Efficient Approaches to Subset Construction

by

Ted Leslie

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1995

Acknowledgements

I would like to thank both Jeffrey Shallit and Howard Johnson for reading and making suggestions on this work. I would especially like to thank my supervisor Derick Wood for his guidance in my research and for suggesting that this work be presented as a thesis. Last but certainly not least, I would like to thank Darrell Raymond for his help with all aspects of this thesis and the related research.

Finally, the financial support from the Department of Computer Science, the Information Technology Research Center, and the Natural Sciences and Engineering Research Council of Canada is gratefully acknowledged.

Abstract

This thesis addresses important issues in the conversion of nondeterministic finite automata into deterministic finite automata: the development of the conversion process, the behavior of the conversion process with respect to input automata, and the relationship of output automata to input automata. In developing a robust, efficient, and modular conversion program, it is necessary to analyze searching and sorting routines, abstract data types, and other issues in order to determine appropriate routines and structures for use with automata and their manipulation. *Subset construction* is a well-known procedure for converting a nondeterministic finite automaton into a deterministic finite automaton. A large part of this thesis explains the improvements made to an existing, inefficient, implementation of subset construction based on a pseudo-code specification. The testing phase of this project led to a study of the behavior of the conversion process with respect to different types of automata. Very few automata were available for testing, therefore, methods to generate test automata were examined. Some interesting behavior in the output automata, with respect to the input automata, prompted concurrent study in this area. A conjecture based on the density of automata enables us to estimate the cost of subset construction prior to actually performing the construction. In conclusion, recommended implementations of subset construction are suggested based upon information about the input automaton.

Contents

Chapter 1

Introduction and Overview

1.1 Introduction

We address important issues in the conversion of nondeterministic finite automata into deterministic finite automata: the development of the conversion process, the behavior of the conversion process with respect to input automata, and the relationship of output automata to input automata. In developing a robust, efficient, and modular conversion program, it is necessary to analyze searching and sorting routines, data structures, and other issues in order to determine appropriate routines and data structures for use with automata and their manipulation. The conversion program resulting from this project is one of the automaton tools in GRAIL. GRAIL is a research project at the University of Waterloo, part of which is the establishment of tools that manipulate finite automata. The testing phase of the subset construction project led us to study the behavior of the conversion process with respect to different types of test data. In addition, it prompted the concurrent study of the behavior of output automata with respect to input automata.

1.2 Basics of Conversion

A finite automaton (or just automaton) is the standard machine model for specifying a recognizer for regular languages. A finite automaton (NFA) consists of a finite set of states Q, a finite set of

Figure 1.1: A nondeterministic finite automaton.

input symbols Σ, a start state S, a set of final states F, and a transition relation $\delta \subseteq Q \times \Sigma \times Q$ which describes the connections between the states. We normally draw finite automata as directed graphs, in which vertices correspond to the states and the edges correspond to the transitions, where each edge is labeled with the symbol of its transition. In figures, we indicate the start state with a wavy arrow, the states with circles, and the final states with double circles. The automaton recognizes a word w if, starting at the start state s, there is a path through the automaton that spells w and ends at one of the final states.

Figure 1.1 shows a nondeterministic automaton for a system that accepts 20 cents in exact change from any combination of nickels (n) and dimes (d). The automaton incorporates each correct sequence $\{dd, dnn, ndn, nnd, nnnn\}$ into the machine. It is said to be nondeterministic because there is a state which has more than one transition originating from it with the same input symbol: there are two transitions from state 0 labeled with the input symbol d. Since there is a choice of what transition should be taken, simulating such a machine is difficult. Backtracking, the mechanism typically used to simulate a nondeterministic machine, is not very efficient.

If an automaton is deterministic (for each symbol, all states have at most one transition originating from them), then its execution is straightforward because the input determines a single transition at each step of execution. A standard construction [7, pages 124–125] shows that

Figure 1.2: A deterministic finite automaton equivalent to the automaton of Figure 1.1.

every nondeterministic machine can be converted to a deterministic machine. Figure 1.2 gives a deterministic machine that accepts exactly the same input as the machine of Figure 1.1 or, in other words, the two machines accept the same language. *Subset construction* [6] is a well-known procedure for converting a nondeterministic automaton to a deterministic automaton.

In subset construction we eliminate nondeterminism by finding the set of target states that are reachable by a single transition symbol from a given set of source states, and treating the target set as a single state. The deterministic machine in Figure 1.2 has states of the form: (deterministic state number , $\{state_1, state_2, \ldots\}$). The deterministic state number is used as a state label in the deterministic machine. Each subset of nondeterministic states has a unique deterministic state number. So, for example, state "1" in the deterministic machine corresponds to states "5", "7", and "9" in the nondeterministic machine. Being in a given state in the deterministic machine corresponds to being in one of, perhaps, many states in the nondeterministic machine. Determining this correspondence is the key to converting a nondeterministic automaton to a deterministic automaton. An example conversion, using the coin machine automaton, is detailed below and tabulated in Table 1.1.

The conversion algorithm begins with the set consisting of deterministic start state $\{0\}$ (the set represented by the nondeterministic start state 0) as the *source set* of states. For each state within a *source set* and for each symbol (column b) from that state, the transitions are noted (column c). The set of *target states* reachable using these transitions represents a state in the deterministic machine. Each unique set of target states is assigned a unique number. Thus, if a target set was previously encountered, then we use the number that was assigned to it at that time. The deterministic state (column e) associated with the source set of states (column a) becomes the source state of a new transition (column f) in the deterministic machine, the current symbol (column b) is the symbol, and the target state (column e) is the number of the target set (column d). The conversion method ensures that all states in the deterministic machine are reachable from start state. Unreachable states are never created. The classic construction [6], sometimes referred to as the "big bang" construction, allocates all states; the connected method only allocates states which are needed. In this paper we will mean the connected method when we refer to subset construction.

For the example in Table 1.1, the source set in the deterministic machine is the start state "0" represented by $\{0\}$. From state "0" on symbol n (in the nondeterministic machine) the transitions are $(0\ n\ 5)$, $(0\ n\ 7)$, and $(0\ n\ 9)$. The target states give the set $\{5,7,9\}$ which becomes the new state "1" in the deterministic machine. Once all the symbols coming from $\{0\}$ have been considered, the source set becomes the next *target set* (column d) that has not been considered. So in our example, $\{5,7,9\}$ is considered next. The conversion continues until there are no more new sets (column d) to be considered. When a set of states has been created that has already been considered, its state number (column e) is determined and the new transition (f) is created using this state number. In the example, this occurs with the state set $\{2\}$ (marked *) which is the deterministic machine state "6".

1.3 Coarse Analysis of Subset Construction

A high-level pseudo-code version of subset construction is given in Algorithm 1.1 (Figure 1.3). The algorithm formally describes the activity documented in Table 1.1. As a pseudo-code algorithm and for small automata, subset construction appears to be relatively efficient. Difficulties occur with large automata, however, because we must organize the data to allow for efficient retrieval. Observe that we form sets of states and must be able to determine if they have been seen before. Over the course of a conversion, thousands of sets may be formed and each set could contain

nondeterministic machine				*deterministic machine*	
(a) Source set	(b) Symbol	(c) Transitions	(d) Target set	(e) State	(f) Transition
GIVEN			{0}	0	
{0}	n	0 n 5			
		0 n 7			
		0 n 9	{5,7,9}	1	0 n 1
{0}	d	0 d 1			
		0 d 3	{1,3}	2	0 d 2
{5,7,9}	n	7 n 8			
		9 n 10	{8,10}	3	1 n 3
{5,7,9}	d	5 d 6	{6}	4	1 d 4
{1,3}	n	3 n 4	{4}	5	2 n 5
{1,3}	d	1 d 2	{2}	6	2 d 6
{8,10}	n	10 n 11	{11}	7	3 n 7
{8,10}	d	8 d 2	{2}*	6	3 d 6
{6}	n	6 n 2	{2}*	6	4 n 6
{6}	d	none			
{4}	n	4 n 2	{2}*	6	5 n 6
{4}	d	none			
{2}	n	none			
{2}	d	none			
{11}	n	11 n 2	{2}*	6	7 n 6
{11}	d	none			

Table 1.1: Nondeterministic to deterministic automaton conversion by the tabular method.

```
Algorithm 1.1:  NFA to DFA conversion by Subset Construction
Input:          A NFA with a finite set Q of states, a finite
                alphabet S of input symbols, a transition relation,
                a designated start state(0) and a set of final states.
Output:         A deterministic automaton with set R of states that
                accepts the same language as the input automaton.

last=0;
$R_0$={0};  /* Given start state ''0'' as start state in DFA */
   for($i = 0; i <= last; i = i + 1$) do
   begin
      for each symbol $a$ in the set of symbols do
      begin $W$ := {};  /* empty set */
         for each $b$ in $R_i$ do
            for each transition of the form ($b$ $a$ $Q_j$) do
               add $Q_j$ to set $W$;
         if $W \neq \{\}$ then
         begin
            if there exists an $R_k$=$W$ then
               output DFA transition:  (i $a$ k);
            else
               begin last=last+1;
                  $R_{last}$ := $W$; /* Set of j states from above */
                  Output DFA transition:  (i $a$ last)
               end
         end   /* of symbol loop */
   end   /* i loop */
```

Figure 1.3: Subset construction algorithm

thousands of states. Arriving at an efficient method of searching for sets is an important factor in the overall performance of an implementation of subset construction. At each iteration of the construction, the state(s) within the *source set* are referenced (line 8). All transitions originating from these states are processed, and for each symbol, the target states of the transitions (Q_j) form a set W. From each state within a set we must be able to find the transitions in the NFA. An efficient link from a state in the current set to that state in the nondeterministic automaton must be established and efficient methods must be sought to retrieve all related transitions. These problems are examples of the subtle concerns involved in achieving an efficient implementation of subset construction.

1.4 The Performance of Subset Construction

With only a cursory knowledge of subset construction, one might guess that its performance would depend on input automaton size and that it would behave in a linear fashion (e.g. twice the input size implies twice the execution time). One might also guess that converting large automata (e.g. hundreds of states and thousands of transitions) is easily within the bounds of modern day computers. However these guesses can be quite inaccurate. As is well known, the maximum number of states that can be formed in the output (deterministic) automaton is 2^n, where n is the number of states in the input automaton. So, for example, an input automaton with 64 states could produce a output automaton with 184,467,400 trillion states. A nondeterministic automaton with n states and m symbols can have, at most, n^2m distinct transitions. In [4] the n^2 bound is shown to be optimal, and [5] explains that a one-input NFA has a reduced, connected subset machine with $(n-1)^2+2$ states. An interesting corollary from [5] states that for each $k \geq 2$, there is a k-input n-state NFA for which the corresponding reduced DFA has 2^n states. In general, the bound cannot be improved. Once converted to a deterministic automaton, the maximum number of transitions is at most 2^nm, an exponential increase in the number of transitions. From this we can conclude that not only are some automata hard to convert, but are, perhaps, in practice, impossible (even on a super-computer). Also, because subset construction must deal with possibly exponential growth, the overall performance of subset construction is, at worst, exponential with respect to input size. This problem will be addressed in later chapters.

