CSE 432 Automata and Computability

PROJECT DOCUMENTATION

Submitted to:

Prof. Dr. Gamal Abdel Shafy

Eng. Sally Shaker

Submitted by: Team 1

Youssef George Fouad	19P9824
Mostafa Nasrat Metwally	19P4619
Kerollos Wageeh Youssef	19P3468
Anthony Amgad Fayek	19P9880
Marc Nagy Nasry	19P3041
Youssef Ashraf Mounir	19P4179

Table of Contents

Table Of Figures

1. INTRODUCTION

In the field of computer science, the Turing machine and pushdown automaton are two important computational models that have been used to study the nature of computation and the limits of algorithmic complexity. These models have played a central role in the development of modern computing systems and have been used to solve a wide variety of problems in fields ranging from linguistics to cryptography. In this report, we will explore the implementations, advantages and disadvantages, problems they solve or overcome, and the history of the Turing machine and pushdown automaton.

2. HISTORICAL BACKGROUND

The Turing machine was named after Alan Turing, who introduced the concept in his 1936 paper " ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM". Turing's paper presented the idea of a theoretical machine that could perform any computation that could be performed by a human "computer". The Turing machine was initially conceived as a thought experiment to explore the limits of computability and the nature of algorithmic complexity. However, it quickly became a central tool in the study of computability theory.

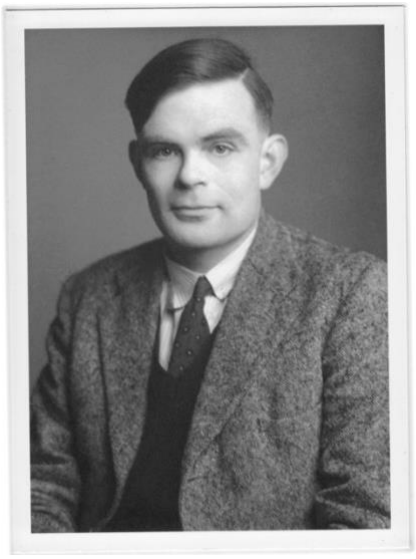

Figure 1: Alan Turing

Pushdown automata were first introduced in the 1950s as a way to model context-free grammars, which were becoming increasingly important in computer science and linguistics. John Backus and Peter Naur, two computer scientists, independently introduced the idea of using pushdown automata to describe the syntax of programming languages. Pushdown automata were also used to study the formal properties of natural languages and to develop algorithms for mac hine translation.

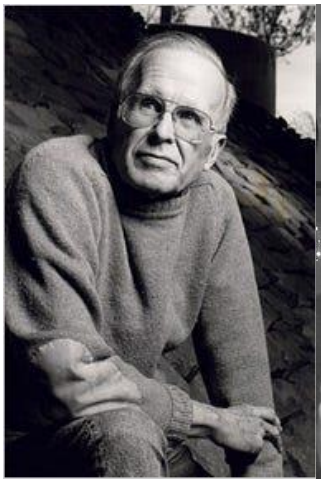

Figure 2: John Backus

Figure 3: Peter Naur

Today, both the Turing machine and pushdown automaton remain important tools in the study of computability and computational complexity. They have been used to develop programming languages, compilers, and other software systems, and have had a profound impact on the field of computer science.

3. TURING MACHINE

3.1. How It Works

A Turing machine consists of several components, including a tape, a head, state register, and a transition function.

1. **Tape:** The tape is an infinite sequence of cells, where each cell can hold a symbol. The symbols can be letters, numbers, or any other discrete values. The tape serves as the input and output medium for the Turing machine.

2. **Head:** The head is responsible for reading and writing symbols on the tape. It can move left or right along the tape, one cell at a time.

3. **State Register:** The state register represents the internal state of the Turing machine. It holds the current state of the machine, which determines its behavior at any given point in time.

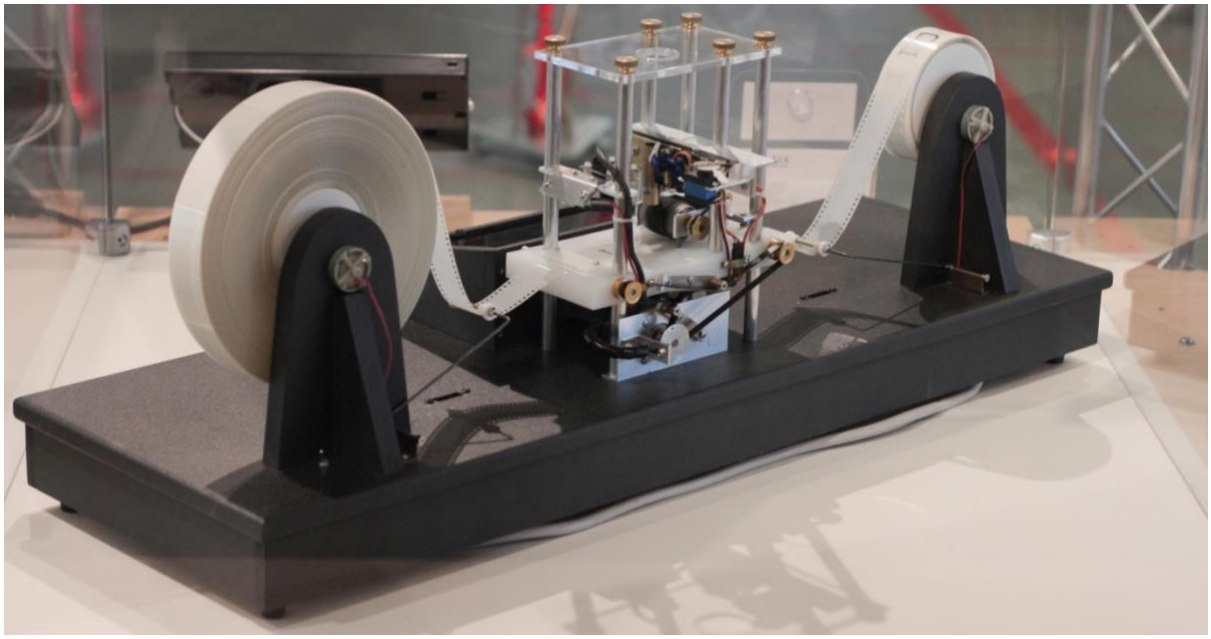

Figure 4: Turing Machine

4. **Transition Function:** The transition function defines the behavior of the Turing machine. It specifies how the machine should transition from one state to another based on the current symbol read from the tape and the current state of the machine.

The operation of a Turing machine consists of a number of steps that can be described briefly as follows:

1. **Initialization:** Initially, the input is placed on the tape, and the head is positioned at the beginning of the input.

2. **Execution:** The Turing machine starts in an initial state and begins reading symbols from the tape. Based on the current state and the symbol read, the machine consults the transition function to determine the next state and the action to perform. The action can include writing a new symbol on the tape, moving the head one position to the left

or right, and changing the internal state of the machine. After performing the action, the machine transitions to the next state and continues the process.

3. **Halting:** The Turing machine halts when it enters a designated halting state. At this point, the computation is complete, and the final contents of the tape represent the output of the Turing machine.

3.2. Implementations

The concept of a Turing machine allows us to reason about the theoretical limits of computation. However, the Turing machine remains a foundational concept in computer science and helps us understand the fundamental principles of computation and algorithmic processes.

Turing completeness refers to the ability of a computational system, such as a programming language or a computer, to simulate a Turing machine. If a system is Turing complete, it can theoretically solve any problem that a Turing machine can solve.

Turing machines have been used as a theoretical basis for developing practical computing systems. For example, the modern computer can be viewed as a collection of Turing machines working together. The tape in a Turing machine can be thought of as a computer's memory, while the read-write head is akin to a processor. Turing machines have also been used to develop programming languages, compilers, and other software systems. However, various real-life technologies and systems are inspired by the principles of Turing machines and offer similar computational capabilities. Here are a few examples:

1. **Universal Turing Machines:** Universal Turing Machines (UTMs) are hypothetical Turing machines capable of simulating any other Turing machine. They are used as a theoretical basis for modern computers and programming languages. Real-life computers, such as desktops, laptops, and smartphones, are designed to be programmable and can simulate any algorithm, making them akin to UTMs.

2. **Programming Languages:** Programming languages, such as Python, Java, and C++, provide a means to write algorithms and execute them on real-life computers. These languages offer constructs and data structures that allow developers to manipulate symbols and perform computations, similar to the operations of a Turing machine.

3. **Abstract Machines:** Abstract machines, such as the Universal Turing Machine (UTM) and Random-Access Machines (RAM), are used in theoretical computer science to analyze the complexity of algorithms and study computational models. These machines provide insights into the efficiency and feasibility of algorithms but are not physical implementations.

