UNBC Laboratory Assignment
Computer Science 101—Winter 2018

Searching Machines

Purpose

To practise systematizing code; preparation for interfaces.

Due Date

The completed lab assignment is due Friday, 2018-02-09 *by the beginning of lecture*.

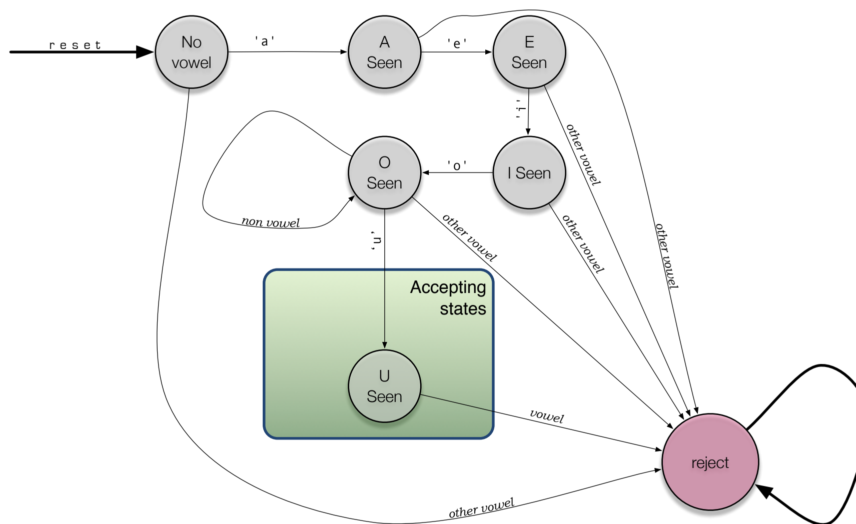

Figure 1: A Finite State Machine

Finite State Machines

Finite state machines are a theoretical machine studied in CPSC 242 and CPSC 340. They are frequently useful for classifying and manipulating strings. For instance when you ask a text editor to search for a pattern in a large file, it probably builds a finite state machine to do so.

© 2018 David Casperson. This document may be freely copied, provided that this copyright notice is preserved.

The finite state machine shown in Figure 1 on the preceding page is designed to find "abstemious" words, that is, words where each vowel ('a', 'e', 'i', 'o', or 'u') occurs exactly once and in order.

A finite state machine has a certain fixed number of states (circles in Figure 1). When the number of states is quite small, it makes a lot of sense to represent the state type by an **enum**, for instance,

```
public enum State = { NoVowel, ASeen, /* ... */ USeen, REJECT ; }
```

A finite state machine also has a special start state. We can code this as[1]

class Machine

```
1 public class Machine {
2     private State currentState ;
3
4     public void reset() { currentState = State.NoVowel ; }
5
6     public Machine() { reset() ; /* .... */ }
```

The arrows between states in a state machine diagram indicate what happens when we are in a particular state and we read a particular character. In Figure 1, only some of the lines are shown, as non-vowels leave the state unchanged.

Starting from the start state, the machine accepts characters one by one. Each time it accepts a character, its state may change based on the transition lines. For the machine shown in Figure 1 we can code this as

class Machine

```
10     public void acceptChar(char c) {
11         switch(c) {
12         /* ... other cases */
13         case 'e':
14             currentState
15               = (currentState==State.ASeen) ? State.ESeen : State.REJECT ;
16             break ;
17             /* ... other cases, default */
18         }
19     }
```

Some of the states are *accepting states*. If the machine is in an accepting state, then it accepts the word that got it into that state. Note that it is possible to go from an accepting state to a non-accepting state. For the machine shown in Figure 1 we can code this as

class Machine

```
8     public boolean isAccepting() { return (currentState==State.USeen) ; }
```

[1]Note that the coding style used in this lab handout *is not my preferred style*. In the interest of making the code compact, I have used much less vertical white-space than I typically do.

Once we have a machine (say as a class variable somewhere) it is quite easy to write a method to determine if a word is acceptable:

class Main
```
4      public /* static */ boolean isAbstemious(String s) {
5        machine.reset() ;
6        for (char c: s.toCharArray()) { machine.acceptChar(c) ; }
7        return machine.isAccepting() ;
8      }
9  }
```

⇒ Steal code from laboratory assignment 2, and write a complete program that finds all of the "abstemious" words in a word list file. Use separate files for `State.java` and `Machine.java`.