1.5 Development Support

This thesis explores the difference between theoretical and practical approaches to subset construction. The implementations described in this paper were constructed in an environment which supported the development of automata tools. The key components of this environment are INR and GRAIL, both of which contain an implementation of subset construction. Each tool helped us to test and to carry out performance evaluation for the subset construction algorithm that we developed. These tools also gave initial insight into existing methods for subset construction. INR is a command-driven program written in C and running on UNIX. INR consist of a single program containing a collection of functions for manipulating automata. The program was written by J.H. Johnson [1] to provide an efficient environment for manipulating automata, including multi-tape automata. Inspecting INR's code reveals efficient data structures and algorithms; however it is difficult to modify and enhance because there is little documentation and it is not very modular. Enhancing INR with new functions is challenging because access to the storage structures is difficult and undocumented. The INR environment requires that the automata be represented in a specific format. This format has been adopted by the GRAIL project and in turn by the subset construction algorithm.

GRAIL [Grammar, Regular expression, Automata: Input Language] is a set of tools which builds on and extends the work begun by INR. The environment of GRAIL is a collection of processes (created by different authors) that perform single functions on automata. The programs are currently written in C and C++ and run under the UNIX operating system. Each process acts like a UNIX filter. They receive input via standard input (keyboard or file indirection), send output via standard output (terminal or file indirection), and can be piped together to produce chained commands. The GRAIL environment is made up of independent processes; therefore, enhancing the environment with a new function involves the compilation of the function and not the whole environment. The open architecture of the tools makes it particularly attractive as a test bed for developing functions for automata. GRAIL processes are written in a structured and documented fashion so they are readable and easy to modify.

Another automata/regular language package is REGPACK [3] developed at the University of Waterloo in 1977.

1.6 Overview of Thesis

In the next few chapters we deal directly with resolving the problems arising from the implementation of subset construction, such as those noted in Section 1.3, in addition to presenting the data structures which evolved. Chapter 2 presents an analysis of the conversion routine, discusses the basic abstract data types and explains how we represent them, and discusses the initial implementations. Chapter 3 deals with the generation of test data. Chapters 4, 5 and 6 deal with the optimization of the conversion routine and discuss how the resulting subset construction algorithm may benefit from different implementations depending on the characteristics of the input automata. A performance evaluation of the various implementations is also presented. In Chapter 7 we define the notion of the density of an automaton and we give a conjecture based on density that enables us to estimate the time and space taken by subset construction. Chapter 8 presents some concluding remarks and discusses future work.

Chapter 2

Subset Construction Explored

2.1 Introduction

The subset construction algorithm as described by Wood [7, page 126] is a high-level symbolic specification that glosses over data structuring, control flow, and other design issues that are important when considering an implementation of the method. Once identified, each design issue can be studied separately and in conjunction with other issues. Our goal is to produce a robust, efficient, and modular program.

2.2 Complexity of Subset Conversion

2.2.1 Internal Structure of Subset Construction

Algorithm 1.1 consists of four nested loops. The outermost loop, spanning lines 4 to 20 we call the *deterministic state loop*. It ranges over every state that is generated for the deterministic automaton. As we have seen in the examples, the number of deterministic states grows as the algorithm progresses. The deterministic state loop operates as long as deterministic states are generated. The next loop, spanning lines 5 to 19, is called the *symbol loop*. It initiates processing for every symbol in the alphabet of the input automaton. The next loop, spanning lines 7 to 9, is called the *state loop*. It initiates processing for each state in a given set. The final loop, spanning lines 8 to 9, is called the *transition loop*. It produces transitions with a specific source state and symbol. In addition to these loops, there are three important modules. The first is the *add state* routine at line 9; it takes the target states, from transitions with a specific source state and symbol, and adds them to a set that represents a deterministic state. The second is the *set search* routine at line 12, that determines whether a particular deterministic state is in a collection of deterministic states. The third is the *add set* routine at lines 15 to 16 that adds a deterministic state to the collection of states. In the algorithm, the variable `R` represents a collection of deterministic states; the deterministic state loop is executed for each deterministic state in the collection.

2.2.2 Worst-Case Analysis

Algorithm 1.1 is essentially the technique which the existing GRAIL conversion routine employs. We will refer to this as the brute-force implementation. We compute the complexity of the brute-force implementation by analyzing it in terms of the various loops and routines defined above. A nondeterministic input automaton `N1` has n states and m symbols and it is converted into a deterministic automaton `D1`. The deterministic state loop processes every state in the resulting automaton `D1`. We know that the maximum number of states that can occur in `D1` is 2^n. Therefore, the deterministic state loop is executed at most 2^n times. The symbol loop has a constant complexity of m, the number of symbols. The state loop is executed for every state in the set R_i. The number of states in each set can be as high as n, so the state loop is repeated at most n times. The transition loop is executed for every transition; however, in the brute-force implementation we search the list of transitions because they are not ordered. In the worst case we have to search through all the transitions and there can be as many as n^2m. Therefore, the transition loop has worst-case complexity n^2m. Within the symbol loop we invoke routines for set searching and adding a state to a set. In the brute-force implementation the sets are kept in a vector in generated order. The number of sets compared is at worst 2^n. In the brute-force implementation, set comparison involves element-by-element comparisons (state-by-state); this involves, in the worst case, n state comparisons. In other words, set search has time complexity $O(2^n n)$. Since a new set is simply added to the vector, the time complexity of adding a set is constant.

The total worst-case complexity of subset construction, as implemented in a brute-force fashion, is $O(2^n m(nn^2 m + 2^n n))$. We do not expect to reduce the exponential complexity of the algorithm in the worst case; however, reducing the complexity of some of the minor factors will result in a faster algorithm in many cases.

We can also analyze the complexity with respect to output size. Suppose automaton `D1` has T output transitions and N states (the result of the subset construction on N1). We know that T can be at most Nm. The deterministic state loop is executed N times, the symbol loop m times, and the state loop at most n times. Finally, at most n transitions can be accessed for each symbol. If the cost of accessing a transition is p, then the worst-case complexity (ignoring the add-state routine) is $O(Nmn^2p)$. Later we will see that we can replace T by Nm and we can make p a constant.

2.3 Main Activity in Subset Conversion

There are two main areas of subset construction that seem appropriate for improved data structures. In the transition loop, a search is made for transitions that have a given source state and input symbol. This is one of the most frequently executed parts of the algorithm, as it is nested in the deepest loop. A search for transitions occurs, for all states of each new set, within the range of symbols and within the number of elements found for `j` (line 8). We search for all transitions of the form (start state, input symbol, ?). This suggests that the transitions of the input automaton should be indexed so that the target states can be retrieved quickly. Searching for transitions is so frequent that a preprocessing step to index the transitions should yield a performance benefit with each access. However, the space required to index the transitions must also be considered.

A second problem area is the maintenance of sets of states. Maintenance involves not only storing the states, but testing for set existence and equality. An easy, but inefficient way of determining if a set exists (line 12), is an element-by-element comparison of each existing set with the newly created set (`W`). If the set exists, we expect to compare `W` against half the existing sets before a match is found. If the set does not exist, then we compare all the existing sets with `W`. Since there can be a large number of sets, each containing a large number of elements, it would be beneficial if a set existence test could be performed quickly. However the sets are stored, if the query set exists, then we search (using, e.g., a hash table or a search tree) to prove its existence and retrieve the associated deterministic state number. When the query set does not already exist, however, it is added to the collection and the index variable `last` provides the deterministic state number for the new set. It suggests that simpler and efficient additional tests should be made to determine that a set is nonexistent. For instance, a vector of sizes could be kept. When a set is added to the list, its size is determined and the corresponding bit in the vector is set. Before searching for a set, if its size has not yet occurred, then it can be added to the list — no further query is required.

Part of the simplicity of Algorithm 1.1 is the loop over the whole set of input symbols (line 5). The symbol loop is often inefficient because time is spent considering symbols that do not occur on transitions from the state being evaluated. It is more efficient to work with only those symbols that will yield successful queries (on line 10). This improvement can be achieved only if a set of symbols, which yield successful queries, is provided for each state in the nondeterministic automaton. The final set of symbols used in the routine would then be the union of the sets of symbols used from each state within a given set of states. We reorganized lines 5 – 9 into a single function as the first step in obtaining an efficient subset construction algorithm.

2.4 Abstract Data Types

An important subgoal of GRAIL's implementation is to design programs that are easy to understand and modify and which provide a good environment for developing future programs. In order to meet these requirements, subset construction makes heavy use of abstract data types or ADT's. This approach makes the code more readable and it should minimize the problems that occur when modifying the code. These ADT's are also available for use in future GRAIL projects. Three ADT's were developed: Deterministic State, Collection of States, and Sorted Set of Transitions. The purpose and the required operations for these ADT's are described below, as well as the first implementation of each ADT. Improvements to the implementation of the ADT's are described in Chapters 4, 5 and 6.

2.4.1 The Deterministic State ADT

Deterministic State is the fundamental ADT of subset construction. Each instance of Deterministic State corresponds to a state of the deterministic automaton and this state corresponds to a set of states of the nondeterministic automaton; we regard the nondeterministic states to be elements of the deterministic state. There are six operations for Deterministic State:

- Create: The creation of a deterministic state involves memory allocation.
- Insert: This operation adds a nondeterministic state to the state set.
- Clear: This operation sets the size of the deterministic state to zero.
- Compare: This operation compares two deterministic states to determine if they are equivalent.
- Size: This operation returns the number of elements in a deterministic state.
- Copy: This operation makes a copy of a deterministic state.

An outline of the initial implementation of the Deterministic State ADT is now presented. The representation of Deterministic State has three parts: the number of nondeterministic states, an array of nondeterministic states, and the maximum number of nondeterministic states the array can contain. The nondeterministic states are stored in sorted order. An instance of this data type — a single deterministic state — is used repeatedly in subset construction to assemble nondeterministic states. We then copy this deterministic state and add it to a Collection of Deterministic States (explained in the next section). The maximum number of allowable nondeterministic states is the number of states in the input automaton. We chose to allocate space of the maximum size, to avoid the cost of reallocation. Adding an element involves a linear scan of the deterministic state to find the appropriate place to insert the element. If the element is already in the array, then nothing is done; otherwise elements are shifted to make room for the new element and the size of the deterministic state is incremented by one. In addition, if the size of the deterministic state is larger than the maximum size, then the entire state is moved to a larger region and a new maximum size is assigned. Clearing a deterministic state means that we set the size to zero. The comparison of two arrays is straightforward because the states appear in sorted order. First, we check whether the two deterministic states are the same size. If they are not, then we know the deterministic states cannot be equal. If they have the same size, we do an element-by-element comparison to determine if the states are equivalent. When a deterministic state is copied, the new copy is created with only the space required to contain the original data, that is, *maximumsize* := *size*. In the initial implementation, nondeterministic states were represented by integers, and as these are represented in words, we can have up to 2^{32} states. Given the restrictions hinted at in Section 1.4, the use of words is more than sufficient for most automata. A deterministic state is, thus, represented as a sorted array of integers.

