A blue and white logo

Description automatically generated with low confidence

Automata Final Project

**Submitted By:**

**Ammar Yasser 19P1126**

**Mohamed Waleed 19P8122**

**Mohamed Hesham 19P9280**

**Mohamed Hani 19P6696**

**Omar Adel 19P8355**

**Tasneem Osama 18P3208**

**Ain Shams University**

**Faculty of Engineering**

**International Credit Hours Engineering Programs (I-CHEP)**

**Spring Semester – 2023**

Table of Contents

[1. PART 1 2](#_Toc134136056)

[1.1. Report: 2](#_Toc134136057)

[1.1.1. Introduction: 2](#_Toc134136058)

[1.1.2. Turing Machine: 2](#_Toc134136059)

[1.1.3. Pushdown Automata: 2](#_Toc134136060)

[1.1.4. Implementations: 3](#_Toc134136061)

[1.1.5. Advantages and Disadvantages: 3](#_Toc134136062)

[1.1.6. Problems Solved and Overcome: 3](#_Toc134136063)

[1.1.7. Shortages and Drawbacks: 3](#_Toc134136064)

[1.1.8. Extra Knowledge: 3](#_Toc134136065)

[2. PART 2 4](#_Toc134136066)

[2.1. Part 2 Section 1: 4](#_Toc134136067)

[2.1.1. Code: 4](#_Toc134136068)

[2.1.2. Code documentation: 4](#_Toc134136069)

[2.1.3. Screenshots from output: 4](#_Toc134136070)

[2.2. Part 2 Section 2: 4](#_Toc134136071)

[2.2.1. Code: 4](#_Toc134136072)

[2.2.2. Code documentation: 4](#_Toc134136073)

[2.2.3. Screenshots from output: 4](#_Toc134136074)

A table of figures will be added once the report is finalized.

# PART 1

In this part we will write a report on how the Turing machine and Pushdown automata are used in the computation.

## Report:

### Introduction:

Computation models are used to describe how machines perform computations and solve problems. Two important computation models are the Turing machine and pushdown automata. These models provide a framework for understanding the limits of computation and the kinds of problems that can be solved by different types of machines.

### Turing Machine:

The Turing machine is a mathematical model of a hypothetical computing machine. It was invented by Alan Turing in the 1930s as a way to explore the limits of computation. A Turing machine consists of a tape, a read-write head, and a set of rules for manipulating the tape. The tape is divided into cells, each of which can hold a symbol. The read-write head can move left or right along the tape and read or write symbols on the tape.

A Turing machine can perform any computation that can be performed by a computer. It is a powerful model of computation that can solve many kinds of problems. However, it is also a very abstract model that is not easy to implement in practice.

### Pushdown Automata:

Pushdown automata are another model of computation that are used to recognize context-free languages. A pushdown automaton consists of a stack and a finite state machine. The stack is used to keep track of the context of the input string. The finite state machine determines which state the automaton is in and what action to take next.

Pushdown automata can recognize a larger class of languages than finite state machines. They are particularly useful for recognizing nested structures such as matching brackets in a programming language. However, they are still limited in their computational power compared to Turing machines.

### Implementations:

Turing machines and pushdown automata can be implemented using various programming languages. For example, a Turing machine can be implemented in Python using a class that represents the tape and read-write head. Similarly, a pushdown automaton can be implemented in C++ using a stack data structure and a switch statement to determine the next action.

### Advantages and Disadvantages:

The advantages of Turing machines and pushdown automata are their ability to solve a wide range of problems and their clear mathematical formalism. They provide a framework for understanding the limits of computation and the kinds of problems that can be solved by different types of machines. However, their disadvantages are their abstractness and complexity, which can make them difficult to understand and implement in practice.

### Problems Solved and Overcome:

Turing machines and pushdown automata can solve many kinds of problems that are relevant to computer science and mathematics. For example, they can recognize regular and context-free languages, solve optimization problems, and simulate other types of machines. They have also helped to establish the limits of computation and the kinds of problems that are intractable.