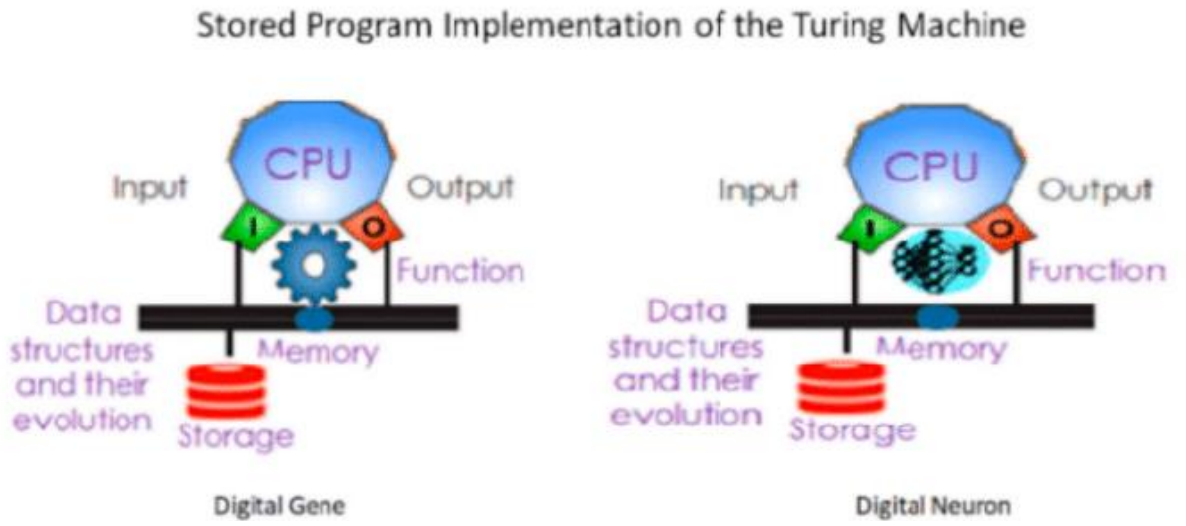

Figure 5: Turing Machine Implementations

4. **Finite State Machines:** Finite state machines (FSMs) are computational models that operate in a similar manner to Turing machines but with limited memory and tape size. FSMs are widely used in various applications, including digital circuit design, language processing, and protocol specifications.

5. **Turing-complete Systems:** Many real-life systems and programming languages are Turing complete, meaning they can simulate a Turing machine and perform any computation that a Turing machine can perform. For example, the JavaScript

programming language, which powers web browsers, is Turing complete and can simulate a Turing machine.

6. **Computer Architecture:** While modern computer architecture does not directly resemble a Turing machine, it incorporates concepts inspired by Turing's work. Components like the central processing unit (CPU), memory, and input/output systems collectively enable computation and information processing, although with finite resources and limitations.

It's important to note that these real-life implementations deviate from the theoretical ideals of Turing machines by operating on finite resources, having practical limitations, and being subject to physical constraints. Nevertheless, they draw upon the fundamental principles of computation and provide powerful tools for solving a wide range of practical problems.

3.3. Advantages and Disadvantages

One significant advantage of the Turing machine is its ability to simulate any other computational model. This universality property means that any problem that can be solved by any other computational model can be solved by a Turing machine. Turing machines possess several advantages as a theoretical model of computation. Turing machines can simulate any algorithm that can be described in a step-by-step manner, making them powerful tools for studying the limits and capabilities of computation. This universality allows researchers to explore various computational problems and develop insights into algorithmic solutions.

Another advantage of Turing machines is their mathematical simplicity. The model's abstraction, consisting of a tape, a head, and a transition function, provides a clear and elegant representation of computation. This simplicity aids in the analysis and theoretical reasoning about algorithms and their behavior. By precisely defining the operations and transitions of a Turing machine, researchers can study the properties and complexities of algorithms in a rigorous and formal manner.

Turing machines also serve as a framework for algorithmic analysis. By counting the number of steps or transitions required by a Turing machine to solve a problem,

researchers can evaluate the time complexity of algorithms. This analysis helps in determining the efficiency and scalability of algorithms for solving larger instances of problems. The ability to reason about complexity enables the development of efficient algorithms and the identification of computational bottlenecks.

Moreover, Turing machines provide a theoretical foundation for computer science and the theory of computation. They offer a common language and conceptual basis for discussing algorithms, complexity, and computability. The Turing machine model allows researchers to establish fundamental results, proofs, and theoretical frameworks that form the backbone of computer science as a discipline.

However, Turing machines also have certain disadvantages. One primary limitation is their abstract and idealized nature. Turing machines assume infinite tape length, unlimited memory, and unbounded resources, which do not align with the constraints and limitations of real-life computers. Directly implementing Turing machines in practical computing systems is not feasible, as real-life machines have finite resources and face physical constraints.

Additionally, Turing machines are not well-suited for real-time or parallel computations. They operate sequentially, processing symbols one at a time, which limits their efficiency in scenarios that require immediate response and simultaneous processing of multiple tasks. Real-life computing systems often employ specialized architectures and algorithms to achieve real-time performance and handle parallelism effectively.

Furthermore, the representation of data and computation in a Turing machine can be non-intuitive for humans. The tape and symbols used by Turing machines do not directly correspond to natural representations of data, such as text, images, or real numbers. This abstraction can make it challenging to understand and reason about complex algorithms, particularly for those without a strong background in theoretical computer science.

Lastly, Turing machines do not provide a solution for intractable problems. The halting problem, for instance, highlights the inherent limitations of Turing machines. It refers to the inability to determine, in general, whether a given Turing machine will eventually

halt or run indefinitely. This demonstrates the existence of problems that are inherently unsolvable or undecidable within the scope of Turing machines.

Despite these limitations, Turing machines remain a foundational concept in computer science and continue to play a crucial role in understanding the fundamental principles of computation and algorithmic processes.

3.4. Problems It Solves

Turing machines serve as powerful tools for solving a diverse array of computational problems in the field of theoretical computer science. They are adept at addressing decision problems, which require a binary "yes" or "no" answer. By examining input strings, Turing machines can determine whether they belong to a specific language or satisfy certain properties. This allows for solving problems such as primality testing, palindrome verification, and assessing the provability of mathematical statements.

Additionally, Turing machines facilitate the analysis of algorithmic complexity. By counting the number of steps or transitions required to solve a problem, researchers can assess the time and space efficiency of algorithms. This analysis aids in understanding the scalability and resource requirements of algorithms for solving problems of varying sizes.

Turing machines also play a crucial role in studying computability and undecidability. They enable investigations into whether a problem is solvable by an algorithm or whether it falls into the realm of undecidable problems, for which there is no universally applicable solution. Prominent examples of undecidable problems include the halting problem, which seeks to determine whether a given program will eventually halt or run indefinitely, and the Entscheidungsproblem, which deals with determining the decidability of logical statements.

Formal language recognition is another area where Turing machines excel. They can act as acceptors, determining whether a given string belongs to a language defined by a formal grammar. Additionally, Turing machines can function as generators, producing valid strings adhering to the rules and structure of a particular language.

Furthermore, Turing machines provide valuable capabilities for simulation and modeling in various domains. Researchers can use Turing machines to simulate and analyze the behavior and properties of algorithms, protocols, and systems. This allows for a deeper understanding of the inner workings, limitations, and potential improvements of computational systems in fields such as computer science, mathematics, and artificial intelligence.

While it is essential to note that practical implementations often require adjustments to accommodate finite resources and time constraints, Turing machines remain an indispensable theoretical framework. They provide foundational knowledge about computation, algorithmic problem-solving, and the boundaries of what is computationally feasible. By studying Turing machines, researchers gain valuable insights into the possibilities and limitations of computational systems.

3.5. Shortages and Drawbacks

Turing machines have several limitations and shortages. Firstly, they abstract away real-world constraints by assuming infinite tape length, unlimited memory, and unbounded resources, which do not align with the limitations of practical computing systems. Additionally, Turing machines are not well-suited for real-time or parallel computations as they operate sequentially. The representation of data and computation in Turing machines can be non-intuitive, making it challenging for humans to understand complex algorithms implemented on them. Furthermore, Turing machines cannot solve certain problems like the halting problem, indicating the existence of inherently unsolvable or undecidable problems. Lastly, Turing machines do not explicitly prioritize efficiency, and real-life computing systems require algorithmic optimizations and specialized data structures to achieve practical efficiency goals.

4. PUSHDOWN AUTOMATA:

4.1. How it works

Pushdown automata are theoretical computational models that extend the capabilities of finite automata by incorporating a stack. They are used to recognize and accept context-free languages, which are a type of formal language defined by a context-free grammar.

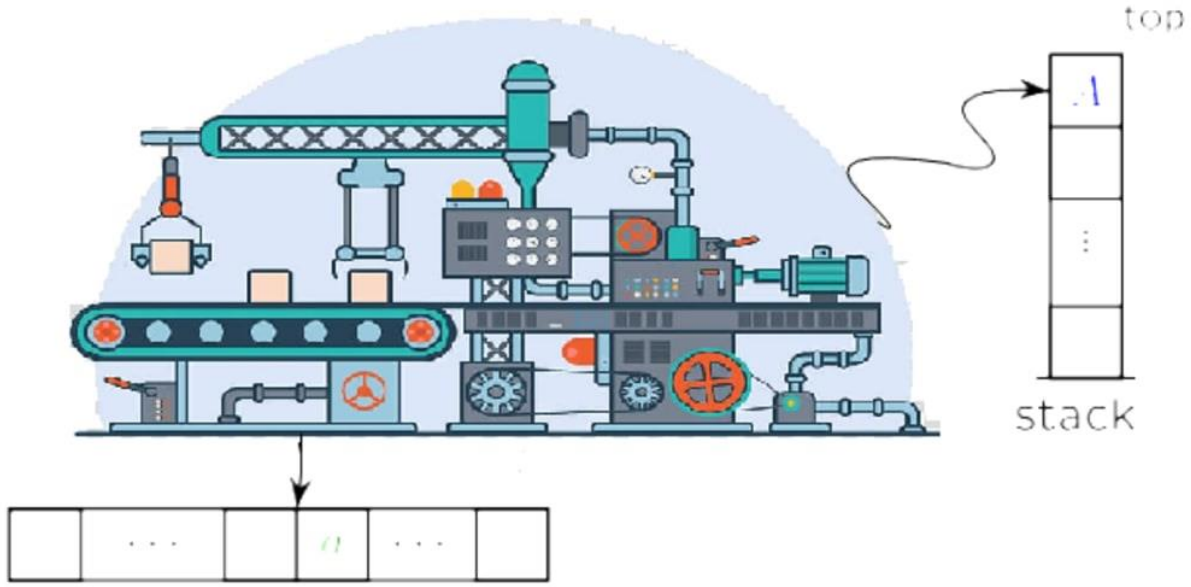

Figure 6: Pushdown Automata

A pushdown automaton consists of the following components:

1. **Input Alphabet:** A finite set of symbols that make up the input string.