2.4.2 The Collection of States ADT

A collection of deterministic states represent the set of states in the output automaton. In Algorithm 1.1, we see that `R` is a *Collection of (Output) States*, the deterministic state loop is executed for each one of the output states, and we want to represent the collection so that we can carry out rapid membership testing for a given deterministic state. In Algorithm 1.1 we see that a collection is accessed in two ways. A collection of states holds the deterministic state (a set of input states) and provides a way to search for the states, and the *deterministic state array* is an array which, instead of holding the whole deterministic state, holds a pointer to that state in the collection. When a state is added to a collection, a pointer to that state is returned and added to the deterministic state array at position *last*. The deterministic state loop is executed once for each state in the deterministic state array.

Whenever a deterministic state is created we check to see if it is in the collection, and if it is not in the collection, then we add it. Otherwise, we return its related deterministic state number. In total there are only three operations on Collection of States:

- Create: The creation of an empty collection.
- Insert: This operation adds a deterministic state to the collection.
- Member: This operation takes a deterministic state and returns the deterministic state number of that state, if the state is in the collection. Otherwise, `FALSE` is returned.

We now present an outline of the initial implementation of the Collection of States ADT. Creating a set of states involves allocating an array of pointers to deterministic states and initializing the pointers to NULL. A size variable is set to zero and a maximum size variable is set to the size of the pointer array. When we insert a deterministic state into the collection, we also store its deterministic state number. A copy of the state is stored and not a reference to it. The copy has only the required amount of memory needed to store the set, that is, the maximum number of nondeterministic states the array can contain is set to the actual number copied. In the first implementation, we copy the state and assign the first non-NULL pointer to the copy. The size of the collection is then increased by one. If the pointer array is full, then a new one of larger size is allocated. In other words, the implementation of the ADT is an array of deterministic states that are stored in arrival order. The deterministic state number is the index of the pointer to the state. In the implementation of Member, each deterministic state in the pointer array is compared to the query state. First the size of each state is compared with the size of the query state. If the sizes are the same, then an element-by-element comparison is conducted. Otherwise, the next state in the array is considered. If a state that matches the query state is found, then the pointer-array index is returned, as it is the deterministic state number.

2.4.3 The Pile ADT

The deterministic state and collection of states ADT's have many similarities. Both of these types manage a set of objects. However, unlike a standard set, we want to maintain not just integers but arbitrary objects; we must manage very large sets; and we do not require deletion. We call such an ADT a **pile**. Deterministic State and Collection of States are piles. Formally, we define the Pile ADT as having the operations:

- Create and Clear: These operation are basic.
- Insert: Insert an element. Optional data may be stored with the element. For instance, in the Collection of States we want to store the deterministic state number with the set.
- Member: Determine if a query element is in a pile. If it is, then extract the optional data when it is appropriate to do so.
- Size: Return the number of elements in the pile.
- Examine: Return a particular element in the pile. Examine[i] returns the ith element in the pile, where $0 \leq i < Size$.

From these basic operations we can build Copy and Compare operations. For efficiency, however, Copy and Compare can also be implemented as basic atomic operations. In future chapters we will examine various implementations of the Pile ADT.

2.4.4 Sorted Set of Transitions ADT

During subset construction transitions are randomly accessed (Algorithm 1.1, line 9) and the corresponding transition symbols and target states are acquired. In the initial implementation, transitions are unsorted and a linear search is required to find a given transition. This representation was changed, almost immediately, to a sorted set of transitions with an index for fast access. The input automaton is represented as a sequence of transitions in the form

$$(1, a, 2), (2, b, 3), \ldots, (\text{source state, symbol, target state}).$$

To create an instance of the Sorted Set of Transitions ADT, we sort the input transitions by source state, secondarily by input symbol, and thirdly by target state. We then set up a vector of pointers such that the pointer indexed by a source state gives the first transition starting with that state. In our implementation, the UNIX utility `qsort` is used to sort the transitions and a further pass over the sorted transitions is used to generate the index, the *transition index array*, `T_Index[]`. This will be explained in more detail in the next section.

2.5 Implementation of Multiway Merge

2.5.1 Multiway Merge

Our first attempt at improving subset construction was to replace the symbol and state loops by a multiway merge. One of the main drawbacks of Algorithm 1.1 is that it checks for symbols (line 7) that do not occur in any transition from the source state. These checks can be avoided with a merge. For example, in Table 2.1 we provide snapshots of Algorithm 1.1 with the data of Figure 2.1. There are two obvious inefficiencies with Algorithm 1.1. First, all input symbols are considered even when only a few appear in transitions. Second, when we sort the collection of states (column (c)) to give a sorted sequence (column (d)) we do not take advantage of the fact

Memory Location	Source State	Input Symbol	Target State
*	*	*	*
P4	4	a	3
	4	c	8
	4	r	11
*	*	*	*
P7	7	a	8
	7	b	1
	7	r	6
*	*	*	*
P9	9	a	3
	9	a	9
	9	c	4
	9	r	22
*	*	*	*
P11	11	z	2
*	*	*	*

Figure 2.1: Indexed Transitions: The set of source states is {4,7,9,11}. Only indexed transitions of relevance to the example are given.

(a) For Symbol s	(b) Consider transitions (Ψ s ?) where $\Psi \in$ {4,7,9,11}	(c) Collect target States {?}	(d) Order states and make set
a	(4 a 3)		
	(7 a 8)		
	(9 a 3)		
	(9 a 9)	[3,8,3,9]	{3,8,9}
b	(7 b 1)	[1]	{1}
c	(4 c 8)		
	(9 c 4)	[8,4]	{4,8}
d	none		
.	.		
.	.		
q	none		
r	(4 r 11)		
	(7 r 6)		
	(9 r 22)	[11,6,22]	{6,11,22}
s	none		
.	.		
.	none		
z	(11 z 2)	[2]	{2}

Table 2.1: Snapshots of the set generation process in Algorithm 1.1 with the input automaton of Figure 2.1.

```
loop
   Find $i$ | $i \in$ [0..size],
   where Finger[$i$] points to the transition
   with the smallest symbol and smallest target state in case of a tie.
   return_value = Finger[i]);
   If Finger[i+1] = <end of transition list> or
      Finger[i+1].symbol <> Finger[i].symbol
      then
         Finger[i]=NULL; size = size -1;
      else
         Finger[i] = Finger[i] + 1; %points to next transition
   Output(return_value);
until size=0;
```

Figure 2.2: The multiway merge routine.

that the input transitions, in Figure 2.1, are already sorted. We would like to avoid these two inefficiencies.

A method was developed in which all the processing in Table 2.1 occurs in one structure. This procedure is a multiway merge sort and we provide a single structure to support it. First, pointers to the first transition for each state are created. For the example in Table 2.1, the pointers are: $T\_index[4] = P4$, $T\_index[7] = P7$, $T\_index[9] = P9$, $T\_index[11] = P11$, where $P4$, $P7$, $P9$, $P11$ are the locations of the corresponding sets of transitions; see Figure 2.1. These pointers are part of the Sorted Set of Transitions ADT and are set up at the start of the conversion routine. Each time we begin a merge, temporary pointers or fingers are assigned from the relevant T_index pointers. So, for our example we have:
$Finger[1] = T\_index[4], Finger[2] = T\_index[7]$,
$Finger[3] = T\_index[9]$, $Finger[4] = T\_index[11]$;
hence, we have a multiway merge of size 4. The code for the multiway merge is given in Figure 2.2. For the example of Table 2.1, the merge proceeds as follows:

$Finger[1] = (4a3); Finger[2] = (7a8); Finger[3] = (9a3); Finger[4] = (11z2);$

The smallest finger is $Finger[1]$ (because the tie between $Finger[1]$ and $Finger[3]$ is broken by choosing the lower index)

$return\_value = (4a3)$

$Finger[1]$ is then changed to point to the next transition (4 c 8)

$Finger[1] = (4c8); Finger[2] = (7a8); Finger[3] = (9a3); Finger[4] = (11z2);$

$return\_value = (9a3)$

$Finger[3]$ is then changed to point to the next transition $(9a9)$

$Finger[1] = (4c8); Finger[2] = (7a8); Finger[3] = (9a9); Finger[4] = (11z2);$

$return\_value = (7a8)$

$Finger[2]$ is then changed to point to the next transition $(7b1)$

$Finger[1] = (4c8); Finger[2] = (7b1); Finger[3] = (9a9); Finger[4] = (11z2);$

$return\_value = (9a9)$

$Finger[3]$ is then changed to point to the next transition $(9c4)$

The algorithm continues and returns $(7b1), (9c4), (4c8), (7r6), (4r11),$ $(9r22), \text{and}(11z2)$.

The output produced by this algorithm is:

```
4   9   7   9   7   9   4   7   4   9   11
a   a   a   a   b   c   c   r   r   r   z
3   3   8   9   1   4   8   6   11  22  2
```

We obtain the deterministic states from these transitions by ignoring the source states to give:

```
a   a   a   a   b   c   c   r   r   r   z
3   3   8   9   1   4   8   6   11  22  2
```

and removing duplicate target states with the same input symbol:

```
a   a   a   b   c   c   r   r   r   z
3   8   9   1   4   8   6   11  22  2
```

Finally, we partition the remaining transitions, by the input symbols:

```
a   a   a  |  b  |  c   c  |  r   r   r  |  z
3   8   9  |  1  |  4   8  |  6  11  22  |  2
```

and these blocks give the deterministic states, in sorted order, as
{3,8,9},{1},{4,8},{6,11,22}, and {2}.
The corresponding DFA transitions are obtained by using the input symbols of each block of the partition:

({4,7,9,11} a {3,8,9})
({4,7,9,11} b {1})
({4,7,9,11} c {4,8})
({4,7,9,11} r {6,11,22})
({4,7,9,11} z {2})

We implemented the multiway merge in a slightly different way from our description. The source set is created and the DFA transition generated as soon as a symbol partition is available. That is, in each loop of the multiway merge, whenever the symbol of the output transition changes, all previous output transitions give a symbol block. In addition, the removal of duplicates is performed as transitions are generated by multiway merge.

Multiway merge completely eliminates the symbol loop (lines 5–19 of Algorithm 1.1). This change has little effect on performance if most states have transitions on all symbols.

Multiway merge also takes advantage of the sorted transitions and, thereby, reduces sorting time. The following example illustrates an extreme case. Suppose we have the DFA source state {1} and sorted transitions

```
1  1  1  1  1  1
a  a  a  a  a  a
2  3  4  5  6  7
```

When this is processed by multiway merge (size=1), the transitions are extracted directly (in order) with no processing needed to find the smallest finger. The resulting set {2,3,4,5,6,7} is produced without any sorting.

Figure 2.3: A tournament example. If the data elements are removed from (a) one at a time the tournament becomes (b).

