Comparison of Deterministic Automata Representations

Abrie Greeff

February 24, 2006

Contents

1 Introduction

The purpose of this project and report is to compare different methods of representing DFAs[1] in a computer environment. There are five different methods of representing DFAs. Only three of these methods were considered for this project. These methods will be discussed in the chapters that follow.

The implementation of these methods was done in Java, in a Linux environment. The DFAs were constructed from text files that confirms to the Grail input format. Each of the three considered methods read these files and loaded their respective structures according to the input. Through the user interface of the program the user is able to select through which method he or she wants to test text input on the DFA.

[1] Deterministic Finite Automata

2 Transition Matrices

The transition matrix is the simplest method of implementing the storing of transition functions. I implemented it with a two-dimensional array. The x-axis contains all the possible states of the DFA and the y-axis contains all the possible characters of the DFA's alphabet.

There are two problems with this implementation that limits it's usefulness. The first problem is that the input file must be processed twice because the alphabet and number of states needs to be known before the matrix can be generated. After this has been done the file needs to be parsed again so that all the necessary transitions can be added into the matrix.

The second problem is space constraints. When the size of the alphabet and the number of states are large, a big matrix is needed. The required size of the matrix is always the size of the alphabet multiplied by the number of states. When these big sizes occur a lot of space is wasted because very few states have transitions on every character of the alphabet.

The advantage of the transition matrix is that it's easy to implement and can be visualized quite easily. Search time on the matrix is very fast because it is indexed and thus the delay when searching is *O(1)*.

3 Adjacency Lists

Adjacency lists improves on transition matrices on the usage of space but increases search time. I implemented it with an array of linked lists. The array is indexed with all the states the DFA contains. The linked list then contains all the transitions that occur from this indexed state.

The problem with this approach is that the input file needs to be parsed two times. The first parse needs to count all the states and the second constructs the linked lists. The space needed to store the DFA is very small because space is only needed for all the transitions. The drawback to this approach is the delay in searching for a specific transition. The indexing of the state is *O(1)* but the searching of the transition in this linked list can be very slow because the list needs to be traversed until the correct transition is found. The worst case thus is *O(d)*, where d is number of characters in the alphabet of the DFA.

4 Transition Lists

Transition lists are the most effective of the approaches considered in this project. I implemented transition lists with a hashing function. A composite string consisting of the state and the character the transition occurs on was used to compute the key for indexing. An object containing these two properties and the state the transition occurs to is then stored in the hashtable.

This approach means the delay in seeking for a transition is minimal because the key is already known. The space used by the hash table depends on the effectiveness of the hashing function. The space usage won't be as effective as the Adjacency list but improves vastly on the usage of a transition matrix.

5 Conclusion

Every one of the methods that was considered in this project has it's advantages and disadvantages. The transition matrix isn't very effective for use in a computer program and is much better suited for non-computer usage. The adjacency list was the most difficult to implement and has the best space usage. The time complexity needed to search for a transition may be the biggest factor that counts against it. The transition list was the easiest function to implement and is also the most effective method of the three. It has a good combination of space usage and search time and is therefore my choice of method.