2. **Stack Alphabet:** A finite set of symbols that can be pushed onto and popped from the stack.

3. **Transition Function:** A function that defines the behavior of the PDA. It takes the current state, the input symbol, and the top symbol of the stack as input and determines the next state, the symbol to be pushed onto the stack (if any), and the symbols to be popped from the stack (if any).

4. **Start State:** The initial state of the PDA.

5. **Accepting States:** A set of states in which the PDA can halt and accept the input string.

The implementation of a pushdown automaton typically involves designing and coding the transition function, as well as creating the necessary data structures to represent the stack and track the current state. Here is a step-by-step overview of how a pushdown automaton processes an input string:

1. The PDA starts in the start state with an empty stack.

2. It reads the input string symbol by symbol from left to right.

3. For each input symbol, the PDA consults the transition function to determine the next state and the actions to be performed.

4. The PDA may push symbols onto the stack or pop symbols from the stack based on the transition function.

5. If the PDA reaches the end of the input string and the stack is empty, it halts and accepts the input string if the current state is one of the accepting states. Otherwise, it halts and rejects the input string.

The transition function plays a crucial role in the behavior of the PDA. It specifies the next state and the stack operations based on the current state, the input symbol, and the top symbol of the stack. The transition function can be represented as a table, a set of rules, or a function in the implementation.

4.2. Implementations

Pushdown automata (PDA) find practical applications in various areas of computer science and software engineering. These are just a few examples of how pushdown automata are implemented in real-life applications. The versatility of PDAs in recognizing context-free languages makes them valuable in various domains where structured or hierarchical data processing is required.

1. **Programming Language Parsing:** PDAs are widely used in the implementation of compilers and interpreters for programming languages. Compilers often use pushdown automata to perform the syntactic analysis phase, known as parsing. PDAs can efficiently check the syntax of the input program by recognizing the language's context-free grammar. The stack in the PDA helps in maintaining the context and handling nested structures, such as parentheses or brackets.

2. **XML and HTML Parsing:** PDAs are utilized in parsing and processing Extensible Markup Language (XML) and Hypertext Markup Language (HTML) documents. XML and HTML documents are context-free languages, and PDAs are effective in recognizing their structures. The PDA stack helps keep track of nested tags and ensures that the document is well-formed according to the defined grammar rules.

3. **Natural Language Processing:** Pushdown automata can be used in natural language processing tasks, such as syntactic parsing and grammar analysis. PDAs are capable of processing context-free grammars, which are commonly used to represent the syntax of natural languages. By using PDAs, it becomes possible to validate and analyze the syntactic structure of sentences, enabling applications like grammar checkers or language understanding systems.

4. **Protocol Analysis:** PDAs are employed in protocol analysis and verification tools. Network protocols, such as TCP/IP, HTTP, or FTP, can be modeled using context-free grammars. PDAs can be designed to verify the compliance of a network packet or message with the specified grammar rules. This enables the detection of protocol violations, ensuring the integrity and security of network communications.

5. **Syntax Highlighting and Code Editors:** Many text editors and integrated development environments (IDEs) use pushdown automata for syntax highlighting. PDAs are utilized to analyze the code structure based on the language's grammar rules and provide visual cues to developers. This feature enhances code readability and helps in identifying syntax errors or inconsistencies.

6. **Document Processing:** Pushdown automata can be applied to process structured documents, such as XML, JSON, or Markdown files. PDAs can validate the document

structure, extract specific data elements, or perform transformations based on predefined grammar rules. For example, a PDA can be used to parse and extract information from XML-based configuration files.

4.3. Advantages and Disadvantages

Pushdown automata (PDA) offer several advantages and disadvantages in their implementation. One of the key advantages of PDAs is their expressive power. They extend the capabilities of finite automata by incorporating a stack, which allows them to recognize and accept context-free languages. This expressive power is particularly beneficial in areas such as programming language parsing, natural language processing, and document processing. PDAs can effectively handle complex grammatical structures and nested contexts, enabling them to analyze and process a wide range of languages and documents.

Another advantage of PDAs lies in their ability to manage context. The stack in a PDA enables it to keep track of previous states and handle nested structures, such as parentheses, brackets, or tags in markup languages. This feature is particularly valuable in applications like syntax highlighting in code editors or validating the well-formedness of structured documents like XML or HTML. PDAs can ensure that the context is properly maintained and enforce the correct nesting of elements.

Additionally, PDAs provide a formal framework for analyzing and verifying languages and protocols. They can be used in the verification of network protocols, where the PDA can check if network packets or messages comply with the specified grammar rules. This helps ensure the integrity and security of network communications by detecting protocol violations or anomalies.

However, there are also some disadvantages associated with pushdown automata. One limitation is their computational complexity. The acceptance problem for PDAs is PSPACE-complete, which means that deciding whether a given input string is accepted by a PDA can be computationally expensive for certain languages. In practical terms, this can lead to performance challenges when dealing with large inputs or complex grammars.

Another challenge with PDAs is the potential for nondeterminism. PDAs can have multiple possible paths of computation, leading to multiple potential outcomes. Managing nondeterminism can be complex and may require additional algorithms or techniques to ensure the correct behavior of the PDA. Determinizing a PDA can lead to an exponential increase in the number of states, further impacting performance and implementation complexity.

4.4. Problems It Solves

Pushdown automata (PDA) are designed to solve a range of problems related to language recognition and processing. One key problem that PDAs address is the recognition of context-free languages. Context-free languages are characterized by a set of grammar rules that define their syntax. PDAs provide a formal framework for determining whether a given input string belongs to a particular context-free language. They can effectively analyze the syntactic patterns and structures of languages, making them invaluable in areas such as programming language parsing, where the correct interpretation of code relies on understanding its context-free grammar.

Another problem that PDAs help solve is the management of nested structures. Many languages and document formats, such as XML and HTML, require the handling of nested tags or elements. PDAs, with their stack-based memory structure, excel at tracking and managing these nested structures. They can ensure that the opening and closing tags or brackets are correctly balanced and nested, thereby guaranteeing the well-formedness of documents and facilitating their further processing or manipulation.

4.5. Shortages and Drawbacks

Pushdown automata (PDA) come with certain limitations and shortcomings that can impact their implementation and usage.

One significant drawback of PDAs is their computational complexity. The acceptance problem for PDAs is known to be PSPACE-complete, which means that determining whether a given input string is accepted by a PDA can be computationally expensive for certain languages. As the complexity of the language or the grammar increases,

the performance of a PDA may suffer. This can result in longer execution times or resource-intensive computations, particularly when dealing with large inputs or complex grammars. As a result, optimizing the performance of PDAs for certain scenarios can be a challenging task.

Another limitation of PDAs is the potential for nondeterminism. PDAs can have multiple possible paths of computation at any given point, leading to multiple potential outcomes. While nondeterminism can increase the expressive power of PDAs, managing and resolving it can be complex. Nondeterministic behavior can require additional algorithms or techniques, such as nondeterministic simulation or converting a nondeterministic PDA to a deterministic one. Determinizing a PDA may result in an exponential increase in the number of states, which can impact both the implementation complexity and the performance of the PDA.

Furthermore, PDAs have a specific focus on context-free languages. While PDAs can effectively handle the syntactic structures and nested contexts present in context-free languages, they are not capable of recognizing or processing languages with more complex grammar types. For example, they cannot handle languages that require context-sensitivity or context-free languages with recursive or ambiguous rules. This limitation restricts the applicability of PDAs in certain language processing scenarios.

5. APPLICATION INTERFACE

The application starts by letting the user choose which mode he wants to utilize from the 2 available modes:

1- **NFA to DFA convertor**, converts any non-deterministic finite automata to a deterministic one.
2- **CFG to PDA convertor**, converts context free grammar to pushdown automata.

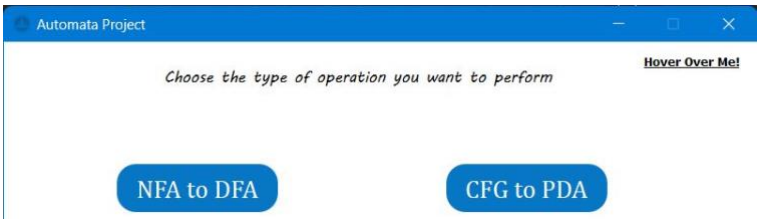

Figure 7: Application Intro Screen

6. NFA to DFA

The first mode of our built application, fully built and implemented using python language, converts any non-deterministic finite automata (ϵ-NFA or NFA) to deterministic form (DFA).

