left\lfloor \frac{M(d')}{P} \right\rfloor \leq p \wedge \left\lfloor \frac{M(d' + s(d'))}{P} \right\rfloor \geq p \right\}$. Using this definition, the following must hold for p to be a valid preceding page:

    (a) All the elements have compatible memory type to d: $\forall d' \in D_p, y(d') = y(d)$

    (b) The last byte of the page is empty: $\forall d' \in D_p, (l(d') + s(d')) \leq (l(d) - 1)$.

  If either of these cases is true, then d can be further compressed, and the requirement is not satisfied.

Elements that satisfy these criteria are considered to be *compressed elements*, while those that do not are considered *uncompressed elements*.

This definition of canonical candidate can be used to define the membership criteria of an equivalence class of candidates as the set of all candidates that result in the same compressed mapping after removing any unnecessary space. All three of the candidates in Figure 3.1 would be part of the same equivalence class, with Candidate 1 being the compressed representation for that class.

Using the compressed mapping for the canonical candidate allows us to preserve the most important aspects of the candidates, the order in which elements are present in memory, while factoring out insignificant commitments. We will show that the gaps between elements are unimportant to any of the validity constraints by proving that any equivalence class that contains a valid mapping will have a corresponding compressed candidate that is also valid. This makes the space of compressed candidates just as good as the space of all candidates, but with far fewer states to be searched. Figure 3.4 shows a portion of an uncompressed mapping and its canonical representative in the same equivalence class. The relative position of each element is maintained. Additionally, elements that start earlier in logical address space are not altered.

3.3.2 Using Oracles to Represent Compressed Candidates

The space of compressed mappings is a useful substitute search space, but a method for enumerating elements in the space is needed. The compressed candidates can be easily generated from any given candidate, but the goal is to generate the compressed candidates without first generating the space of all candidates. We use an *oracle* to represent a compressed candidate in the search space. An oracle is an element sequence that contains all elements $d \in D$, indicating where each element is stored in logical memory with respect to the other elements in the sequence. If a problem instance only contains three elements, an oracle o for the instance could be defined as $\langle d_1, d_2, d_3 \rangle$. The oracle also provides a function to access its elements, where $o[0]$ would return the first element in the oracle, in this case d_1. Oracles are a good way to explore the space of compressed candidates because they are easy to work with, since they are simply a sequence that can be easily modified. This allows the space of oracles to be easily enumerated.

In addition to the standard, fully formed oracle which defines a sequence of all elements $d \in D$, we also need the concept of *partial-oracle*, which only defines the sequence for some $D' \subset D$. This is necessary to facilitate representing the partial-candidates discussed as part of the depth-first search.

Informally, we use an oracle to capture the essential details of their placement in logical address space. If element d_1 comes before d_2 in the oracle, then d_1 can be expected to have a logical address that starts no later than $l(d_2)$.

3.3.2.1 *Mapper function* Consider an oracle o. The information about the relative order of elements in o can be used to construct a compressed candidate c. The construction greedily places elements as early as possible in the logical address space of c while still satisfying all validity constraints.

Consider each element, d, in the order that they appear in o. Per the definition of oracle, we require that $l(d) \geq l(d-1)$. Beginning at $l(d-1)$, check to see if assigning d to the logical address would satisfy all validity constraints. If the assignment would break any validity constraint, consider each successive logical address that could be assigned to d.

Once d has been assigned a logical address, the physical address space and page tables must be updated. Since an element can only occur once in physical memory, each task in $T_d = \{\forall t \in T \mid \exists \langle t, d \rangle \in R\}$ must reference the same pages in its page table to access d. For each slot q that contains d, if any $t \in T_d$ has a page p assigned to slot q, specifically $M(t, q) = p$, verify that $\nexists t_1, t_2 \in T_d$, where $l(t_1, q) \neq l(t_2, q)$. If two tasks have different pages mapped to q, then a new logical address must be found for d. If the slot does not have a page allocated: $\forall t \in T_d$, where $M(t, q) = \perp$, then allocate the next page from physical memory, p', store the element on the page, and reference p' from q for each task, specifically $\forall t \in T_d$, specify that $M(t, q) = p'$.

Once an assignment has been found that satisfies all validity requirements, move on to the next element in o. Once all elements have been assigned a logical address, and all necessary pages have been created and mapped to slots in the page tables, the compressed candidate c is considered to be fully constructed. The following psuedocode demonstrates this process.

```
MapperFunction(o)
```

```
currLogical ← 0
currPhysical ← 0
Foreach d in sequence o
    $T_d \leftarrow \{forallt \in T \mid \exists \langle t,d \rangle \in R\}$
    Foreach $\langle t,d' \rangle \in R$ where $d' = d$
        While setting ($l(t,d)$ =currLogical) violates any validity constraints
            currLogical ← currLogical+1
l(d) ← currLogical
For each slot $q \in \left\{ \left\lfloor \frac{l(d)}{P} \right\rfloor, \ldots, \left\lfloor \frac{l(d)+s(d)-1}{P} \right\rfloor \right\}$
    If $\exists t \in T_d$ where $M(t,q) \neq \bot$
        Ensure that $\forall t_1, t_2 \in T_d$, either $M(t_1,q) = \bot$ or $M(t_2,q) = \bot$ or $M(t_1,q) = M(t_2,q)$
    If $\nexists t \in T_d$ where $M(t,q) \neq \bot$
        Create a physical page starting at currPhysical to store d
        $\forall t \in T_d$, set $M(t,q) \leftarrow currPhysical$
        currPhysical ← currPhysical + P
```

3.3.2.2 *Example* An example helps illustrate the process of generating an oracle from a compressed mapping, and applying the mapper function to the oracle to construct the original, compressed mapping. Consider an oracle $o = \langle d_3, d_1, d_2 \rangle$ that corresponds to the compressed candidate c presented in Figure 3.2. Application of the mapper function to o will produce a compressed candidate. The construction places d_3 at the earlist possible logical address, which is the first logical address in the page tables for Task 1 and Task 2. Since there were no pages created before this placement, a page is created to store d_3, which is then referenced by the first slot in each task's page table. The next element to be placed is d_1. Since d_3 and d_1 have a common task requirement, Task 1, the Mutual Exclusion constraint specifies that they cannot be placed into the same logical address. Additionally, since they do not have the same memory type, the Type Homogeneity constraint requires that they not be placed onto the same page. The mapping procedure will consider successive logical addresses until d_1 is assigned its own page, and that page is mapped to Slot 2 of the page table for Task 1. Finally, d_2 is placed into the candidate. Since it does not share a task requirement with d_1, it can be placed into the same logical address as d_1. It is assigned its own page, and loaded into Slot 2 for Task 2 and Task 3. The resulting candidate is the same compressed candidate that is presented in Figure 3.2.

3.3.3 Formal Examination of Compressed Mappings and Oracles

We have suggested that a search for valid mappings in the space of candidates need not explore the entire space of candidates. Compressed candidates can be used as canonical representatives for their equivalence class, resulting in a smaller search space. Oracles can then be used to facilitate the enumeration and search of the compressed space. We have given an overview of how these concepts work together, but proofs are needed to verify that the exploration of these reduced spaces is sufficient. There are two specific points that need to be proven:

(1) If a mapping exists in the space of candidates, a compressed mapping also exists. This proof will provide a demonstration of how to generate a compressed mapping for any given mapping.

(2) Each compressed mapping has a corresponding oracle that contains enough information to reconstruct the original compressed mapping. Given a compressed mapping, M', there exists an oracle o that will yield M' under the mapping function.

3.3.3.1 *Compressed mapping existence* We can show that a compressed mapping will exist if a mapping exists. In order to show the existence of this compressed mapping, we present a construction that produces a compressed mapping given a mapping.

The first part of the construction is the definition of an elementary move that, if the mapping is not already compressed, moves elements in a way that eliminates some unnecessary space without compromising validity constraints. This elementary move consists of two parts: a compression step, potentially followed by a reordering step to maintain validity. Repeated application of the elementary move will eventually yield

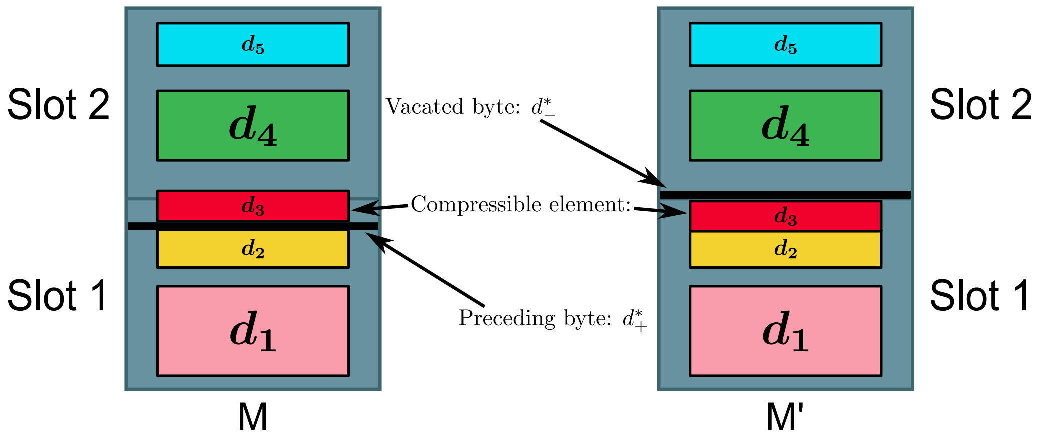


Figure 3.5: Application of a compression step to a non-compressed mapping M that produces M'. For this example, $d^* = d_3$. Note that all other elements occupy the same physical and logical address space, and only two bytes, d^*_+ and d^*_- are ultimately changed after compression.

a compressed mapping. In order to ensure progress, we identify a *bound function* that ensures that each elementary move is progressing towards a compressed mapping. Each move must produce a mapping that is closer to the bound, with a reduced bound function value.

An example helps illustrate the compression process. Let M be a non-compressed mapping. According the definition of compressed mapping, there must exist some element $d^* \in D$ that violates the compressed mapping requirement by not starting on a page boundary and having at least one empty byte immediately preceding it.

The compression step, when applied to M, produces M', which has the same page table as M, with physical and logical addresses modified as:

$$M'(d) = \begin{cases} M(d) & \text{if } d \neq d^* \\ M(d) - 1 & \text{if } d = d^* \end{cases}$$

The physical and logical address space M' is identical to the M, except that the compressible element d^* is moved forward by one byte.

Observe that this single-byte move ultimately modifies only the occupation status of two bytes in physical memory. First, the previously unoccupied preceding byte $d^*_+ = M(d^*) - 1$ now contains data for d^*. Second, moving the element forward in memory causes the succeeding byte $d^*_- = M(d^*) + s(d^*) - 1$, which contained the last byte of d^*, to be vacated. All other bytes in physical memory remain unchanged with respect to the elements they store.

Lemma 3.3.1. *Application of the compression step to a non-compressed mapping M preserves validity.*

Proof. We can consider each validation constraint individually to demonstrate that a compression step will not compromise validity.

(1) *Element Visibility Constraint:* All non-compressed elements occupy the same physical and logical address in M' that they do in M. Since their address doesn't change, they will still be fully visible to all tasks that referenced them in their page tables in M. The compressed element, d^*, occupies the same bytes that it does in M, with the exception of the new starting byte d^*_+, and the vacated byte d^*_-.

Pretend task t requires d^*, and in M was capable of seeing every byte of d^* via its page table allocation. We must show that in M', task t is still capable of seeing every byte of d^*. Since d^* no longer occupies d^*_-, we are not concerned with the visibility of d^*_- to t. All locations from $d^*_+ + 1$ up to $d^*_- - 1$ are still visible to t in M', since d^* occupies them in both M and M'. The byte of concern is d^*_+.

The starting logical address of d^* is new, but the definition of a compressible element ensures that d^*_+ will be accessible to t. The first possibility is that d^*_+ occurs on the same page as the starting logical address of d^* in M, where $\left\lfloor \frac{d^*_+}{P} \right\rfloor = \left\lfloor \frac{M(d^*)}{P} \right\rfloor$. In this case,$d^*$ starts on the same page in M' that it does in M, $\left\lfloor \frac{M'(d^*)}{P} \right\rfloor = \left\lfloor \frac{M(d^*)}{P} \right\rfloor$. Since t references that page in its page table in M', d^*_+ will be visible to t.

The other possibility is that d^*_+ is in the slot immediately preceding the slot that d^* occupies in M, $\left\lfloor \frac{d^*_+}{P} \right\rfloor = \left\lfloor \frac{M(d^*)}{P} \right\rfloor - 1$. The definition of a compressible element explicitly specifies the circumstances under which this scenario can occur.

(a) *Empty preceding page*: The preceding slot is vacant in all tasks which require d^*. Simply allocate a page, and modify the page table to use that page in the preceding slot. Since the compression step only moves in increments of a single byte, d^*_+ will be the last byte on the created page, which will be vacant since the page contains no other elements.

(b) *Single compatible preceding page*: The definition of compression explicitly defines the situation in which the page stored in the preceding slot is the same across all tasks which require d^*, or empty. One requirement on this page is that the last byte be vacant. Thus, d^*_+ will not overlap with any elements on the preceding page. If t maps the page to the preceding slot, than d^*_+ is immediately visible. If t is does not map the page, than the definition requires that it be vacant, which can be easily remedied by mapping the page into the vacant preceding slot.

(2) *Mutual Exclusion Constraint:* Since M is valid, it contains no overlapping elements, so the only places where M' might violate mutual exclusion are at d^*_+ and d^*_-. The address d^*_- is unoccupied in M', so it can't be the cause for a new element collision. The address d^*_+ is empty in M, but, in M', it contains the first byte of d^*. Since it was empty in M, storing an element in d^*_+ in M' will not cause an element overlap since d^* is the only element that changes position.

(3) *Type Homogeneity Constraint:* During the compression step, all elements $d \in D$ where $d \neq d^*$ keep the same physical and logical address in M' that they have in M. The only element which moves is d^*. Again, there are two cases for the location of d^*_+. If $\left\lfloor \frac{d^*_+}{P} \right\rfloor = \left\lfloor \frac{M(d^*)}{P} \right\rfloor$, then d^* starts on the same page in M' that it does in M. Since the elements stored on the page in M are known to all have compatible memory type, and no elements were added to the page, all elements on the page in M' will also have compatible memory types.

If $\left\lfloor \frac{d^*_+}{P} \right\rfloor \neq \left\lfloor \frac{M(d^*)}{P} \right\rfloor$, then d^* will start on the preceding page. The definition of a compressible element explicitly specifies the circumstances under which this scenario can occur.

(a) *Empty preceding page*: The preceding slot is vacant in all tasks which require d^*. Simply allocate a page, and modify the page table to have all tasks which require d^* use that page in the preceding slot. The newly created page will only contain d^* in M', so the Type Homogeneity constraint is trivially met for d^*_+.

(b) *Single compatible preceding page*: The definition of compression explicitly defines the situation in which the page stored in the preceding slot is the same across all tasks which require d^*, or empty. One requirement on this page is that all elements which occur on the page have a compatible memory type to d^*. Thus, adding d^* to the page will not violate the Type Homogeneity constraint for d^*_+.

The set of elements located on the page that contains byte d^*_- may change. If d^*_- is the first byte on the page, such that $d^*_- \bmod P = 0$, then d^* is no longer present on the page in M', and the set of elements on page $\left\lfloor \frac{d^*_-}{P} \right\rfloor$ is smaller. Since all elements in the set have the same memory type in M, and the set is identical in M' with the exception of d^*, all the elements in set still have the same memory type in M'. This ensures that the homogeneity is maintained.

(4) *Ordering Constraint:* The single-byte compression move does not alter the order that elements occur within the page on which d^* starts, since d^* is moving into a previously vacant d^*_+. In M', d^* is either moved forward in the same page it starts in M, or onto a compatible preceding page. Per the definition of a compressible element, d^*_+ is either the first byte on the page, last byte on a compatible preceding page, or a vacant byte between elements. If d^*_+ is the first byte on the page, such that $d^*_+ \bmod P = 0$, then d^* is the first element on the page in both M and M', and satisfies the Ordering Constraint. If d^*_+ is the last byte on a preceding page, then the page has either been newly created and only contain d^*, or the page is required by at least one task which requires d^*. If d^*_+ is a byte between elements, d^* simply moved forward by one byte. Since d^*_+ is vacant in M, this movement will not change the order that elements occur on the page. Since d^* shares a task with an element that precedes it on the page in M, and the order of elements did not change during compression, the Ordering Constraint will be satisfied with this page in M' as it was in M. Unfortunately, there are cases where the vacated byte d^*_- can violate the Ordering Constraint when d^*_- is the first byte on the page, $d^*_- \bmod P = 0$. This scenario must be handled by a partition/reorder operation to restore validity. □

There are cases when a compression step breaks the Ordering Constraint of a valid mapping. This occurs when d^* spans multiple pages in M, and d^*_- is the first byte of its page. The compression move changes the contents of d^*_- to be vacant. Depending on the order that elements occur on the page containing d^*_-, there may be elements that relied on a task requirement of d^* to satisfy the Ordering Constraint. Since d^* is no longer present on the page, the Ordering Constraint would not be satisfied in M', even if it is satisfied in M. Modifications need to be made to M' to fix the Ordering Constraint without breaking any of the other validity constraints.

