Dictionary Matching Automata
The Aho-Corasick Algorithm

Carl Crous
14059967
BScHons

March 10, 2006

Description

Searching through documents with a dictionary (a list of words) is a common task in any computer environment. It is important that you have facilities to do this efficiently and accurately. One may think this is a simple task of trying to match each word you are looking for at all locations in the documents. This method, known as brute force (Goodrich & Tamassia [1] pp. 545), is extremely inefficient since you are throwing away lots of information each time a word doesn't match a specific position in the document. The time complexity of this method is $O(n \times m \times w)$ where n is the amount of letters in the document, w is the amount of words you are searching for and m is the longest one. This time complexity may be acceptable if you are searching for only one relatively short word in a small document.

To illustrate this point, say you are looking for the word "animal" in a document at a position which contains "animation". You will match the word until "anima", get a mismatch, advance a character and proceed to search at "nimation". This just discarded a lot of information. The better choice would have been to proceed to search from "ation" with the first character "a" already matched.

The Aho-Corasick Algorithm uses all the information obtained and will only look at each character of a document once. It achieves this by using a tri-like deterministic finite automata (DFA) with an appropriate failure function. Searching using this algorithm will have a time complexity of $O(n)$ where n is the amount of letters in the document. This is a great improvement from the previous method, and allows you to search for as many words of any length as you like without an impact on searching performance. Another advantage is that once the DFA and failure function have been constructed, you can use them on any amount of different documents without having to reconstruct them. The algorithms used to construct and use these can be found in the following section.

Algorithms

The Aho-Corasick algorithm needs a tri-like DFA and a failure function. The construction of these is described below.

DFA

The required DFA needs to accept all the words in the dictionary and nothing else. This can be done by using the simple algorithm given in figure 1 which is written in a java-style pseudo code. A few conceptual classes and methods are used and their definitions are as follows.

- **Dictionary** is a class which contains a list of words as Strings.
- **Dictionary.remove()** removes a word from the dictionary and returns it.
- **State** is a class which represents a State.
- **State.isValid()** returns true if the state is valid otherwise it returns false.
- **DFA** is a class which represents a DFA and stores all its states, transitions and final states.
- **DFA.newState()** creates a new state in the DFA and returns it.
- **DFA.setStartState(State s)** sets the DFA's starting state to the one specified.
- **DFA.getStartState()** returns the DFA's starting state.
- **DFA.getTransition(State s, char t)** returns the destination state for the transition starting at state s on alphabet symbol t or an invalid state if the transition doesn't exist.

- **DFA.addTransition(State s, char t, State d)** Adds a transition to the DFA starting at state s on alphabet symbol t with state d as the destination state.
- **DFA.addAcceptState(State s)** Sets the given state to be an accept state for the DFA.

```
/*
 * INPUT: d - the dictionary of words.
 * OUTPUT: The DFA which accepts all the words in the dictionary d.
 */
DFA dfaConstruction(Dictionary d) {
    DFA dfa = new DFA();
    String w;
    State state, nextState;

    state = dfa.newState();
    dfa.setStartState(state);
    while ((w = d.remove()) != null) {
        state = dfa.getStartState();
        for (int i = 0; i < w.length(); i++) {
            nextState = dfa.getTransition(state, w.charAt(i))
            if (! nextState.isValid()) {
                nextState = dfa.newState();
                dfa.addTransition(state, w.charAt(i), nextState);
            }
            state = nextState;
        }
        dfa.addAcceptState(state);
    }
    return dfa;
}
```

Figure 1: DFA Construction Algorithm

The algorithm basically follows the states of the DFA along each character of a word in the dictionary, creating states and transitions where they don't exist and setting the state at the end of the word to an accept state. This is repeated for each word in the dictionary.

The process has a time complexity of $O(n \times m)$ where n is the amount of words in the dictionary and m is the length of the longest word. The time complexity can also be written as $O(l)$ where l is the sum of all the words characters. This means that if the dictionary was read in from a file, then the running time of this algorithm will be linear and in proportion with the size of the file.

Failure Function

The failure function defined below is constructed directly from the DFA.

Given a state s which can be reached by traversing the DFA using the word w. The failure function's value for s is the state which is reached by longest suffix of w, i.e., it is a prefix for another word in the DFA.

The method for constructing the failure function is given in figure 2. The following additional conceptual methods and classes were used:

- **Queue** is a simple first-in-first-out queue class.
- **Queue.add(State s)** adds a state to the end of the queue.