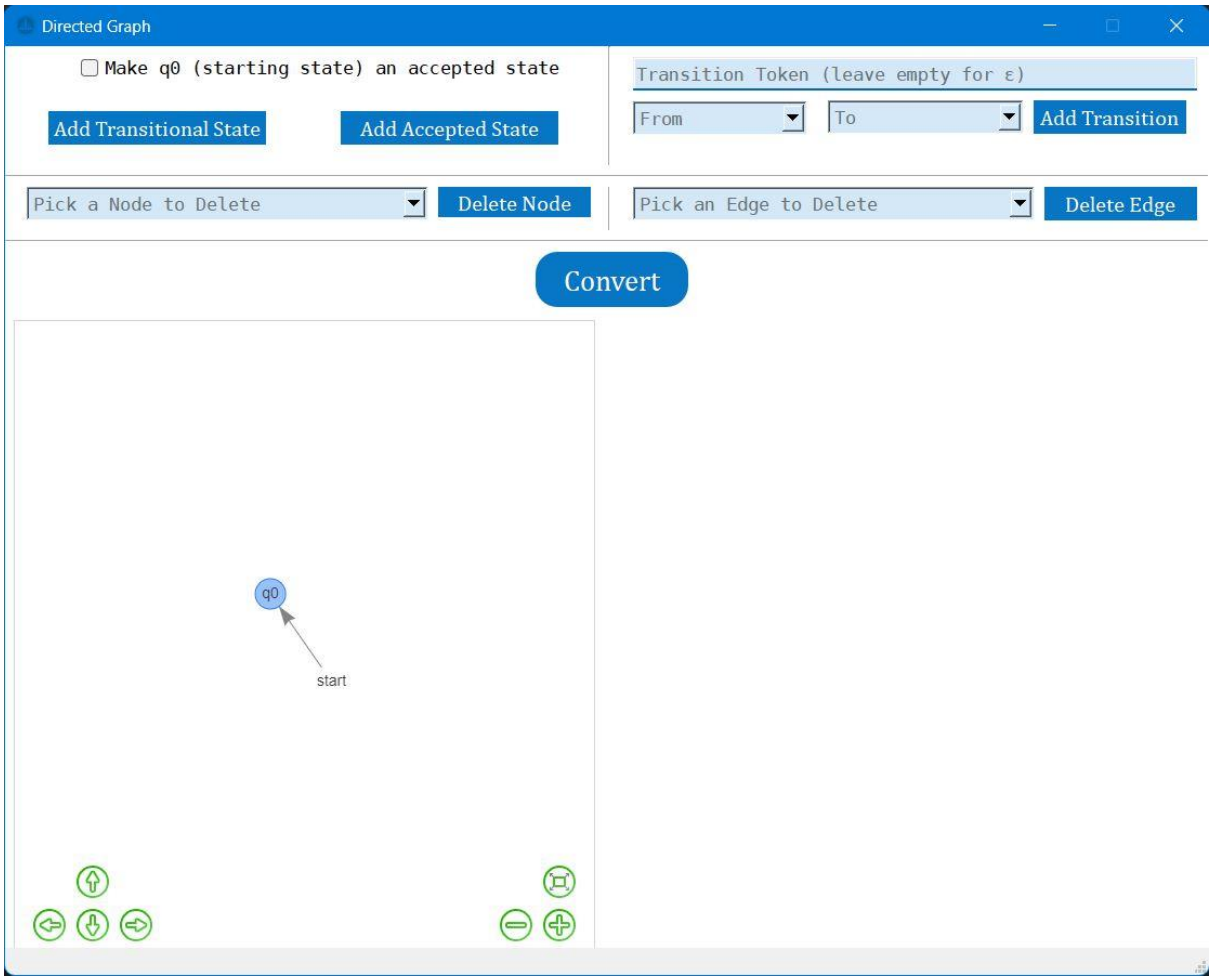

Figure 8: NFA to DFA GUI Screen

6.1. User Input

User fully describes the NFA he wants to convert through the application interface. He inputs states, whether acceptance (final), or transitional (non-final), in addition to any transition (edge) the NFA has.

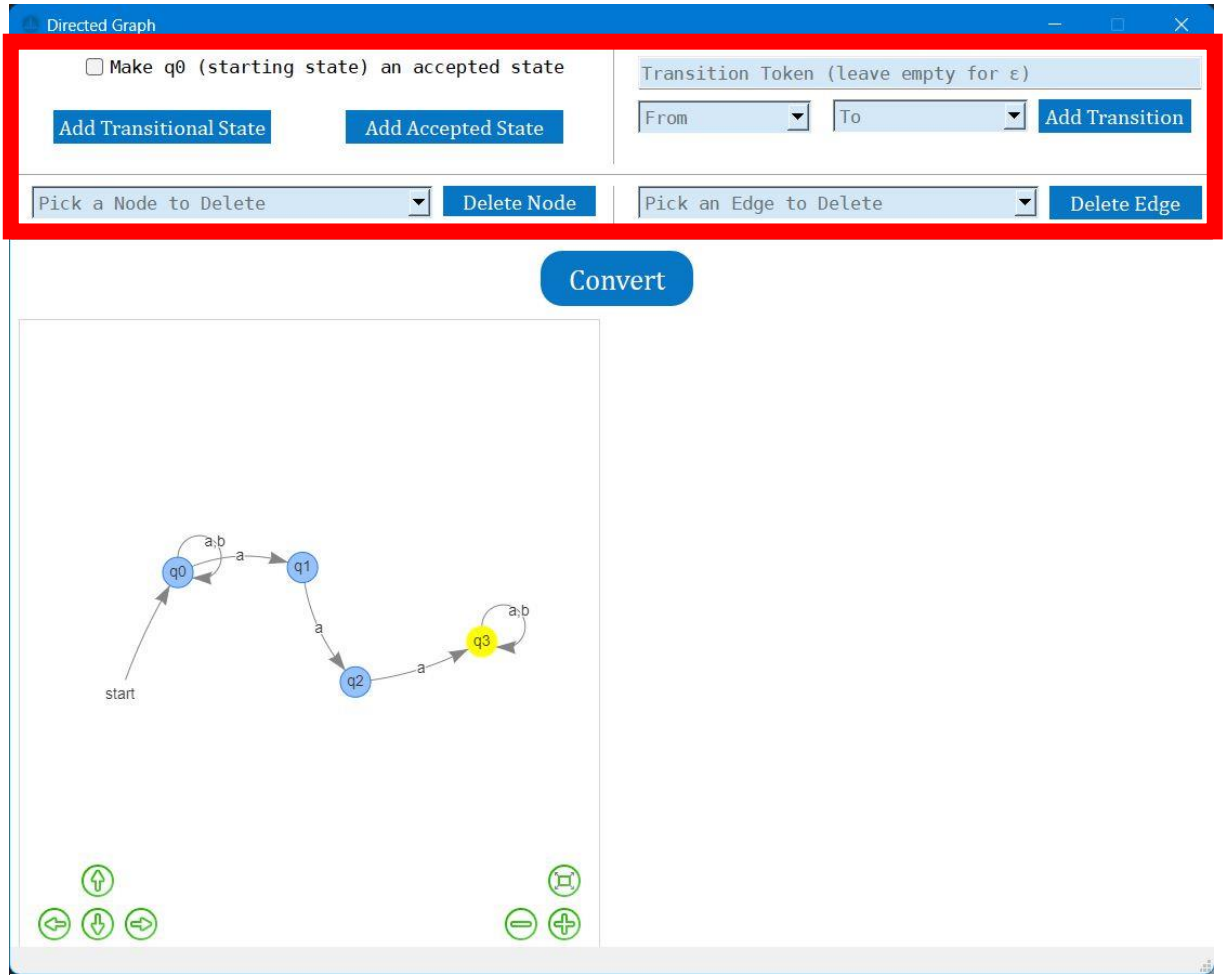

Figure 9: How user inputs NFA

6.2. How It works

The code in “NFA2DFA.py” file follows a sequence of steps to process the user input and convert it to DFA before viewing the output on the user’s screen again.

Before the main conversion function, we implement a helper function, `getClosure()`, that gets the epsilon closure of each node and returns the list of reachable nodes from it.

```
# gets closure for a node (in epsilon-nfa)
# returns new merged nodes
# helps convert e-NFA to NFA
def getClosure(self, node):
    nodes = {node}
    nodes2disc = [node]
    for curn in nodes2disc:
        for n in self.AdjLi[curn]:
            try:
                if (not {n}.issubset(nodes)) and (self.Edges.index({'from':curn, 'to':n, 'cost':"ε"}) != None):
                    nodes.add(n)
                    nodes2disc.append(n)
            except:
                pass
    print(nodes)
    return nodes
```

Afterwards, we implement our main conversion function, `onClickConv()`, that does the full function.

1. The code initializes two lists: `NFAnodes` and `NFAedges`. These lists will store the nodes and edges of the NFA.

```
def onClickConv(self):
    NFAnodes = []
    NFAedges = []
```

2. The first loop iterates over each node (`n`) in `self.Nodes`. It retrieves the closure of the node's name using the `getClosure()` method. The closure is a set of nodes that can be reached from the current node.