⇒ Use your program to find all of the "abstemious" words in `/usr/share/dict/web2`, and print them to a file `facetious.txt`.

Larger Numbers of States

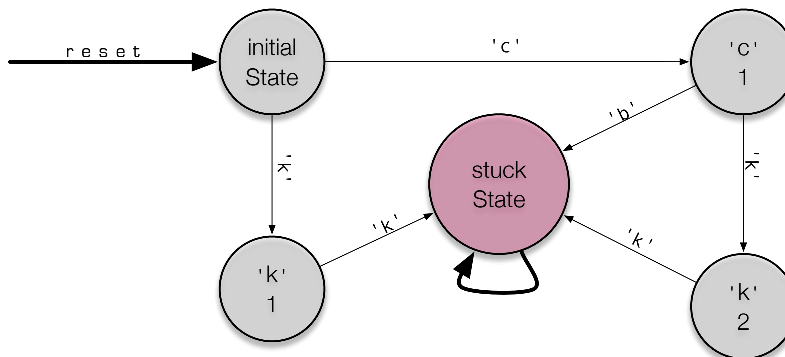

Figure 2: A Few of the States and Transitions for "Degenerative" words

Suppose that we want to find "degenerative" words, that is words that contain six or more consonants, all in increasing order. For our purposes, consonants are letters that are neither vowels, nor '`y`'. As above, a vowel is any one of '`a`', '`e`', '`i`', '`o`', or '`u`'.

We can also create a finite state machine for this problem, but the number of states is quite large (minimally $7 \times 20 + 2$, see Figure 2). Here is makes a great deal of sense to encode states as a class (`State2`).

The state of State2 can be encoded as

class State2
```
2      private boolean inInit ;
3      private boolean amStuck ;
4      private int consonantCount;
5      private char lastConsonant;
```

It's helpful to name some special states:

class State2
```
7      public static State2 initialState = new State2(true,false,0,'@') ;
8      public static State2 stuckState   = new State2(false,true,0,'@') ;
```

where somewhere there is a 4-argument constructor that sets the private variables to its arguments.

Thinking ahead to creating a machine, we add helper methods to our class:

class State2
```
13     public State2 nextState(char c) {
14         if (amStuck || ! isConsonant(c))    return this ;
15         else if (inInit || c>lastConsonant)
16                       return new State2(false,false,1+consonantCount,c) ;
17         else          return stuckState ;
18         }
```

and

```
20     public boolean isAcceptingState()
21         { return (!amStuck && consonantCount>=6) ; }
```

and a helper method (*needed in* nextState)

```
10     private static boolean isConsonant(char c)
11         { return ("bcdfghjklmnpqrstvwxz".indexOf(c,0)>-1) ; }
```

⇒ Complete the coding of the State2.

⇒ Now create a Machine2 class that has the same public methods as the Machine class, but which uses a State2 member variable, and which isAccepting() of "degenerative" words.

⇒ Following the previous pattern, write a complete program that finds all of the "degenerative" words in a word list file.

⇒ Use your program to find all of the "degenerative" words in /usr/share/dict/web2, and print them to a file demipriest.txt.

Lab 2 Revisited

Suppose that we try to build a finite state machine for Lab 2. One way to encode state is as

```
class State3
2     private int pairCount;
3     private int maxPairCount;
4     private int lastCharCount;
5     private char lastChar;
```

However, there is something tricky about state transitions here. After we have read "`ggoobookkee`" the state needs to be {`pairCount` = 2, `maxPairCount`= 2, `lastCharCount`= 2, `lastChar`=`'c'`}. That is, we cannot conclude that "...`ee`" is a pair *until we see the following character*.

To accommodate this, we probably want to change the main program word test to send a spurious extra character, something like

```
7     public boolean isBookkeeping(String s) {
8       machine.reset() ;
9       for (char c: s.toCharArray()) { machine.acceptChar(c) ; }
10      machine.acceptChar('\u1f63b') /* a smiling cat */;
11      return machine.isAccepting() ;
12    }
```

⇒ [Bonus] Recode the program from Lab assignment 2 to use a state and machine class as suggested above, and test it on Finnish words, looking for 4 or more consecutive pairs.