Theory of Computation

Bret Black

1 Introduction

1.1 Why Machines?

The word "machine" is often used to describe a mechanical device that exists in the real world. The word "machine" can also be used to describe a theoretical device that helps mathematicians and computer scientists conceptualize problems. These machines form the foundation of computer science and can more simply describe which problems can and cannot be solved by a physical computer.

Four machines will be examined across the following pages: DFAs, NFAs, PDAs, and Turing Machines. Each machine will be examined first individually, then compared to other machines and languages. Advantages and shortcomings of each machine will be discussed, as well as their potential equivalences.

As each machine is dissected, the source code for the authors implementation will be provided and discussed. Sample inputs and outputs for this code will be provided.

1.2 Overview of Machines

The most basic machine to be discussed is the Deterministic Finite Automata, or DFA. A DFA is a simple state machine that accepts a series of instructions and returns a single output. State links must have exclusive key-value pairings. These machines can be used to solve very simple problems.

The next machine to be discussed is the Non-Deterministic Finite Automata, or NFA. The NFA is also a state machine, but unlike the DFA the NFA allows for multiple outputs. Links need not conform to exclusive key-value pairings - an infinite number of values can be paired with a particular key. The NFA also features epsilon transitions, which allow a machine to advance to another state without consuming any input.

Pushdown Automata, or PDA, takes the concept of a NFA and implements a stack. Unlike the input, which can only be consumed, the stack grows and shrinks as the machine passes through states. Links are no longer simple key-value pairings, but also consume from the stack and push to the stack. Like the NFA, PDAs are capable of multiple outputs. Instantaneous Descriptions are used to describe the machine at any given point. PDAs also allow the use of epsilon transitions.

The final machine to be discussed is the Turing Machine, which most closely simulates problems that can be solved on a physical computer. The

Turing Machine is conceptualized as an infinite tape on which data can be read and written. A tape head moves across this tape, reading and writing along the way.

1.3 Overview of Languages

Two types of languages will be discussed: context-free languages and regular languages. While neither of these languages were implemented, they will be referenced and used to explain the advantages of the machines.

Regular languages are simply defined as languages that can be expressed by regular expressions. Regular expressions are a popular tool for parsing text, instructions, and programming languages. They are particularly useful for expressing finite automata. Unfortunately, they are limited in power and incapable of expressing advanced computation.

Context-free languages rely on combinations of terminals and non-terminals. This allows for interpretations of palindromes. Context-free languages can also be ambiguous, which means they can have multiple derivations. These properties make context-free languages more powerful than regularly languages. They are particularly useful for expressing PDAs. It is worth noting that all regular languages are context-free languages.

Pumping Lemma
Regular and context-free languages contain what is known as the pumping lemma. The pumping lemma is a necessary property which states that the middle-section of a word in a regular language must be able to be pumped infinitely while producing a word that is still a part of the same language.

The pumping lemma is most commonly used to test the regularity of a particular language. If the pumping lemma fails, the languages regularity has been disproven. The pumping lemma was initially developed for regular languages, but was generalized to apply to context-free languages as well.

2 Machines

The following sections will discuss machines as well as the authors implementations of them. These implementations were written in Python 2 and make use of plain-text input files. In addition to the python classes used by individual machines (dfa.py, nfa.py, pda.py, turing.py) two additional classes were written. “stateMachine.py“ contains three classes that are used by one or more of the machines as well as two helper functions to convert state labels between integers and characters. “stateMinimize.py“ is used

only by DFAs and, as the name implies, minimizes the number of states in the machine.

2.1 DFA

Deterministic Finite Automata are simple state machines that travel from state-to-state along links that match the consumed input. These machines are incredibly basic, and limited in functionality, and were thus simple to implement. The authors implementation automatically minimizes these machines, ensuring the most efficient solution.

Input File
langSize - Determines how many symbols are in the language
links - Will be read until end is detected. These lines represent the links from each state - the first link read will be tied to the 0th state, the second link read will be tied to the first state, and so on. The first character represents the end-state if a 0 is consumed, the second character represents the end-state if a 1 is consumed, and so on. If an asterisk is provided at the end, this state is marked as an accepting state.
input - Will be read until an empty line is detected. Each line represents an instruction set.

Example input:
2
CB*
AC*
BA
end
111
101
1010

Implementation
The script first queries the user for the locations of the input file. Then, the script parses the input file and builds the machine described. The machine is then minimized using the state Minimize script previously described. The provided instructions are then read, one line at a time, and passed through the constructed machine. For each instruction, the script outputs whether or not the instruction set ends in an accepting state, as well as what state it ended in.