3. The code then filters the `NFAnodes` list to check if any node matches the closure obtained in the previous step. If there are no matching nodes, it proceeds to determine if the closure contains any goal nodes. If it does, the `goal` variable is set to `True`. Finally, a new node is appended to `NFAnodes` with the name of the closure, the goal value, and the main node name.

```
# converts e-NFA to NFA
# get eps closure of each node, checks if final state, append all to NFAnodes array
for n in self.Nodes:
    clos = self.getClosure(n["name"])
    Nos = list(filter(lambda nod: (nod['name'] == clos), NFAnodes))
    if len(Nos) == 0:
        goal = False
        for no in clos:
            gN = list(filter(lambda nod: (nod['name'] == no) and (nod['goal'] == True), self.Nodes))
            if len(gN) > 0:
                goal = True
        NFAnodes.append({"name":clos ,"goal":goal, "main":n['name']})
```

4. After completing the first loop, the code moves on to connect the NFA nodes and create NFA edges. It iterates over each node in `NFAnodes` and retrieves the individual nodes from the closure. It then filters the edges of the original graph (`self.Edges`) to find edges that match the current node. If a matching edge is found, it further filters the `NFAnodes` list to find the corresponding destination node. Finally, a new edge is appended to `NFAedges` with the source node, destination node, and edge cost.

```
# connect NFA nodes, creating NFA edges
for no in NFAnodes:
    for n in no["name"]:
        nE = list(filter(lambda edge: (edge['from'] == n) and (edge['cost'] != "ε"), self.Edges))
        for edge in nE:
            for nod in NFAnodes:

                if edge["to"] == nod["main"]:
                    NFAedges.append({"from":no["name"], "to":nod["name"], "cost":edge["cost"]})
```

5. The next part of the code converts the NFA to a DFA. It initializes two lists: `DFAnodes` (stores DFA nodes) and `DFAedges` (stores DFA edges). The first node of the NFA (`NFAnodes[0]`) is added as the starting state of the DFA.

```
DFAnodes = [NFAnodes[0]] # add starting state of NFA to DFA
DFAedges = []
```

6. The outer loop iterates over each node in `DFAnodes` to build the DFA. It initializes two lists: `nN` (stores nodes reachable from the current node) and `nE` (stores the original edges of the current node).

```
# outer loop
for node in DFAnodes:
    nN = [] # nodes reachable from current node
    nE = [] # current node original edges
```

7. Within the outer loop, the code iterates over each original edge (`edge`) of the current node and prints the source node, destination node, and edge cost.

8. The code then filters the `nN` list to check if any node has the same cost as the current edge. If there are no matches, it appends a new node (dictionary) to `nN` with the destination node as a set and the edge cost. If there is a match, it merges the destination node with the existing node in `nN`.

```
for n in node["name"]:
    nE += list(filter(lambda edge: (edge['from'] == n) and (edge['cost'] != "ε"), self.Edges))
for edge in nE:
    print(str(node['name'])+"$"+str(edge['to'])+"$"+str(edge['cost']))
    fN = list(filter(lambda no: (no['cost'] == edge['cost']), nN))
    if len(fN) == 0:
        nN.append({'name':{edge['to']}, 'cost':edge['cost']})
    else: # adds new node if first appearance
        nN[nN.index(fN[0])]['name'] = nN[nN.index(fN[0])]['name'].union({edge['to']})
```

9. Next, the code filters the `NFAnodes` list to find the NFA node corresponding to the current DFA node. If a matching node is found, it retrieves the corresponding NFA edges and iterates over them.

```
nfN = list(filter(lambda no: (no['name'] == node['name']), NFAnodes))
if len(nfN) > 0:
    nfE = list(filter(lambda edge: (edge['from'] == nfN[0]['name']), NFAedges))
    for edge in nfE:
        print(str(node['name'])+"$"+str(edge['to'])+"$"+str(edge['cost']))
        fN = list(filter(lambda no: (no['cost'] == edge['cost']), nN))
        if len(fN) == 0:
            nN.append({'name':edge['to'], 'cost':edge['cost']})
        else:
            nN[nN.index(fN[0])]['name'] = nN[nN.index(fN[0])]['name'].union(edge['to'])
```

10. Similar to step 8, the code filters the `nN` list to check for matching nodes based on the edge cost. If there is no match, it appends a new node to `nN` with the

destination node and edge cost. If there is a match, it merges the destination node with the existing node in `nN`.

11. After building the `nN` list, the code iterates over each node (`n`) in `nN`. It filters the `DFAnodes` list to check if the current node already exists. If the current node (`n`) does not exist in the `DFAnodes` list, the code proceeds to determine if the node contains any goal states. It iterates over each individual node within the current node's name and checks if there is a matching node in the original graph (`self.Nodes`) with the same name and a goal value of `True`. If such a node is found, the `goal` variable is set to `True`.

```
for n in nN:
    dfN = list(filter(lambda no: (no['name'] == n['name']), DFAnodes))
    if len(dfN) == 0:
        goal = False
        for nn in n['name']:
            gN = list(filter(lambda nod: (nod['name'] == nn) and (nod['goal'] == True), self.Nodes))
            if len(gN) > 0:
                goal = True
```

12. After determining the goal value, a new node is appended to the `DFAnodes` list, containing the name of the current node (`n['name']`) and the goal value.

```
DFAnodes.append({'name':n['name'], 'goal':goal})
```

13. Finally, the code appends a new edge to the `DFAedges` list, representing the transition from the current DFA node (`node['name']`) to the current node (`n['name']`) with the corresponding edge cost (`n['cost']`).

```
DFAedges.append({"from":node['name'], "to":n["name"], "cost":n["cost"]})
```

The code essentially performs the conversion of an NFA to a DFA by exploring the reachable nodes and their transitions. It constructs a new set of DFA nodes (`DFAnodes`) and edges (`DFAedges`) based on the connections and costs of the original NFA.

6.3. Output Form

After conversion, the DFA nodes and edges are viewed to the user in graphical representation.