For multiway merge to be efficient, finding the location of the smallest finger has to be implemented efficiently. The initial implementation used a tournament. We discuss this approach in the next subsection.

2.5.2 Tournaments

The tournament was initially used in the multiway merge because it is known to be a good data structure for merging data, in particular for merging data from secondary storage [2, pages 142–143, 145–147]. A tournament is a tree structure which contains the data (in our case, fingers), at the leaf nodes. The root of the tree, after the tournament is processed, contains the smallest finger. In a tournament, every node that is not a leaf node contains the minimum data element in the leaves of its subtrees. Figure 2.3 gives an example of a tournament for integer data. The multiway merge uses the tournament to "bubble-up" the smallest fingers and then remove them. A tournament has two operations: setup and remove_element. The setup of the tournament is exactly as in Knuth [2]. The leaf nodes are assigned the fingers and then the nodes at higher levels are assigned, for our implementation, the smaller of their childrens' fingers. The root of the tournament then contains the smallest finger. The removal of an element from the tournament implies that we should restructure the tournament. The removed finger is checked to see if it gives further transitions from the same source state. If not, then that finger is set to NULL and the fingers on the path to its corresponding leaf node are recomputed. If the finger has other transitions to be considered, then the finger is moved to the next transition and the path is reorganized in a similar fashion.

More rigorously, we restructure the tournament by following the path from the root until we get to the leaf from which the current finger originated. We then assign this leaf node the new finger (pointer to a transition or NULL) and work back up to the root reassigning each parent node with the smaller value of its children. A NULL finger is assumed to be larger than all other transitions. The tournament now has the smallest finger at its root. When a NULL finger reaches the root the tournament is considered to be empty, since every finger in the tournament is NULL at this point.

Chapter 3

Testing and Evaluation

3.1 Introduction

We have experimented with numerous implementations of subset construction. Some versions were clear improvements, while others performed well for some inputs and not for others. Each time a new version of the algorithm was created, it was essential to verify that it worked correctly. We also compared each new implementation with other implementations, including the brute-force implementation and the INR subset construction implementation. Immediately upon completion of the first modifications to the brute-force method, we confronted the problem that there were very few automata available for use in testing. Thus, the generation of test automata became an important aspect of the project.

3.2 Test Automata Generation

We did most of the performance evaluation of the implementation of subset construction using computer-generated automata. These automata were generated in two ways: randomly and by cross-product. There were only a few small hand-constructed automata available for testing. These small automata, of between three and twenty states each, were designed to test correctness, not to tax the abilities of subset construction.

3.2.1 Manual Generation

The initial tests for boundary conditions used an empty automaton and an automaton with no start state. More general input errors were tested with small hand-crafted automata. Automata that we found valuable, particularly when tracing coding problems with the multiway merge, were deterministic, nondeterministic by one transition, nondeterministic many times at one state, and nondeterministic many times at one state on more than one symbol. The validity of each implementation was first tested using the hand-crafted automata for which the resulting deterministic automata were easy to verify.

3.2.2 Cross-Product Generation

The cross-product $(\times)$ of two automata F_1 and F_2 is defined by

$$F_1 \times F_2 = (Q_1 \times Q_2, \Sigma_1 \cap \Sigma_2, \delta, (s_1, s_2), F_1 \times F_2),$$

where δ, the new transition relation, is given by

$$\{((x_1, x_2), a, (z_1, z_2)) \mid (x_1, a, z_1) \in \delta_1 \ , \ (x_2, a, z_2) \in \delta_2, \text{ and } a \in \Sigma_1 \cap \Sigma_2\}$$

In general, $F_1 \times F_2$ is an automaton F that accepts a string if and only if it is accepted by both automata F_1 and F_2. For example, if F_1 accepts $(aa)^*$ and F_2 accepts $(aaa)^*$ then F accepts $(aaaaaa)^*$. The cross-product of an automaton with itself (e.g. $F_1 \times F_1 = F$) gives an automaton that accepts the same language as F_1; however, F may not be isomorphic to F_1. We say that two automata $F_1 = (Q_1, \Sigma, \delta_1, s_1, F_1)$ and $F_2 = (Q_2, \Sigma, \delta_2, s_2, F_2)$ are *isomorphic* if there is a bijection $h : Q_1 \rightarrow Q_2$ such that $h(s_1) = s2$, $F_2 = \{h(f) : f \text{ in } F_1\}$, and $\delta_2 = \{(h(p), a, h(q)) : (p, a, q) \text{ in } \delta_1\}$. In particular, if F_1 is nondeterministic, then F_1 and F are not isomorphic (this characteristic should be apparent from the definition), but clearly when F_1 is deterministic, $F_1 \times F_1$ is isomorphic to F_1. We made extensive use of the following result concerning the relationship between subset construction and the cross-product of an automaton with itself. An automaton is *connected* if, for every state, there is at least one input sequence that can reach that state from the start state.

Proposition 3.1 *Let M be a connected finite automaton. Then,*

$$\textit{subset}(M \times M) \textit{ is isomorphic to } \textit{subset}(M) \times \textit{subset}(M).$$

Figure 3.1: Commutative diagram — the subset construction of the cross-product of two identical automata is isomorphic to the cross-product of the subset construction of two identical automata.

The proposition states that the cross-product and subset operations commute when applied to a single automaton as indicated by the communative diagram in Figure 3.1. The proposition also holds when we are given two different nondeterministic automata.

Figure 3.2 displays the cross-product $FA2$ of two identical automata $FA1 \times FA1$; $FA2$ has twice as many transitions as $FA1$. In general, new states must be created that represent all possible combinations of routes that can be taken out of each state. It should be apparent that the states (6,7) in $FA1$ would be represented as the deterministic state {6,7} and states (2,3,4,5) in $FA2$ would represented as the deterministic state {2,3,4,5} . As the cross-product is continually reapplied to the resulting automata, their sizes increase. The example automaton in Figure 3.2, regardless of how many times cross-product is applied, will continue to generate only two sets of states in the subset construction algorithm. Thus, we can generate arbitrarily large automata whose equivalent deterministic automaton is known in advance.

The cross-product method creates test automata that tax the multiway merge and deterministic set, and in addition, they tax the hash-function routine that creates a hash table index from each set. The cross-product method does not tax the collection of sets because the number of generated sets is the same, and a large number of states collapse into a single state, analogous to the way in which they were created.

Figure 3.2: The cross-product $FA1 \times FA1 = FA2$. When subset construction is applied to $FA1$ and $FA2$ it yields isomorphic deterministic automata.

3.2.3 Random Generation

The cross-product operation generates large automata, but such automata have a restricted structure. In order to ensure that we were not exploiting this restriction, we decided to use randomly generated automata. Such an approach also enables us to estimate the expected performance of the implementations of subset construction. In the method we developed, the number of states, number of symbols and percentage density of the automata are user definable. The *density* of an automaton is the ratio of the number of transitions in the automata and the largest possible number of transitions in an NFA with the same state set and input alphabet. The procedure for random generation is given in Figure 3.3. The expected number of transitions generated by the algorithm is

$$((number\, of\, states)^2 \times (number\, of\, symbols)) \times density.$$

By changing the three input parameters a variety of automata can be generated.

Randomly-generated automata usually have unreachable states. Although subset construction works with disconnected automata, it creates some problems for the purposes of evaluation. First, INR does not work with some disconnected automata (actual testing failed). Second, the cost of

```
begin
Input num_states, num_symbols, density;
Output ``(START) |- 2''
For $p$ := 2 to num_states + 1 % loop over number of states
begin
   For $a$ := 2 to num_symbols + 1
   begin
      For $q$ := 2 to num_states + 1
      begin
         If((rand()%100)<density)
            then Output ``$p$,$a$,$q$''
      end
   end
end
end
```

Figure 3.3: Algorithm for generating random automata with a given number of states, symbols, and a percentage density in the range 0–100 percent.

unreachable transitions, states, and symbols is not directly reflected in the output. Third, performance comparison is difficult because the automaton may be much larger than the connected sub-automaton. Since random generation does not guarantee connectedness, the generated automata are checked for connectedness. Only automata that are connected are used for evaluation and testing.

An alternative method for generating automata randomly that avoids connectedness checking is: *Connect all the states together randomly, to ensure connectedness, then generate random transitions.* We generate a transition from the start state to any other state and mark that state as "visited." The start state is also marked as "visited." Then, we repeatedly generate a transition from any "visited" state to any state not yet visited and mark that state "visited," until all states have been visited. This completes the connectedness phase of the generation. We continue the generating process by generating random transitions, until we obtain an automaton that is dense enough. We generate a random state number (source state), input symbol and state number (target state) to form a random transition. We check to see if this transition already occurs and if so, we discard it and generate another. We call this method of random generation the *connected method*. The method illustrated in Figure 3.3 is called the *all-density* method because it can generate random (possibly disconnected) automata with a given density. The connected method is inefficient when generating high-density random automata because there will be many duplicate transitions to be discarded. The connected method is used when evaluation requires a percentage density in the range 0–80 percent, whereas the all-density method is used in all other cases.

3.3 Correctness Testing

As we implemented subset construction algorithms, we tested them for correctness. We created a UNIX script to test each implementation with several hand-crafted automata. We next tested each implementation with large automata obtained by cross-product and random generation. The results with large automata were compared to those obtained with INR. The INR subset construction routine was used with larger automata because it is very fast and its correctness is not in doubt. If all the tests succeeded, then the new version of subset construction was considered correct.

3.4 Performance Evaluation

Once an implementation has been certified correct, its performance was compared with other implementations. Different implementations are affected by different characteristics of input automata. There are four primary factors which play an important role in the performance of subset construction: the input automaton, output automaton, size of deterministic states, and size of the collection of states. There are some important details which should be borne in mind, regardless of other characteristics the automata may have. For example, the transitions must be sorted and the symbols, which can be character strings, must be assigned numeric identifiers. The size of the automaton is important because the larger it is, the less workspace is available for subset construction.

The output automaton is the result of subset construction on the input automaton. If we hold the numbers of transitions, symbols, and states constant, then we can still produce output of vastly different sizes by changing the structure of the input automaton. For example, one automaton can collapse to only a fraction of the input size (as we see with the cross-product test cases), whereas another may expand exponentially. The expansion factor indicates how often multiway merge is used because all output transitions are formed by it. In addition, it is always the case that the more an automaton expands, the more deterministic states are generated and added to the collection of states.

The size of a deterministic state depends on the number of states in the input automaton. Automata with n input states can generate deterministic states only as large as n. Assembling and comparing these states becomes more time consuming as the number of states grows. We expect the Collection of States ADT to consume more time as the number of states increases; each member and insert function will become more expensive. The size of the collection can make a significant impact on the performance of the algorithm.

When analyzing the performance of an implementation of subset construction, it is important to understand how the individual factors contribute to the overall performance of the algorithm. In particular, we want to know whether a new modification improves the performance of the conversion uniformly or only for some types of automata.