### Shortages and Drawbacks:

One of the main drawbacks of Turing machines and pushdown automata is their computational complexity. They are often used as theoretical models rather than practical ones due to their high time and space complexity. They can also be difficult to understand and implement, particularly for people who are new to computer science.

### Extra Knowledge:

Turing machines and pushdown automata are important tools in the field of theoretical computer science. They provide a way to understand the limits of computation and the kinds of problems that can be solved by different types of machines. While they are not always practical for solving real-world problems, they have helped to establish the foundations of computer science and mathematics.

# PART 2

In this part we will have two sections each containing a code with its own documentation and screenshots from the output.

## Part 2 Section 1:

convert any non-deterministic finite automaton to deterministic finite automaton.

### Code:

import tkinter as tk

*#Global Variables*

F\_dfa=[] *#Accepting states of the DFA*

d = {} *#Dictionary to store DFA's transition function*

def delta\_nfa(state, alphabet):

    return trans[state][alphabet] *#transition function of the nfa. Returns a list*

def main():

*#----------------------------------Input processing---------------------#*

    f = open("nfa.txt", "r")

*#-----------Reading States-----------#*

    line=f.readline()

    line=line.strip()

    length=len(line)

    state\_comma=line[1:length-1] *#Removing braces*

    state\_comma = state\_comma.replace(" ", "") *#Removing whitespaces*

    global Q\_nfa

    Q\_nfa = state\_comma.split(",") *#Reading CSVs into a list*

*#-----------Reading Alphabets-----------#*

    line=f.readline()

    line=line.strip()

    length=len(line)

    letters\_comma=line[1:length-1]

    letters\_comma = letters\_comma.replace(" ","")

    global sigma

    sigma=letters\_comma.split(",")

*#-----------Reading Transition Function-----------#*

    line=f.readline()

    line=line.strip()

    line=line.replace(" ", "") *#Removing whitespaces*

    qline=""

    oq=False *#Open(Unpaired) Quote*

*#Enclosing all states and alphabets in quotes*

    for i in line:

        if(i == '{'):

            qline+=i

            continue

        if(i==':' or i==',' or i=='}' or i=='[' or i==']'):

            if(oq):

                qline+="'"

                qline+=i

                oq=False

            else:

                qline +=i

            continue

        if(oq==False):

            oq=True

            qline +="'"

            qline +=i

            continue

        if(oq==True):

            qline +=i

    global trans

    trans=eval(qline) *#String to dictionary conversion*

*#-----------Reading Start State-----------#*

    line=f.readline()

    line=line.strip()

    line=line.replace(" ", "")

    line = str(line)

    global start\_nfa

    start\_nfa =line

*#-----------Reading Final States-----------#*

    line=f.readline()

    line=line.strip()

    length=len(line)

    final\_comma=line[1:length-1]

    final\_comma = final\_comma.replace(" ","")

    global F

    F = final\_comma.split(",")

*#Getting the power set of Q\_nfa, and filling*

*#it in Q\_dfa*

    global Q\_dfa

    Q\_dfa = sub\_lists(Q\_nfa)

    construct\_delta\_dfa() *#Calling the function to create the transition function of dfa*

*#Start State of dfa is the e closure of the start state of nfa,*

*#plus start state itself.*

    start\_eclose = e\_closure\_state(start\_nfa,[])

    start\_eclose.append(start\_nfa)

    start\_state\_dfa = find\_dfa\_state(start\_eclose)

    f.close()

*#----------------------------------Output processing---------------------#*

    fo = open("dfa.txt", "w")

*#-----------Writing States-----------#*

    statesStr = str(Q\_dfa)

    statesStr = statesStr.replace("'", "") *#Remove quotes around states*

    fo.write('States: '+statesStr+"\n")