6.4. Sample Outputs

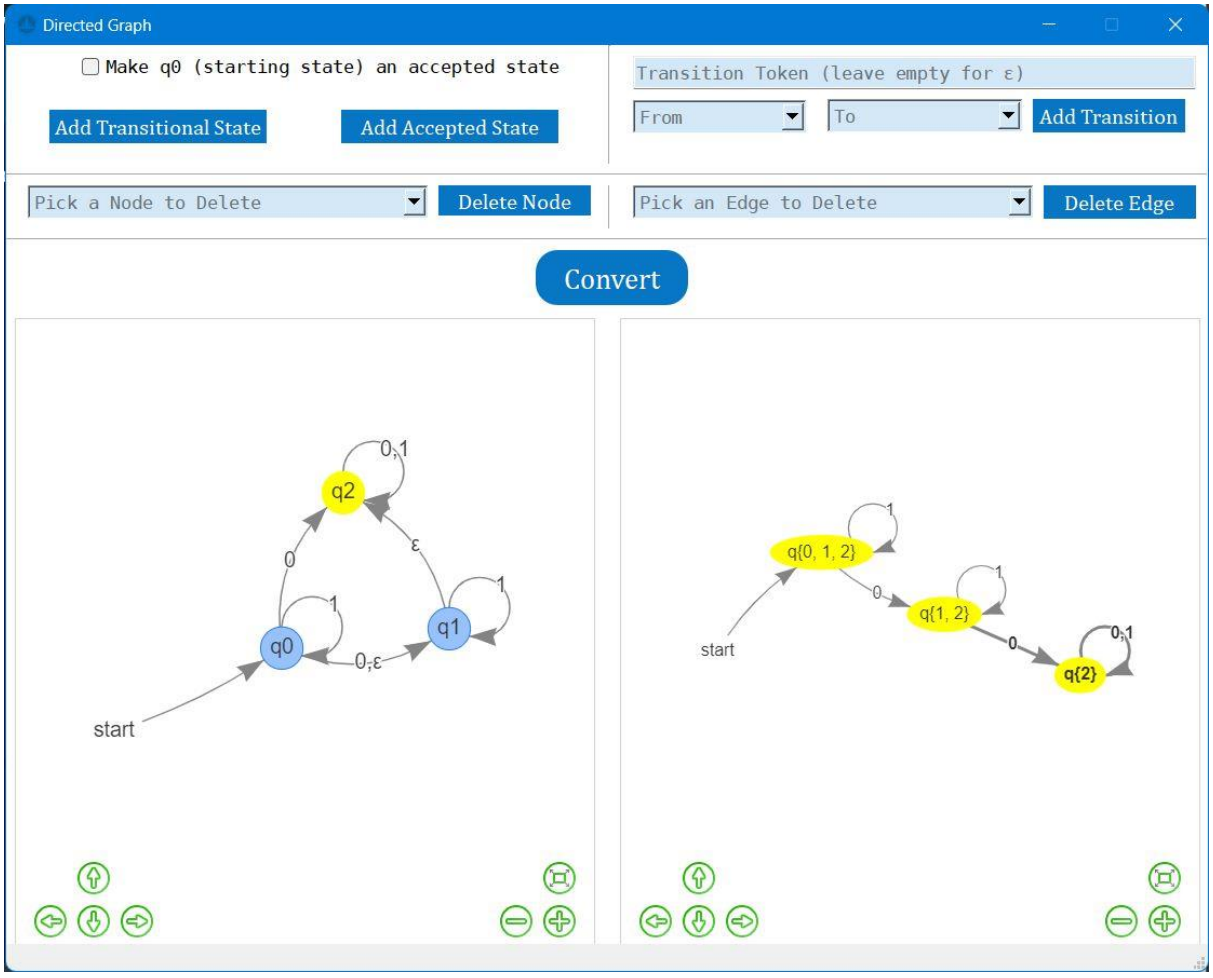

Figure 10: epsilon-NFA to DFA example 1

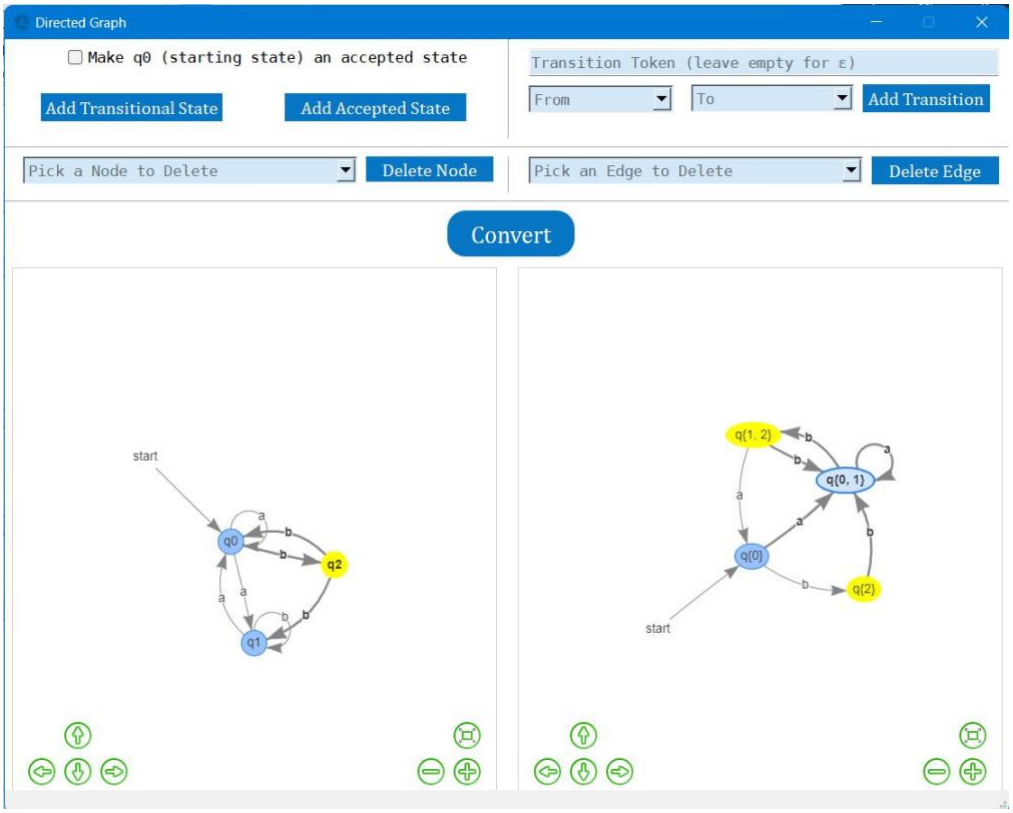

Figure 11: NFA to DFA example 2

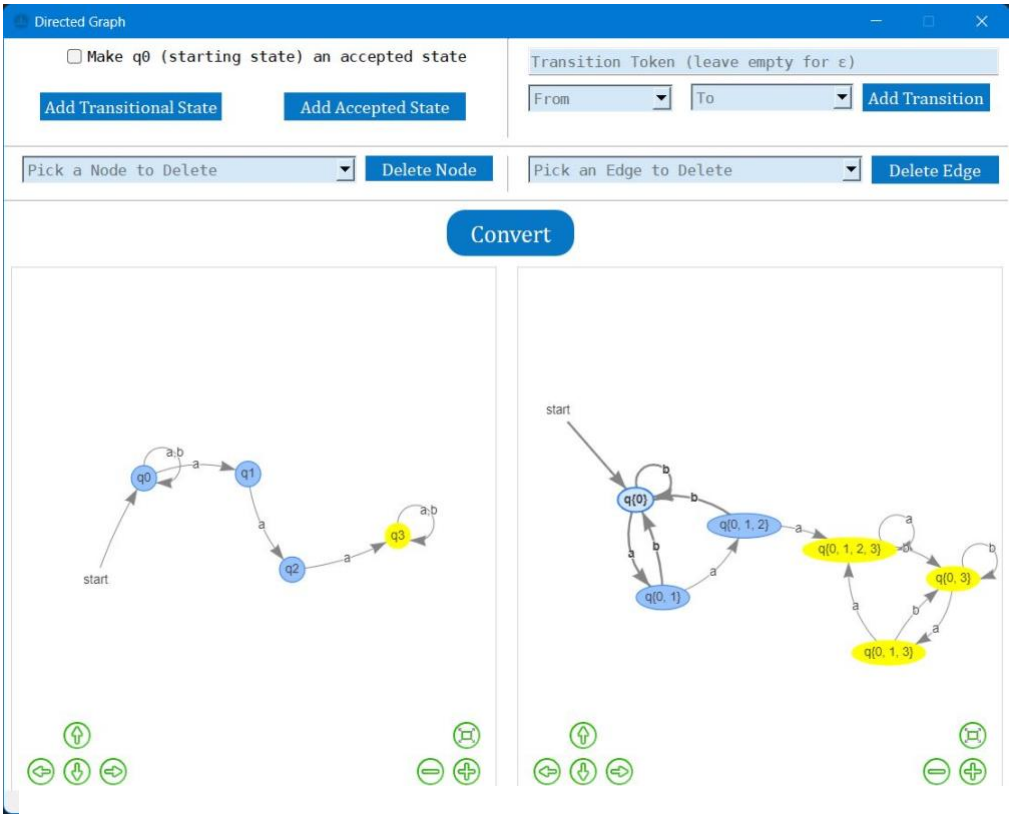

Figure 12: NFA to DFA example 3

7. CFG TO PDA

The second mode of our application converts context free grammar rules to pushdown automata. It takes context free grammar rules from the user and convert it using the implemented algorithm to PDA.

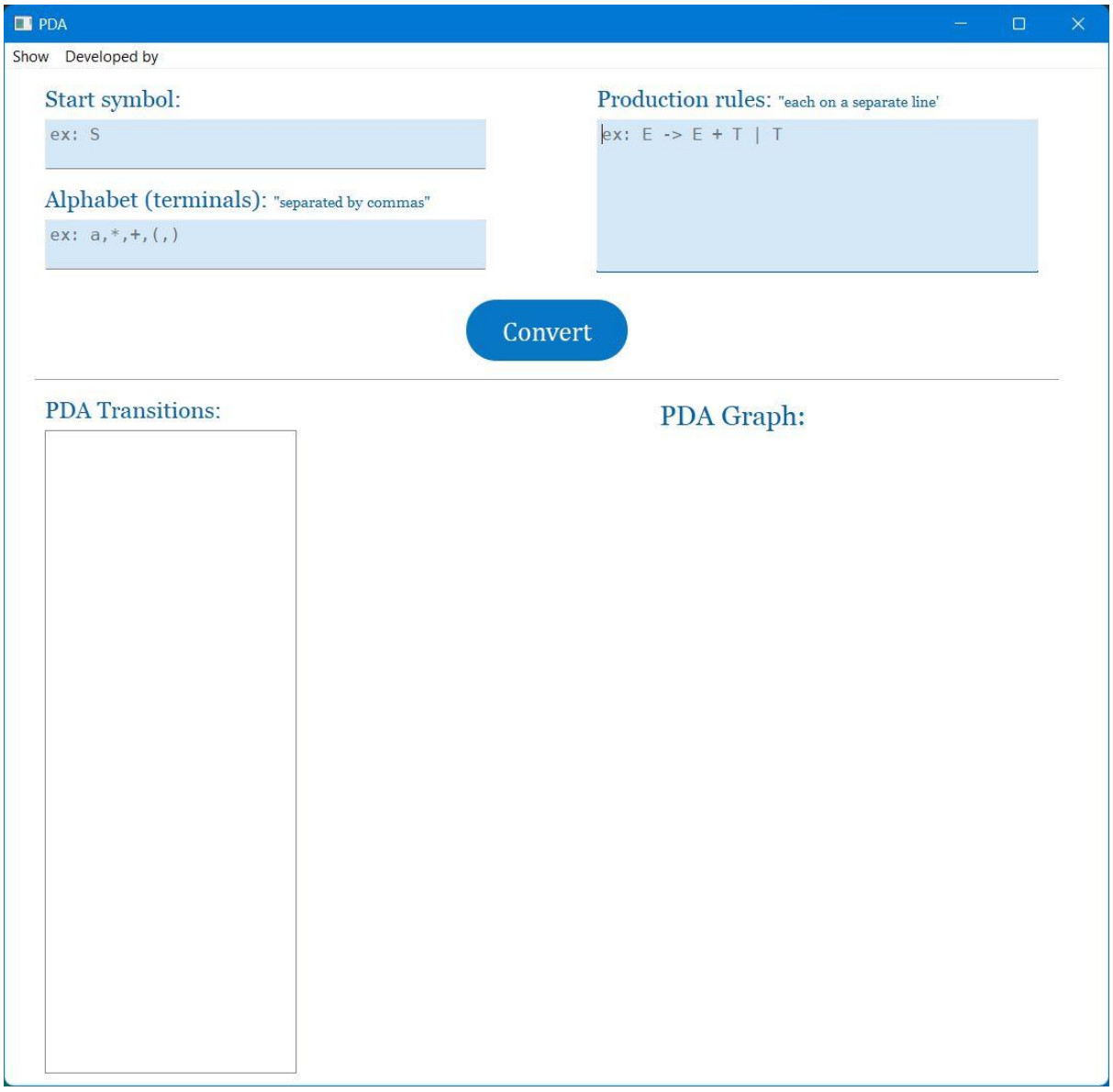

Figure 13: CFG to PDA GUI Screen

7.1. User Input

User enters the context free grammar he wants to convert through the application interface. He inputs the production rules, the alphabet, and the starting non-terminal.