3.5 Profiling

The UNIX profiling function, `profile`, is useful for determining where time is spent in a program. A compiler option can be set that causes a program to keep track of the amount of time spent in the program's functions and the number of times they are called. We can analyze the performance of various implementations working with various automata and locate functions that are frequently called or are taking a large percentage of the execution time. We may then attempt to improve the coding of these functions.

3.6 Evaluation of Results

The implementation developed in Chapter 2 is identical to the brute-force implementation except that the tournament-based multiway merge replaces the original symbol loop and the transitions are sorted and indexed directly. We call this implementation *Ch2*. We now analyze *Ch2's* performance. Three graphs (Figures 3.4–3.6) illustrate the performance of INR, *Ch2*, and the brute-force (BF) implementation on randomly generated automata of 10 states, with 10 and 15 symbols, and 15 states and 10 symbols. We have already suggested that the BF implementation is inefficient because it considers symbols that do not appear in transitions. The graphs show that the BF implementation is, indeed, slower than the other two methods when the number of symbols is increased. If we look at a density of 20 percent, then we see that the BF method deteriorates with respect to *Ch2*. With 10 states and 10 or 15 symbols the ratios of the times for BF and the times for *Ch2* are approximately 3.0 and 3.5, respectively. The ratios get larger as the number of symbols increases. For this small range of symbols, we see that the performance already differs substantially. The improved performance of *Ch2* occurs over the entire range of density and for different numbers of states and symbols. The profiling of random test cases shows that, for automata taking at least a few seconds to convert, the BF method spends 70–98 percent of its total time locating transitions and 1–15 percent locating states in the collection of states. *Ch2*, however, takes 27–83 percent of its total time in the multiway merge and 1–23 percent locating states in the collection of states.

A number of cross-product tests were also applied to the three implementations. Beginning with the automaton NFA_cross with 10 states, 20 transitions and 4 symbols, we formed the cross-product of NFA_cross with itself twice to give the automaton NFA_cross2 and with itself thrice to give automaton NFA_cross3. NFA_cross3 has 32,770 states, 81,920 transitions and 4 symbols. As mentioned earlier, applying the subset construction to NFA_cross gives the same deterministic automaton as applying the subset construction to NFA_cross3, and this deterministic automaton has 4 states, 5 transitions, and 4 symbols. When subset construction is applied to NFA_cross3, 4 deterministic states are generated from the 32,770 states in the input automaton. More precisely, we obtain a START state, a FINAL state and two other deterministic states each of which contain 16,384 states. This type of test automaton gives not only large deterministic states and large tournaments, but also a very small collection of states and very few merges. The times for subset construction, on NFA_cross3, by the three implementations, are: INR – 29.7 seconds, *Ch2* – 46.4 seconds, BF – many hours (33 minutes for NFA_cross2). There are only 4 symbols, so we know that the multiway merge does not play a large factor in the dramatic performance improvement of *Ch2* over BF. Thus, it must be the sorted and indexed transitions that play the more important role. Profiling BF with NFA_cross2 shows that the time to locate transitions takes 86 percent of the total time and adding to the collection of states takes 13 percent. Profiling *Ch2* shows the multiway merge takes 36 percent of the total time, and reading and sorting the transitions takes 23 percent, with other functions taking less than 40 percent. With NFA_cross3, *Ch2* accesses the transitions over 100,000 times. The BF method searches a larger number of transitions, while the improved version does not search the transitions because they are indexed by source state.

The graphs show that *Ch2* performs well in comparison to the original implementation, but not as well as INR. INR uses a hash table to store the collection of states, whereas *Ch2* uses the method employed in the BF implementation: an array of states arranged in generation order. In the random tests with only 10 states, certain density values can produce nondeterministic automata which, after subset construction, can have over 1,000 deterministic states. With this many states, the Collection of States ADT, as currently implemented, is quite inefficient and needs to be improved.

The cross-product test with NFA_cross3 generates only four deterministic states; therefore, the collection of states handles this efficiently. But, other parts of the algorithm need improvement. Profiling indicates that the multiway merge is a frequently called function in *Ch2* and it takes the most time in subset construction. When the input automata are large, the multiway merge takes more time than INR uses for the whole subset construction. A closer examination of INR reveals that it uses a heap rather than a tournament for the multiway merge.

The multiway merge and the Collection of States ADT are two important items that need improvement, and these improvements, among others, are the subject of the next three chapters.

Figure 3.4: A 10-state, 10-symbol random automata test.

Figure 3.5: A 10-state, 15-symbol random automata test.

Figure 3.6: A 15-state, 10-symbol random automata test.

Chapter 4

Set Signatures and Hashing

As deterministic states are created, the brute-force implementation adds them to the end of an array; a search for a given state is implemented as a sequential search in the array. The brute-force implementation compares the sizes of two deterministic states, since the size is stored with the state. If the sizes of the two states are different, the states are different. Even with the size check, this implementation of subset construction takes too much time to maintain the collection of deterministic states; therefore, we examine alternative approaches in this chapter.

4.1 Set Signatures

A *signature* is a number associated with a set that has the following properties: If two sets have the same signature, then they may be equal, otherwise the sets are different. The advantage of a signature is that it can be used to quickly determine if sets are different and avoid the costly element-by-element comparisons. The size of a set can be regarded as a signature. A simple signature function such as set size is adequate for small automata but becomes ineffective for larger automata because there are too many sets of the same size. It would not be uncommon, for instance, for an input automaton having 20 states to generate thousands of deterministic states of the same size. For example, for a 20-state automaton, there can be a maximum of 20 states of size 1, 190 states of size 2, and 184,756 states of size 10. An array containing 184,756 states with the same signature implies that searching is very slow. An element-by-element comparison is required on half of these states, on average.

Signatures can be arbitrarily complex, but we restrict the size of a signature to be no larger than a word (or integer). Signatures should provide a large range of values and the values should be 'well-distributed,' meaning that we do not want a large number of sets to have the same signature, but rather, we want as few sets as possible with the same signature. The fewer sets with the same signature, the fewer element-by-element comparisons are necessary. To obtain signatures that provide good dispersion and a large range, we use most of the information that is contained in a deterministic state. For instance, we can use a polynomial expression based on every element in a set. Given a set $S = \{s_0, s_1, ..., s_n\}$, one possible signature is given by

$$Sig(S) = s_n 37^n + s_{n-1} 37^{n-1} + \cdots + s_0$$

modulo the number of distinct signatures. The polynomial signature improves subset construction, but as it was being implemented we realized that even if signature generation is perfect, traversing the array and comparing signatures is still too costly. We need to avoid traversing sets each time we want to add or search for a set. We should group the sets by their signature and restrict searching to this group.

4.2 Deterministic State Hashing

With the possibility of millions of deterministic states being generated, it seems that hashing is best suited for the task of storage and retrieval. The information in a deterministic state can be used to form a signature for that state, which can then be used as an index into a hash table. We decided to handle collisions with separate chaining, since this groups together the deterministic states with the same signatures.

4.2.1 Hash Table Size

The hash table is an array of pointers to chains of deterministic states. The signature generated from each deterministic state is scaled to the size of the hash-table and this value is used as an index into the table. The hash table index is the signature modulo the table size. The hash table is accessed many times during the conversion, so we expect that a large hash table will make the conversion faster. After some experimentation with hash table size, we determined that for the automata that we can convert, a hash-table size of the order of 2^{16} or 64K is adequate. The larger the hash table, the faster the states can be found because there are fewer collisions and shorter chains. The justification for not having a larger table size is that if the average chain length is 10, then the hash-table can hold up to 640K states. Automata of this size approach or even exceed the storage capacity and execution ability of the machines that we used. The hash-table is not too large when compared with the memory that may be used to store the deterministic states. The hash-table uses only a small fraction of memory, even when a 30-state NFA is converted into a DFA.

4.2.2 Hash Functions

The signature value for each set is scaled to the size of the hash table and it is then used as an index into the hash-table of pointers, which point to linked lists. If a pointer is NULL, it means that there are no states with that hash-function value. Otherwise, we search sequentially through the corresponding linked list. Each list element consists of a deterministic state, deterministic state value, and pointer to the next node in the list. Since a deterministic state contains the size of the state, we first compare the size of the query state and the deterministic state. If the sizes are equal, then we compare the two states element-by-element. Every time a set is accessed or stored, we place it at the beginning of the chain. Although this requires very little processing time, it can improve the performance for popular sets. This is the well known move-to-front heuristic.

The hash function we first used was the polynomial signature. It works well as a hash function for the same reasons that it works well as a signature. This hash function scatters the deterministic states evenly over the table and even with small sets it provides a large range of values. Implementing a pile with a hash table and polynomial signature worked so well that we wondered whether a simpler polynomial or some other method would make as good a signature, thereby decreasing the time for computing the signature, resulting in an overall improvement in performance. It seems reasonable to suppose that the multiplication and modulus operations are time-consuming operations. Instead of using every element of the deterministic state in the polynomial, we considered using every second or third element. The time saved in signature calculation is, however, insignificant and does not outweigh the benefit of good table allocation.

Testing the various alternatives gave very similar results and, therefore, generating a polynomial signature using all the elements has been retained as the preferred implementation. The multiplication and modulus operations are not as costly on some of the architectures we used because they have mathematical coprocessors and some even have pipelining for mathematical operations. Testing was performed on VAX (DEC 5500) and MIPS machines.

4.3 Set-Existence Checks

So far we have checked for set membership by searching for a given set. If a set is present, then we need to locate it because we must obtain its deterministic state value. If the query set is not there, we would like to avoid searching if at all possible. When we have short chains, searching costs hardly anything, but if the chains are long, it would be beneficial to eliminate some of the searching that takes place. In earlier implementations existence checks were used to try and eliminate some of the time taken in verifying that a set was absent. There are two simple and fast existence checks that can be used.

4.3.1 Universe Check

One existence check is the *universe check* that uses a bit vector to hold the nondeterministic states that have been seen. When a deterministic state is generated, we check whether each of its nondeterministic states have been seen before. If one has not been seen before, then the deterministic state is not in the collection. In this case, we can add this deterministic state directly to the collection. If the check fails, then we search for it in the usual way.

4.3.2 Size Check

Another existence check is the *size check*; which holds in a bit vector the sizes of the deterministic states that have been generated so far. When a deterministic state is generated, if its size has not been seen before, then it is a new state. Otherwise, we search for it in the usual way.

4.3.3 Diminishing Returns of Existence Checks

Although existence checks seem helpful in reducing the processing time of some automata, once the conversion process generates many sets of each size and enough to cover the universe of states, no benefit is obtained from using these checks. As automata with ten or more states and hundreds of transitions are common, we concluded that the existence checks were useless. For the conversion of large automata, the existence check even becomes harmful; therefore, the existence checks were removed.