*#-----------Writing Alphabets-----------#*

    alphaWrite = str(sigma).replace("'","")

    fo.write('Alphabets: '+alphaWrite+"\n")

*#-----------Writing Transition Function-----------#*

    dString = str(d).replace("'","")

    fo.write('Transition Function: '+dString+"\n")

*#-----------Writing Start State-----------#*

    StartString = str(start\_state\_dfa).replace("'","")

    fo.write('Start State: '+StartString+"\n")

*#-----------Writing Accept States-----------#*

    AcceptString = str(F\_dfa).replace("'","")

    fo.write('Accept States: '+AcceptString)

    fo.close()

def construct\_delta\_dfa():*#Constructing the transition function for the DFA*

   for i in Q\_dfa: *#For each state in the dfa, we figure out the transitions*

      dict\_state = {}

      for k in sigma:*#For each alphabet*

         to\_composite = []

         for j in i: *#For each dfa state,corresponding to multiple nfa states, find E(q) i.e. closure*

            to = []

            tos\_alpha = delta\_nfa(j,k) *#Add states reachable from the main state by the alphabet*

            to.extend(tos\_alpha)

            '''tos\_epsilon = e\_closure\_state(j,[]) #Apparently these states are not counted in the transition,

            #so it is commented out.

            for l in tos\_epsilon: #Add states reachable from the e-closure states by the alphabet

               tos\_epsilon\_alpha = delta\_nfa(l,k)

               to.extend(x for x in tos\_epsilon\_alpha if x not in to)'''

            for l in to: *#Add states e-reachable from states reachable by alphabet*

               tos\_alpha\_epsilon = e\_closure\_state(l,[])

               to= to + list(set(tos\_alpha\_epsilon) - set(to))

            to\_composite = to\_composite + list(set(to) - set(to\_composite))

         to\_composite\_state = find\_dfa\_state(to\_composite)*#Sort the states in the nomenclature*

         dict\_state[k] = to\_composite\_state

      d[str(i)] = dict\_state

   for i in Q\_dfa: *#Accept states of the DFA*

      for j in i:

         if j in F:

            F\_dfa.append(i)

            continue

def delta\_dfa(state, alphabet):

    t=tuple(state, alphabet)

    return d[t]

def sub\_lists(l):

*#Returns the power set of the input list*

   def decimalToBinary(n):   *# converting decimal to binary*

       b = 0

       i = 1

       while (n != 0):

           r = n % 2

           b+= r \* i

           n//= 2

           i = i \* 10

       return b

   def makeList(k):       *# list of the binary element produced*

       a =[]

       if(k == 0):

           a.append(0)

       while (k>0):

           a.append(k % 10)

           k//= 10

       a.reverse()

       return a

   def checkBinary(bin, l):

       temp =[]

       for i in range(len(bin)):

           if(bin[i]== 1):

               temp.append(l[i])

       return temp

   binlist =[]

   subsets =[]

   n = len(l)

   for i in range(2\*\*n):

       s = decimalToBinary(i)

       arr = makeList(s)

       binlist.append(arr)

       for i in binlist:

          k = 0

          while(len(i)!= n):

             i.insert(k, 0) *# representing the binary equivalent according to len(l)*

             k = k + 1

   for i in binlist:

      subsets.append(checkBinary(i, l))

   return subsets

def e\_closure\_state(state,eclose):

*#Recursive function. Input -*

*#a state and a list of states which are reachable from it(initially empty)*

   primary = trans[state]['epsilon'] *#A list of states one hop away*

   primary\_no\_duplicates=[]

   [primary\_no\_duplicates.append(x) for x in primary if x not in primary\_no\_duplicates] *#Remove duplicates*