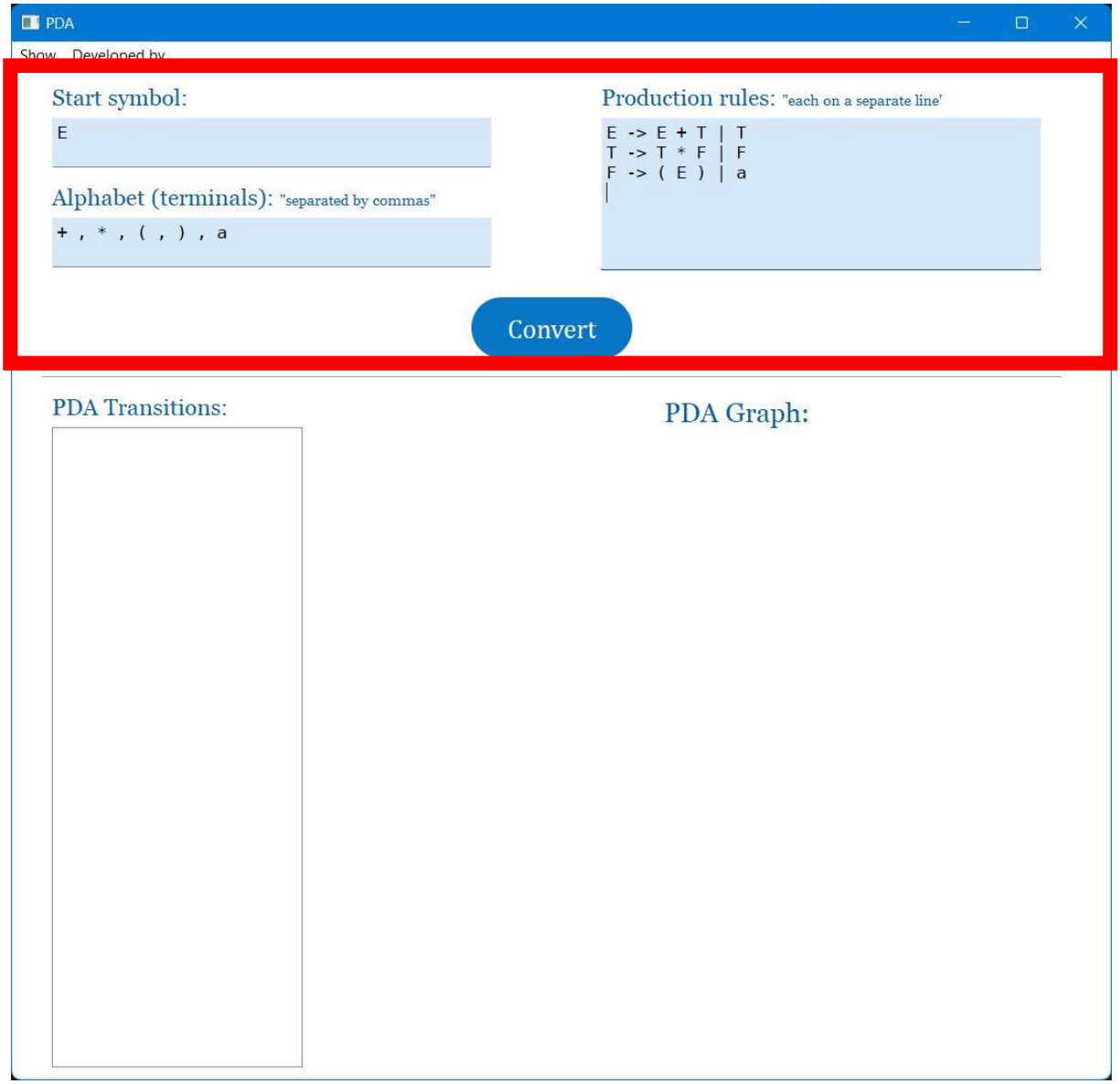

Figure 14: How user inputs CFG

7.2. How It works

The code in “PDA.py” file follows a sequence of steps to process the user input and convert it to PDA before viewing the output on the user’s screen again.

All the logic is implemented in one function, `getStatesTransitions()` in addition to 2 Classes, `State` and `Transition` that are implemented as follows.

The class `State` defines a state that has a name, and whether it is an initial, final state.

```
class State:
    name = ""
    isInitial = False
    isFinal = False

    def __init__(self, name, isInitial, isFinal):
        self.name = name
        self.isInitial = isInitial
        self.isFinal = isFinal

    def setFinal (self, isFinal):
        self.isFinal = isFinal

    def setInitial (self, isInitial):
        self.isInitial = isInitial

    def __str__(self):
        string = self.name
        return string
```

The class `Transition` defines the input symbol, states that the transition goes from and to, in addition to pop and push symbols.

```
class Transition:
    inputSymbol = ''
    currState = ''
    nextState = ''
    popSymbol = ''
    pushSymbol = ''

    def __init__(self, inputSymbol, currState, nextState, popSymbol, pushSymbol):
        self.inputSymbol = inputSymbol
        self.currState = currState
        self.nextState = nextState
        self.popSymbol = popSymbol
        self.pushSymbol = pushSymbol

    def __str__(self):
        string = self.inputSymbol + ", " + self.popSymbol + " → " + self.pushSymbol
        return string
```

1. The main function `getStatesTransitions()` starts by defining 4 main states (q_0, q_s, q_l, q_f) as (q_0, q_1, q_2, q_3) that are in every PDA in addition to the 3 transitions connecting them.

```
def getStatesTransitions(self):
    # 4 initial main nodes
    self.states = [ State('q0', True, False),
                    State('q1', False, False),
                    State('q2', False, False),
                    State('q3', False, True) ]

    # 3 main transitions hard coded
    self.transitions = [ Transition("ε", 'q0', 'q1', "ε", "$"),
                         Transition("ε", 'q1', 'q2', "ε", self.startSymbol),
                         Transition("ε", 'q2', 'q3', "$", "ε") ]
```

2. It retrieves all the rules that the user inputs and divide them to unit rules that have no “OR” in the right-hand side of the rule.

```
for rule in self.rules:
    # parse every rule in every line into unit rules (no ORs)
    rule = rule.strip().replace(' ', '')
    [lhs, rhs] = rule.split('->')
    rhs = rhs.split('|')
    for term in rhs:
        unit_rules.append((lhs, term))
```

3. For every unit rule, create the needed **n** states and **n-1** transitions to push every symbol on the right-hand side of the rule into the stack, where **n** is the number of terminal and non-terminals in the rule. The first transition of every rule must start at state “q2”, while the last transition must end at “q2” as well.

```
for rule in unit_rules:
    lhs, rhs = rule[0], rule[1]
    rhs = rhs[::-1] # Necessary for the terminals to pushed in correct order
    # always starts from q2
    start = 'q2'
    # push into stack from left to right
    for i, char in enumerate(rhs):  #####!!!!!!! rhs.reverse()
        poped = lhs if i==0 else "ε" # pop on first transition only, else pop epsilon
        # adds n-1 states
        # adds n transitions
        # n = number of elements on RHS
        if i==len(rhs)-1: # last element must transition back to q2
            trans = Transition("ε", start, 'q2', poped , rhs[i])
        else:
            end = 'q'+str(len(self.states))
            state = State(end, False, False)
            self.states.append(state)
            trans = Transition("ε", start, end, poped , rhs[i])
            start = end

        self.transitions.append(trans)
```

4. Lastly, create transitions for all non-terminals of the alphabet that start and end at “q2” and pops from the stack.

```
for terminal in self.terminals: # pops terminal
    self.transitions.append(Transition(terminal, 'q2', 'q2', terminal, 'ε'))
```

7.3. Output Form

After conversion, the DFA nodes and edges are viewed to the user in graphical representation as a graph in addition to a transition list in a separate column.