- **Queue.isEmpty()** returns true if the queue is empty otherwise returns false.
- **Queue.remove()** removes a state from the front of the queue and returns it.
- **FailureFunction** is a class which stores the failure function.
- **FailureFunction.setFailure(State s, State f)** sets the failure function to go to state f if state s fails.
- **FailureFunction.getFailure(State s)** returns where failure function goes if state s fails.
- **DFA.getAlphabetLength()** returns the amount of alphabet symbols.
- **DFA.getAlphabetSymbol(int i)** returns the alphabet symbol at index i.
- **State.isAcceptState()** returns true if the state is an accept state otherwise returns false.

```
/*
 * INPUT: dfa - the DFA to construct the failure function for.
 * OUTPUT: The failure function.
 */
FailureFunction failureConstruction(DFA dfa) {
    FailureFunction f = new FailureFunction();
    Queue q = new Queue();
    State state, nextState, s;
    char ch;

    q.add(dfa.getStartState());
    f.setFailure(dfa.getStartState(), null);
    while (! q.isEmpty()) {
        state = q.remove();
        for (i = 0; i < dfa.getAlphabetLength(); i++) {
            ch = dfa.getAlphabetSymbol(i);
            nextState = dfa.getTransition(state, ch);
            if (nextState.isValid()) {
                s = f.getFailure(state);
                while ((s != null) && ! dfa.getTransition(s,ch).isValid()) {
                    s = f.getFailure(s);
                }
                if (s != null) {
                    f.setFailure(nextState, dfa.getTransition(s,ch));
                } else {
                    f.setFailure(nextState, dfa.getStartState());
                }
                if (f.getFailure(nextState).isAcceptState()) {
                    dfa.addAcceptState(nextState);
                }
                q.add(nextState);
            }
        }
    }
```

Figure 2: Failure Function Construction Algorithm

This algorithm runs a breadth first search on the DFA traversing to each node and setting each failure function. The breadth first search has the advantage over a depth first search because you are able to use previous information, such as parent node's failure functions. This is possible because the failure function will always point to a node with a lesser depth (fewer parents).

The way the algorithm calculates the failure function, is it visits each state in the DFA and calculates the failure function of each of its children (the states it has transitions to). This calculation is done by following its own failure function to another state and seeing if that state has a transition on the same alphabet symbol the child is on. If it has one, then the destination of that transition becomes the child's failure state. If not, the failure function is followed again and this process is repeated until either a transition is found, or the starting state is reached. If the start state is reached then it becomes the child's failure state.

Another thing you may notice is that if a state's failure function points to an accept state, then that state becomes an accept state. The reason for this is that if one word is a sub-string of another word in the dictionary, then it will not be recognized in certain cases without this feature. For example, if "heard" and "ear" are in the dictionary and "hear" is in the document, then a match for "ear" must still be found.

The time complexity of this algorithm is $O(n \times m \times w)$ where n is the amount of states in the DFA, m is the amount of alphabet symbols and w is the length of the longest word in the dictionary. This conclusion is reached by analyzing the loops in the algorithm. The outer while loop essentially traverses through each state ($O(n)$). Next is the for loop which checks each transition symbol ($O(m)$). There is another while loop within the for loop, which can at most be executed w times.

However, the innermost while loop will only have this running time for a state which the dictionary contains every suffix of it. This makes it extremely unlikely for most dictionaries, and experimentation reveals that in large dictionaries (over 10000 words) this loop is executed at most ten times for each transition and less than once on average. This shows us that the true running time is closer to $O(n \times m)$ for dictionaries where the words don't have an excessive amount of suffixes in the it.

An alternative implementation may be to do a depth first search, constructing the words that each state represents then finding the longest suffix of those words in the DFA and use that as the failure function. This method easy to understand since it sounds more like the definition but its running time is $O(n \times (m + w^2))$, where n is the amount of states, m is the amount of alphabet symbols and w is the length of the longest word in the dictionary. Traversing through each state and building up the words has a running time of $O(n \times m)$. Finding the longest suffix of a single word in the dictionary has a running time of $O(\frac{1}{2}w(w-1)) = O(w^2)$ because you have to take a character off of the word, test it, and repeat the process throughout the length of the word. Finding the longest suffix of n words therefor has a running time of $O(n \times w^2)$. These results make up the total running.

It is clear that the breadth first search method has an advantage over the depth first search if used on large dictionaries with large alphabets.

Searching

Searching through a document using the Aho-Corasick algorithm is done by traversing the DFA and using the failure function when ever a transition fails. The algorithm in figure 3 shows how this is done exactly. The following conceptual methods and classes were used:

- **Document** a text document class.
- **Document.eof()** returns true if the end of the document has been reached otherwise false.
- **Document.getChar()** returns the next character of the document.
- **Document.getPosition()** returns the current position in the document.
- **Results** a class which houses all the results.