Some additional notation is helpful to formally examine this scenario and the associated modifications. First, let page p be the page that is affected by the vacation of d^*_- during compression of d^* in M, such that $p = d^*_-$. Next, let D^* be all elements that start on page p in M', such that $D^* = \{d \in D \mid p \leq M'(d) < p + P\}$. Finally, let T^* be all tasks that require an element in D^*, such that $T^* = \{t \in T \mid \exists d \in D^*, \langle t, d\rangle \in R\}$.

We specify a partition/reorder process that partitions the set of tasks in T^* and elements in D^* into subsets. We can show that each subset can be reordered so that the Ordering Constraint is satisfied, while continuing to satisfy the other validity constraints and requiring no additional pages.

The first step is to partition T^* into *independent subsets*. We define two subsets, T^*_i and T^*_j to be independent with respect to D^* if no element in D^* is required by both a task in T^*_i and a task in T^*_j. Next, we define a partitioning of D^* into independent subsets by utilizing the partitioning of T^*. Each subset $T^*_i \subseteq T^*$ has a corresponding element subset $D^*_i \subseteq D^*$ which represents the set of all elements that are required by the tasks in T^*_i. Formally, $D^*_i = \{d \in D^* \mid \exists t \in T^*_i, \langle t, d\rangle \in R\}$.

Partitioning T^* using this definition of independence is critical to satisfying the Ordering Constraint. The independent task subsets do not have any element requirements in common. This allows each task subset to map the elements in the corresponding element subset to the same slot as other element subsets without causing any elements to overlap. Each element subset is assigned to its own page and mapped to the same slot in the required tasks. Figure 3.6 shows this process after the compression of d_4.

Once the tasks and elements have been partitioned into independent sets, more work is needed to satisfy the Ordering Constraint. Each element subset must be

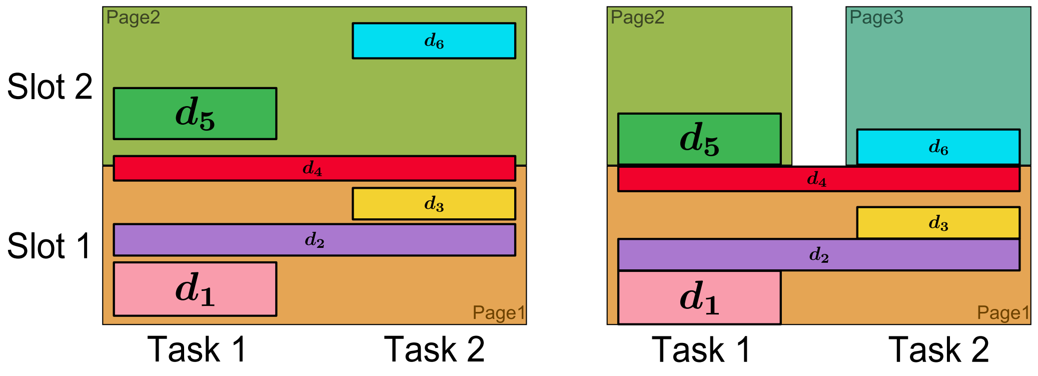


Figure 3.6: Application of the partion/reorder process to Page 2 following the compression of d_4 can result in multiple pages being created. Elements that do not share any task requirements, such as d_5 and d_6 can then occupy the same logical address, since they are stored on separate pages.

ordered appropriately. We specify a construction of a new page for each element subset D_i^*. This page will satisfy the Ordering Constraint without invalidating any of the other validation constraints.

An independent partitioning of tasks must exist for any T^*. Let ${T^*}_1, \ldots, {T^*}_k$ be a partitioning of T^* into k independent subsets with respect to D^*. At a minimum, T^* can be trivially partitioned into a single subset, such that $k = 1$ and $T_1^* = T^*$. This satisfies the independence requirement since there is only one set, and there is no other task set with which elements may share task requirements. The singular partitioning is not always useful. We need another notion to indicate that each subset has been partitioned as much as possible.

We define a partitioning to be *minimally dependant* if there is no way to further partition any of its subsets into independent subsets. The partitioning ${T^*}_1, \ldots, {T^*}_k$ is minimally dependant if each partition cannot be partitioned further, into subsets $T' \subset T_i^*$ and $T'' \subset T_i^*$ which are independent with respect to D^*.

Ensuring that a partitioning is minimally dependant guarantees that each element subset can be ordered in a manner that satisfies the Ordering Constraint. Consider a non-minimally dependant partitioning ${T^*}_1, \ldots, {T^*}_k$. Let T_i^* be a subset which can be further partitioned into independent subsets $T' \subset T_i^*$ and $T'' \subset T_i^*$, which makes the partitioning non-minimal. These additional task subsets have corresponding element subsets, $D' \in D_i^*$ and $D'' \in D_i^*$. By the definition of independent task subsets, there does not exist a $d \in D_i^*$ that is a member of both D' and D''. Since such a d does not exists, there is no way to order the elements in D_i^* in a manner that will satisfy the Ordering Constraint. If the first element on the page is a member of D', then no element in D'' can be placed because there can never be an element that shares a task requirement earlier in the page. Starting the page with an element from D'' presents similar problems.

Producing an order of elements that satisfies the Ordering Constraint from a minimally dependent subset is straightforward. This order can then be used to build an assignment of physical and logical addresses to the elements.

Lemma 3.3.2. *If a compression step invalidates the Ordering Constraint for M', an M'' can be constructed which fits in memory and satisfies the Ordering Constraint while preserving all other validity constraints.*

Proof. The construction of M'' procedes as follows. First, let all elements not affected by the compression of d^* occupy the same physical and logical addresses in M'' that they do in M', specifically $\forall d \in D$, if $d \notin D^*$, then $M''(d) = M'(d)$. Determining the addresses for elements that are part of D^* requires more work.

A new page p_i is constructed for each $D_i^* \subset D^*$ in a minimally dependant partitioning of D^*. This construction is specified inductively, which provides a straightforward method for allocating elements while also facilitating verification of the validity constraints.

Placing the first element on a page is easy. The base case for the inductive construction occurs when p_i is empty. In this case, select any element $d_1 \in D_i^*$ and assign it to be the first element on the page, specifying that $M''(d_1) = p_i$. Being the first element on the page, d_1 trivially satisfies the ordering constraint.

The inductive case follows when one or more elements have been placed onto p_i and more elements must be placed onto the page. To keep track of which elements have been placed on a page and which tasks have been covered by the placed elements during construction, we define more notation. Let $D_i^c \subset D_i^*$ be the set of elements from the ith subset that have been placed onto a page, while letting $D_i^u = D_i^* - D_i^c$ be the elements that have not been placed onto the page. Additionally, let T_i^c be the set of all tasks that require an element that has been placed, specifically $T_i^c = \{t \in T_i^* \mid \exists d \in D_i^c, \langle t, d\rangle \in R\}$. It then follows that $T_i^u = T_i^* - T_i^c$.

More elements need to be placed when $D_i^u \neq \emptyset$. Choose an element d_n from D_i^u which is required by at least one covered task $t^c \in T_i^c$, such that $\langle t^c, d_n\rangle \in R$. Place d_n onto page p_i immediately following d_{n-1}, such that $M''(d_n) = M''(d_{n-1}) + s(d_{n-1})$. It is important to note that such a d_n must exist in D_i^u by the definition of a minimally dependant partitioning, because each task requires at least one element that is required by another task in T_i^u. This prevents D_i^u from having no tasks in common. If such a case were to exists, the elements would have been further partitioned. Selecting d_n which is required by a covered task ensures that satisfaction of the Ordering Constraint is maintained.

Once all elements in D_i^* have been placed onto p_i, p_i can be added to the page table of each task in T_i^*. Since the partitioning produced independent task subsets, the tasks can place the page into the slot q that had previously contained p, specifically $\forall t \in T_i^*$, $M''(t, q) = p_i$.

Application of the partition/reorder process to M' produces a M'' which satisfies all validity constraints and restores the validity contraint that was compromised when d_-^* was vacated. Inspection of each constraint verifies that it is still satisfied in M''.

(1) *Element Visibility Constraint:* Assume task $t \in T$ requires element $d \notin D^*$. Since $M''(d) = M'(d)$ for all non-affected elements and the slot for t that

references the page containing d was not altered, the page table of t will continue to contain the required non-affected elements.

Next, assume that $t \in T^*$ requires element $d \in D^*$. The definition of the independent partitioning ensures that if $t \in T_i^*$, then $d \in D_i^*$. Once the page for D_i^* has been constructed, the construction specifies that the page be referenced by the page table for t in the same slot that p had previously been located. The allows d to be fully visible to t.

(2) *Mutual Exclusion Constraint:* Each element $d \notin D^*$ will retain the same physical and logical address in M'' that it does in M'. Since they do not change location, no non-affected element will overlap with another. The affected elements need to be inspected to verify that they do not overlap with one another, or with a non-affected element.

Utilizing a proof by contradiction assists this inspection. Pretend d_a and d_b are two elements that overlap in memory, such that $M''(d_a) \leq M''(d_b) \leq (M''(d_a) + s(d_a))$. There are a variety of cases that need to be inspected.

(a) *Case: Both d_a and d_b are part of the same minimally dependent subset, $d_a \in D_i^*$ and $d_b \in D_i^*$.* If the d_a and d_b are both members of the same minimally dependant set, the construction explicitly prevents the overlap from occurring. An element either starts the page, or is placed immediately following the previously placed element, such that $M''(d_n) = M''(d_{n-1}) + s(d_{n-1})$. This prevents elements from the same minimally dependant subset from overlapping.

(b) *Case: Both d_a and d_b members of D^*, but not in the same subset.* If both of the d_a and d_b are affected, but not members of the same subset, then by definition of independent subset, $\nexists t \in T$ such that $\langle t, d_a \rangle \in R$ and $\langle t, d_b \rangle \in R$. Additionally, the definition of the construction builds a separate page for each subset D_i^*. This guarantees that the elements occupy different pages.

(c) *Case: One element is affected, but the other is not.* The definition of the construction ensures that affected elements are placed onto a page only with elements from the same subset. During this page construction, affected elements are not placed onto pages with unaffected elements. As such, an overlap cannot occur between d_a and d_b when only one of them is affected.

(3) *Type Homogeneity Constraint:* Once again, non-affected elements maintain the same physical and logical address and page membership in M'' that they do in M'. Additionally, the subsets of D^* are each mapped to their own page. This construction of new pages prevents any affected elements from being stored on a page with non-affected elements. Since M' is valid, all the elements stored on p must have the same memory type. We defined D^* as all elements that start on p, so its elements will also be of homogeneous memory type. Regardless of the manner in which the elements are partitioned, any page that includes only elements that are part of D^* will also have homogeneous memory types. □

The partition/reorder process ensures that a page that lost an element due to compression will satisfy the Ordering Constraint. Additionally, the virtual memory footprint of the elements will not increase during this process, and in many cases will be reduced, requiring fewer pages in the page table. This reduction in memory footprint can cause an Ordering Constraint issue on the page that immediately follows, if an element spanned from the affected page onto the following page in M', and no longer spans in M''. Figure 3.7 shows an example of such an element, d_6 in M', which is removed from Page 3 after partition/reorder is applied to Page 2, resulting in M''. This situation can be remediated by simply applying the partition/reorder process to the following page, ultimately resulting in M'''. Since the process cannot cause Ordering Constraint issues on any preceding pages, at most $Q - 1$ partition/reorders will be required.

Application of the partition/reorder process, potentially repeatedly, to M' produces a new mapping M'' which satisfies the Ordering Constraint while maintaining the satisfaction of all other constraints that were satisfied in M'. When used in conjunction with the compression step, this forms an elementary move that can be used to compress any mapping into a compressed mapping that is a canonical representation for its equivalence class of candidates.

The elementary move, consisting of a compression step and a partition/reorder process, results in a single compressible element being moved one byte earlier in logical memory, while maintaining all validity constraints. A single application of an elementary move results in a mapping M' which is more compressed than the base mapping M, but in most cases there are still compressible elements in the mapping. Fortunately, since the satisfaction of all validity constraints is maintained in M', it can be used as a base mapping to which another elementary move can be applied. This process can then be repeated as many times as necessary until the produced M' does not contain any compressible elements.

Lemma 3.3.3. *Repeated application of the elementary move will result in a compressed mapping within a finite number of steps.*

Proof. Repeated application of the compression move is only useful if it continues to make progress. A minor change to the selection of which compressible element d^* to compress provides this guarantee. Application of the compression move to d^* moves it forward in logical address space. This move does not affect any elements that come before it in the space, but can affect elements that come after it. Ensuring that each compression move selects the first compressible element possible, that which has the

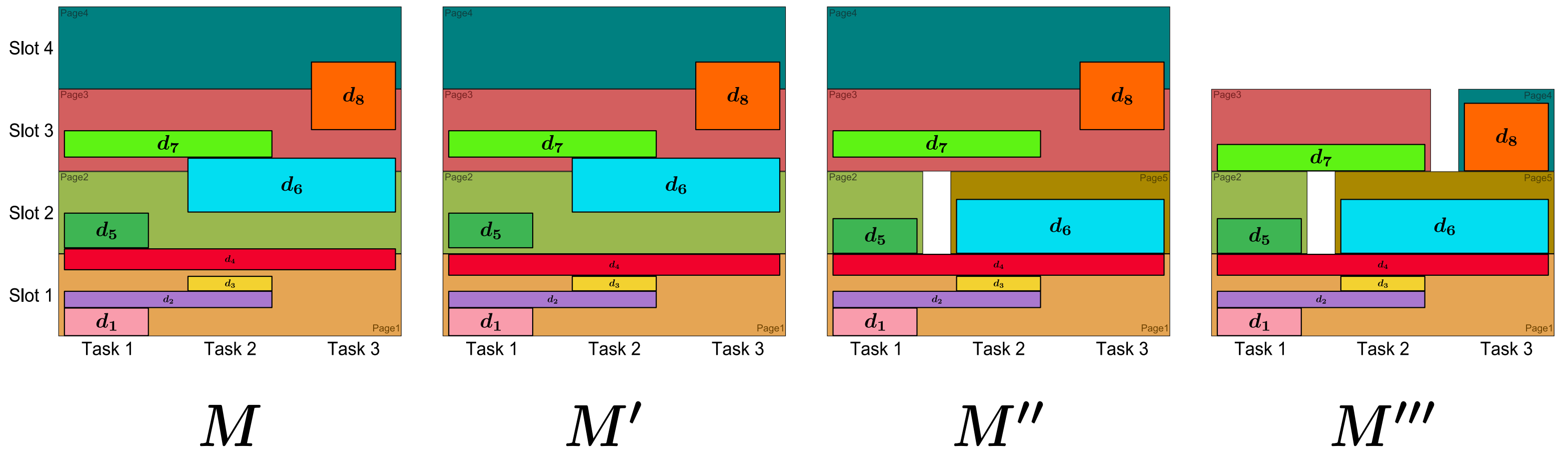


Figure 3.7: Application of a partition/reorder process to a page can result in the removal of an element from the following page. Element d_4 is compressed, producing M', which does not satisfy the Ordering Constraint on Page 2. Application of partition/reorder to the elements on Page 2 results in M'', with a newly created page storing d_6. The removal of d_6 from Page 3 causes another violation of the Ordering Constraint, this time on Page 3. Partition/reorder is applied again, resulting in the final, fully compressed mapping M'''. Note that there is significantly less polluted space in this final mapping.

lowest logical address, guarantees that repeated application of the elementary move will terminate with a compressed mapping in a finite number of steps.

Applying an elementary move to d^* will not alter the logical address of any elements that precede it. The definition of the compression step of the elementary move specifies that all elements $d \neq d^*$ maintain the same physical and logical address, specifically $M'(d) = M(d)$. The partition/reorder process can change the physical and logical address of other elements, but it only operates on elements that begin after d^*, since they are elements that start on the page that begins with d^*_- in M. Thus, the only elements that can change logical address during the elementary move are d^* and other elements that have a later starting logical address and have the same memory type, since they shared a page in M.