7.4. Sample Outputs

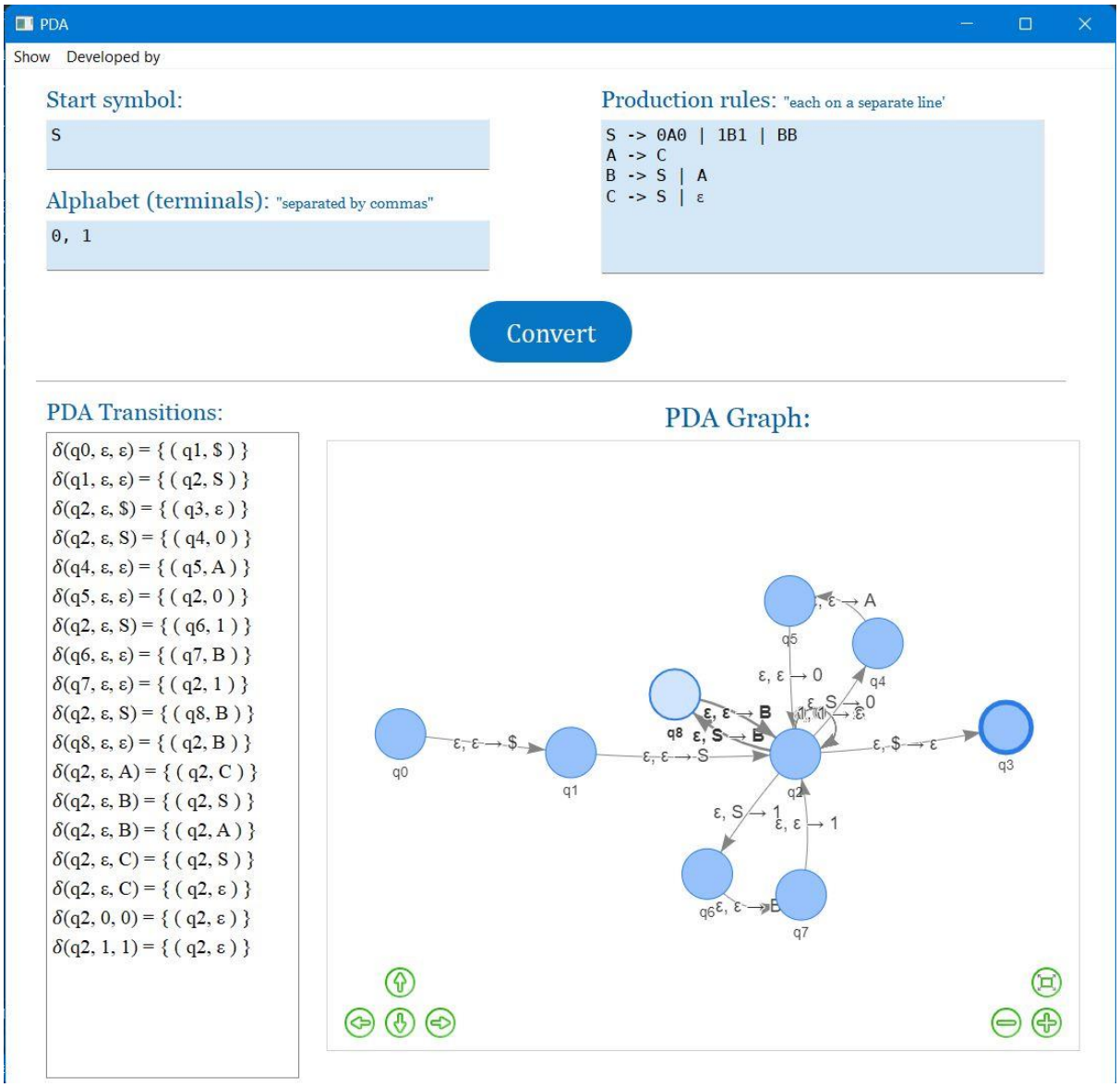

Figure 15: CFG to PDA example 1

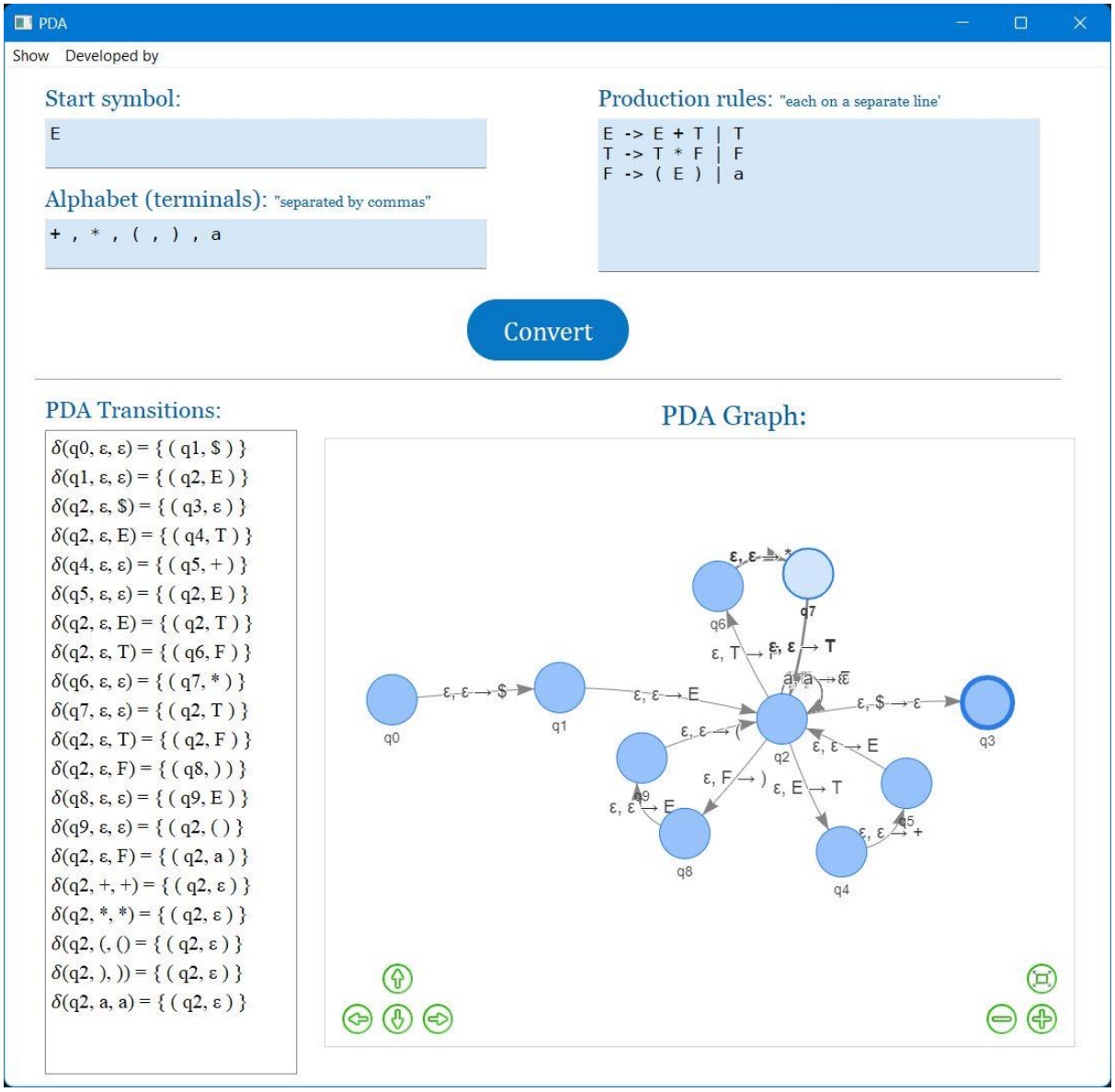

Figure 16: CFG to PDA example 2

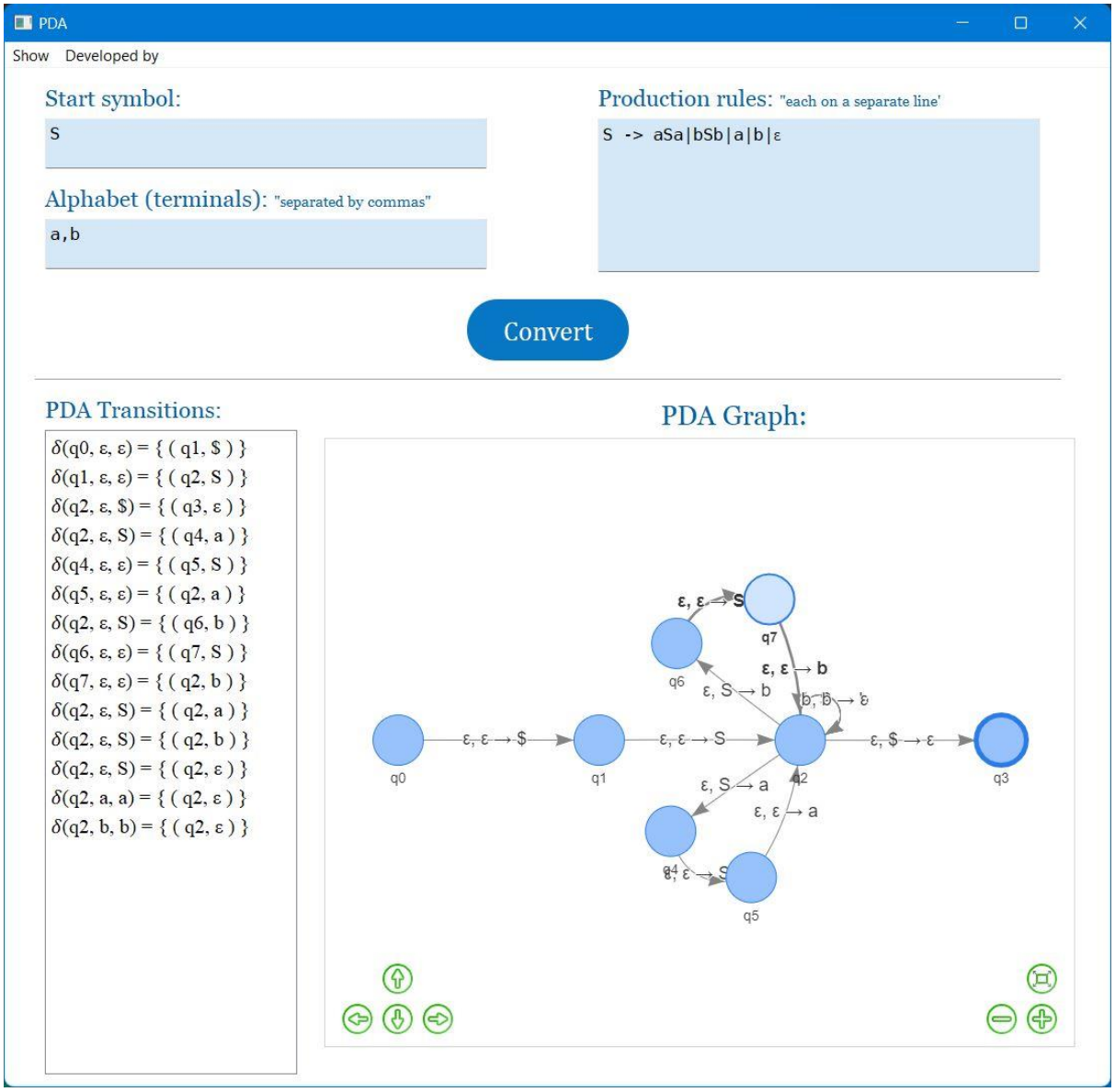

Figure 17: CFG to PDA example 3