Chapter 5

Multiway Merge with a Heap

5.1 The Heap

The multiway merge replaces three loops in the brute force implementation. Profiling shows that 30–90% of the time can be spent in multiway merging; therefore, an improvement in multiway merge can cause a substantial overall improvement. The multiway merge takes a lower percentage of time (30%) when the input automaton is large and the processing is uncomplicated, as is the case with large automata generated by the cross-product method. Small input automata that generate large output automata make heavy use of multiway merge, as is the case with some randomly generated automata. The first implementation of multiway merge used a tournament, which was convenient because it performed only enough sorting to ensure that the root contained the smallest transition. Each time the root transition was extracted, the corresponding finger would be advanced to the next transition and the tournament would be reordered if necessary. One problem with the tournament is that the data is stored only at the leaf nodes and thus extraction initiates the traversal of a root-to-leaf path to advance the finger of the corresponding leaf node, and a reverse traversal to reorder the tournament.

A heap [2] is also an appropriate structure for producing transitions in sorted order, with the advantage that it stores each transition once in an internal node, rather than in both internal and external nodes. The use of a heap eliminates some of the reordering time that is needed in a tournament. As with the tournament, the heap orders transitions first by symbol and then by target state. Starting from the leaf nodes, the heap is ordered from bottom to top ensuring that every parent node is smaller than its two children. Our heap differs from the standard heap only in the removal of a transition.

As with the tournament, when the root transition is removed, the corresponding finger is advanced to the next transition. If there are no more transitions under that finger then, as with a normal heap, the last finger in the heap, in the rightmost leaf node, is moved to the root, the size of the heap is reduced by one, and the fingered transition is trickled down. Otherwise, the finger points to another transition that becomes the new root transition which is trickled down if necessary. In either case the root transition is changed and the heap is restructured to reestablish the heap property — the transition in every parent node is less than the transitions in its children. To do this, we compare the root to its children. If the root is smaller than both children, then no restructuring is necessary. Otherwise, the root is larger than at least one of its children, so the smaller of the two children and the root are swapped. This trickling process continues down the tree until the original root transition reaches a position where either it is less than both of its children or the frontier of the heap is reached. In comparison with the tournament, heap reordering involves a smaller number of nodes. As shown in Figure 5.1, the heap performs much better than the tournament for the same set of transitions.

The multiway merge with the heap is still not competitive with INR. Analysis of the INR subset construction code revealed that INR's heap is being used to sort transitions only with respect to the transition symbol; it ignores the target state. This implies that the transitions removed from INR's heap occur in proper symbol order, but for each group of transitions with the same symbol, the transitions are not necessarily in sorted order. It is necessary to sort the transitions by the target states to obtain the correct representation of the target set (remember that all sets are represented in sorted order). If the heap only orders transitions on symbols, then, when the transitions with the same symbol have been removed from the heap, they must be sorted separately by target state. We describe such a modification in the following section.

5.2 The Symbol Sorting Heap

We have modified subset construction so that a group of target states pulled from the heap are placed in an array and then sorted by the UNIX `qsort` routine. The heap only sorts transitions

Figure 5.1: Multiway merge — heap versus tournament.

with respect to the input symbol. Performance testing of the new version showed substantial improvements over the previous version which sorted the transitions fully in the heap. It is apparent that the full sorting heap spends excessive time reordering transitions that only differ in their target state. The extra reordering occurs primarily when the finger at the root has multiple transitions with the same symbol. The symbol sorting heap returns the transition at the root and then advances the finger to the next transition which has the same symbol as the previous one. This new transition is returned and again the finger is advanced to the next transition. This cycle continues until the symbol of the root transition differs from the symbol of the previous root transition. The transitions with the same symbol are, in effect, returned immediately with no intervening heap reordering. The full sorting heap also returns the transition at the root and advances the finger to the next transition, but because the transitions are sorted according to both symbol and target state, the root transition is compared with the transitions of its two children to see if one of them has a smaller target state. If one child has the same symbol but a smaller target state, it is exchanged with the root transition and the trickle down continues until the transition reaches its place in the heap. We see that sorting transitions by only the symbol allows transitions to be pulled from the heap much earlier. Even though these transitions must undergo additional sorting, it is more efficient than doing the sorting in the heap.

The use of low density automata with the symbol sorting heap also shows an improvement in performance. Low density automata do not provide many situations of the kind described above, therefore, it is apparent that in general, doing a separate sort is always more efficient. This may be the result of the fast UNIX `qsort` routine or that sorting on two fields is more efficient if done in separate processes. Regardless of why the separate sort is faster, the development of the bit vector store and sort method explained in the next section justifies this approach in another way.

5.3 Bit-Vector Transition Sort

Bit vectors are used extensively in our latest and best implementation of subset construction. The first way in which a bit vector is used is to store the states produced by the multiway merge. Concurrent research into the behavior of automata (Chapter 7) indicates that automata with thousands of states usually exceed the limits of the computer systems available to us. If we assume that automata do not usually have more than ten thousand states, then we can use a bit vector to store the states. After reading in the input automaton we know how many states there are, so a bit vector can be created that is large enough to hold all the states. For even very large automata this involves only a few kilobytes. The bit vector is a Pile since we insert into it but never delete from it; it is cleared before each use. As we remove states from the heap, we store them in the bit vector by setting the corresponding bit to 1. When the symbol changes, the bit vector contains the states which make up a deterministic state. We now scan the bit vector and create a deterministic state by adding the corresponding integer for each 1 bit. Note that the integer states are added in sorted order, and that this sorting method is much faster than `qsort` used in the previous implementation. We know that in dense automata, the group of target states retrieved from the heap contains many duplicates. When `qsort` is applied to the resulting array, time is wasted in sorting the duplicate states which are eventually removed by a linear scan. The bit vector reduces the time spent in sorting and by its very nature, takes constant time to eliminate each duplicate appearance of a state, the best we can hope for.

The bit vector approach is very successful, primarily because it simplifies sorting. There are situations, however, where this method is not beneficial. If the number of states is large and transitions are bunched or very sparse, the number of bits set in the bit vector is small in comparison to its size. Since we must scan the entire bit vector regardless of how few bits are set, it becomes less efficient than `qsort`. For example, we might scan thousands of bits to find only a few 1-bits. Such a small number of states is more efficiently handled by `qsort`. This is, however, not considered a problem because automata with this property take little time in subset construction. Automata that require much time normally have large deterministic states, and this implies that the bit vector will have many marked bits.

Chapter 6

Bit Vector Implementations of Piles

6.1 Introduction

Since we have used a bit vector to represent the states computed by each phase of the multiway merge, why not use a bit vector to represent deterministic states wherever they appear? We examine how the use of bit vectors can be exploited in subset construction.

6.2 A Bit Vector Representation of Pile

We represent a deterministic state with the same information as before: the size, the maximum allowable size, and an array of states. We replace the array of integers, however, with a bit vector. For example, in our implementation, if the input automaton has state values ranging from 0 to 63, we must allocate deterministic states with $64/32 = 2$ words. It is also apparent, from this example, that the new representation can save substantial memory. We have replaced integer state values by single bits. The down side of this choice is that we need the full bit vector to represent only one state. In Figure 6.1, the bit vector representation of the set {1,2,4,5,7,9,10,12,14,17,19,24} clearly uses less memory than the integer representation. However, in Figure 6.2 the bit vector

Figure 6.1: Set of nondeterministic states {1,2,4,5,7,9,10,12,14,17,19,24} stored in two forms: Integer and bit vector. The bit vector representation uses only 1 word of memory; integer representation uses 12 words.

representation of the set {13, 149} uses more memory than the integer representation. Fortunately, the difficult conversions (the output DFA are much larger than the input NFA) frequently benefit from the bit vector approach because the number of elements in a deterministic state, on average, exceeds the number of words used to represent the bit vector.

One major advantage of the bit vector representation is that the elements of a deterministic state are maintained trivially in sorted order. The implementation of the clear operation is, however, more complex than the previous implementation; previously we had to reset the size to 0 and overwrite the old data. In the bit vector implementation, we must clear the entire bit vector in addition to setting the state size to 0. Although this operation takes more time, clearing a state is done infrequently.

As noted, in Section 5.3, we modified the multiway merge by entering the target states directly into a bit vector to speed up sorting. Using the new bit vector representation of a deterministic state, the output of the multiway merge is a deterministic state. This eliminates the transformation of the deterministic states from bit vector form into integer array form.

6.3 Power-Set Bit Vector

If we use a bit vector of size n to represent a deterministic state, we can then use the integer represented by a bit vector as the unique index into a table of size 2^n that represents a set of

Figure 6.2: Set of nondeterministic states $\{13, 149\}$ stored in two forms: Integer and Bit vector. Integer representation uses only two word of memory — Bit vector uses five words.

deterministic states. Moreover, because each bit vector gives a unique index, the table of size 2^n can be represented also as a bit vector of size 2^n. We call such a representation a *power-set bit vector representation*; it is illustrated in Figure 6.3. Naturally, this approach is out of the question for all but small automata; even an automaton with 32 states requires bit vector of size 2^{32} (more than four billion states). Given such a representation, we can check easily if a deterministic state has already occurred by taking the integer represented by the state's bit vector and using it as an index into the power-set bit vector. If the corresponding bit is 1, then the state has already occurred; otherwise, we set the bit to 1. In either case, we return a deterministic state number for the given state. We use the table index as the deterministic state value, both for convenience and to save space. The bit vector tells us only that the set has occurred and unlike the original integer array representation of the pile, we do not store the corresponding output state number. The purpose of the power set approach is to illustrate an extremely efficient way to store states when we apply subset construction to automata that generate a large number of output states. By assigning integers to output states in sequence, it allows us to inspect the output automata and see in what order the states were generated. This order can be useful for students studying the steps of the construction or for debugging purposes. However, for the extremely large automata

Figure 6.3: A set K in integer and bit vector form. The integer representation of K is used as the index in the power-set bit vector (collection of sets)

that are generated via the power-set implementation, sequential numbering of output states will not be of any particular benefit because the automaton is too large to examine. Therefore, the output states are numbered by the integer representation of the set. This numbering is simple to accomplish because the set size is restricted, in practice, to be less than 32, so the bit vector that stores a state is in fact already an integer.

Although the power-set method can be used for only small input automata, it is extremely fast because the existence of a set can be checked instantly — there are no comparisons. It is also special, because the memory needed for the deterministic states of a small input automata (27 states was the limit reached on the best machine used in testing this implementation) is acquired at the beginning of the conversion. This acquisition means that a solution is guaranteed, provided that there is enough space for the deterministic state array, which is generated during subset construction. This is not the case with the other implementations, which dynamically acquire memory to store each deterministic state in the hash table.