Since elements that start before d^* are not affected by the elementary move, the move cannot introduce any compressible space before any of these elements. This guarantees that once an element has been compressed to the point that it no longer has empty space before it, compressing other elements will not create any gaps. This is useful because each elementary move can be applied at most $P - 1$ times to each compressible element, each moving it one byte forward in the page until it is adjacent to another element, or is the first element on the page. At most P elementary moves will be applied to each of at most $|D|$ elements, so the upper bound on elementary moves performed is $P \cdot |D|$. □

Theorem 3.3.4. *If a mapping exists, a compressed mapping also exists.*

Proof. We have defined a set of methods that can be used together to produce a compressed mapping given any valid mapping. The elementary move consists of two steps. The compression step, presented in Lemma 3.3.1, moves a compressible element one byte earlier in the mapping, while guaranteeing that most validity constraints are still satisfied, with the potential exception of the Ordering Constraint. In cases that result in the Ordering Constraint being invalidated, the partition/reorder construction defined in Lemma 3.3.2 shows that it is always possible to adjust and reorder elements on a subsequent page so as to satisfy the Ordering Constraint without invalidating any other validity constraints. This elementary move can be repeated until a compressed mapping is produced, as shown in Lemma 3.3.3. □

3.3.3.2 *Candidate enumeration via oracle* Theorem 3.3.4 shows that a valid compressed mapping will exist in the space of compressed candidates if a valid mapping exists in the space of candidates. This allows the compressed mapping to serve as a canonical representation for its equivalence class of mappings. As long as we can consider all compressed mappings, we are guaranteed to find a representative for any valid mapping in the search space. Of course, this requires an effective procedure for traversing the space of compressed mappings. Fortunately, we can show that there is a natural correspondence between compressed mappings and oracles. This lets us explore, and even prune, the space of candidates by exploring the space of oracles.

Theorem 3.3.5. *Exploration of the space of oracles O produces a mapping in the space of candidates C if one exists.*

Proof. Theorem 3.3.4 guarantees that, for every valid mapping in C, a compressed mapping exists which is also valid. We show a correlation between the space of compressed mappings and O.

We can construct an ordering, o, that yields M under mapping. To do this, the elements in o must be ordered based on their logical address in M, with ties broken arbitrarily. In constructing ordering o, we define the rank of element d_i as the number of elements that must come before d_i in the ordering.

$$\text{rank}(d_i) \quad = \quad \Big| \{d_j \in D \mid l(d_j) < l(d_i) \vee (l(d_j) = l(d_i) \wedge j < i)\} \Big|$$

We can then build the ordering as the sequence of elements ordered by rank:

$$o[\text{rank}(d_i)] \quad = \quad d_i$$

In order to ensure that M will be examined during a search of O, we must show that application of the mapper function specified in 3.3.2.1 to o will produce a compressed mapping M' such that $M' = M$. For the purposes of this proof, we need to easily reference the logical address of elements with respect to both M and M'. Let $l_M(d)$ to be the logical address of d in the mapping M. Similarly, let $l_{M'}(d)$ be the logical address of d in the constructed mapping M'.

Since the mapper function proceeds through the data elements according to the order specified in o, we can use induction over the placement of elements to show that each element is placed in the same logical address in M' that it occupies in M. The base case is trivial; when no elements have been placed in M', have the placed elements been placed into the same logical address in M' that they do in M? Since no elements have been placed, all the mapped elements agree with their location in M.

For the inductive case, we examine the status of M' after the placement of an arbitrary element d^*, where $d^* = d_{o[i]}$ for some $i \in \{0, \ldots, (|D| - 1)\}$. For any such d^*, assume that $\forall j \in \{0, \ldots, i-1\}$, logical address $l_{M'}(d_{o[j]})$ has been assigned such that $l_{M'}(d_{o[j]}) = l_M(d_{o[j]})$. To ease validation, we define the set of placed elements to be $D^p \subset D$. This subset is the first i elements in D according to the ordering, specifically, $D^p = \{d_{o[0]}, \ldots, d_{o[i-1]}\}$. Similarly, we define the set of elements that have not been placed to be $D^s = \{D - D^p - d^*\}$.

Since o is defined in terms of M, elements occur in o in the order of their logical addresses in M. The placement of d^* is highly dependent on the logical address of the element that precedes it in the oracle, $d_{o[i-1]}$. Since d^* can start no earlier than $d_{o[i-1]}$ in logical memory, we can be sure that $l_M(d^*) \geq l_M(d_{o[i-1]})$. Additionally, since M is valid, we can be sure that $l_M(d^*) \leq (QP - s(d^*))$, since starting d^* any later would not allow it to fit into page table space, thus violating the Element Visibility constraint. These restrictions provide a range of possible logical addresses to be considered: $\{l_{M'}(d_{o[i-1]}), \ldots, QP - s(d^*)\}$.

The mapper function examines each logical address $a \in \{l_{M'}(d_{o[i-1]}), \ldots, QP - s(d^*)\}$. It starts with the first possible logical address, and continues to examine successive logical addresses until it finds a logical address which satisfies all validity constraints. Such a logical address is considered to be a *valid logical address* with respect to M'. Once a valid logical address is found with respect to M', $l_{M'}(d^*) \leftarrow a$, and the next element is considered.

In order to verify that the logical address chosen by the mapper function for $l_{M'}(d^*)$ is the same as $l_M(d^*)$, we consider each value for a that is tested during execution of the mapper. These logical addresses can be divided into two cases: either $a < l_M(d^*)$ or $a = l_M(d^*)$. Each logical address that matches the first case is clearly not the correct assignment, so we must show that the mapper function will never find that address to be valid. Conversely, the second case is the correct value for a, so the mapper function must find it to be valid and use it for $l_{M'}(d^*)$.

Case 1 ($a < l_M(d^*)$): This is the incorrect logical address for d^*, so the mapper should find a to be invalid. This can be demonstrated via proof by contradiction. Pretend that a is valid in M'. We show that placing d^* into $l_M(d^*)$ makes it an uncompressed element, which causes a violation of the compressed quality of M, a contradiction.

Since M is a compressed mapping, it must satisfy all criteria for a compressed mapping, specified in Section 3.3.1. Specifically, we examine whether d^* is a compressed element in M. If d^* is uncompressed, then M would also be uncompressed, resulting in a contradiction. The element d^* is a compressed element in a task when either:

(1) d^* is not referenced by the task

(2) d^* immediately follows another element

(3) d^* starts on a page boundary, and cannot be placed on the preceding page

We consider d^* with respect to the tasks that require it, specifically $T_{d^*} = \{t \in T \mid \langle t, d^* \rangle \in R\}$. This definition of T_{d^*} allows us to focus on the second two requirements of a compressed element, since the first criterion would not apply to these tasks. It is also important to note that since a is a valid logical address for d^* in M'', which differs only in the assignment of logical address to d^*, all elements in D^p must be placed into logical address space of M before a in a manner that does not overlap with a. At a minimum, all elements in D^p which are required by a task in T_{d^*} must end before a. We call this requirement *Precedence Requirement #1.* Additionally, any elements in D^p which occur in the page on which a starts, even those that are not required by a task in T_{d^*}, must also end before a, specifically $\forall d \in D^p$ where $\left\lfloor \frac{l_M(d') + s(d')}{P} \right\rfloor = \left\lfloor \frac{a}{P} \right\rfloor$, end before a, $l_M(d) + s(d) < a$. We call this requirement *Precedence Requirement #2.*

The second and third requirements are easily examined if we consider two cases: either $l_M(d^*)$ is in the middle of its page and immediately follows another element, or $l_M(d^*)$ is the first address on its page and cannot be placed onto the preceding page, $l_M(d^*) \bmod P = 0$. We show that neither of these cases can occur, and thus d^* is uncompressed in M.

(1) **Immediately follow another element**: No element $d \in D$ can exist which immediately precedes d^* when placed at logical address $l_M(d^*)$. Our definition of o requires $d \in D^p$, which ensures that it has the same address in M' that it does in M. There are two scenarios that need to be examined:

(a) *a is on the same page as $l_M(d^*)$, such that* $\left\lfloor \frac{a}{P} \right\rfloor = \left\lfloor \frac{l_M(d^*)}{P} \right\rfloor$: Since all elements in D^p which occur on page $\left\lfloor \frac{a}{P} \right\rfloor$ must end before a, and $a < l_M(d^*)$, none of them can occupy $l_M(d^*)$-1, and thus cannot precede d^* in M.

(b) *a is on an earlier page than $l_M(d^*)$, such that* $\left\lfloor \frac{a}{P} \right\rfloor < \left\lfloor \frac{l_M(d^*)}{P} \right\rfloor$: In this scenario, d may or may not have any common requiring tasks with d^*, but the Ordering Constraint requires that there be some preceding element in the page that is required by a common task with d^*. This preceding element is in D^p and is required by a task in T_{d^*}, thus it must end before a, by Precedence Requirement #1. Since a does not occur on the same page as $l_M(d^*)$, it must occur in a preceding page. Thus, such a preceding element cannot exist.

(2) **Start on page boundary, and cannot be placed on the preceding page**: If $l_M(d^*)$ occurs on a page boundary, such that $l_M(d^*) \bmod P = 0$, it is compressed if one of the following criterion is true. We show that none of the criterion can be satisfied for placement of d^* into $l_M(d^*)$, which makes d^* uncompressed in M.

- Any two tasks in T_{d^*} map different pages, p_1 and p_2 where $p_1 \neq p_2$, into the preceding slot, specifically $\exists t_1, t_2 \in T_{d^*}$ where both are defined and $M\left(t_1, \left\lfloor \frac{l_M(d^*)}{P} \right\rfloor - 1\right) \neq M\left(t_2, \left\lfloor \frac{l_M(d^*)}{P} \right\rfloor - 1\right)$.
- The preceding page p is incompatible with d^*
- There is no space available at the end of the preceding page p.

Each task t_1 and t_2 must require at least one element stored on their page in order to map it into the slot. Any such element would be required by at least one task in T_{d^*}, and also be a member of D^p. Since all elements in D^p which are required by a task in T_{d^*} must end before a by Precedence Requirement #1, this scenario requires that a occur after these elements end. Since a must be mapped to the same page in all tasks, it cannot occur on either p_1 or p_2, since $p_1 \neq p_2$. Thus, this situation cannot occur.

Like the first scenario, the p must contain at least one element which is required by a task in T_{d^*}. Precedence Requirement #1 necessitates that a must occur after these required elements, which would place it on p. Since p is incompatible with d^*, a cannot occur on p as such a placement would cause a violation of the Type Homogeneity constraint. Thus, such an incompatible preceding page p cannot exist.

Finally, the last scenario provides two options. If a is not on p, there must exist some element d on p which is required by a task in T_{d^*} for p to be mapped. Precedence Requirement #1 requires a to occur after such a d, which cannot occur since a is not on p, and p immediately precedes the page that $l_M(d^*)$ starts. Thus, such a d cannot exist. If a does occur on p, then p cannot be full. Any element which occupies the last byte in p would be a member of D^p and also occur on the same page as a, which means that it must end before a, by Precedence Requirement #2.

We have shown that d^* cannot satisfy any of the criterion necessary for a compressed element. Thus, in M, it is an uncompressed element. The definition of a compressed mapping necessitates that it not contain any uncompressed elements, thus, this case requires that M be an uncompressed mapping: a contradiction. This proof by contradiction is based upon the premise that a is valid in M' while $a < l_M(d^*)$. The contradictions provided show that a cannot be valid in M'. Each time a logical address less than $l_M(d^*)$ is encountered in the mapper function, it will be found invalid and discarded.

Case 2 ($a = l_M(d^*)$): This is the correct logical address which should be assigned to $l_{M'}(d^*)$, resulting in $l_{M'}(d^*) = l_M(d^*)$. In order for the mapper to select a for assignment, a must be valid in M'. We examine each validity constraint to ensure that it is satisfied in M' when using a.

(1) *Element Visibility:* Since $l_M(d^*)$ is assigned such that all tasks can access all the bytes of d^*, a will also allow every task in M' to access d^*. Assigning d^* to a in M' cannot violate this constraint.

(2) *Type Homogeneity:* Validity of a in M indicates that no element in D violates homogeneity when d^* is assigned to a. Since $D^p \subset D$, no element in D^p, which are the only elements mapped in M', can violate homogeneity in M'.

(3) *Mutual Exclusion:* Validity of a in M ensures that no element in D can overlap with d^* if it is assigned to a. Since $D^p \subset D$, no element in D^p, which are the only elements mapped in M', can overlap with d^* in M', which would violate Mutual Exclusion.

(4) *Ordering Constraint:* Since a is valid in M, either a is the first logical address on its page where $a \bmod P = 0$, or $\exists d \in D$ where $\left\lfloor \dfrac{a}{P} \right\rfloor P \leq l_M(d) + s(d) < a$ which has a task requirement in common with d^*, such that $\exists t \in T$ where $\langle t, d^* \rangle \in R$ and $\langle t, d \rangle \in R$. Recall that D^p is a very specific subset of D, containing all elements which have a lower rank than d^*. Since $l_M(d) < a$, we can be sure that $\text{rank}(d) < \text{rank}(d^*)$, which requires that $d \in D^p$ and the Ordering Constraint be satisfied in M'.

Since none of the validity constraints can be violated in M' when a is valid in M, we know that a also must be valid in M'. This results in the mapper function correctly setting $l_{M'}(d^*) \leftarrow a$.

Analysis of these two cases shows that the mapper function will assign logical address a to $l_{M'}(d^*)$ only when $l_M(d^*) = a$. This extends the set of placed elements which have the same logical address in both M and M'. Since the inductive step holds when applied to the placement of any element in o into the mapping, we can be sure that after all elements have been placed, the constructed mapping M' will be identical to M.

$\square$

CHAPTER FOUR

Design and Implementation

We have shown via formal examination that MEM-MAP requires search. We have also presented some techniques to reduce the number of nodes that need to be examined during the search, such as only considering compressed candidates, but even this reduced search space is likely to overwhelm naïve, exhaustive search methods. The search space of all possible oracles is asymptotically bound as $O(n!)$, where n is the number of data elements.

We provide a search framework that is extensible enough to evaluate a variety of search-based procedures with the goal of identifying a class of procedures that are best suited to finding solutions to the problem. This framework must be well designed to support the different needs of the various search procedures. A well-written implementation based on this framework provides the vehicle for analyzing and testing the procedures.

4.1 Search Framework Design

The first step of the search framework design is to define primary design elements that the design should adhere to. Since the goal of the framework is to be extensible for evaluating various procedures, the design should focus on making the framework as configurable as possible. We use C++ for our implementation, which permits the use of objects and abstraction to build the extensible modules.

The size of the problem necessitates that the design also take into account the performance of the implementation. Since we expect to evaluate each procedure experimentally, as many inefficiencies as possible need to be eliminated from the implementation. Using C++ for the implementation helps with the speed because the code can be highly optimized for performance once the class of most effective search procedures is identified. This varying amount of optimization allows the code to be designed using extensible modules, which can be interchanged during prototyping and initial tests. These modules can be manually modified and slimmed down to a present more efficient approach during performance tests.

One of the most important ways to make the search procedures more efficient is to reduce the search space to the smallest possible increment that still allows solutions to be found if they do, in fact, exist. In Chapter 3, we showed that searching the space of oracles is sufficient for finding solutions if they exist in the space of all candidate. The implementation utilizes this relationship in all search procedures. The base representation and search procedures work with oracles to explore the search space.

Using these goals, we have designed a framework that is divided into low-level representation and modification modules, and high-level search procedure logic. The low-level modules provide the key functionalities necessary to perform a search, without relying on details about the search procedure. Search procedures can be expected to reuse the low-level modules to interface with the representation, while providing their own approach to exploring the space. This allows search procedures to be rapidly prototyped and tested.

The working implementation includes the following low-level modules that form the foundation for the framework.

- Problem Representation
- Problem File Parser
- Mapping Representation
- Fitness Function

- Mapper Function to convert from an Oracle to a Mapping
- Mapping Visualization
- Solution File Emitter

4.2 Framework Modules

The implementation of the search framework provides the necessary functionality in a manner that is both extensible and efficient. The major design elements in the framework that need to be extensible are implemented as either C++ classes or structs. Different versions of these classes can be implemented and interchanged as desired to test various performance characteristics.

