George Miller

Natural Language Processing – Final Project Report

Regular Expressions

We learned in class that any regular expression can be made into a finite state machine. This machine consists of states that represent matchable characters at any given point in time. States are connected to each other with edges, and there can be multiple edges coming to and from a state. Some states are marked as start and end states, where any valid instance of the regular expression would be able to traverse the graph from a start state to an end state. Based on this knowledge given in class, I was able to create a working regular expression compiler and parser.

A big picture view of my system consists of three major parts, first being the syntax checker. The syntax checker takes a regular expression as input and returns a number between -1 and the length of the given regular expression. A return value of -1 means that the regular expression was valid. Any other return value means that there was an error at the given index. For example, if I input the regular expression “A|B|C|”, the syntax checker would output 5, since this regular expression is invalid, it has an extra pipe character at position 5.

The second part of my system is the graph creator. This part inputs a valid (must be valid or else the graph creator has undefined behavior) regular expression, and outputs a collection of interconnected nodes (aka a graph) that fully represents a finite state machine as described in the first paragraph of this report. The general method behind this part is to parse the regular expression from start to finish, creating the graph as we parse. The graph is created in chunks, where a chunk is defined as a single character, a range of characters ([a-h] for example), or a regular expression contained in parenthesis. Since each special character has very different effects on the graph, the parsing algorithm comes down to a large switch case, where special behavior is defined for each case.

The last part of my system is the search program. This program has two inputs: a valid graph

(created in the previous step) and a piece of text to be searched for instances of the regular expression. It then outputs all instances of the regular expression, along with their position in the given piece of text.

On a grand scale, each step relies on the step before it. The graph creator can't function properly unless the given regular expression passes the syntax checker. The searcher won't work properly if the given graph contains invalid nodes, or the graph is simply an incorrect representation of the regular expression. In summary, my program uses the Chain of Responsibility pattern to distribute the work of the program among the different parts of the system. For the system to be run, the user simply has to put his/her inputs through the correct phases, and the output of the searcher will be correct.

I will now go into detail about how each part works, and give some improvements that would make the system better. The syntax checker is divided into four main cases of where errors can occur in the regular expression. Just like the syntax checker, each case is implemented such that it returns -1 if the given regular expression passes the case, and returns a positive number less than the length of the regular expression if the regular expression was invalid, notifying where the error occurred. Thus, the syntax checker is simply a vehicle for these four main cases, and doesn't do much on it's own. The first case makes sure each character in the regular expression is valid. Valid characters are: | [ ] { } ( ) + * ? - , \ . and alphanumerics. If any character (that isn't immediately after an escape character) isn't in this list of characters, then it is a bad character, and this case fails. The second case ensures that all open parenthesis (or curly braces or square brackets) have matching closing character. This check is implemented as a simple list, where we record all the opening characters we have seen. If we encounter a closing character with no matching opening character, that is an error. If we find a matching closing character, we remove the open character from the list and continue. If at the end of the regular expression the list is empty, then all open characters had matching closing characters, meaning this case passed. The third case checks whether the internals of {} and [] have correct values.

Currently my system supports ranges of the form {x}, {x,y}, and [a-d]. Any other representation of range is considered an error that this case would catch and return. The fourth and last case makes sure that all special characters have legal positions. The characters * + ? and {} must all come after valid regular expressions. Pipe characters must come in between valid regular expressions, and not at the beginning or end of the regular expression. This case loops through the regular expression making sure the special characters are valid based on the given rules, checking neighboring characters to determine if they are valid. A better implementation of this fourth case would actually divide the regular expression into chunks, making sure that * + ? {} all come after valid chunks, and that pipe characters come in between valid chunks. That way the fourth case would not have the downfall of checking only neighboring characters, and not neighboring chunks. Another improvement could be made in the second case, where it could ensure that closing curly braces and closing square brackets come immediately after their openers. This must be true because regular expressions like [a-(abd)] and {[a-b], 7} are invalid, and this would make the syntax checker all the more careful.

By far the most complex and interesting part of my system is the graph creator. The key to my implementation is the encapsulation of any regular expression (or chunk of a regular expression) into something I will call a Graph. A Graph consists of input nodes and output nodes, with some encapsulated complexity in between the input and output nodes. Representing a chunk of a regular expression with a Graph allows my system to treat all chunks in the same way. By treating all chunks in the same way, I am able to create the finite state machine a lot easier, since I get to treat all instances of special characters (* for example) in the same way. The graph creation algorithm essentially consists of a loop and a large switch case. It starts by creating a Graph instance that will represent the entire regular expression. It loops through the indexes in the regular expression: when it comes across an alphanumeric, an escape character, or a [] statement, all it has to do is add one node representing that character (or range of characters) onto the end of the current graph. On special characters (like ? * + |) it connects previous graphs/nodes to accurately portray the statement. At the end you end up with one

filled Graph that represents the whole regular expression. The process is also recursive. If an open parenthesis is encountered, the system gets the regular expression that is in between the opening and closing parenthesis and recursively creates a Graph with it. Then when that Graph is returned, it connects the graph that was just created to the initial Graph and continues on. It should be noted that a chunk can be more than one character, and more than one node. For example, the initial Graph treats inner Graphs (like the one created by a parenthesis) as chunks, even though they really are full graphs, and can have long lengths. To keep everything in line correctly, the algorithm uses three lists: previousOutput, previousInput, and twoPreviousOutput. If these three lists are kept up to date, all of the special characters (like * + ? |) can be easily represented. For example, when the algorithm encounters a ? character, instead of having to go back through the regular expression to see where the last chunk began, the algorithm can trust that all it has to do is connect twoPreviousOutput to previousOutput, creating an edge that passes through the previous chunk. This thus accurately creates the behavior of a ? character. Improvements for this phase of the system are simply to create more features. More special characters that have different effects would make regular expressions more verbose and more efficient. However, I didn't have enough time to implement any new features besides the basic regular expression features.

Lastly, the searcher uses a DFS algorithm to find instances of the regular expression in a given piece of text. The algorithm initiates a search at every index in the given input, which creates the possibility for finding multiple instances of the regular expression in the same text. For example if I used the regular expression [0-9]+ to search through text “12345”, the results would be “12345”, “2345”, “345”, “45”, and “5”. When I realized this effect, I thought it was a mistake and so I changed the algorithm to start the search at the end of the last found instance, but I changed it back because in reality all 5 of those instances given in the example are valid instances, there's no reason to leave them out. It should be noted that each node in the graph has a “match” element, which is a list that contains the character codes of characters that are valid (I found character codes by using the built in ord()

python function). The search process is as follows: one character is read from the input text, if that character matches an input node A, we read another character. If that character matches any node that is connected to A, then we read another character. This process is repeated until we reach an output node such that the next character in the input text does not match any edge on said output node. Thus this process finds any substring in the given input that traverses from any input node to any output node, and returns it. The algorithm returns Found objects, which contain the text that matched, and the starting and ending indexes of where it was found in the input text. The big improvement that can be made here is speed. The algorithm is completely correct, but when a large regular expression is used, it can run quite slow.

I used Test Driven Development, with tests for each subsystem, and a test runner script that runs all of the tests seamlessly. This method of development helped me to decide on the structure of my system before I started developing, and gave me confidence in knowing (once I was done) that my system works as planned. Any and all help is appreciated! https://github.com/george-miller/regex