Project Report

Theory of Automata

Push-down Automata (PDA)Simulator

Team Members:
Mudasir Mujtaba Rana (15765)
Abdur Rehman (15715)
Instructor Name : Miss Misbah Anwer
Class ID: 118692

December 29, 2025

1 Introduction

1.1 Background

Push down automata (PDA) plays a vital role in computer science, particularly in compiler construction. It is a core topic which is used to recognize the context free language. It is the extension of finite automata by introducing a stack. The stack in push down automata enables to manage recursive patterns as it is basically a finite automata but with a memory and stack is the memory used in push down automata. Previously the work done on this simulator was that it only was generating transition function and transition of the few (pda) languages and it was all console-based work which made more difficult to understand the parsing of the expression.

1.2 Motivation

The motivation for this project was to properly learn push down automata (PDA) through the simulation and workflow of the PDA by completely visualizing the each step or transition made made by the PDA and also the changes implemented on the stack at every transition when any string is accepted by an language.As the learning theory part of the push down automata can be confusing and time consuming.So this project fills the gap of understanding between theory practical implementation.

1.3 Significance

There are some key contributions which make this simulator significant such as,The simulator has an clear visualization as it provides an effective learning for students by enabling students to observe real time stack operation which provides better understanding of PDA.Also it works on the multiple language which are pre-defined but it demonstrates correct implementations of eight different context-free languages (CFL), serving as a reference for both students and the teacher.

Utilizing the latest python web framework (Flask) and modern day (java script) which ensures it to be maintainable and extensible.

2 Related Work

Push down automata (PDA) is an important part in automata and Many online tutorials and lecture explain push down automata (PDA) definitions and transition rules.There are some simulators available online which simulates on behalf of the user input like JFLAP which provides PDA simulation but is not friendly for beginner as it requires installation and a proper guide to use it or like FSM Simulator which only focuses on Finite Automata lacking the stack based context free languages and etc.

The focus of this project is support multiple PDA languages in a single platform which makes it more suitable for learning PDA.

3 Methodology

3.1 System Architecture

The system follows a client-server architecture.The backend is developed using python where each PDA is defined using states, transition and stack. The user send input from front-end to backend which perform the simulation and return the result. It show step by step PDA behavior.

The Backend (Flask/Python) Handles PDA definitions and simulation logic.While the Front-end (HTML/CSS/JavaScript) Manages user interface, visualization, and interaction

3.2 PDA Representation

Each PDA is represented as a 5-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where:

Q is a finite set of states Σ is the input alphabet δ is the transition function $q_0 \in Q$ is the start state $F \subseteq Q$ is the set of accept states Γ is the stack alphabet $Z_0 \in \Gamma$ is the initial stack symbol

In the implementation, transitions are stored as a dictionary mapping tuples (q, a, X) to lists of possible next configurations (q', γ), where γ is the string to push onto the stack.

3.3 Simulation Algorithm

The simulator implements breadth-first search (BFS) to explore all possible non-deterministic computation paths which lies in the block of code of a python function def simulate(npda):

Algorithm 1 Non-deterministic PDA Simulation

```
Initialize queue with (q_0, 0, Z_0)
Initialize visited set
while queue is not empty do
    (state, position, stack) ← dequeue()
    if state ∈ F and position = |input| and stack = Z_0 then
        return ACCEPT with trace
    end if
    top ← stack[0]
    symbol ← input[position]
    for each transition δ(state, symbol, top) or δ(state, ϵ, top) do
        Compute new configuration
        if configuration not visited then
            Add to visited set
            Enqueue with updated trace
        end if
    end for
end while
return REJECT
```

This approach ensures that if any computation path leads to acceptance, the PDA accepts the input string.