- **Results.add(int pos)** add a result at the given position.

```
/*
 * INPUT: dict - dictionary of words to search for.
 * doc - the document to search.
 * OUTPUT: The results found
 */
Results ahoCorasickSearch(Dictionary dict, Document doc) {
    DFA dfa = dfaConstruction(dict);
    FailureFunction f = failureConstruction(dfa);
    State state, nextState;
    char ch;
    Results r = new Results();

    state = dfa.getStartState();
    while (! doc.eof()) {
        ch = doc.getChar();
        nextState = dfa.getTransition(state,ch);
        if (! nextState.isValid()) {
            nextState = f.getFailure(state);
            while ((nextState ! = null) && ! dfa.getTransition(nextState,ch).isValid()) {
                nextState = f.getFailure(nextState);
            }
            if (nextState ! = null) {
                nextState = dfa.getTransition(s,ch);
            } else {
                nextState = dfa.getStartState();
            }
        }
        if (nextState.isAcceptState()) {
            r.add(doc.getPosition());
        }
        state = nextState;
    }
    return r;
}
```

Figure 3: Aho-Corasick Search Algorithm

Implementation

The Aho-Corasick algorithm was to be implemented for use as a document searcher. Large documents and dictionaries were to be used and run in a reasonable amount of time.

Data Representation

With the above factors in mind the data representation method had to be chosen carefully. Since each character in the document causes a transition, the lookup time for each transition is extremely important. A transition list, where all the transitions are stored in a hash table, was used since it has a constant lookup time. The hash function had to be well chosen as to avoid collisions

and preserve this running time. A cyclic shift hash code (Goodrich & Tamassia [1] pp. 364) was implemented because it does a good job of avoiding collisions provided the amount you shift by is chosen carefully.

A test was run to choose the optimal shift number. The hash table was populated for various values of the shifting number, counting the amount of collisions in the process. For each value of the shifting number, 100000 random transitions were inserted with at most 10000 states and 26 alphabet symbols. The data for this test can be found in table 1. Since least amount of collisions occur when shifting 7 bits, that value was used for the cyclic shift hash code.

Shift	**Total Collisions**	**Maximum Collisions**
0	99821	3457
1	99479	679
2	96258	72
3	63336	21
4	50788	12
5	47886	13
6	77921	23
7	**36461**	**9**
8	65556	14
9	38828	9
10	66807	21
11	92885	90
12	85660	31
13	58687	16
14	70626	18
15	93139	52

Table 1: Hash Function Test

The hash table was implemented to be resizable, but the initial size it still had to be chosen to avoid rehashing. Assuming some relationship exists between the amount of words in the dictionary and the amount of transitions in the DFA constructed from those words. To calculate this relationship, various dictionaries were loaded and the amount of words and transitions were recorded and can be seen in table 2. A linear function was fitted to the data which produced the equation $Transitions = 2.5577 \times Words$. This equation was implemented to approximate the amount of transitions there would be for a certain amount of words.

Words	**States**
164	779
815	3007
3906	9180
4960	10889
13044	35733
27607	69900

Table 2: Hash Table Size Test

Results

An important part of searching is how the results are displayed to the user. It isn't going to be useful to tell the user that a result was found at character 5930 of the file. Ideally, the results should provide the absolute position, line number, position on the line and the matching text, with all the values relative to the start of the match. The algorithms given in the previous section

only provide the position of the end of a match. To get all the needed information, the algorithms have to be altered a bit.

First of all, the prefix length of each state (the length of the word used to get to that state) needs to be known. These prefix lengths can be calculated easily while calculating the failure function.

Secondly, when searching, the current matched word needs to be maintained in order to be able to get the matched text. On any failure to a new state, this word must be shortened by removing characters at the beginning of it to be the same length as the new state's prefix length.

Thirdly, no additional final states may be introduced in the failure function calculation. These final states will be simulated in the searching method. When we search for a word, we do it as before, but if we reach a state which has a failure to a final state, we must add a result for the failure state.

A graphical user interface was implemented where the user can select the dictionary and documents using a file browser. When a dictionary is loaded, its DFA and failure function are constructed. These will then be used on any document that is chosen. When a document is loaded, a portion of it is displayed in a scrollable text pane. When the user scrolls to a different part of the document, that part is then loaded. This allows for arbitrarily large documents.

Once both the dictionary and document are chosen, the search is performed and the results are displayed in a list. The user has the opportunity to select any result and the document will scroll to that position and highlight the matching text.

Testing

The implementation was tested in two phases to ensure correctness, completeness and efficiency.

Initial Testing

Before the actual testing began, a small test dictionary was created which produced the following:

- failure to a state other than the start state.
- failure from a final state to a state other than the start state.
- a word which is a substring of another word, i.e., a failure to an accept state must exist.
- a word with a suffix which is a prefix for another word with the suffix/prefix length longer than one character.