4.2.1 Symbol Table

A symbol table is used to convert strings used in the problem, such as names of elements, tasks, or memory types, into integers. The integers are used instead of the strings since they are more easily indexed, stored, compared, and referenced. The symbol table is built during parsing by looking up strings via the `lookup` function. This has log performance during the initial lookup and generation of the symbol, but constant time performance when comparing symbols during time-critical portions of the search procedures. A symbol can be easily converted back to a string when a corresponding string is needed during the output of a solution.

```
class SymbolTable {
...
public:
   int lookup(string const &s);
   string const &getString(int i);
};
```

4.2.2 Problem Representation

The input file, which is in Map2 file format, provides a wealth of information about the problem description. This file format, however, is redundant in its specification. Since the implementation needs to be efficient, we need an internal representation of the problem that captures the minimal amount of information necessary to perform the search. The list of elements, `elist`, and list of tasks, `tlist`, capture this necessary information.

```
struct Problem {
    ...
    vector<Element> elist;
    vector<Task> tlist;
};
```

4.2.2.1 *Element* The `Element` represents a data element from the mapping problem; it is the structure that is mapped into pages of the page table. It contains a `name` and memory type, `memtype`, for the element, each stored as an **int** index into the symbol table. It also has a `size`, which is the number of bytes that it requires in both logical and physical memory for it to be useful. The `pred` reference indicates the element that it should immediately follow in the mapping, per the specification of the Map2 input file. The `dctype` indicates if the element is a data or code element, which is again references as an int against the symbol table. The biggest hurdle in mapping current instances of the problem revolve around mapping the data elements, although the code elements also need to be tracked. Finally, the element keeps a list of all tasks that require it, `ref`, stored as a sequence of task indices into the problem's `tlist` vector.

```
struct Element {
    int name;
    int size;
    int memtype;
    int pred;
    int dctype;
    vector<int> ref;
};
```

4.2.2.2 *Task* A `Task` is a particular slice in the time-slice architecture of the mission computer. It is also identified by an integer `name` field, which references a name stored in the `SymbolTable`. The `req` vector stores the integer indices into the problem's `elist` of all elements that are required by the task. Each of these elements must be completely mapped into the task's page table in order for the mapping to be considered valid. The `size` integer is a convenience variable for quickly identifying how many total bytes the task requires. This can be useful when deciding which task's page table is the most critical to save space in. A task is identified by its index in the `tlist` vector, which is considered to be this the task's `taskID`.

```
struct Task {
    int name;
    vector<int> req;
    int size;
};
```

4.2.3 Parser

The `Parser` builds an instance of `Problem` from a given input problem file in Map2 format, specified by `filename`. During construction, the parser creates each element and task specified in the input file and adds them to the problem. It also populates all necessary reference variables so that they are internally consistent and match the input problem. During this process, any unnecessary, redundant information present in the file is eliminated, so the problem contains only the minimal information necessary to represent the problem instance in `filename`. In particular, any existing physical and logical address assignments specified in `filename` are discarded, allowing the implementation to start with a clean slate of address assignments.

```
bool parseProblem( Problem &prob, string filename );
```

4.2.4 *Generalized Representation*

The information directly parsed from the Map2 file format provides an explicit definition of the problem, but the `Problem` module distills this information to a minimal representation. Creating additional data structures from the base representation facilitates efficient search procedure design. The `EBlock` module, for example, is not a structure specified in the Map2 file format, but it is used to represent groups of contiguous elements. This additional data is stored in the `Representation` module, which also has a copy of the internal `Problem` definition. Note that the `Representation` contains the `parse` method, described in Subsection 4.2.3, which allows it to parse a Map2 problem specification file into its associated problem instance.

```
class Representation {
 public:
   Problem prob;
   vector<EBlock> blist;

   void parse(string filename);
   void buildGenericMemMap();
   int lookupGenericMemType(int memType);
   map<int, int> genericMemMap;
   ...
};
```

4.2.4.1 *Blocks* The Map2 input file format allows the specification that certain elements must immediately follow other elements. This is facilitated in the Element module by `pred`. Ensuring that all of these precedence rules are satisfied by inspecting each element is not an efficient representation. A *Block* is a grouping of multiple elements together into a single representation. This allows procedures to deal with the groups of elements, rather than individual elements. When a block is mapped, the elements in the block are mapped contiguously, which satisfies the precedence requirements. The `EBlock` struct provides this aggregation.

In addition to the precedence relationship defined in the file format, blocks are also used to group elements that have a compatible memory type and are required by the same set of tasks. This task requirement *fingerprint*, the tasks which require the element, is important to consider since elements with the same fingerprint will be required in the exact same tasks. Since they are compatible for storing on the same page, and are needed in the same tasks, mapping them contiguously is an efficient use of space.

Each `EBlock` contains a list of all elements that are in the block, `elist`, as well as all tasks constitute the fingerprint of the elements in the block, `ref`. In certain cases, due to predefined precedence relationships, not all elements in the block will have the same task requirement fingerprint. In this case, the fingerprint of the block is the union of the tasks in each element's fingerprint. The convenience members `memType`, `genericMemType`, and `size` provide convenient access to details about the elements stored in the `EBlock` during mapping.

Every `EBlock` is stored in the `blist` vector of the mapping. The index of the block in the vector is that block's *blockID*.

```
struct EBlock {
    vector<int> elist;
    vector<int> ref;
    int memType;
    int genericMemType;
    int size;
};
```

4.2.4.2 *Generic Memory Types* The Type Homogeneity constraint specifies that all elements on a `Page` must have a compatible memory type. Rather than specifying a function to check compatibility of each memory type during mapping, which would be heavily used, we pre-compute a generic memory type for each memory type. This precomputed generic memory type serves as a canonical representative for its class of equivalent memory types. The blocks then keep track of the generic memory type of the elements stored on the page. This allows a constant-time comparison of compatibility during element assignment to a page, which helps performance. The mapping of a specific memory type to its generic counterpart is stored in the `genericMemMap` vector. The `buildGenericMemMap` and `lookupGenericMemType` functions provide the means for building the mapping from a specific memory type to the generic counterpart, and performing the lookup during parsing and block generation.

4.2.5 Mapping Representation

Once the problem has been built by the parser, a means of representing both the physical and logical address space is necessary. The `Mapping` class provides the structures necessary to represent an assignment of physical addresses to blocks of elements, and the population of the page table for each task. Once a mapping has been fully constructed, it can be tested for satisfaction of the validity constraints. If all validity constraints are satisfied, a mapping can be combined with information from the problem to completely specify a solution to the problem instance. The mapping is only valid in the context of the problem and representation and thus it maintains the `Representation rep` on which it is applicable.

```
class Mapping {
 public:
   Representation rep;
   vector<Page> pages;
   vector<vector<int> > pageTable;
   vector<pair<int, int> > addressTable;
   vector<vector<int> > block2page;

   // Visualization functions
   void writeCommonFormat(string filename = "solutionCandidate.txt");
   void writeSVG(string filename);
   ...
};
```

4.2.5.1 *Page* The `Page` class is needed to ensure that elements are only stored in one physical location in the entire mapping. A `Page` represents a section of physical memory that is mapped into the page table for one or more tasks. Each `Page` keeps track of how much space is used on the page, represented by `size`. Since the mapper function builds compressed mappings, it is understood that any free space on a page is at the end. As each `Page` is allocated, the mapping stores it in the `pages` vector of the mapping. The page is assigned an index based on its location in the `pages` vector, called the *pageID*.

To facilitate checking the Type Homogeneity constraint, each page keeps track of the memory type of the elements stored on it via the `genericMemType` variable. Since the mapper function obeys the validity constraints during construction of the page table, no elements are added to a page that would conflict with the `genericMemMap` of that page.

The mapper function allocates the physical addresses after the logical addresses have been assigned. As such, each page does not keep track of its specific location in memory. Simply maintaining which page follows the current page via the `nextPage` variable is sufficient for placing the page in physical memory.

The most important aspect of a page is the location of each block that is stored on it. The `blockPositions` vector maintains this information by storing pairs of blockID's, and their starting position on the page. If an element started on an earlier page and continued onto the the current page, its starting position is indicated by -1. Since every element starts on a page, the the logical address of each element can be recovered from the page on which it starts. Once the pages have been placed into physical memory, the actual physical address of each element can be determined by adding the page's physical starting address to the location of the element on the page.

```
struct Page {
   unsigned int size;
```

```
    int genericMemType;
    int nextPage;
    vector<pair<int, int> > blockPositions;
};
```

4.2.5.2 *Page table* The page table in the mapping specifies where each task should load pages necessary to satisfy execution requirements. Since both tasks and pages have integer ID numbers, a simple two-dimensional vector is used to maintain a record of which page each task loads into each slot of the page table. The integer `pageTable[i][j]` stores the index of the page that should be loaded into the `j`'th slot of task `i`'s page table.

4.2.5.3 *Block to page* While each page keeps track of the blocks that are stored on it, it is also very helpful to keep a reverse lookup index of this information for verification purposes. The `block2page` vector of vectors keeps track of all pages that each block is stored on. `block2page[i]` is a vector of all pages on which blockID `i` is stored on. When validating requirements, this allows each task to verify that it maps all necessary pages into its page table to satisfy its requirements.

4.2.6 Mapper Function

We have shown in Chapter 3 that enumerating mappings is straightforward, but not practical. Instead, we enumerate oracles during search. These oracles cannot be directly examined for satisfaction of validity constraints. Instead, they must be converted into a mapping for such examinations, a conversion process that we introduced in Chapter 3 as a mapper function. This conversion is a critical piece, since each oracle that is examined must be converted. An efficient mapper function is crucial to the performance of the implementation.

There are a variety of approaches to constructing the mapping. The abstract `MappingMethod` class is used to facilitate developing different methods. Each subclass of `MappingMethod` class must provide both an explicit and implicit method for performing the mapping, in addition to methods to calculate the fitness of the mapping. The explicit mapper, called `slowMap`, must explicitly define the location of every element, while the implicit mapper, called `fastMap`, is allowed to take shortcuts in an attempt to optimize. Since they use take different approaches to placing elements, a means of verifying that that elements are placed into the same place is needed. Each method has its own fitness calculation function which can be compared to provide this verification. Additionally, the `validateSlowFitness` function verifies that that all requirements are truly satisfied during after explicitly mapping the oracle by inspecting every byte in logical and physical memory. Since this byte-by-byte examination requires an explicit definition of where each element is stored, it must be run after `slowMap` has been performed.

```
class MappingMethod: public Mapping {
 public:
   virtual void slowMap(const Oracle &oracle) = 0;
   virtual void fastMap(const Oracle &oracle) = 0;
   virtual Fitness slowFitness(const Oracle &oracle) = 0;
   virtual Fitness fastFitness(const Oracle &oracle) = 0;

   bool validateSlowFitness();
   ...
};
```

4.2.6.1 *Oracle* Since the implementation deals with blocks during the mapping process instead of individual elements, each search procedure considers an `Oracle` to be a sequence of blocks, instead of a sequence of elements, as described in Chapter 3. Since a block is simply an aggregation of elements with compatible memory types, potentially only containing a single element, this modification of the definition of oracle is very useful. There are typically fewer blocks than elements, and the blocks can be assigned physical and logical addresses just like an element. The oracle is simply a vector of IDs for each block. Figure 4.1 shows an example oracle, in which

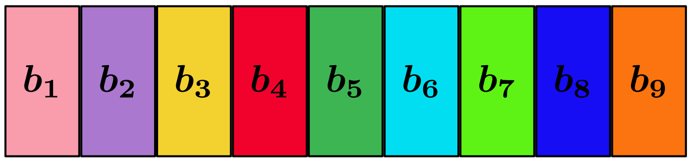


Figure 4.1: An example oracle, which specifies the insertion blocks $b_1, \ldots, b_9$ into the mapping. This representation of an oracle depicts block insertion order moving from left to right.

blocks are are inserted into the mapping as they appear left to right, starting with block b_1 and ending with block b_9.

```
struct OracleStruct {
   int blockID;
   ...
};
typedef vector<OracleStruct> Oracle;
```

4.2.6.2 *Explicit mapper* The accuracy of the Mapper Function is critical to the correct functioning of the implementation. The Explicit Mapper, defined in the `slowMap` function of a `MappingMethod`, converts an oracle into a mapping by assigning each element a logical address in the page table. Each block position in the page table is kept until the mapping has been fully generated. Each task that references the element stores the block in that logical address of its page table. When the Explicit Mapper has finished, every block has been placed into the page table.

4.2.6.3 *Implicit mapper* Explicitly mapping the entire contents of the page table is necessary for building a mapping that has enough information to produce a full Map2 output file, but it stores more information than is strictly necessary to evaluate quality of the oracle. Typically, each fully-formed oracle is evaluated based on the oversubscription fitness measure, which is not concerned with the individual placement of each element.

The Implicit Mapper, defined in the `fastMap` function, only maintains enough information to evaluate the oversubscription fitness measure of each oracle. It accomplishes this by only storing the last slot and page used for each task, the `addressTable` of the mapping. The assignment `addressTable`[`i`] = `pair`(`j`,`k`) would indicate that task `i` has last used slot `j` with page index `k` stored there. This is sufficient to determine the placement of the next block, since each block can start no earlier than any block that precedes it in the oracle. This simplification of the representation results in a significant performance increase, since each task need only keep track of its last logical address used, instead of storing all blocks that were placed into each task during the mapping.

4.2.7 Fitness Function

We have discussed how some oracles will produce a valid mapping, while other will not. In most cases, the majority of oracles analyzed will be invalid. We need some metric to distinguish the quality of these invalid oracles, which we call the fitness of the oracle. The value of this metric is defined as a `vector`<**unsigned int**>, with each element of the vector representing a different measure of fitness for the mapping. This supports a flexible fitness measure, allowing secondary measures to break ties between primary measures, and providing the ability to easily sort mappings based on more than one specific measurement.

There are a variety of approaches to measuring the fitness of an oracle. We define a set of fitness components that measure different aspsects of an oracle. Each search procedure has its own requirements for how the fitness should be calculated, so having a set of fitness components to build a custom fitness function provides some flexibility. Figure 4.2 provides an example oracle and resulting mapping that demonstrate each of the fitness measures.

- *Oversubscription:* One method of measuring the fitness of an oracle is to loosen the restrictions on a specific validity constraint that is easy to measure: the size of the page table. When using this method, the mapper function

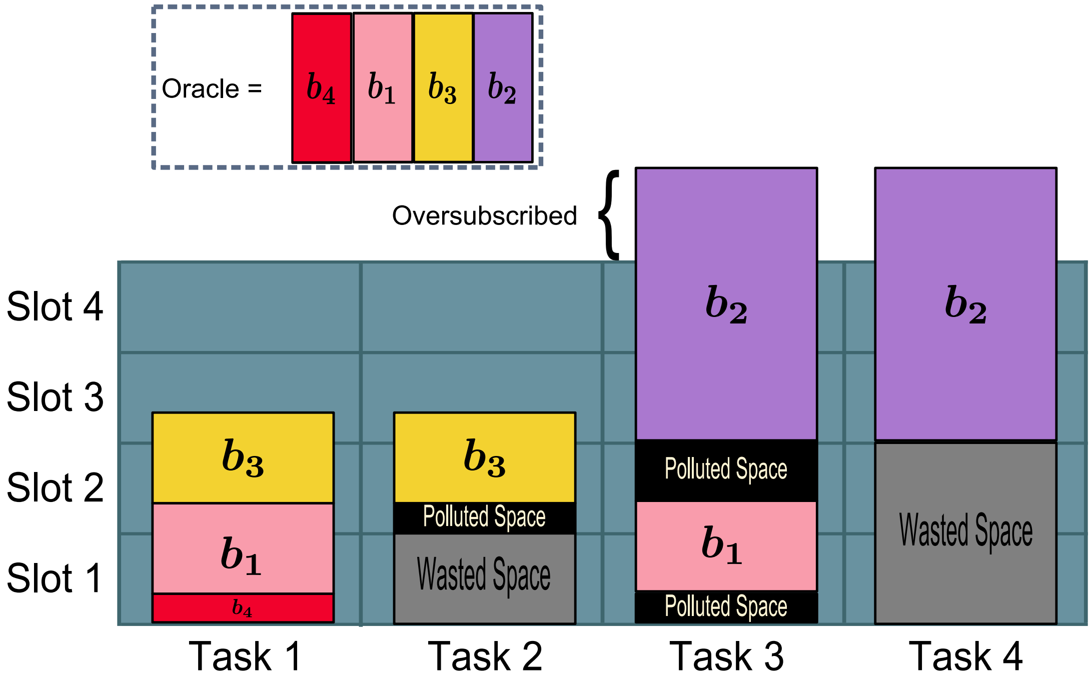


Figure 4.2: An example mapping of an oracle that shows each of the fitness measures. The oracle (b_4, b_1, b_3, b_2) is mapped into a page table containing four tasks with four slots. Oversubscribed space is the portion of the blocks that does not fit in the page table. Polluted space is indicated in black, while wasted space is indicated by grey.

must make sure that the produced mapping satisfies all validity constraints, except that it may allow the mapping to overrun its page table. We call a task that has overrun its page table an *oversubscribed* task. The degree to which the page table length is overrun can serve as a basis for calculating the fitness metric.

There are a variety of ways to utilize this oversubscription to measure the fitness of an oracle. These include measuring the number of elements that do not fit into the page table, counting the number of bytes that are required in excess of the page table capacity, or finding the maximum number of bytes that any task requires beyond the page table capacity. Each `MappingMethod` can provide a different method for calculating the oversubscription fitness score. The oversubscription fitness score cannot be effectively calculated on a partial-oracle since, in most cases, a partial-oracle will not oversubscribe the page table, preventing this score from differentiating between partial-oracles.