3.4 Implemented Languages

The system implements eight context-free languages:

1. **Balanced Parentheses**: $\{w \in \{(,)\}^* | w \text{ is balanced}\}$
2. **No of a's followed by same No. of b's** $a^n b^n$: $\{a^n b^n | n \geq 0\}$
3. **Palindromes**: $\{w \in \{a, b\}^* | w = w^R\}$
4. $(a + b)^*$: $\{w \in \{a, b\}^*\}$
5. **No of a's followed by No of b's with optional occurence of c** $a^n b^n c^*$: $\{a^n b^n c^m | n \geq 0, m \geq 0\}$
6. **Mirror Strings** (ww^R): $\{ww^R | w \in \{a, b\}^*\}$
7. $a^{2n} b^n$: $\{a^{2n} b^n | n \geq 0\}$
8. $a^n b^m c^{n+m}$: $\{a^n b^m c^{n+m} | n, m \geq 0\}$

3.5 Visualization Components

Three synchronized visualization components are implemented in the front-end:

1. **State Diagram**: An SVG-based representation of the PDA's state transition graph that highlights the active state

2. **Stack Visualization**: Push and pop operations are color-coded and the contents of the stack are represented vertically.

3. **Execution Trace**: A list of all configurations visited and parsed during computation that can be scrolled

4 Experiments and Results

4.1 Implementation Details

The system was implemented using: Python 3.8+ with Flask 2.0 and JavaScript works as an backend while HTML and CSS works for front-end and for creating the state diagram SVG is used for dynamic diagram rendering.

Total lines of code: approximately 800 lines (Python: 350, JavaScript: 300, HTML/CSS: 150).

4.2 Test Cases and Validation

Comprehensive testing was performed for each implemented language:

4.2.1 Balanced Parentheses

Input	Expected	Result
()	Accept	Pass
(())	Accept	Pass
()(())	Accept	Pass
(()	Reject	Failed
)(	Reject	Failed

4.2.2 $a^n b^n$ Language

Input	Expected	Result
ϵ	Rejected	Failed
ab	Accept	Pass
aabb	Accept	Pass
aaabbb	Accept	Pass
aab	Reject	Failed
abb	Reject	Failed

4.2.3 Palindrome vs Mirror String Comparison

To get the better Understanding we properly understand-ed the difference between the mirror strings and the palindrome

Input	Palindrome PDA	Mirror String PDA
aba	Accept	Reject
abba	Accept	Accept
a	Accept	Reject
aa	Accept	Accept
abcba	Accept	Reject

This demonstrates that all mirror strings are palindromes, but odd-length palindromes are not mirror strings.Mirror strings can be said that they are the even-length palindrome.

4.2.4 $a^{2n}b^n$ Language

Input	Expected	Result
ϵ	Reject	Fail
aab	Accept	Pass
aaaabb	Accept	Pass
ab	Reject	Fail
aaab	Reject	Fail