*#eclose.append(primary\_no\_duplicates)*

   eclose = eclose + list(set(primary\_no\_duplicates) - set(eclose))

   for i in primary\_no\_duplicates:

      if i not in eclose:*#Base case = the state is already recorded in the e-closure*

         eclose.append(i)

         secondary = e\_closure\_state(i,eclose.copy()) *#Recursive step*

         for j in secondary:*#Sort through the e-reachable states of the secondary state*

            if j not in eclose: *#Add it to list of reachable states if it is not already added*

               eclose.append(j)

   return eclose.copy()

def find\_dfa\_state(state\_list):

*#Given a list of nfa states in random order, find the corresponding single dfa state in correct order*

   for i in Q\_dfa:

       if(set(i) == set(state\_list)):

          return i.copy()

main()

def show\_output():

*# Call main function*

    main()

*# Create GUI window*

    root = tk.Tk()

    root.title("DFA Output")

*# Create label to display output*

    output\_label = tk.Label(root, text="DFA Output:\n\n")

    output\_label.pack()

*# Open and read output file*

    with open("dfa.txt", "r") as f:

        output\_text = f.read()

*# Create text box to display output*

    output\_textbox = tk.Text(root, height=15, width=150)

    output\_textbox.pack()

    output\_textbox.insert(tk.END, output\_text)

    root.mainloop()

*# Create GUI button to run show\_output function*

button = tk.Button(text="Run Program", command=show\_output)

button.pack()

*# Run GUI loop*

tk.mainloop()

### Code documentation:

This Python script converts a Nondeterministic Finite Automaton (NFA) to a Deterministic Finite Automaton (DFA). It takes the following steps:

* Processes the input file **nfa.txt** containing the NFA's characteristics such as states, alphabets, start state, and final states.
* Constructs a DFA using the powerset construction method. The states of the DFA are the power set of the states of the NFA.
* Constructs the transition function of the DFA using the NFA's transition function and the Epsilon closure of states. The Epsilon closure of a state is the set of states that are reachable from it using Epsilon transitions. The transitions are evaluated for each alphabet in the language.
* The final states of the DFA are those that contain one or more final states of the NFA.
* Writes the output to the file **dfa.txt**.

**Requirements**

This script requires the following dependencies:

* tkinter
* Python 3.x

**Usage**

The script expects the input file **nfa.txt** to be present in the same directory as the script file. The output file **dfa.txt** will be created in the same directory.

To run the script, execute the following command in the terminal:

bashCopy code

python nfa\_to\_dfa.py

**Variables**

The script has the following variables:

* **Q\_nfa**: list of states of the NFA
* **sigma**: list of alphabets in the language of the NFA
* **trans**: dictionary representing the transition function of the NFA
* **start\_nfa**: start state of the NFA
* **F**: list of final states of the NFA
* **Q\_dfa**: list of states of the DFA
* **d**: dictionary representing the transition function of the DFA
* **F\_dfa**: list of final states of the DFA

**Functions**

The script has the following functions:

**delta\_nfa(state, alphabet)**

Given a state and an alphabet, returns a list of states reachable from the state using the alphabet in the NFA.

**sub\_lists(lst)**

Given a list, returns the power set of the list.

**e\_closure\_state(state, visited)**

Given a state and a list of visited states, returns the Epsilon closure of the state.

**find\_dfa\_state(states)**

Given a list of states, returns a sorted, concatenated string representation of the states.

**construct\_delta\_dfa()**

Constructs the transition function of the DFA using the NFA's transition function and the Epsilon closure of states. The Epsilon closure of a state is the set of states that are reachable from it using Epsilon transitions. The transitions are evaluated for each alphabet in the language.

**main()**

The main function of the script that reads the input file, constructs the DFA, and writes the output file.

### Screenshots from output:

Input text file :

A screen shot of a computer

Description automatically generated with medium confidence

Output :

A screenshot of a computer

Description automatically generated with medium confidence

## Part 2 Section 2:

convert context free grammar to Pushdown automaton.

