Looking for patterns with regular expressions that make mistakes

Saad Mneimneh
Computer Science
Hunter College of CUNY

General description

We implemented a basic regular expression matching algorithm based on NFAs for one of the homework assignments. The program for that assignment identifies lines that contain a pattern provided by a regular expression. In this project, we would like to add the following features:

- Allow the matching to make up to k mistakes, where k is a parameter provided to the program. The definition of a mistake will be provided shortly.
- Instead of simply reporting a line that contains the regular expression pattern (possibly with up to k mistakes), the program should exactly highlight the beginning and end of the occurrence of the pattern in the line (possibly multiple and overlapping occurrences exist).
- When the program is reading from the terminal, the ending positions of those occurrences must be highlighted in real time (this will require to use non-canonical mode and suppress echoing on the terminal for the program to display the appropriate output). Upon pressing return, the program waits for a new input.

How to model mistakes

A mistake is either a character mismatch or the absence of a character or the addition of a character. An example of each appears below, assuming the regular expression is 'abc'.

- mismatch: axc is a match with 1 mistake, x and b don't match
- absence: ac is a match with 1 mistake, b is missing
- addition: abdc is a match with 1 mistake, d is added

Each of the occurrences above can be interpreted as a match with more than 1 mistake; for instance, ac represents a mismatch (c and b), and an absence of c.

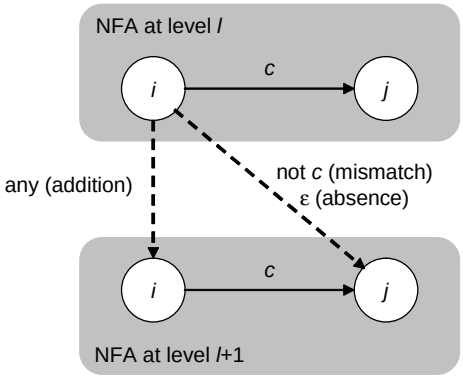
```
ac-
||
abc
```

But we take the number of mistakes to be the smallest possible. For the same example:

```
a-c
| |
abc
```

Modifying the NFA

Mistakes will be accounted for by creating $k+1$ identical NFAs. Each NFA can be thought of as being at some level $l = 0 \ldots k$ (the level indicates the number of mistakes made so far). For a given NFA, a state i transitions to a state j upon reading a character (in addition to ϵ transitions for non-determinism). These transitions stay the same within each of the created identical NFAs. Transitions across NFAs are added and exist only from an NFA at level l to an NFA at level $l+1$. The following figure illustrates the three additional transitions required for each state, note that one of them is an ϵ transition.



The pattern is accepted whenever any of the $k+1$ accept states is reached.

Mapping identical states across levels

One particular challenge is to identify, for state i at level l, state i at level $l+1$. Since all NFAs are created identical **but independently**, this may be difficult. One suggestion is to provide a function to map state i at level l to state i at level $l+1$ by using an extra pointer. The pointer is initially set to NULL. The mapping proceeds by visiting the states of the two NFAs in an identical way, starting from the start state of each, and setting the extra pointer appropriately. Here's an example motivated by the homework assignment.

```
void mapNFA(State * s, State *t) {
  if (!s || s->mapped)
    return;
  s->mapped=true;
  map(s, t);
  mapNFA(s->next, t->next);
  mapNFA(s->alt, t->alt);
}
```

Finding occurrences

The actual occurrence of a pattern can be found as follows. First, the regular expression pattern E is modified by adding $!*$ to become $!*.(E)$. In this way, when an accept state is reached, the ending position of the occurrence (or multiple occurrences) is immediately determined to be the character just read, say the j^{th} character. To determine the starting position, we search the reverse string starting at the j^{th} character, for the regular expression $\bar{E}$, where $\bar{E}$ is the reverse of E. This identifies all starting positions that correspond to the previously identified end.

Further detail

Further detail on the project and implementation can be obtained through discussion between the student and the professor.