Each state is stored as an instance of the State class from stateMachine.py. The state class holds an array of links and a boolean that marks it as an accepting state or a non-accepting state.

There were no tricks to this implementation - each instruction set is converted into an array, and then the machine iterates through each element, following the links described by the input file.

Minimizing States

The stateMinimze.py file contains a set of helper functions used by the DFA implementation to minimize the machine. The implementation for this process mimicked the process described in the book very closely.

The process starts by filling in a two dimensional array. Both the X and Y axis represent states. First, cells that are tied to one accepting state and one non-accepting state are marked with a 1. All other cells are marked with a 0. The updateTable function is then called repeatedly until no changes are made to the table. This function iterates through all cell combinations within the table and checks for distinguishable pairs. When a value is changed, hasChanged is marked as true, which signals the the function must be run again.

Once all pairs have been marked, the partition function iterates through cells and identifies equivalent states. These states are then added to a list, which is sent to the constructMinimizedMachine function and and turned into a new list of states. This list of states is what is ultimately returned to the DFA, and used as the new state machine.

2.2 NFA

Non-Deterministic Finite Automata are only slightly more complicated than Deterministic Finite Automa - in fact, the implementation was only about 30 lines longer. NFAs support multiple links from each consumed input - for this reason, NFAs also support multiple outputs. NFAs also provide support for epsilon transitions, which allow a machine to traverse to a different state without consuming any input.

Input File

langSize - Determines how many symbols are in the language
links - Will be read until end is detected. These lines represent the links from each state - the first link read will be tied to the 0th state, the second link read will be tied to the first state, and so on. Inputs are separated by commas - the first set represents the end-states if a 0 is consumed, the

second set represents the end-states if a 1 is consumed, and so on. If an asterisk is provided at the end, this state is marked as an accepting state.
input - Will be read until an empty line is detected. Each line represents an instruction set.

Example:
2
A,BC
C,D
A,B
B,C*
end
11
1101
00
1001

Implementation
This script was very similar to the authors implementation of the DFA. As stated previously, the NFA only required the addition of 30 lines of code. Similar to the DFA, the NFA first queries the user for an input file, parses the file, and builds the machine. States are, again, stored as an instance of the State class. It then cycles through input strings and prints the output. Unlike the DFA, the NFA may provide several outputs for an instruction set. For each output, both the state ID and whether or not that state is an accepting state is printed.

Epsilon transitions are handled as an input equal to the langSize - or one higher than what would ever be read in (as languages are treated as [0...langSize-1].) Whenever a state is reached, the script checks for the existence of an epsilon link, and if it does exist, it follows.

The queue of states is handled as an array called nextState. The states in this array are to be run on the next consumed input. After each input is consumed, the nextState array is moved to activeState, and nextState is emptied. The script then iterates through all states in activeState and adds a new set of states of nextState. This continues until no instructions are left.

2.3 PDA

The Push-Down Automata is much more complicated than the DFA or the NFA. The script for this machine was almost twice as long as the script for the DFA and used two additional classes from stateMachine.py. While the machine itself is, really, just an extension of the NFA, the author was required to drastically modify his script for this machine. Perhaps the most significant addition brought by the PDA is the stack - the stack grows and shrinks throughout the machine and is just as important as the input for determining which link to use. Links become more complicated, as they have two values to match up with, and also must describe what they consume and add to the stack. These links may consume nothing or may add nothing, as represented by epsilon. PDAs also introduce Instantaneous Descriptions (IDs). IDs are used to represent the state of the machine at any given point. IDs hold the state, remaining input, and stack. These are often treated as nodes on a tree that represents the output for a given instruction set.

Input File
langSize - Determines how many symbols are in the language
startStack - The initial value of the stack
stateCount - The number of states
acceptingStates - The list of accepting states
links - Will be read until end is detected. These represent transition arcs. The first value is the start state, the second value is the end state, the third value is the consumed input, the fourth value is the consumed stack, and the fifth and final value is the produced stack.
input - Will be read until an empty line is detected. Each line represents an instruction set.

Example Input
2
1
4
C
A,B,1,1,2
A,C,0,1,0
C,B,1,0,0
C,D,2,0,0
D,C,0,0,1

end
11
1101
00
01
1001

Implementation
While this script certainly bears resemblance to the authors implementations of DFA and NFA, the design was significantly overhauled to accommodate stacks. Similar to previous machines, the script first queries the user for the input file and then builds a machine from this fine. Note that this script uses a for loop to generate states from a provided state count, rather than reading each state in as in previous machines. Transition arcs are constructed from the input file and stored in their appropriate state. States, again, use the State class from stateMachines.py. In this machine, the links array stores transition arcs rather than state IDs. This accounts for the additional complexity of the machine.