### Code:

1. from typing import List, Tuple  
   import sys  
     
     
   def CFG\_to\_PDA(gram: List[Tuple[str, List[str]]], nonterm\_states: List[str], term\_states: List[str],  
    start\_state: str) -> None:  
    print("Grammar given as input:")  
    print(f"Start State: {start\_state}")  
    print("Non Terminal States: ")  
    print(\*nonterm\_states, sep=" ")  
    print("Terminal States: ")  
    print(\*term\_states, sep=" ")  
    print("Production Rule: ")  
    for rule in gram:  
    state, productions = rule  
    print(f"{state} -> {productions[0]}", end="")  
    for i in range(1, len(productions)):  
    print(f" | {productions[i]}", end="")  
    print()  
    print("-" \* 73)  
     
    print("\nCorresponding PDA: ")  
    for rule in gram:  
    state, productions = rule  
    print(f"δ(q, ε, {state}) = {{ (q, {productions[0]})", end="")  
    for i in range(1, len(productions)):  
    print(f", (q, {productions[i]})", end="")  
    print(" }")  
    print()  
    for term in term\_states:  
    print(f"δ(q, {term}, {term}) = (q, ε)")  
     
     
   if \_\_name\_\_ == "\_\_main\_\_":  
    gram = []  
    nonterm\_states = []  
    term\_states = []  
     
    start\_state = ""  
    with open("cfg.txt") as fin:  
    for line in fin:  
    line = line.strip()  
    if line.startswith("NT"):  
    nonterm\_states = line.split()[1:]  
    start\_state = nonterm\_states[0]  
    elif line.startswith("T"):  
    term\_states = line.split()[1:]  
    else:  
    state, productions = line.split(maxsplit=1)  
    productions = productions.split("|")  
    productions = [p.strip() for p in productions]  
    gram.append((state, productions))  
     
    CFG\_to\_PDA(gram, nonterm\_states, term\_states, start\_state)

### Code documentation:

Sure, here's a detailed report explaining the functionality of the code:

The code is used to convert a context-free grammar (CFG) to an equivalent pushdown automaton (PDA). The main function of the code is `CFG\_to\_PDA()` which takes the following arguments:

- `gram`: A list of tuples where each tuple represents a production rule. The first element of the tuple is the non-terminal state and the second element is a list of possible productions for that state.

- `nonterm\_states`: A list of strings representing the non-terminal states of the grammar.

- `term\_states`: A list of strings representing the terminal states of the grammar.

- `start\_state`: A string representing the start symbol of the grammar.

The function first prints the input CFG in a human-readable format. It then converts the CFG to an equivalent PDA and prints the transition functions for the PDA.

The function first prints the start state, non-terminal states, terminal states, and production rules of the input CFG. The production rules are printed in the format `<non-terminal> -> <production1> | <production2> | ... | <productionN>`.

After printing the input CFG, the function converts it to an equivalent PDA. It does so by creating a new state `q` and adding a transition for each production rule. For each production rule of the form `A -> w`, the function adds a transition `δ(q, ε, A) = {(q, w)}`. This transition pushes the first symbol of the production onto the stack.

Next, the function adds a transition for each terminal symbol of the grammar. For each terminal symbol `a`, the function adds a transition `δ(q, a, a) = (q, ε)`. This transition pops the top symbol from the stack.

Finally, the function prints the transition functions for the PDA in the format `δ(q, X, Y) = {(q', Z1), (q'', Z2), ... ,(qN, ZN)}`. Here, `q` is the current state, `X` is the input symbol, `Y` is the symbol at the top of the stack, and `{(q', Z1), (q'', Z2), ... ,(qN, ZN)}` is the set of next states and stack symbols to push onto the stack.

The code does not use any external libraries or functions. It only uses the built-in functions and data types provided by Python.

### Screenshots from output:

Input from the cfg file :

A screenshot of a computer

Description automatically generated

Output :

A screenshot of a computer

Description automatically generated