6.3.1 Block Power-Bit Vector

One of the problems with the power-set approach is that it requires substantial time to clear a bit vector of size 2^n. Even when very small automata are being converted, clearing takes too much time; other approaches out perform it. We can avoid the clearance time, however, by partitioning the power-set bit vector into equal-sized blocks that are cleared when needed. This is an application of the *laziness principle*: we do not do today what we never need to do tomorrow. To keep track of the blocks that have been cleared, we use another bit vector called the *clearance vector*. This method is called the *block power-bit method*. Each time we access a bit, BIT, in the power-bit vector, we must determine which block, $BLOCK$, it is in. A function $Bfunc$ maps bits into blocks — $BLOCK = Bfunc(BIT)$. We then use $BLOCK$ as an index for the clearance bit vector. If the bit at position $BLOCK$, in the clearance bit vector, is 0, then that block of memory has not yet been accessed and, so it has not been cleared. If the bit is 1, then the block has already been cleared (and used), so we can access the power-bit vector. If we want to access a bit in a block that hasn't yet been cleared, we first clear the block of memory, set the appropriate bit in the clearance bit vector to 1, and then access the power-bit vector. In our implementation, $Bfunc$ returns the top 16 bits of the power-bit index. For example, [0000 0000 0000 1010 0000 0000 0000 0110] is a deterministic state containing nondeterministic states 1,2,17,19. The state is represented by the integer $786444 = (2^1 + 2^2 + 2^{17} + 2^{19})$. We want to set the 786444th bit in the power-bit vector. The top 16 bits of the deterministic state [0000 0000 0000 1010] give the integer 10. Therefore, we check the 10th bit in the clearance vector to determine whether that 64 Kbyte block of memory, containing the deterministic state, has been cleared. In other words, the presence or absence of states 16-31 in the deterministic state uniquely identifies a block of 64 Kbytes. If an input automaton has less than 17 states, then only one 64 Kbyte block of memory is ever used (e.g. $Bfunc(x) = 0$, for all x). The block power-bit method is an improvement over the straight power-set method when input automata are small and have between 18-27 states.

6.4 Virtual Power-Bit Vector

We can use the power-bit method for arbitrary-sized sets, in which case the 64 Kbyte hash table becomes a *virtual power-bit vector.* The integer value given by a deterministic state can be used directly as a hash value by scaling it to provide a value in the range of the hash table. To implement this method we use the first word of the deterministic state as the hash value and scale it to the size of the hash table. Therefore, the first 32 states provide the information used in the hash function. This method is beneficial because the set does not have to be converted to another form to generate a polynomial signature. One disadvantage is that only the first 32 states in each set are involved in the calculation of the hash table index. If the number of states is large, the method may produce poor allocation because the first 32 bits may be the same for many sets. The method could be improved, for a large number of states, if the signature is formed from elements ranging throughout the set. This may appear to contradict the results obtained from the previous polynomial signature experiments (where a smaller polynomial did not help), but because we have sets in bit vector form, generating a polynomial signature would take extra time. Combining this with the fact that we are obtaining signatures immediately suggests that we can obtain a more efficient implementation.

Figure 6.4 shows the results of Virtual Power-set, Power-set, and Polynomial Signature, performing subset construction on automata with 20 states and 10 symbols.

Figure 6.4: Performance comparison of Power-set, Virtual Power-set and Polynomial Signature.

Figure 6.5: Illustration of a bit plane. Shaded row represents a single deterministic state.

6.5 Multiway Merge and Bit Planes

As our experience with subset construction grew, we used more memory. We realized that because subset construction may require many megabytes of memory, and an improved implementation may involve only a few kilobytes of more memory, the extra memory use will not be significant. We now describe how multiway merge can be replaced by a *bit plane merge*. From the multiway merge we receive deterministic states and the symbols which lead to them. Therefore, we can view the multiway merge as providing a set of deterministic states with respect to input symbols. If an input NFA has n states and m symbols, then the multiway merge returns, at each step, at most m deterministic states, each of size at most n. Rather than forming the collection of deterministic states with a multiway merge, the *bit plane merge* takes each individual finger and "empties it" into an m by n bit plane. The (i, j)th entry in the bit plane is 1 if there is a transition to state j with symbol i, from some given set of states. Figure 6.5 illustrates a bit plane. An $m \times n$ bit plane is used to represent deterministic states — the shaded row in Figure 6.5 represents a single deterministic state. For each finger, the necessary state would be marked at the corresponding symbol. Once all fingers have been emptied, the bit plane holds all the deterministic states for that merge. The bit plane is composed of rows of bit vectors which are in fact the deterministic states for each symbol and are already in proper state form. This bit plane merge uses direct indexing to locate the particular state from a particular symbol that needs to be marked — the merge involves no sorting or comparisons. If an input automaton has 1,000 symbols and 1,000 states, then the bit plane has 1 million bits (125Kbytes). This is not much memory when we realize that each of the deterministic states acquired from the bit plane has to be stored in the collection of states, if it is not already there. Or put more simply, the memory required for the bit plane is usually only a fraction of the memory is required to store the deterministic states formed from the bit plane merge.

Each use of the bit plane merge involves clearing the bit plane. It is not known whether this clearing is overly time consuming, but results show that the bit plane merge is superior to all other methods of merging, on all automata except for those of very small size. The bit plane has a check bit for each row. The bit is set once a state has been marked in that row. This check bit shows, upon completion of the fingers, which symbols have states. Only target states from symbols having a non-empty set are returned. These check bits also have to be cleared when a bit plane merge is completed.

Figure 6.6 shows the initialized bit plane used in a subset conversion with an input automaton that has 22 states and 4 symbols. We give, for an instance of the deterministic state loop, a set of fingers, which are then emptied into the bit plane (FINAL BIT PLANE).

Figure 6.7 compares the bit plane approach with INR. The bit plane approach is superior to INR and is even better with large automata. INR is more efficient when the automata are small and the bit plane approach spends too much time working with and clearing the bit plane.

Figure 6.6: Bit Plane Example

Figure 6.7: Bit Plane compared with INR.

Chapter 7

Density of Automata

7.1 Introduction

We have tested the performance of the various implementations of subset construction with randomly generated automata. We tested average execution time, for a fixed number of states and symbols, with different transition densities. The resulting graphs have a single peak and are Poisson-like. The position of the peak, with respect to the density, appears to vary with the number of states, while the amplitude appears to vary with the number of symbols. For the tests, we measured only the execution time of subset construction as we varied the three parameters, but the occurrence of the Poisson-like shape prompted us to gather statistics about the number of generated states and transitions. The number of output states and transitions also give a similar curve. In this chapter we conjecture that, for random automata, the number of states and transitions (and hence the execution time of subset construction) can be predicted with reasonable accuracy. In addition, we will explain the use of this conjecture in the implementation of subset construction.

7.2 State Collapse

Recall that the density of an automaton is the ratio of the number of transitions and the maximum possible number of transitions — sometimes expressed as a percentage. The plots of density against execution time have a Poisson-like shape in Figures 7.1–7.3. The rise in the curve is easily explained. As the number of transitions increases, the work involved in merging and managing sets increases. The fall in the curve is not as obvious. Consider how subset construction works on a completely connected NFA. From the start state we have transitions to all other states (including the start state) on all symbols, so a DFA state R_1 is created that includes all NFA states. The state R_1 is the target state for all transitions on all symbols from the start state of the deterministic automaton, because every state has transitions with all symbols to all states. Now consider R_1 as a source state; the target state for R_1, for any symbol, is again R_1. Subset construction stops here since we have added no states. The resulting DFA has only one state, R_1. Intuitively speaking, we expect that, for automata that are almost fully connected, the final DFA will have very few states and the subset construction will terminate quickly. When different sets produce the same result from the multiway merge, we say that these sets have *collapsed* into one state. The worst case in subset construction (the highest point on the Poisson-like curve) occurs when very few states collapse. We expect that collapsing is more likely to occur as density increases. Now we realize why the graphs are Poisson-shaped — as the density increases, more DFA states are generated until, at some point, this expansion peaks, then state collapse starts to occur, and fewer states are generated.

7.3 Deterministic Density

Density is one measure of the connectivity of an automaton. This measure, however, has the drawback that it is computed relative to the absolute number of transitions in an automaton. Thus, we cannot meaningfully compare measures of density across automata unless they have the same numbers of states and symbols. A 5-state, 5-symbol automaton that is minimally connected has a density of $\frac{4}{125}$, whereas a 10-state, 10-symbol automaton that is minimally connected has a density of $\frac{9}{1000}$. We would like a unit of measure that captures the density of an automaton independent of the number of states and symbols. One such measure is *deterministic density*. We now refer to the old measure of density as absolute density. Deterministic density is the ratio of the number of transitions in a given NFA to the number of transitions of a fully connected deterministic automaton having the same number of states and symbols. Specifically, the deterministic density of an automaton *FA* of n states, m symbols, and p transitions is $\frac{p}{nm}$. For example, if *FA* has 5 states, 5 symbols, and 55 transitions, the deterministic density of *FA* is 2.2, which is 44 percent absolute density. A deterministic density of 2.2 tells us that the automaton has slightly more than twice the number of transitions of a maximally connected DFA with 5 states and 5 symbols. If *FA* is a randomly generated automaton with deterministic density 2.2, we expect it to have 2.2 transitions per symbol per state. Deterministic density

$$dd = \frac{\#transitions}{nm} \tag{7.1}$$

and the absolute density

$$ad = \frac{\#transitions}{n^2 m} \tag{7.2}$$

are related by

$$dd = n \times ad. \tag{7.3}$$

Deterministic density was initially conceived as a way to factor out n and m; one result is that when graphs are plotted against deterministic density, they are automatically scaled. This use of deterministic density makes it easier to compare the graphs.

7.4 Analysis of Deterministic Density

The first testing method logged the execution time of subset construction for randomly generated automata. Now we modify this process so that the number of states and transitions resulting from subset construction for randomly generated automata are also logged. Three graphs, Figures 7.1, 7.2, and 7.3, show the results obtained with randomly generated automata of 15, 20 and 25 states, with 10, 15, 20, and 25 symbols. The number of output states is plotted on the ordinate; deterministic density is plotted on the abscissa. At each density 7 test trials are run on randomly generated automata with a specific number of states and symbols. It is clear that the number of output states is determined by density. We know execution time is very closely tied to the number of output states and transitions, so we can focus our attention on the behavior of these parameters rather than on the execution time.

Examining each graph individually, we see that the graphs, for each specific number of symbols, all peak at the same density and have a similar shape. As the number of symbols increases, the height of the curve increases. In fact, we see that with a small number of input states (Figure 7.1)

Figure 7.1: A plot for 15-state automata.

Figure 7.2: A plot for 20-state automata.

Figure 7.3: A plot for 25-state automata.

the curve peaks at values close to the maximum possible number of output states for a 10-state automaton (1023). Clearly, as the number of input symbols increases with respect to the number of input states, the peak of the curve approaches the maximum attainable number of output states, $2^n - 1$. This property occurs because as we increase the number of symbols while keeping the density and the number of states constant, the number of input transitions increases. For example, a random 10-state, 10-symbol automaton with a deterministic density 1, consists of, on average, 10 transitions from each state, while a 10-state, 20-symbol automaton with deterministic density of 1, will produce, on average, 20 transitions from each state. As there are more transitions from each state, we expect larger numbers of output states in the resulting deterministic automaton.