- *Wasted Space:* There are cases where an element is mapped in a manner that leaves unallocated space between the element and its predecessor. This can occur due to a variety of reasons. The most common reason is the elements having different memory types, which prevents them from being mapped to the same page. The issue is that once an element is placed in a manner that leaves this unused space, the strict ordering of an oracle dictates that no other element can be assigned to the empty space, which is effectively wasted.

  The wasted space fitness method keeps a running total of all wasted space that occurs during the mapping process. In most cases, wasted space will be unavoidable, even in valid solutions. However, significant amounts of wasted space can quickly prevent mappings from being valid. Typically, mappings with less wasted space are more desirable. Since the wasted space can be tracked during the application of the mapper function, the wasted space method can be applied to any oracle, including partial-oracles. Wasted space is displayed in Figure 4.2 in grey. In this example, wasted space is found in whole-page increments, although that is not necessary.

- *Polluted Space:* The concept of pollution is defined in Chapter 1 as the presence of elements with differing task requirements on the same page. The space is considered polluted because the page is loaded into memory for all the required tasks, but any element that is not required by the specific task is effectively wasted space for that task.

  Like wasted space, polluted space is generally unavoidable, even in valid mappings, since it is not feasible to ensure that all elements on a page have the exact same task requirement list. Polluted space is undesirable, but it has less of a negative impact than wasted space, since the polluted space is used by at least some task, as opposed to wasted space which is not used at all. It is, therefore, generally better to pollute a page with an element that has a different task requirement list than to eliminate pollution by starting a new page and wasting the space. The polluted space method measures the degree to which elements with differing task requirements are assigned to the same page. Reducing this pollution will generally lead to a mapping that is closer to being valid. Similar to the wasted space method, the pollution method maintains how much space has been polluted during the application of the mapper function, so it is able to provide fitness information for both oracles and partial-oracles. Polluted space is displayed in Figure 4.2 in black.

4.2.8 Search Procedures

An important aspect of the implementation is to provide the ability to test a variety of search procedures for performance and effectiveness. The abstract `GenericAlgorithm` class is designed to provide a common base for developing these procedures. The `runMain`() and `initialize`() functions provide bootstrapping capability, while the `createMappingMethod`() function creates an instance of a `MappingMethod` to be used for converting oracles into mappings and calculating fitness. The `process` function parses the input file and kicks off the search. The intent is that these will be overridden or extended by the derived classes as necessary. The `algName` string is kept so that `runMain` can display which algorithm is being used during execution. The primary piece that must be extended by each search procedure is the `doSearch` function. This contains the details of how each search procedure operates.

```
class GenericAlgorithm {
 public:
   virtual int runMain(int argc, char* argv[]);
   virtual void initialize();
   virtual void createMappingMethod();
   virtual void process(string filename);
   virtual void doSearch() = 0;

 protected:
   string algName;
   Representation rep;
   MappingMethod *method;
};
```

4.2.9 Visualization

The Map2 file format is very precise, but reading through a file in this format does not give a reader a good idea for how the page table is allocated or where spaces occur. The `Mapping` class provides a pair of visualization functions for converting the assignments made to the page table into a more human-readable format. The `writeCommonFormat` function produces a file that concisely specifies the elements

```
# Common Output Format
...
Page 574 R 0
  0 CKSSDB 3
  3 %CSLADB1 1
  4 Q2D20R 1

Page 575 RW 0
  0 Q2EABO 3

Page 576 R 0
  0 %CSLADB0 1

Page 577 RW 0
  0 Q2EM1O 1
...
 PageMap 161
309 262 -1 -1 -1 -1 -1 -1 -1 327 295 -1 -1 -1 -1 -1 -1 -1 -1 321 333 ...
310 263 -1 -1 -1 -1 -1 -1 -1 328 296 -1 -1 -1 -1 -1 -1 -1 -1 322 334 ...
```

Figure 4.3: An excerpt from a file in common file format. Each page specifies the elements that it stores along with offset. The PageMap defines the way that pages are loaded into virtual memory.

assigned to each page, and the contents of each task's page table. This differs from the Map2 file format in that it more directly specifies which elements are stored on each page, and which page is stored in each slot of the page table. Care is taken to make the common file format both specific and easy to read. Figure 4.3 provides a portion of an example file in common file format.

The `writeSVG` function produces an image file for the mapping in Scalable Vector Graphics (SVG) format. Different elements are colored differently and empty space in the page table is colored white. Polluted space is colored black. Slots that are needed to store elements that have oversubscribed the page table are colored darkly, to give the reader a visual cue for where the page table ends. This visual representation gives the user a more intuitive feel for how elements are stored and what impact assignment of an element has on the mapping. Figure 4.4 shows an example SVG for a solution to an instance of MEM-MAP.

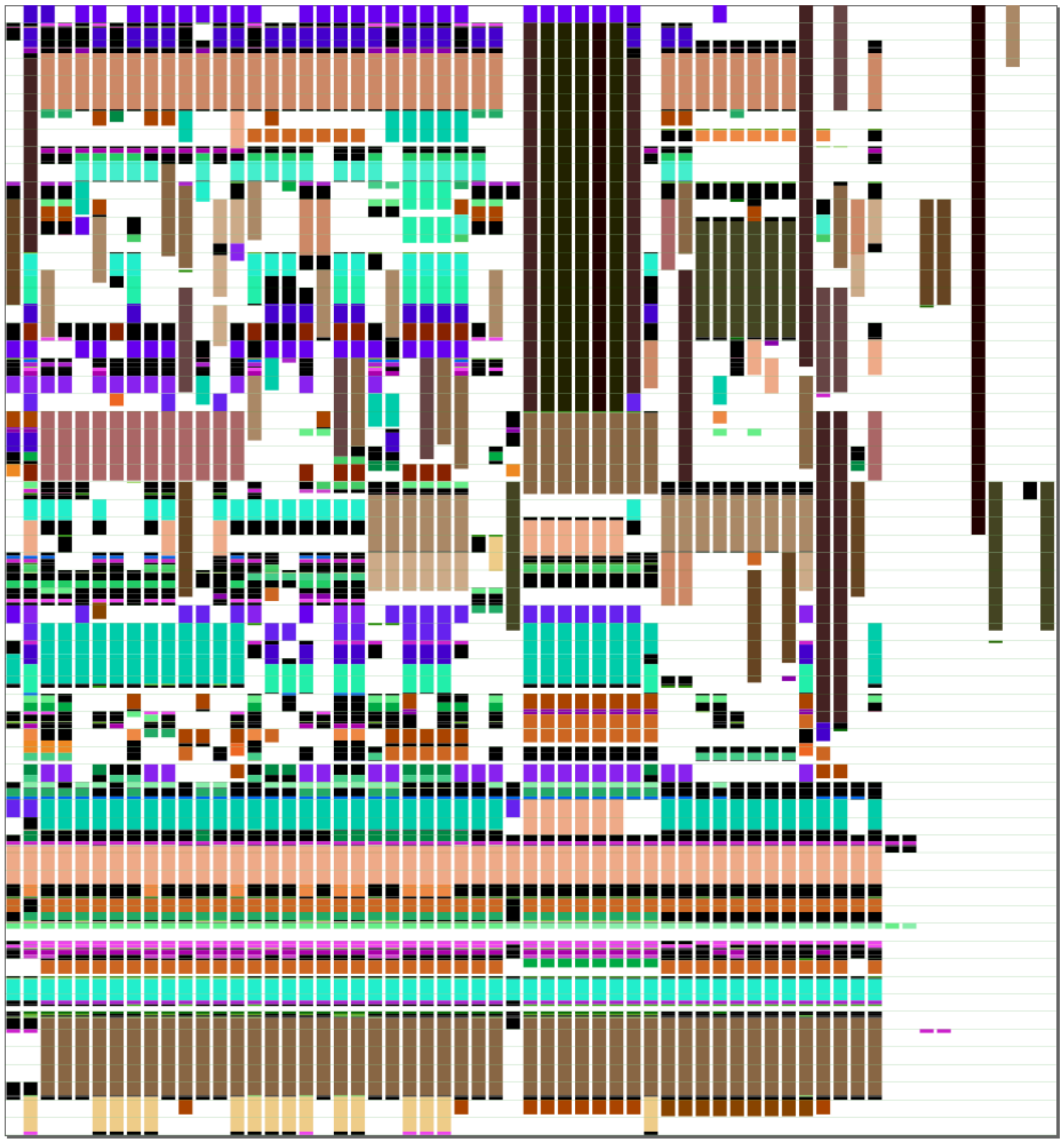


Figure 4.4: An example SVG image for a solution a MEM-MAP problem instance. Data elements are represented by colored blocks, and slots are separated by light green horizontal lines. This example is a valid mapping, and thus all elements fit into the page table.

4.2.10 Map Emitter

Once a valid mapping has been found by a search procedure, the implementation needs a means of generating a Map2 file that corresponds to the assignments made in the mapping. The Map Emitter module translates the page table and element assignments from internal representation to the standard Map2 format. Since the problem was originally parsed from a Map2 input file, many of the details about the problem are available for converting into the file. The portions of the Map2 input file that were discarded during parsing, specifically the logical and physical address of elements, have been determined during the building of the page table using the explicit mapper. Converting the information from problem and page assignments from mapping into Map2 format is straightforward.

4.3 Search Methods Evaluated

The implementation includes a variety of search procedures that are readily adapted to this problem. Due to the specifics of the this problem, each procedure is specifically tailored to apply to this problem, while still adhering to the general principles of the technique described in Chapter 2.

4.3.1 Depth-First Search

Since the problem requires search, including a variety of depth-first search procedures in the implementation provides a good baseline for evaluating other procedures, despite the performance issues associated with the complexity of the problem. In addition to the complete search, we also examine some alterations to the order that child nodes are examined after expansion which attempt to focus the search on the most interesting portions of the search space.

4.3.1.1 *Complete* Our implementation of the depth-first complete search is a straightforward application of depth-first search, with only basic attempts to optimize performance or reduce the size of the space explored, such as pruning. This allows it to serve as a reference implementation which is guaranteed to find a valid oracle, if one exists, when given enough time. The various oracles are examined systematically, without any application of heuristics.

Since it is depth-first, the complete search utilizes a LIFO-stack, with each node in the stack representing a block. This uses a specially defined `TotalOrderMapper` mapping method that provides the ability to incrementally add and remove commitments to a partial-oracle. The removal is accomplished by keeping a vector of commitments made, and providing a backtrack function which restores the mapping to the state specified by an index in the commitment vector. A search based on a LIFO-stack guides us to use a recursive function for the search. As each commitment is pushed onto the stack, the function recurses deeper. When commitments are popped off of the stack, the recursive function backtracks and returns.

```
Mapper myMapper;
search( Oracle O, unmapped )
    if unmapped = $\emptyset$
        return TRUE
    for each element $\in$ unmapped
        Oracle $O' \leftarrow$ O + element
        if myMapper.fitness( $O'$ ) = 0
            if search( $O'$, unmapped - { element } )
                return TRUE
    return FALSE
```

Each recursive step considers every unmapped block as the next block in the partial-oracle. Once a block is placed, the resulting mapping is examined to see if it fits into the page table. If the insertion fits, then the function recurses deeper with the partial-oracle that contains the inserted block.

If an insertion of a block causes a validity constraint to be broken, the insertion is retracted, during which all modifications made during the insertion are undone. If none of the unplaced blocks can be inserted into the mapping, then one of the insertions during a previous recursion step caused the mapping to be invalid, so the recursive function backtracks and returns to the previous step. This backtracking effectively prunes all subsequent block assignments that would have occurred after making the invalid assignment.

4.3.1.2 *Heuristic* The complete depth-first search is a good reference implementation, but it pays no attention to the quality of the nodes that it is expanding. In pathological cases, the worst expansions could be explored first, leaving the most promising expansions for much later in the search. This would result in the search taking a significant amount of time to find a solution.

The heuristic-based search implementation is a variation of the complete depth-first search, specifically addressing the order that child nodes are explored during recursion. Instead of systematically expanding child nodes according to their block ID's regardless of their quality, it applies a heuristic to first expand child nodes that are more likely to lead to a solution.

```
Mapper myMapper;
search( Oracle O, unmapped )
    if unmapped = ∅
        return TRUE
    for each element ∈ unmapped
        h[ element ] ← myMapper.fitness( O + element )
    sort unmapped by h
    for each element ∈ unmapped
        if h[ element ] = 0
            if search( O + element , unmapped − { element } )
                return TRUE
    return FALSE
```

The concept of fitness provides a basis for defining this heuristic. The process in which nodes are expanded gives guidance for which fitness components to utilize in this search. It is important to note that, in a heuristic search, the fitness score is required at internal nodes, when only a partial-oracle has been constructed. Oversubscription is generally unable to differentiate between partial-oracles, especially those containing only a few blocks, since, in most cases, none of the partial-oracles would oversubscribe the page table. In these cases, all of the partial-oracles would have the same oversubscription score of 0.

Fortunately, the wasted space and polluted space fitness approaches are directly applicable to measuring the relative fitness of partial-oracles. Both the wasted and polluted space are tracked during block insertion, and it is very unlikely that a mapping will have absolutely no wasted or polluted space. This means that different partial-oracles will most likely have wasted space and polluted space fitness scores that are reasonably comparable, even when they do not contain an order for all elements.

Once fitness has been computed for each block that is eligible for insertion in the mapping, the blocks are ordered according to their fitness score. Each block is then reconsidered for insertion, beginning with the best block. During this pass, each block that satisfies the validity constraints results in a recursive step. This approach allows the blocks with the best fitness functions to be evaluated first, which will hopefully lead to a solution being found sooner. The heuristic-based search will still completely search the space, if given enough time. It simply applies heuristic logic in an attempt to first focus on block placements that are the best choice at each step in the search.

4.3.1.3 *Incomplete* While the heuristic-based search attempts to focus the complete search on the most promising regions of the search space, it is still a complete search. Both the complete and heuristic search systems attempt to exhaust the search space. However, the size of the mapping search space may be so large as to render these approaches impractical. In an effort to focus attention on the most promising regions of the search space, we also investigate an incomplete, depth-first search procedure which ignores large portions of the search that seem unlikely to contain a solution. Although there is some risk that this procedure will miss a solution, the hope is that the heuristic is good enough to guide the search towards a portion of the search space that contains a solution.

It is easy to adapt the complete, heuristic search so that it ignores many of the least promising regions of the search space. Once the children of a search node have been heuristically ordered, only the best k expansions are considered, while the remaining least preferred expansions are discarded from consideration. For our tests, we specify that $k = \frac{\texttt{blist.size()}}{2}$. Using this value for k allows the algorithm to keep a higher percentage of possible expansions at each level. This relies on the premise that, if the best expansions for a node are unable to lead towards a solution, then the least preferred expansion will likely not either. After the subtrees rooted at the best k children have been explored, the search backtracks to the parent node and continues. This limit prevents the incomplete search from searching the entire space, but it aims to reduce the amount of time spent stuck searching uninteresting portions of the search space. This code is the same as the heuristic recursive function, except it imposes the MAX_EXPAND limit.

```
Mapper myMapper;
MAX_EXPAND ← $\frac{|blist|}{2}$
search( Oracle O, unmapped )
   if unmapped = ∅
      return TRUE
   for each element ∈ unmapped
      h[ element ] ← myMapper.fitness( O + element )
   sort unmapped by h
   for each element ∈ unmapped limit MAX_EXPAND
      if h[ element ] = 0
         if search( O + element, unmapped − { element } )
            return TRUE
   return FALSE
```

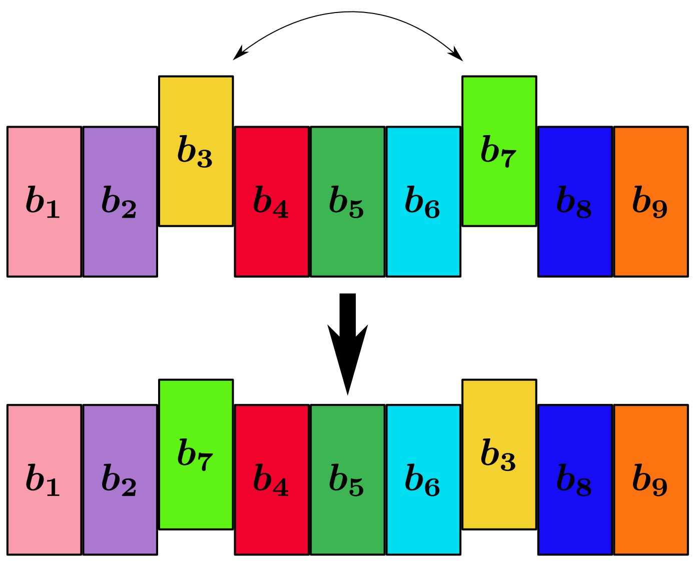


Figure 4.5: An example swap operation, which swaps the location of blocks b_3 and b_7 in the oracle.