Where the NFA stored state IDs in the nextState array, the PDA stores PDANodes (instantaneous descriptions) in the nextNode array. Rather than iterating through states inside of a loop iterating through instructions, this machine uses a single queue that handles both the instruction and the state. Whenever a state is reached without any matching links, an output is printed. Similar to previous machines, this script outputs both the ID of the ending state and whether or not that state was an accepting state.

2.4 Turing Machine

The Turing Machine is the most complex, and also the most useful, machine discussed. The machine reads and writes on a tape of infinite length - initially, this tape is set to the input. A tape head moves across this tape - in one move, it may change states, write a symbol to the scanned cell (replacing the symbol that was there previously), or move left or right.

This machine varies greatly from the other machines in its focus on a single tape - the Turing Machine should not be viewed as a state machine, even though states are used. Transition functions specify a change in state, as with previous machines, but also specify a change in symbol for the current cell and a direction for the head to move. Note the absence of the stack - in its place is an infinite number of cells which can be written on.

Input File
langSize - Determines how many symbols are in the language
stateCount - The number of states
acceptingStates - The list of accepting states *transition functions* - Will be read until end is detected. The first value is the start state, the second value is the end state, the third value is the symbol to be consumed, the fourth value is the symbol to be written, and the fifth and final symbol is the direction for the tape head to move. Note that 0 represents left and 1 represents right.
input - Will be read until an empty line is detected. Each line represents an instruction set.

Example Input
2
3
B
A,B,0,2,1
A,C,1,1,1
B,B,0,1,0
B,C,2,1,1
C,C,1,2,1
C,A,1,2,1
end
11
1101
00
01
0
1001

Implementation
The implementation of this machine was, in some ways, simpler than the implementation for the PDA - there was no queue to maintain, the machine simply moved between states while adjsuting its position on a tape. The tape was implemented as a list, with an int tapePos tracking the tape head‘s location. The machine halted when no more moves were possible given the instruction set and links

3 Machine Relations

The four machines previously described serve a similar purpose - or, rather, are designed to serve the same purpose and do so with varying degrees of success. The following sections will sort these machines into what is known as the Chomsky Hierarchy as well as describe the equivalence of DFAs and NFAs.

3.1 Chomsky Hierarchy

The four machines discussed (and associated languages) are commonly sorted into what is known as the Chomsky Hierarchy. At the center of this hierarchy is the DFA and the NFA (which are equivalent, as is will be later discussed.) These are the machines powered by regular languages. Next are PDAs, which are powered by context-free languages. At the top is the Turing Machine, which is powered by recursively enumerable languages.

This hierarchy can be conceptualized as a series of concentric circles. Every DFA and every NFA can be described as a PDA, and similarly every PDA can be described as a Turing Machine.

The Chomsky Hierarchy should not come as a surprise - each machine was largely implemented as an expansion of the previous machine. These machines were discussed (and implemented) an an outward-expanding order and, naturally, were described by what additional capabilities each machine had. No capabilities were lost moving outward in the hierarchy. The descriptions may have become more complex, but functionality was never lost.

3.2 Equivalence

DFAs and NFAs, the most basic of the described machines, are equivalent. While the NFA is more complicated and may appear to be more powerful, it can be easily reduced down to a DFA. This equivalence is achieved by injecting intermediary states. If a consumed input is mapped to two or more links, an additional state must be introduced. The consumed input will now be mapped to this new state which will hold maps to the displaced links. Note that if the number of links mapped to a single input is greater than the size of the language, several intermediary states will be necessary.

While these machines are equivalent, the NFA does have advantages. It is far more condensed than the DFA - the absence of intermediary states has the potential to drastically reduce the size of the machine. Similarly, the absence of intermediary states reduces the number of instructions required.

4 Conclusion

Four machines (DFA, NFA, PDA, and Turing Machine) were described, along with the authors implementation of them. Although these machines pre-date physical computing machines, these theoretical machines are used to conceptualize which problems can and cannot be solved by a modern computer. Each of these machines is paired with a type of language (Regular, Context-Free, and Recursively Enumerable) that can be used to describe it. These four machines can be set into what is known as the Chomsky Hierarchy. This hierarchy describes each of these machines as being expansions of one another, with DFAs/NFAs and the center and the Turing Machine at the outside.