When comparing the graphs (plotted on absolute density) for different numbers of input states, we discover that the peak of the graph, regardless of the number of symbols, shifts to the left as the number of states increases. Also, the Poisson shape is scaled smaller in the x-axis (the curve becomes narrower). It appears that the location of the peak of the curve is solely a function of the number of input states. That is, it would appear that, for automata with 15 states, the curve will peak at approximately 13 percent absolute density and, similarly, 10 percent for 20 states and 8 percent for 25 states.

The notion of deterministic density assumed greater significance when we discovered that the curves all seemed to peak at the same deterministic density — approximately 2.0. A wide range of test cases show, by experimentation, that randomly generated automata produce predictable results, in terms of the numbers of output states and transitions, as a function of the deterministic density, number of input states, and number of input symbols.

Figure 7.4: Sections of deterministic density.

7.5 The Conjecture

The testing that we have carried out suggests that randomly generated automata exhibit the maximum execution time, and the maximum number of states, at an approximate deterministic density of 2. Most of the area under the curve occurs within 0.5 and 2.5 deterministic density — this is the area in which subset construction is expensive. The smaller tails occur in a range from deterministic densities [0,0.5] and [3.5,max], where subset construction takes time linear in the number of states of the input automaton. We see this illustrated in Figure 7.4.

Conjecture 7.1 *For a given NFA, we can compute the expected numbers of states and transitions in the corresponding DFA, produced by subset construction, from the deterministic density of the NFA. In addition, this functional relationship gives rise to a Poisson-like curve with its peak approximately at a deterministic density of 2.*

It is not yet clear how the number of symbols affects the magnitude of the curve, only that more symbols causes more vertical growth. In our test cases we see that even 10 symbols causes the curve to reach high values at approximately deterministic density 2.

7.6 Implications of the Conjecture

Testing was performed for automata with state sizes of 25 or fewer. When tests were attempted on automata with 30 or more states, the storage capacity of the computer was exceeded with densities near 2. Output automata near a million states and in excess of 10 million transitions were being generated, and the execution time of even the best implementation (bit plane) was taking hours of computing time. In addition, these large automata consumed the available main memory (10-15 megabytes) and, thus, virtual-memory management began to dominate the computing time. These problems show that the conjecture can help us in two important ways: we can decide if a given input automaton is tractable, and if so, we can compute how long we expect subset construction to take.

We can use the conjecture as follows: Given an NFA, compute its deterministic density and assign the automaton to one of the three regions `A`, `B`, or `C` shown in Figure 7.4. If it lies in region `B`, we expect the resulting automaton to have an exponential number of states. Although machines differ, a maximum number of input states can be established as a cut-off for deterministic densities that lie in region `B`. For our tests, we used computers with up to 30 megabytes of main memory and we established a cut-off at 50 states. If subset construction is attempted on an automaton with 50 or more states that has more than 5 symbols, and if the deterministic density is in area `B`, then we almost certainly cannot carry out the subset construction.

Once we decide to proceed with subset conversion, we can estimate, from the conjecture, the time of the conversion, the size of the output and the amount of main memory needed. To do this the curve should be approximated by a polynomial function and scaled and translated according to the deterministic density and number of input symbols. The expected numbers of states and transitions can then be computed from the function. Since the elapsed time is a function of the machine load, one way of relaying information to the user is to calculate the rate at which the conversion is progressing. For example, if we know that the subset construction will generate about 10,000 states, then after we have generated 100 states, we can estimate that approximately 1 percent of subset conversion has taken place. Providing this information will allow a user to estimate the total execution time and to abort subset construction if it seems that it will take too long. By the same method, we can also estimate the main memory usage by considering the amount of main memory used for the fraction of estimated state sets already computed.

Chapter 8

Conclusions

8.1 Additional Comments on Implementations

Originally, both GRAIL and INR stored output transitions in main memory, as did our very early implementations. This became a problem when subset construction was tested on large automata, because too much memory was needed for the transition storage. Output transitions take up a large portion of main memory and are not needed by the algorithm once they are generated. It was clear that the transitions should be directed to secondary storage. By not keeping transitions in main memory, subset construction is successful on much larger automata. INR and GRAIL were both modified so as not to store transitions in order to keep comparisons fair. This modification also resulted in slightly better execution times, because the transitions are not stored and reallocated. If I/O is particularly slow, however, this modification will hinder performance.

8.2 Future Improvements

8.2.1 Memory Allocation

One particular area where our implementations may be inefficient is in memory allocation. Each time a set is stored, an appropriate amount of memory is allocated using `malloc`. Each separate call to `malloc` may involve costly allocation procedures. All we want to do is store a set and receive a pointer to it; clearly, it would be more efficient if we allocated a large section of memory at one time and stored many sets in it. The density conjecture can help in determining the required amount of memory and, thus, helping to reduce the number of calls to `malloc` and `realloc`.

8.2.2 Modification to Bit Plane

In the most deeply nested part of the subset construction algorithm, we access certain input transitions and sort them according to input symbol and target state. By sorting the input transitions and then indexing them prior to subset construction, we save time with each access and take advantage of their sortedness. In the bit plane approach, we avoid costly sorting by simply sweeping through all the fingers and marking bits in the bit plane. It is also possible to take the bit plane method a step further. When we sort and index our input transitions, we convert this sorted transition data into sorted bit vectors of target states for each input symbol. Each finger now points to a set of bit vectors representing all target states from each symbol. Instead of using marking on the bit plane, for each target state on each symbol, we do a logical *or* of all target states with the column of the bit plane that corresponds to the given symbol. Just as with the bit vector representation of sets, the transfer of an entire bit vector to the bit plane will be less efficient if the automaton is sparse.

8.2.3 Complement

If an automaton is very dense, it may be beneficial to work with the complement of the automaton. More specifically, if an automaton is very dense we can sort the input transitions and then generate another sorted set of transitions which is the complement — all the transitions not in the input automaton. The original sorted set of transitions could then be removed from memory. The set of complement transitions is smaller than the sorted set of transitions and, therefore, we have a memory saving. Now instead of clearing all the bits in the bit plane we set all the bits. When the contents of the fingers are accessed we clear the corresponding bit for that target state and symbol on the bit plane. The result will be identical to the old method, except that there is less bit marking on the bit plane, thereby improving the performance of the process. For a fully connected automaton, for example, the sorted set of complement transitions is empty and no transitions are stored. The fingers are null and no bits are cleared on the bit plane. We thereby generate our output states instantly instead of having possibly millions of accesses to the transitions via the fingers. Of course this illustrates the extreme case, but savings in memory and execution time should be apparent in automata with greater than 50 percent absolute density.

8.3 Implementation of Choice

We have analyzed many different ways to implement subset construction, but which implementation is the best? We have constantly improved the execution time of subset construction and eventually arrived at implementations which utilized bit vectors for sets and merging. With bit vectors we obtain a memory savings and faster transfer rates if the sets are dense enough that the bit vector representation is smaller than the integer set representation. The bit plane merge also improved the execution time. If the sets are not sufficiently dense, then excessive amounts of memory and bit plane transfer may occur. Specifically, for our implementation, if the average set size is sparser than $\frac{n}{32}$, we would waste memory and have excess transfers. The use of a bit plane and bit vector sort, in practice, still allow even sparse automata to be converted efficiently. If sets are very sparse, then the integer approach is more appropriate. On smaller machines with little memory, the integer approach might be the only solution as the bit vector implementation requires too much memory.

Clearly there is no single perfect implementation. The best implementation depends upon the amount of memory available, the density of the automaton, and the number of input states. The pseudo-code in Figure 8.1 is a recommendation of how the choice can be made.

The recommended implementation will, therefore, glue together the best data structures and routines to choose a subset construction method which is best suited for a particular input automaton. The recommendation for a fixed implementation would probably be best assembled as `BIT-PILE SET REPRESENTATION`, `HASH TABLE COLLECTION-OF-STATES`, `BIT-PLANE MERGE` and `NORMAL TRANSITION SORT`. This fixed implementation will work well on a majority of automata, but will be somewhat inefficient on very sparse input automata.

```
F is nondeterministic automaton with n input states.
DENSITY(F) is a function that return density of automaton F.
MIN_DENSITY and MAX_DENSITY are constants.
FOR COLLECTION OF SETS
   IF ( $2^n$ bits can be allocated AND sufficient
            memory for storing state list )
      USE POWER-SET BIT VECTOR METHOD
   ELSE USE HASH TABLE
FOR SET REPRESENTATION
   IF ( DENSE(F) $<$ MIN_DENSITY ) USE INT-PILE
   ELSE USE BIT-PILE
FOR MERGE
   IF ( memory for BIT-PLANE cannot be allocated) USE HEAP SORT METHOD
   ELSE USE BIT-PLANE
FOR NORMAL/COMPLEMENT
   IF ( BIT-PLANE )
         If ( DENSITY(F) $>$ MAX_DENSITY) USE COMPLEMENT METHOD
   ELSE USE NORMAL
```

Figure 8.1: Pseudo-code for choosing an implementation.

8.4 Future Work

In this thesis we have demonstrated ways that subset construction can be efficiently implemented and we have discovered that there is a way of predicting the size of the automaton produced by subset construction. Future research can continue in the area of deterministic density. Specifically, it would be beneficial to prove the conjecture of deterministic density — that subset construction produces a maximum number of output states and transitions at deterministic density of approximately 2, for random automata. We have based our conjecture on automata which are 'random,' but in practice we do not yet know what typical automata look like. We know that automata can expand exponentially in size as a result of subset construction and we realize that, for even a small number of input states, subset construction may require massive resources. It is therefore important to analyze an automaton prior to subset construction and discover how it might behave. By further investigating the effects of biased automata we may be able to develop a conjecture which would predict the expected result size based on the distribution of the transitions and their symbols. Ultimately we would like to take *any* automaton and say, with accuracy, how it will behave under subset construction.

Bibliography

[1] J.H.Johnson. *INR: A Program for Computing Finite Automata.* Unpublished manuscript, University of Waterloo, February, 1987.

[2] Donald E. Knuth. *The Art Of Computer Programming Vol. 3 / Sorting and Searching*, pages 142–143, 145–147, 153–158, 209–212, 252. Addison-Wesley Publishing Company, Reading, Massachasetts, 1973.

[3] Ernst Leiss. *REGPACK: An Interactive Package for Regular Languages & Finite Automata.* University of Waterloo, Dept. of Computer Science, Technical Report, CS-77-32, 1977.

[4] A. R. Meyer and M. J. Fischer. Economy of description by automata, grammers, and formal systems, pages 188–191. *Conference Record 1971 Twelfth Annual Symposium on Switching and Automata Theory*, IEEE Computer Society, 1971.

[5] F. R. Moore. On the bounds for state-set size in the proofs of equivalance between deterministic, nondeterministic, and two-way finite automata, pages 1211–1214. *IEEE Transactions on Computers 20*, 1971.

[6] M.O. Rabin and D. Scott. Finite automata and their decision problems, pages 114–125. *IBM Journal of Research and Development 3*, 1959.

[7] Derick Wood. *Theory of Computation*, pages 98–125. John Wiley and Sons, New York, 1987.