4.3.2 Hill Climbing

The hill climbing family of implementations is a major portion of the incomplete search techniques that are part of the implementation. In the application of hill climbing to the problem, each oracle is represented by a state in the space; no partial candidates are considered. We impose a neighbor relation on the set of oracles using a variety of schemes, some dense, some sparse. These schemes consider oracles to be neighbors if one or many elementary operations can be performed to convert one oracle into the other, depending on the scheme. The elementary operations include:

- *Swap:* A swap exchanges the location of two blocks in the oracle, keeping everything else in place. Figure 4.5 presents an example swap operation on an oracle.

```
void swap(Oracle &oracle, int a, int b) {
    swap(oracle[a],oracle[b]);
}
```

- *Move:* A move is simply the movement of a block from one location in the oracle to another. All elements that follow the block are moved forward

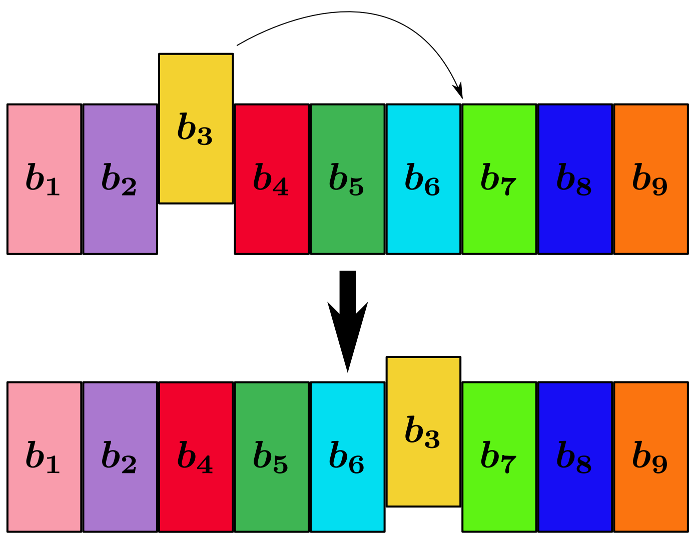


Figure 4.6: An example move operation, which moves block b_3 right before block b_7. The relative order of all other blocks remain unchanged.

to make room for the element and to fill in vacated space. Because of the operation of the mapper, this movement can have a significant effect on the resulting logical address to which the block is assigned. Figure 4.6 shows an example move operation.

```
void move(Oracle &oracle, int a, int b) {
    Oracle::iterator itA = oracle.begin() + a;
    OracleStruct elem = *itA;
    oracle.erase(itA);
    if (b > a)
        b--;
    Oracle::iterator itB = oracle.begin() + b;
    oracle.insert(itB, elem);
}
```

- *MultiMove:* Depending on the desired density of the neighbor scheme, blocks can also be moved in groups of various sizes. This group movement significantly increases the density of each neighbor set, but it has some advantages. If a specific ordering of a subgroup of elements is optimal, this subgroup can be kept intact during the multi-moves, preserving the good subgrouping.

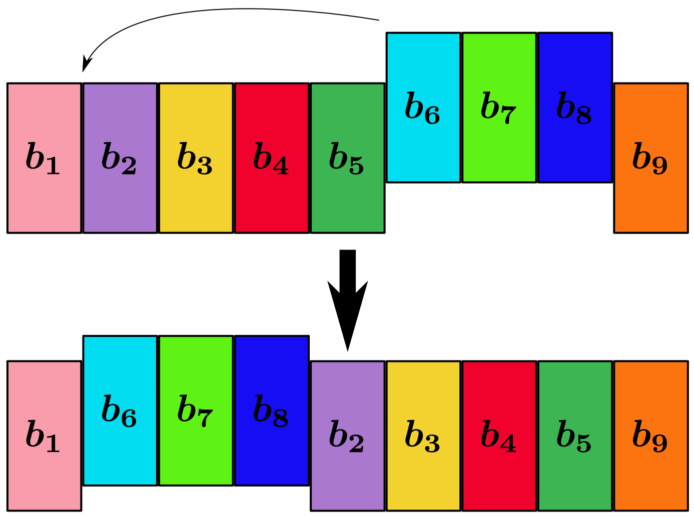


Figure 4.7: An example multi-move operation, which moves the group of 3 blocks b_6, b_7, and b_8 before block b_2 in the oracle. The relative order of all other blocks remain unchanged.

Groups of varying size are inspected, ranging from just a single block, up to `maxMoveGroup` blocks, which, for our tests, is set at 20. This allows group representing up to about 7% of the total blocks in a typical problem instance to be moved during a multi-move. Each possible group of `num` blocks is moved to all possible move locations, and tested for fitness.

```
void multiMove(Oracle &oracle, int a, int b, int num){
    Oracle::iterator groupFirst = oracle.begin() + a;
    Oracle::iterator groupLast = oracle.begin() + a + num;
    Oracle currGroup(groupFirst, groupLast);
    neighbor.erase(groupFirst, groupLast);
    oracle.insert(oracle.begin()+b,currGroup.begin(),currGroup.end());
}
```

We use the fitness function to evaluate the quality of each neighbor in relation to the others when determining which direction to move. In order to produce a search space which is relatively smooth with respect to fitness, the elementary operations are designed to produce neighbors which generally do not have markedly different fitness scores. Hill climbing deals with fully constructed oracles and thus any of the fitness criteria can be used. Since the ultimate goal of the search is to find a mapping that fits into the page table while satisfying all validity constraints, all our hill climbing techniques rely heavily on the oversubscription fitness calculation method which directly measures how well the oracle fits into memory.

In order to enumerate the set of all neighbor oracles, a neighbor is generated for each possible move or swap in the oracle. This set is quite large, but enumerating it is straightforward. As each neighbor is generated, its fitness is calculated for comparison with the other neighbors. Once all neighbors have been generated and evaluated for fitness, the neighbor with the best fitness is chosen for the next candidate in the search.

4.3.2.1 *Standard hill climbing with random restart* The most basic implementation of hill climbing utilizes the standard neighbor generation methods of move and swap. In order to keep the search going after encountering a local minima, random restart is employed which randomly generates a new seed whenever a local minima is encountered. This implementation is simple called `hc`.

4.3.2.2 *Persistent/Deep* The persistent hill climbing approach is a modification to the random restart hill climbing which represents an attempt to reduce the number of local optima encountered. The persistent implementation uses a very dense neighbor scheme which includes all possible single-block swaps and moves, as well as the set of neighbors generated by performing the multi-moves. This dense space provides a larger pool of neighbors to check for the best neighbor.

In addition to the dense neighborhood space, the persistent implementation also includes all the fitness score approaches in its fitness score vector. Since it still places emphasis on the oversubscription score, that score is stored in the first slot of the vector, followed by the pollution and wasted space scores. These additional scores provide additional information for neighbor comparisons, which reduce ties in neighbor selection and allows the process to continue even when the primary fitness criterion of oversubscription cannot discriminate between two similar oracles. Additionally, if a tie is encountered in all fitness scores, the procedure chooses an arbitrary neighbor and continues the search. In order to prevent a needlessly long search of a poor portion of the space, a bound is placed on the number of times that a tie may be selected for the next step in the search. For our tests, this bound is set at 5. Once the limit has been reached, a new seed is randomly selected, and the search begins again. This implementation is called `persistentHC`.

4.3.2.3 *Multi-Stage* The multi-stage variation of hill climbing is based on the concept that, typically, the larger blocks are harder to map that smaller blocks. This approach breaks the search into multiple stages, each containing progressively smaller blocks. Once the blocks in the current stage have been ordered in such a way that all validity constraints are satisfied, that relative order of those blocks is frozen, and the elements of the next stage are inspected. The generation of neighbors at each stage only considers movements or swaps of blocks in the current stage.

The multi-stage implementation accommodates a variety of ways to group blocks into stages. The most easy to test is a static definition of the minimum size for each group. Our implementation of the multi-stage search defaults to using three stages, based upon the size of each block b, $s(b)$, which is defined as the sum of the sizes of all elements that are members of the block. Using these definitions, block b will be inserted into the oracle during stage:

$$stage(b) = \begin{cases} 1 & \text{if } 1024 < s(b) \\ 2 & \text{if } 128 < s(b) \leq 1024 \\ 3 & \text{if } 0 < s(b) \leq 128 \end{cases}$$

This static definition of which elements are in each stage allows the performance of the multi-stage implementation to be easily compared across multiple problem instances. These specific break points were selected in light of the original problem instance, in that they divide the blocks into three stages with roughly the same number of blocks per stage. The first stage contains all blocks that are larger than a page. The second stage is the in-between sized blocks blocks, while the last stage is all smaller blocks that should be easier to place. The goal is to make the first stages sufficiently difficult that a reasonable ordering of difficult elements is produced, while not spending too much time in the first stage. Once a valid partial-oracle is found, it is passed to the next stage. If any stage reaches a local optima that is not valid, or the next stage is unable to find a solution within the drop limit, it re-drops the stage. For our tests, this drop limit is set to 1000 drops per stage. The implementation of multi-stage hill climbing is referred to as `multiHC`.

4.3.3 Genetic Search

The genetic search technique presented in Chapter 2 is an alternative, incomplete search technique that utilizes randomness to build a generation of oracles. This randomness allows different test runs to have very different results. The implementation has many tunable parameters, which allow the search to be easily modified.

The genetic search begins by creating a random common ancestor oracle, `noah`, which is used as the basis for all mutations that occur during the search. A series of generations are then generated by applying random mutations to members of the previous generation. The best member of each generation is compared to the global best member found thus far in the search, allowing the best member found during the search to be kept for the final reporting, in the case that a solution is not found. The search stops when a valid oracle is found, one which has an oversubscription fitness score of 0, indicating that no tasks have oversubscribed page tables. If no valid oracle is found within `NUM_GENERATIONS` generations, then the search terminates.

```
Mapper myMapper
search( )
    Oracle noah ← generateRandomOracle()
    Oracle best ← noah
    Oracle[] generation ← {noah}

    for each $i \in \{0, \ldots, \mathrm{NUM_GENERATIONS}\}$
        generation ← createGeneration(generation)
        if myMapper.fitness(generation[0]) < myMapper.fitness(best)
            best ← generation[0]
        if myMapper.fitness(best) = 0
            return TRUE
    return FALSE
```

New generations are created by applying a random mutation to each member of the parent generation. A certain number of oracles are allowed to survive from one generation to the next. For our tests, this threshold is specified using the `MAX_GENERATION_HOLDOVER` constant set to 100. This holdover is particularly useful when none of the oracles in the new generation are better than the parent generation. It also randomly allows good oracles to persist. Once the new generation has been fully populated, it is sorted according to the fitness of each child oracle, and only the best `GENERATION_SIZE` children are kept as parents for the next generation.

```
MAX_GENERATION_HOLDOVER ← 100
CHILDREN_PER_PARENT ← 100
GENERATION_SIZE ← 500
createGeneration(prevGeneration)
    Oracle[] children ←
        $\{\mathrm{prevGeneration}[1], \ldots, \mathrm{prevGeneration}[MAX\_GENERATION\_HOLDOVER]\}$
    foreach parent ∈prevGeneration
        foreach childNumber $\in \{1, \ldots, CHILDREN\_PER\_PARENT\}$
            Oracle mutant ← mutateOrdering(parent)
            children ← children + mutant

    for each child ∈children
        h[ child ] ← myMapper.fitness( child )
    sort children by h
    return $\{\mathrm{children}[1], \ldots, \mathrm{children}[GENERATION\_SIZE]\}$
```

The mutations are an application of the same elementary operations used for defining the set of neighbors in hill climbing. The difference is that the blocks involved in the operation are selected randomly during each mutation. The `rate` defines an upper bound on the number of operations performed during during a mutation. In our tests, `rate` is set to 10% of the blocks being mutated. After the mutation has been applied to the child, it is returned to be considered for inclusion in the next generation

```
RATE = 10
mutateOrdering(Oracle parent)
    Oracle child ← parent
    randomRate ← randomlySelectFrom({1,...,RATE})
    foreach mutation ∈ {1...randomRate}
        a =randomlySelectFrom({1,...,|blist|})
        b =randomlySelectFrom({1,...,|blist|})
        move(child, a, b)
    return child
```

CHAPTER FIVE

Evaluation

We have shown that MEM-MAP requires search to determine if a solution exists for a specific problem instance. Additionally, we have provided a search framework that serves as a foundation for testing a variety of approaches to finding solutions. The implementation based upon this framework allows the various search procedures to be tested against one another. These tests offer insight into which procedures are most effective.

5.1 Base Problem Instance

The DOD has provided a single problem instance file to support efforts to solve this problem. This problem instance, which we refer to as the base problem instance, is a known solvable problem instance. While we are sure that at least one solution exists for the base problem instance, solutions may not be easy to find using automated techniques. The goal is to use the implementation to produce an equivalent solution within a reasonable amount of time.

Some basic statistics about the base problem instance portray the key elements that affect the number of solutions that can exist. Table 5.1 provides an overview of the key characterization statistics for the base problem instance.

Even though there are 943 elements, only about half of them are data elements with which we are concerned. The remaining code elements are easily mapped into their own, separate page tables. These 485 data elements are grouped into 274 blocks by our search techniques, using the fingerprint of task requirements and precedence relationships specified in the input file to define groups. Element sizes, and the resulting block sizes, vary widely, as shown in Figures 5.1 and 5.2. The requirements

Table 5.1: Base Problem Instance Statistics

Number of Elements	943
Number of Data Elements	485
Number of Blocks	274
Number of Tasks	61
Number of Requirements	8819
Median Element Size	145
Median BlockSize	434

statistic measures the total number of times that a task requires an element. The number of blocks, combined with the number of requirements, help to define how difficult a particular problem is to solve. The search space of all oracles of blocks is 274!, or 3.673×10^{550}.

5.2 Alternate Problem Instances

The base problem instance is the definitive instance, but it is also the only problem instance that is provided by DOD. Since they do not make many changes to the mission computer, there is not a wide selection of problem instances to evaluate the search procedures.

In order to fully evaluate our implemention, we need more data points in the form of more problem instances. Since these are not readily available, we generate our own test suite to evaluate the methods. This is obviously not as desirable as testing on real data, but it provides a more complete testing scope. Since the base problem instance is difficult to solve, we create new problem instances by reducing the complexity of the base instance, thus making them easier to solve.

A way to modify the base problem instance in a manner that makes it easier to find a solution is to delete random elements from the instance. This has the effect of reducing the total number of elements and blocks which need to be placed. It

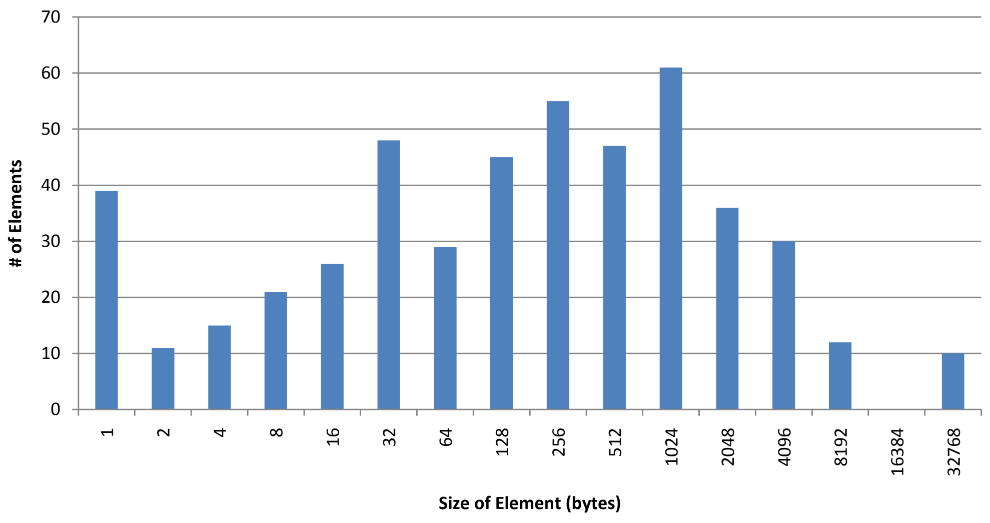


Figure 5.1: A histogram showing the distribution of the number of data elements of each size. The x-axis indicates the bytes represented by the histogram bin for the bar. Each bin contains all elements that are at least as large as the bin minimum, and smaller than the next bin minimum. The height of each bar indicates how many elements are in the size range specified by the x-axis.