These conditions were chosen to test the program for most cases. The resulting dictionary that was chosen contained the words "arrows", "row", "sun" and "under". The DFA for this dictionary was constructed by hand and is given in figure 4 on page 9. The failure function was also calculated by hand and can be found in figure 5 on page 9. The following is the test document which was created and all the matches were found and marked by hand:

```
<- indicates a match
[Actual Words]
arrows  <- x2
row <-
sun <-
under <-
[All Prefixes]
a
ar
arr
arro
```

```
arrow <-
r
ro
s
su
u
un
und
unde
[One Word]
arrowsunderows <- x5
```

With the correct results known, the testing became a simple task of verifying if the DFA and failure function were constructed correctly, and if searching gave the correct results. When an error was found, it was easy to find since the example was well known. A few more small test cases were constructed and used in the same way. One of which included chained failures.

Large File Testing

Now that the program was working correctly, its performance was tested on large dictionaries and documents. Large word lists which varied in lengths from 100 to 100000 words were obtained. For large documents, free online novels were used ranging in size from 50KB to 2.3MB. These dictionaries and documents were loaded and timing tests were run.

Parts of the program were timed so that a profile could be obtained for which parts were taking up most of the time. This helped in removing many unnecessary commands which took up most of the execution time. The profiling was complete when the main time issues could not be reduced any more.

Once this was done, the DFA and failure function for a dictionary with 69900 words was constructed in 2.8 seconds, and a 2.3MB document was searched in 9.7 seconds obtaining 698756 results. This test was done without the graphical user interface which ran out of memory while trying to list all the results.

Conclusion

Or though the Aho-Corasick algorithm is a great improvement on the brute force method, careful consideration needs to be taken when implementing it for large cases. If you don't have a constant lookup time for your transitions, the performance of the algorithm will be severely impacted. If you don't ensure your implementation is optimized then large files may bring the program to a stand still.

However if you do implement the algorithm correctly and efficiently, documents can be searched in an instant, just as quickly as loading them in a text editor. This great speed obtained from the fact that each character in the document is only looked at once.

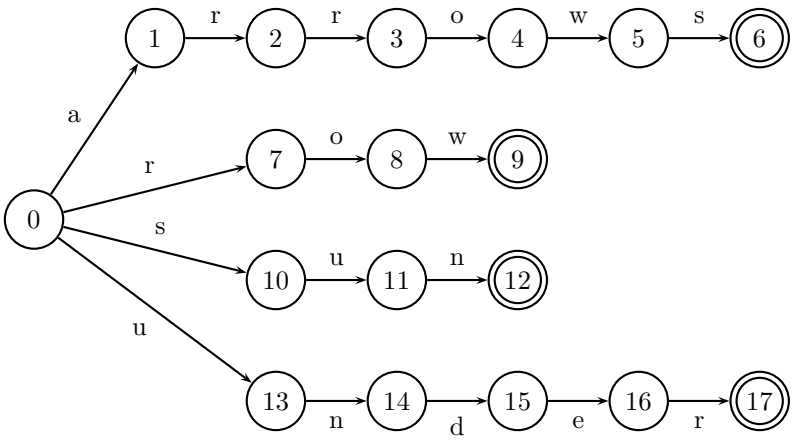

Figure 4:

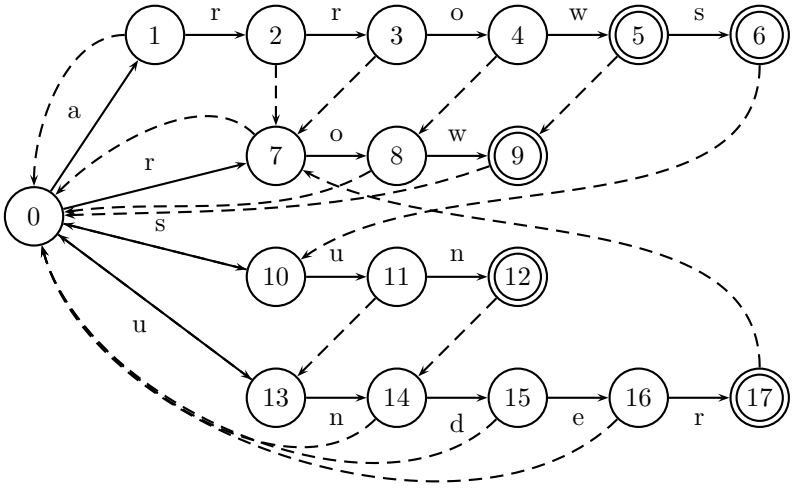

Figure 5:

Bibliography

[1] M.T. Goodrich, R. Tamassia, *Data Structures and Algorithms in Java*, Joh Wiley & Sons, Inc., California, United States of America, pp. 364, 545