Finite Automata and Regular Expressions in Python

Kenneth Sanders

May 12, 2018

Introduction

This project aimed to recreate the unix shell utilities 'grep' and 'ls' in Python. The 'ls' command is relatively simple and self contained, requiring only the ls.py file. The 'grep' command is somewhat more complex, requiring the included FA (Finite Automata) and Regex modules in order to compile a regular expression into an Non-deterministic Finite Automata (NFA) and then process a given input efficiently. The following sections discuss the process behind creating these two utilities.

The 'ls' Utility

As previously stated, the 'ls' utility is fairly simple. Given a list of paths as arguments, 'ls' simply determines whether each path is a directory or a file, and lists the contained folders or the file itself respectively. Additionally, a few of the standard command-line flags for 'ls' have been included, such as the '-a' flag to display hidden files, or the '-l' flag to output in 'long' format. Other available flags can be listed by using 'ls -h'.

The 'grep' Utility

The 'grep' utility is much more complex than 'ls'. It must process a regular expression, generate an automata that recognizes the regular expression's language, and then use that automata to process input text and either accept or reject the input. Therefore, modules were developed in order to emulate an FA and a regular expression parser, named FA and Regex respectively.

The FA module provides an Nfa class that takes and stores a formal description of an NFA, and given an input to interpret, either accepts or rejects. The interpret function for an Nfa is iterative instead of recursive, since large inputs resulted in deep recursion, and extremely poor efficiency. This iteration is done by maintaining a queue of $(state, input)$ pairs that need to be evaluated with regard to the function $nextstatesandinputs$ function, which calculates the next states and their inputs for the automata given a current state and input. Using a queue allows the non-deterministic evaluation tree to be traversed breadth-first, so that any accept state found is the closest possible answer. The recursive method also works for small inputs, and can be found in the RecursiveNfa file in the FA module.

The Regex module is used parse a given regular expression and compile it into an Nfa object that recognizes the same language as the given regular expression. This is done by processing the regular expression input into a tree of 'Section' objects, which represent the possible operations in a regular expression, such as star, concatenate, and union. These operations all operate on a base 'Character' Section, which is the atomic unit in the tree. Each Section object is capable of converting its child Section(s) into an Nfa object by recursively having the child sections convert themselves to Nfa objects, and then combining the created child Nfas as each type of section is programmed to do. The base case for this recursion is the character sections, which are capable of generating a simple Nfa object which recognizes only its respective character. This allows the regular expression to be expressed as the Section tree, and then converted using a single function call in order to generate a Nfa representation of the regular expression. The regex class compiles the regular expression and stores the resulting Nfa object during its constructor call, so that the regular expression can be constructed once and used repeatedly without any cost besides the operation of the NFA.

This minimal cost is extremely important for the efficiency, since the actual compilation is relatively costly, with a runtime of $O(n^3)$, while interpreting a string using a pre-compiled NFA can be done in $O(n)$ time.

Once we have the regex module, all that needs to be constructed is the command-line functionality, which can be simply done using the ArgParse library in Python. The first argument given must be the regular expression that 'grep' should search for. The rest of the arguments may be the files for 'grep' to search in, or a directory to search all the files within. The recursive flag may also be used in order to recursively walk a given directory and use the regular expression on any files found.

Conclusion

In conclusion, I learned about how regular expressions are implemented and how much the implementation efficiency matters for large and complex regular expressions. I am most proud of my implementation of the Iterative NFA, since I spent a substantial amount of time trying to find a way to reduce the number of recursions in the original implementation before realizing that I could do away with the recursion altogether. I am also proud of my implementation of the Section sub-class for regular expressions, as they let me modularize the regular expression into a syntax tree, which made generating the respective NFA possible. The Section object idea also took some time to think about and plan how it should function, and I am glad that I used the subclass instead of trying to directly process the regular expression into a NFA. Overall, I learned a great deal about the intricacies of implementing regular expressions, and will certainly use what I have learned in the future.