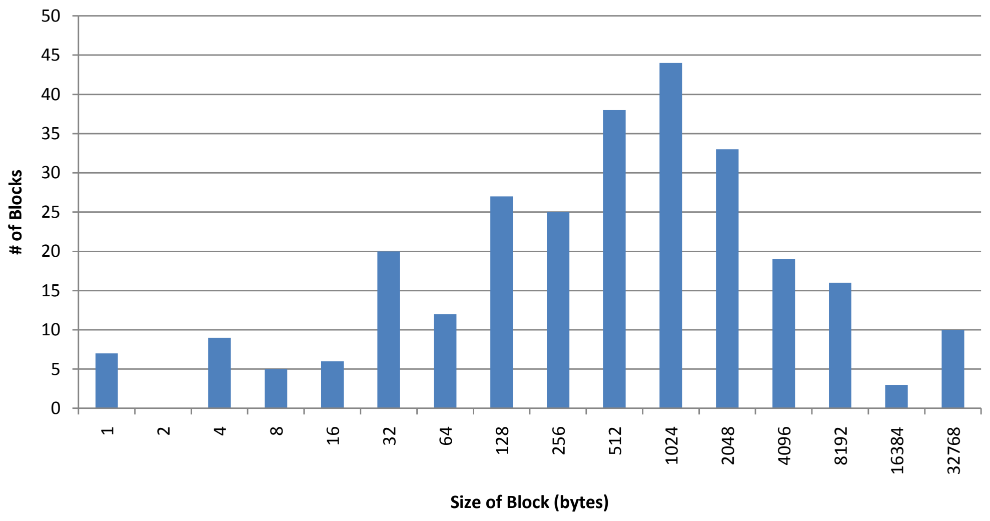


Figure 5.2: A histogram showing the distribution of the number of blocks of each size. The x-axis indicates the bytes represented by the histogram bin for the bar. Each bin contains all elements that are at least as large as the bin minimum, and smaller than the next bin minimum. The height of each bar indicates how many blocks are in the size range specified by the x-axis.

also reduces the number of requirements that need to be satisfied in a solution. Our generated instances are the result of deleting random elements until the sum of all element sizes has been reduced by a specified percentage.

In order to generate these modified problem instances, we create a function, `deleteElements`, that performs these deletions. During this process, `deleteElements` ensures that every task still requires at least one element, in order to prevent tasks from being effectively deleted as well. A modification to this straightforward approach is that `deleteElements` does not consider elements larger than 10,000 bytes in size for deleting, since they would quickly fill the quota and are typically not required by many of the most critical tasks. The other consideration is that it makes no guarantee that an exact number of bytes are deleted; in most cases it goes over slightly during a deletion.

It is important to note that each instance is generated independently of the other generated instances. As such, different elements may be deleted from one instance but not from another. Independent selection of elements for deletion can ultimately generate an instance that may have deleted more elements, but is ultimately harder due to more difficult elements being retained.

5.3 Performance Tests

We use the `deleteElements` function to generate 18 problem instances of varying difficulty in addition to the base problem instance. Table 5.2 provides a description of each generated instance. The difference between each instance is the percentage of the element size that is deleted, ranging from 5% to 90% at intervals of 5%. This distribution of deletion rates provides a basis for comparing even the procedures with poor performance. Instances with higher percentage of elements deleted will have more valid candidates in their search space, and thus be easier to solve than those with lower percentages deleted. For the purposes of the tests and results analysis,

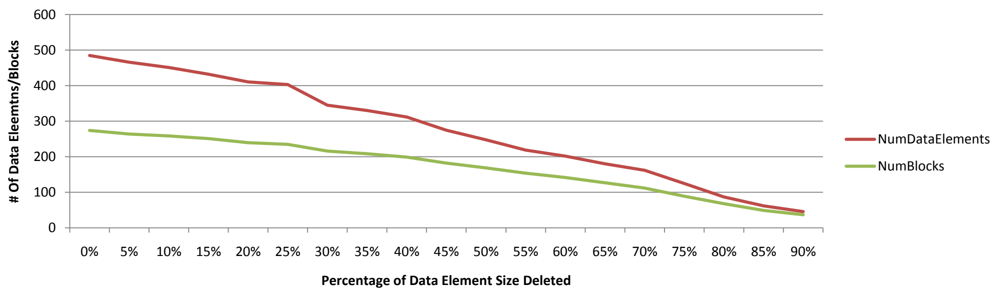


Figure 5.3: Comparison of the number of elements and blocks present in each problem instance. Reduced instances will have fewer elements, which eventually reduces to each element having its own block.

instances are identified by the percentage of element size that is deleted. The 00% instance is the base problem instance, which has no elements deleted.

The 90% instance is the easiest of the generated instances. In this instance, each task requires very few elements, and many oracles in the space are valid. This serves as a trivial test of how fast each implementation finds a solution in a solution-rich search space. Figure 5.3 shows that, as more elements are deleted, the total number of blocks is also reduced. This ultimately reduces the size of the search space, since there are fewer permutations possible for each oracle. Any approaches that have difficulty finding a solution in this space will likely continue to have problems once the problems get more difficult.

As the instances get progressively harder, the implementations are expected to take longer to find a solution. This allows us to measure how performance of each tested system changes with problem difficulty. Running the search procedures against the same test suite allows direct performance comparison.

5.3.1 Test Methodology

The method for running this suite of tests is a straightforward application of each search procedure to each of the 19 generated problem instances, as well as the base problem instance. There are two primary data points that need to be collected

Table 5.2: Key statistics about each of the generated problem instances. As more elements are deleted to reduce the total element size, the number of blocks are also reduced, since there are fewer fingerprints available to combine elements into blocks. Similarly, the total number of requirements declines as more elements are deleted. Since only elements smaller than 10,000 are deleted, the median element size increases as more elements are deleted. The total number of tasks remains constant, since care is taken to ensure that every task continues to require at least one element.

	Percentage of element size deleted																		
Statistic	00	05	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90
# of Elements	943	924	909	890	869	861	803	788	770	733	706	677	660	638	620	583	545	520	504
# of Data Elements	485	466	451	432	411	403	345	330	312	275	248	219	202	180	162	125	87	62	46
# of Blocks	274	264	259	251	240	235	216	209	199	182	169	154	142	127	112	89	68	49	37
# of Tasks	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61
# of Requirements	8819	8575	8402	8091	7763	7677	6845	6738	6518	5878	5663	5272	5064	4872	4637	4137	3475	3142	2992
Median Element Size	145	146	145	146	148	145	152	152	152	174	184	193	204	157	153	153	221	330	374
Median Block Size	434	431	428	407	407	406	414	407	407	427	414	428	428	424	441	401	401	528	625

about each individual test: what was the best candidate found, and how long did it take to find it. The goal is for the best candidate to be a valid mapping, but, in the absence of a solution, the fitness of the best candidate found will give us some basis for comparing results. Each implementation is designed to display the best oracle that it found during the run. Of course, if a solution is found, the search terminates immediately.

The length of time spent looking for a solution is measured in seconds of execution. Since the tests can easily run for multiple days if the problem instance is sufficiently hard, we impose an execution time limit of 24 hours, or 86,400 seconds, on CPU time. After 24 hours have elapsed, if the search has not found a valid solution, the best oracle is returned, along with its corresponding oversubscription fitness score. This allows comparison of techniques, even if they failed to produce a valid solution in the required time.

5.3.2 Test Hardware

Applying the search procedures to each generated problem instance requires a significant amount of processing power. Each of the procedures is single-threaded and can make effective use of one processor core.

Baylor University maintains a High Performance Computer (HPC) named Kodiak for research projects that require significant amounts of processing power. Key information about the Kodiak cluster is shown in Table 5.3. It has 128 nodes, each with dual-quadcore processors. This totals to 1024 total cores available for computation tasks. A batch processing systems allows multiple users to utilize the cluster at the same time. Once a job is submitted to the batch system, the job has an entire core dedicated to its processing for the duration of execution. This is important to consider because it ensures that each test is given the same computing resources for the duration of each test. The memory on each node is shared across

Table 5.3: Kodiak Cluster Specifications

Model	HP C3000BL
Operating System	Red Hat Enterprise Linux
Kernel	2.6.9-67.0.4.EL_SFS2.3_0smp
Cluster Mgt. Software	Platform Manage
Hardware	128 nodes (HP Proliant BL460C blades)
	Dual quadcore Intel Xeon 5355, 16 GB RAM

each of the 8 cores. Fortunately, the 16GB of memory available to each node is more than enough to support 8 concurrent searches, one on each core.

5.4 Performance Results and Comparison

We design a set of experiments to measure the performance of the various search techniques on the set of problem instances. Analysis of these experiments provides insight into the effectiveness of each procedure and guides selection of the procedure best suited for automatic application.

5.4.1 Execution Time Analysis

Figure 5.4 shows a comparison of the execution time for each procedure against each of the test instances. Each chart shows the performance of a different search procedure on the original and generated problem instances. Individual problem instances are ordered on the x-axis from largest to smallest total element size. Solution time for each problem instance is reported on the Y axis, with the height of the line indicating the number of seconds of user time consumed. As the problems become more difficult, some approaches are not able to find a solution within the 24 hour time limit. This failure to find a solution is indicated by an entry at 86400 seconds. Note that the y-axis is presented in log-scale, to accommodate viewing both small and large running times.

As expected, the easier instances, with more elements deleted, are solved much more quickly than the difficult problem instances. Additionally, each of the depth-first search procedures has difficulty finding solutions for all but the most trivial problem instances. The genetic and hill climbing procedures show the most promise for dealing with the larger problem instances.

The complete depth-first search almost immediately finds a solution for instances with greater that 70% of the element size deleted, but fails to find a solution at all for any instances with less than 65% deleted. This indicates that the search space in the higher deletion percentages must be solution-rich. As more elements are added, it becomes more difficult for this technique to recover from a poor choice made closer to the root of the search tree. As expected, complete search does not find a solution to the base case within the 24 hour time limit. During the course of its search, complete search examines 7.104×10^{10} internal nodes of the tree, and does not examine any leaf nodes due to pruning. The pruning function allows it to effectively exhaust 5.813×10^{475} oracles from the search space. While this seems like a large number of oracles, it is only $1.582 \times 10^{-75}\%$ of the 274! possible oracles in the search space. This still leaves 3.673×10^{550} oracles that must be examined. Even though these numbers appear to indicate that the complete search made no progress at all, complete search did make progress, just none that is measurable at this level of detail.

The heuristic and incomplete depth-first searches perform approximately the same, finding a solution extremely quickly for instances with 45% or more of the element size eliminated. This should be expected, since they are optimized versions of the complete depth-first search with the goal of finding interesting portions of the search space, but they still must perform exponential-time search to escape from subtrees of the search that are void of solutions.

The standard hill climbing search does very well on the medium to easy problem instances, but does not find a solution for any of the instances with less than 30% of the element size deleted. The success experienced on the easier instances suggests that it may have found a solution to some of these if it were given an additional 24 hours of search time.

The persistent hill climbing closely mirrors the performance of the standard hill climbing implementation. The search is able to make improvements at each step, but they are relatively minor with respect to the total oversubscription score. It is thus unable to quickly identify solutions to the larger problems that prevent it from being valid. Even though it is able to find a better neighbor many times, each of these neighbor selections ultimately end up in a local minima. Each time a local minima is found that it is unable to escape, it starts over with a new random drop. It finds solutions for instances with greater than 20% deleted, but takes longer than genetic to find them.

The performance of the multi-stage hill climbing is disappointing; it fails to find a solution for any instance harder than 45%, which is on par with the heuristic-based depth-first searches. Unlike the heuristic-based searches, it does not quickly find a solution even to the easier instances. This shows that the stage-based code is sufficient for finding as solution, but more research needs to be done to find better stage partitioning.

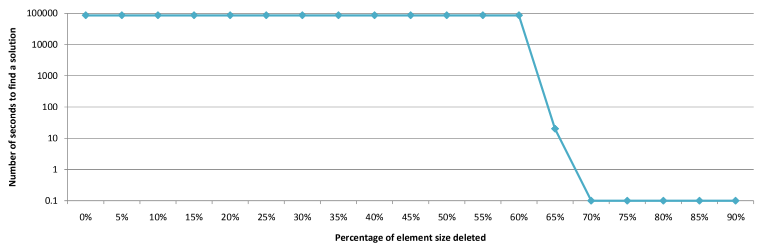
The genetic search procedure provides the most promising results of all the search procedures, finding a solution for all instances with greater than 10% of the element size deleted. This leaves only the base instance, along with the 5% and 10% instances, as unsolved. The fact that it solves most of the instances within 24 hours is encouraging. One anomaly in the results is the amount of time taken to find a solution for the 25% problem instance. This problem takes significantly longer than both the 20% and 30% instances. This can be attributed to a variety of factors, most likely being the random nature of the genetic search, and the fact that each problem instance is independently generated. These random operations can have varying effects on the inherent difficulty of the problem and on the effectiveness of the procedure. It is notable, however, that the genetic procedure is the only one that finds a solution to the 25% instance within the 24 hour testing period. These results indicate that genetic is one of the most effective procedures.

5.4.2 Effectiveness on Base Problem Instance

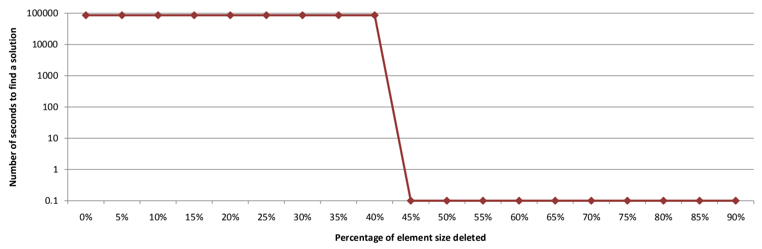
Figures 5.4 and 5.5 provide insight into which procedures are the most promising. However, none of the procedures find a solution to the base problem instance within the 24 hour period specified by the test. In order to run extended tests with the goal of finding a solution to the base problem, we analyze the fitness function over time with respect to the base problem instance for each of the most promising procedures, as identified in Figures 5.4 and 5.5. Figure 5.6 shows this comparison of fitness for the best candidate found by the genetic, standard hill climbing, and persistent hill climbing searches when applied to the base problem instance. This is shown over the course of the 24 hour testing period.

Once again, the y-axis in Figure 5.6 is presented in log scale to facilitate close examination of results as they approach a valid mapping with zero oversubscribed bytes. Time is presented on the x-axis in hours.

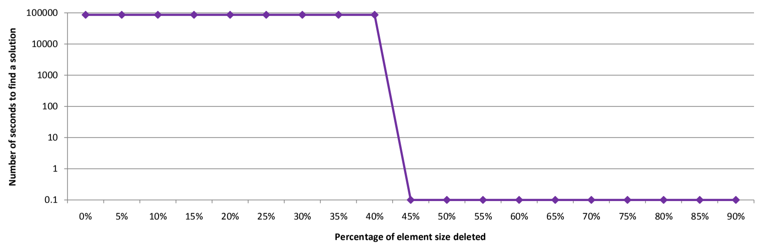
Both the standard and persistent hill climbing procedures report the oversubscription fitness during each roll. This presents sawtooth lines in Figure 5.6 since the procedures generate a new seed when no more expansions can be performed that produce a better oversubscription. These sawtooth spikes do not even approach a valid solution, they are all well above 100 bytes oversubscribed. This means that the flatter green and red lines indicate the best oversubscription fitness level found so far by the search. Each time a hill climbing attempt ends with a better solution for



(a) Complete Search

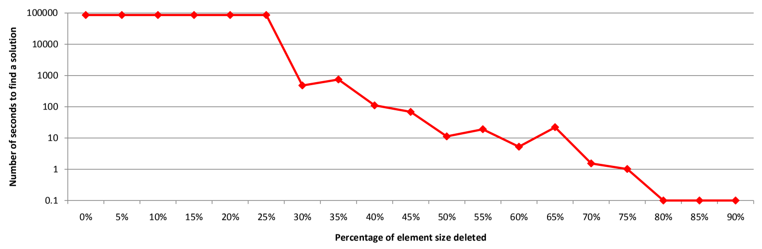


(b) Heuristic Search

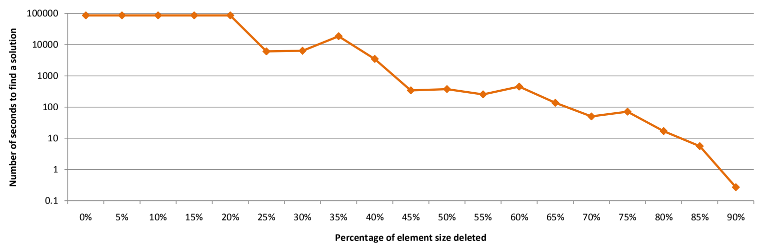


(c) Incomplete Search

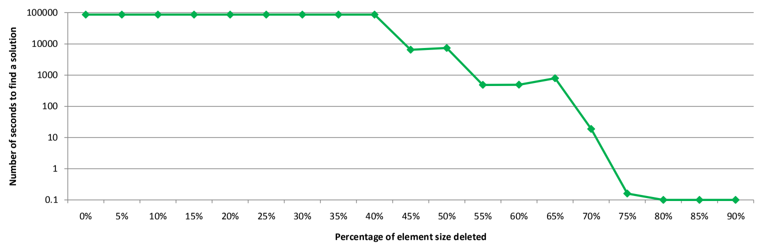
Figure 5.4: Performance results from applying all implemented procedures to the test suite. The y-axis, presented in log-scale, indicates the number of seconds of user time used by the procedure for a specific problem instance. Problem instances are ordered on the x-axis by percentage of total element size deleted. A time limit of 86,400 is imposed; procedures that fail to find a solution in that time simply report 86,400 seconds as return time.



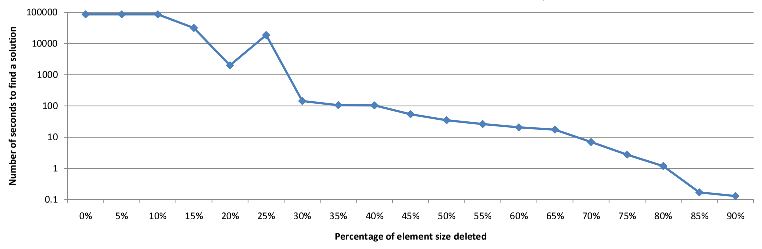
(a) Standard Hill Climbing



(b) Persistent Hill Climbing



(c) Multiple-Stage Hill Climbing



(d) Genetic

Figure 5.5: Performance results for variations on hill climbing search and genetic search.

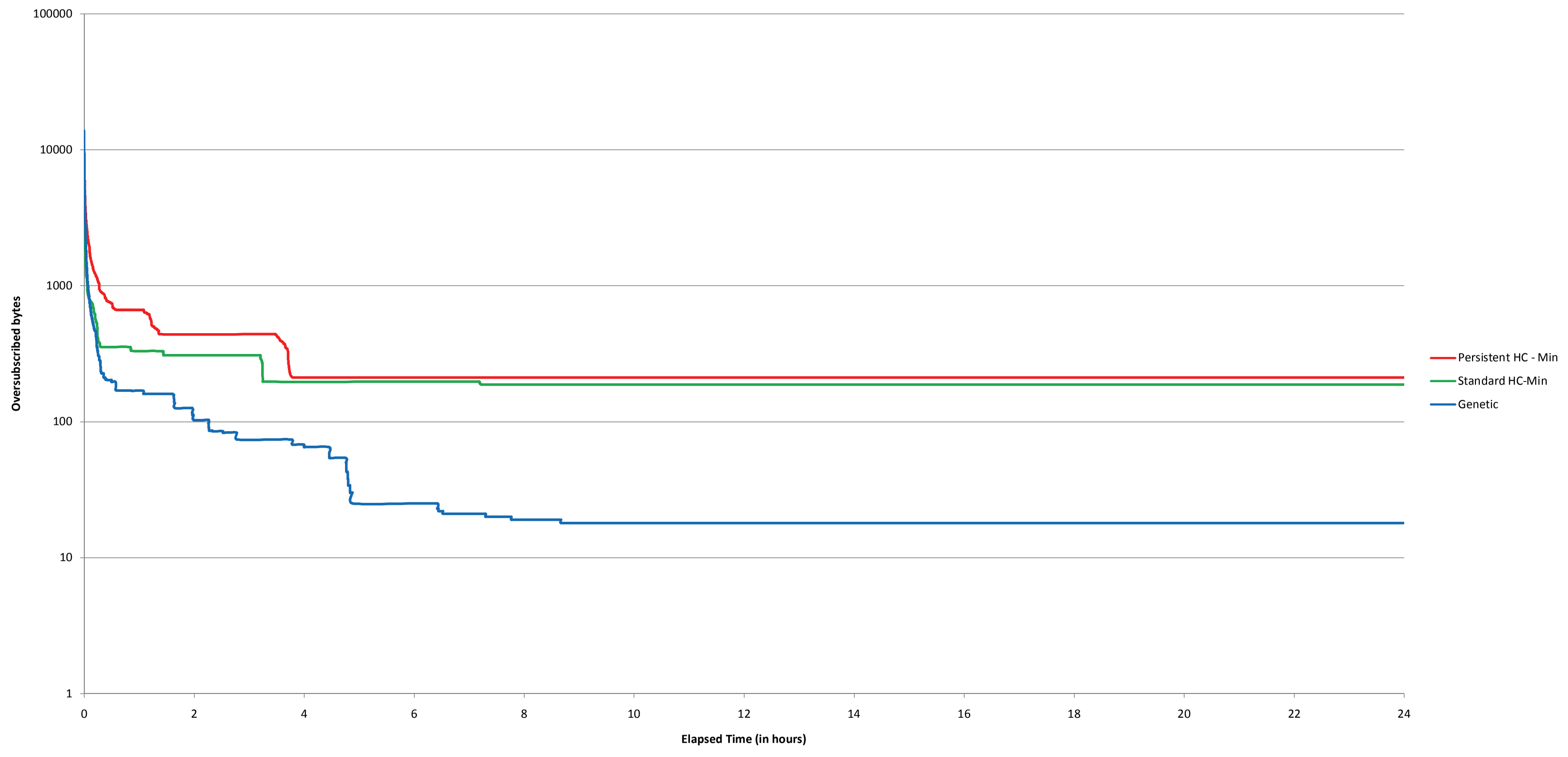


Figure 5.6: Performance during search for each of the three most promising search procedures. The graph plots the oversubscription fitness measure on the y-axis against user CPU time consumed in hours on the x-axis for the base problem instance. As time passes, the three search procedures, genetic search, standard hill climbing, and persistent hill climbing find oracles with better oversubscription scores. Genetic search quickly finds oracles with fewer than 100 bytes oversubscribed, and continues to improve, while both hill climbing searches are limited around the 400 byte mark.

the search, this value is reduced. During the 24 hours of execution, the standard and persistent hill climbing procedures examine 1.515×10^9 and 1.499×10^9 oracles in the search space, respectively. The incomplete nature of these search procedures means that some oracles could be examined multiple times.

The genetic approach, which always maintains its best oracle as part of the population, presents a continually decreasing oversubscription line. Within the first few minutes of the genetic search, it identifies an oracle that is better than the best oracle found by either of the hill climbing procedures in an entire day. Once this initial progress is made, however, progress is significantly slower. At about 4 hours, a good mutation is identified which drops the oversubscription to 20 bytes. This is the last major progress, and the procedure never gets below 18 bytes oversubscribed. During 24 hours of execution, genetic search examines 1.142×10^9 oracles in the space, although the incomplete nature of genetic search means that some oracles could be examined multiple times.

5.4.3 Analysis of Multiple Extended Runs of Genetic Search

Of the most promising procedures identified in Figures 5.4 and 5.5, the genetic search appears to be the most effective in quickly getting close to a valid solution, as shown in Figure 5.6. Based on these results, we perform a set of extended runtime tests using the genetic search procedure on the base problem instance. Figure 5.7 shows the results of these extended tests.

The time limit for the extended tests is 7 days, or 168 hours. As with previous figures, the y-axis in Figure 5.7 is presented in log scale to allow the wide range of oversubscription scores to be easily viewed.

Of the 50 genetic search instances tested, 6 found a solution to the base problem within the 7 day timeframe. Figure 5.7 graphs the fitness score of the best oracle found by each of these instance as time progresses, in addition to a non-solving

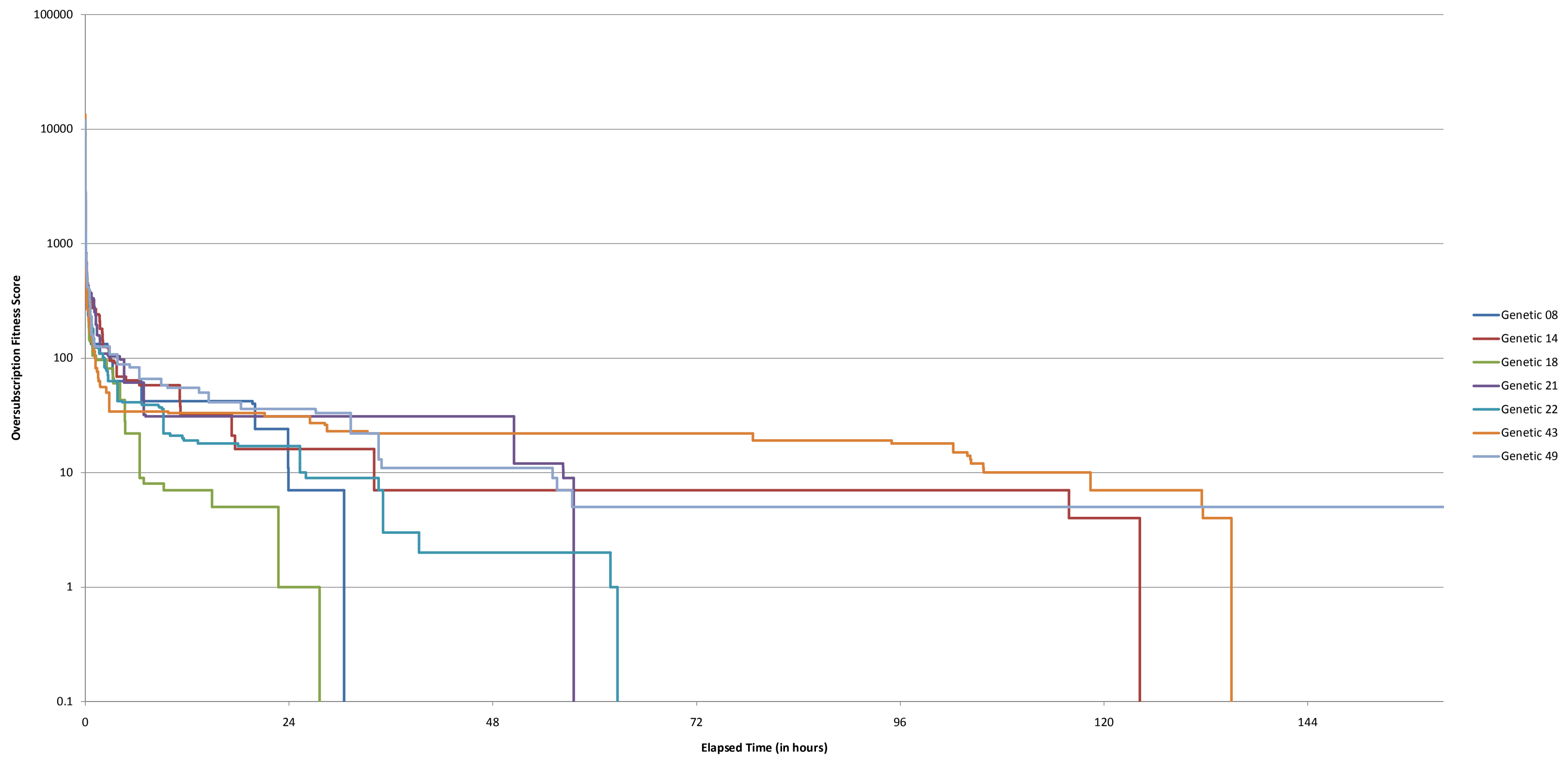


Figure 5.7: Performance of the best performing search instances from 50 separate genetic search instances run over the course of 7 days. The graph plots the oversubscription fitness measure on the y-axis against user CPU time consumed in hours on the x-axis. Of the 50 instances run on the base problem instance, 6 found a solution within the timeframe, with most of the others coming within 10 oversubscribed bytes. Instance 49 is an example instance which finds a best oversubscription score of 4.

instance. Genetic #18 finds a solution the fastest in just under 100,000 seconds; almost 28 hours. Genetic #43 is the last instance to find a solution within the timeframe, solving in 486,000 seconds, which is 135 hours. Genetic #49 is an example instance which does not find a solution within the timeframe, bottoming out at 5 bytes oversubscribed.

Since each instance generates its own set of random generations starting from a randomly generated starting oracle, each instance is an independent test. It is interesting that all of the selected instances converge at approximately the same rate during the first 24 hours. Once the oversubscription gets below 100 bytes, progress becomes more difficult. Ultimately, the instances that find a solution do so via a mutation from an oversubscription of less than 10 bytes.

It is encouraging that some of the test instances found a solution within the specified timeframe on our test hardware, despite the use of a randomized incomplete search procedure. The random nature of the search does not make any guarantees about future performance results on the same base problem instance, but it is encouraging that more than one instance found a solution. Even a 12% success rate in 7 days demonstrates that automated search techniques can be effectively applied to MEM-MAP.

CHAPTER SIX

Conclusion

This work explores a variety of topics related to the MEM-MAP problem, from formal analysis, to framework design and implementation, and finally analysis of performance results. This process provides some contributions to research in this area, in addition to providing guidance for future efforts to develop automated solutions.

Our hypothesis is that the total order depth-first search based algorithms will be the least effective, since there is simply too much search space to be analyzed. The genetic and hill climbing variants are expected to produce the best results.

6.1 Contributions

One of the largest contributions of this project is the formal analysis of the problem. The formal definition of the problem provides a common language for discussing and examining the problem. This formal definition permits a rigorous evaluation of the problem's complexity.

The $\mathcal{NP}$-Completeness proof indicates that search is the only method that can be guaranteed to find a solution if one exists. This knowledge guides the development of search procedures, since we can be assured that we won't find a silver bullet that is guaranteed to efficiently find a solution.

In addition to formal analysis, this work provides a well designed and extensible framework for evaluating a variety of search procedures. The extensible nature of the framework allows it to serve as a basis for future research into search procedures, as well as adapting it to accommodate future changes in the definition of the problem.

A reference implementation based upon the framework is provided for both testing and production-level use for producing mappings. We use this implementation to thoroughly evaluate the performance of a variety of search procedures. These procedures are tested for both speed and quality of search on the base problem instance, as well as our suite of generated test instances. The variety of testing instances provides a method of comparing the performance of the procedures quantitatively against one another. Through these tests, we find that the genetic search method provides the most promising results, often solving the base problem in only a few days. This demonstrates that the problem, although difficult, will respond to automated search techniques.

6.2 Future Work

While this project provides some significant contributions to the problem, more work can be done to improve the performance of the procedures. A good next step is to run more tests on actual historical problem instances, especially ones that are known to be solvable. This will provide a larger test pool to compare the performance of the procedures, since the actual problem instances are more realistic than any instances we could generate. Since the problem has become more complex over the years, the historical instances may contain more solutions, providing additional information into how to customize the procedures.

Customization of the procedures is a major opportunity for future research with this problem. Each of the procedures is designed to be flexible and tunable, but we have limited our testing to a small set of possible values for the tunable parameters. Additional testing on the many refinements to the procedures will likely produce procedures which are more customized to the problem instance, producing a more efficient search.

It is interesting to note from Figure 5.6 that, within 10 hours, the genetic procedures has made all the progress it will make for the entire 24 hour period. This flatlining of progress might be addressed by making the search more aggressive. A possible refinement would be to increase the mutation rate of the genetic search when progress has waned, allowing it to escape a local optima of fitness that is produced by a limited mutation rate.

In addition to making genetic search more aggressive, it might be helpful to parse element ordering from the Map2 input file. This order could serve as the base oracle for genetic search, potentially providing a shorter path to a solution by utilizing intelligent decisions made in previous versions of the instance.

BIBLIOGRAPHY

Map1 Documentation. McDonnell Aircraft and Missile Systems, 2007a. Map1.doc.

Map2 Documentation. McDonnell Aircraft and Missile Systems, 2007b. Map2.doc.

Carder, Philip, and Paul Guse. "Boeing F/A-18E/F Super Hornet Block II." Technical report, 2009.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2003, second edition.

Garey, M[ichael] R., and D[avid] S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W. H. Freeman, 1973.

Ghannadian, F. Alford, and R. C. Shonkwiler. "Applications of Random Restart to Genetic Algorithms." *Information Sciences* 95, 1/2: (1996) 81–102.

Luger, George F., and William A. Stubblefield. *Artificial Intelligence, Structures and Strategies for Complex Problem Solving*. Addison-Wesley, 1998, third edition.

Naval Air Systems Command. "United States Navy Fact File: F/A-18 Hornet strike fighter." http://www.navy.mil/navydata/fact_display.asp?cid=1100&tid=1200&ct=1, 2009.

Reed, R. "Pruning Algorithms-A Survey." *IEEE Transaction on Neural Networks* 4, 5: (1993) 740–747.

Sipser, Michael. *Introduction to the Theory of Computation*. PWS, 1997, first edition.

Stevenson, Jeremy. *Memory Map Editor User's Guide*. Baylor University, 2007.

Stevenson, Jeremy, Russsel W. Duren, and Michael W. Thompson. *Fall Through Gravity*. Baylor University, 2006.

Thede, Scott M. "An introduction to genetic algorithms." *J. Comput. Small Coll.* 20, 1: (2004) 115–123.

Weld, D. S. "An Introduction to Least Commitment Planning." *AI Magazine* 15, 4: (1994) 27–61.