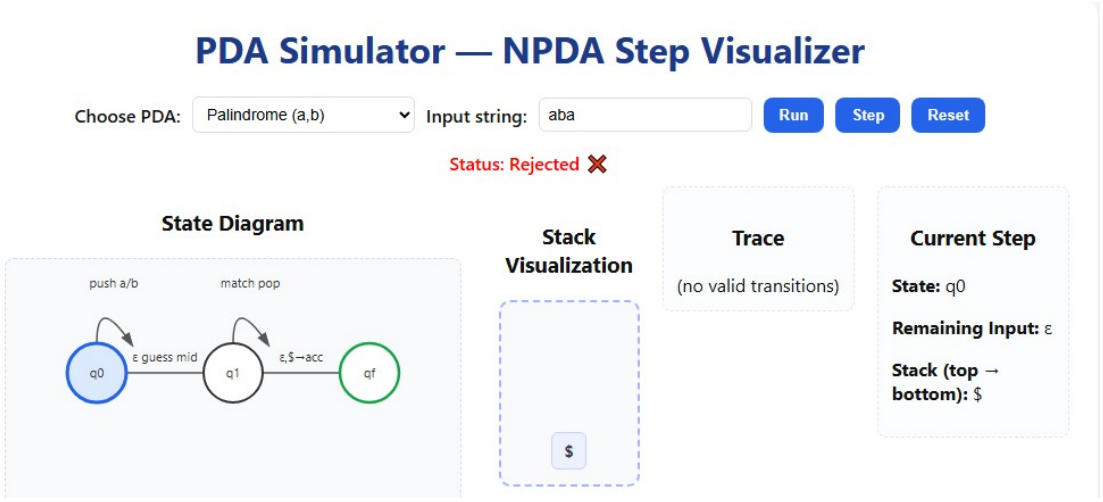
4.2.5 $a^n b^m c^{n+m}$ Language

Input	Expected	Result
abc	Reject	Fail
aabccc	Accept	Pass
abbcccc	Reject	Fail
aaabbbbccccccc	Accept	Pass
abc	Reject (needs 2 c's)	Fail

4.3 Debugging Insights

Several bugs were identified and resolved during development:

1. **Palindrome Issue**: Initial implementation failed on odd-length palindromes like "aba".



   Solution: Added non-deterministic transitions that simultaneously push and skip middle character.

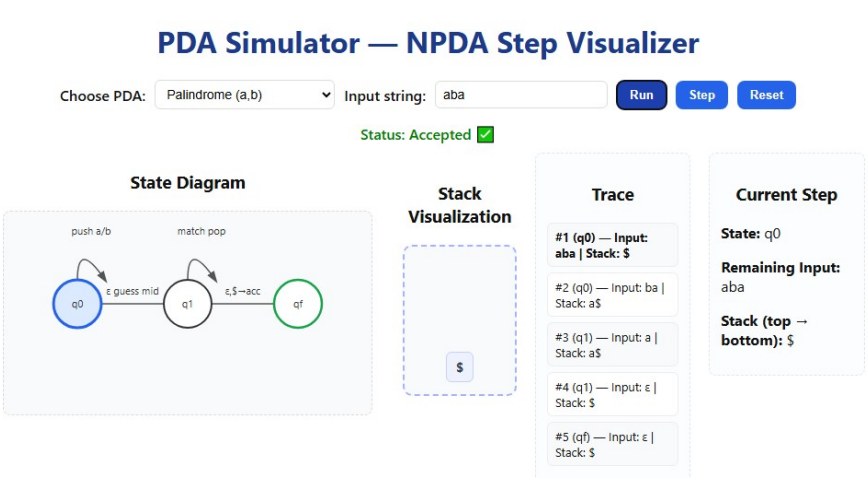
```python
def pda_palindrome():···
    Z = "$"
    t = {}

    # Phase 1: Push symbols onto stack (reading left side)
    # Each transition can either push OR move to matching phase (for odd-length)
    t[("q0", "a", Z)] = [("q0", "a" + Z), ("q1", Z)]  # push OR skip middle 'a'
    t[("q0", "b", Z)] = [("q0", "b" + Z), ("q1", Z)]  # push OR skip middle 'b'
    t[("q0", "a", "a")] = [("q0", "aa"), ("q1", "a")]  # push OR skip middle 'a'
    t[("q0", "a", "b")] = [("q0", "ab"), ("q1", "b")]  # push OR skip middle 'a'
    t[("q0", "b", "a")] = [("q0", "ba"), ("q1", "a")]  # push OR skip middle 'b'
    t[("q0", "b", "b")] = [("q0", "bb"), ("q1", "b")]  # push OR skip middle 'b'

    # Phase 2: Non-deterministically guess we've reached middle (even-length)
    t[("q0", "", Z)] = [("q1", Z)]
    t[("q0", "", "a")] = [("q1", "a")]
    t[("q0", "", "b")] = [("q1", "b")]

    # Phase 3: Match input with stack (pop matching symbols)
    t[("q1", "a", "a")] = [("q1", "")]
    t[("q1", "b", "b")] = [("q1", "")]

    # Phase 4: Accept if we've consumed all input and stack is empty
    t[("q1", "", Z)] = [("qf", Z)]
```



Issue Resolved

2. **Stack Symbol Confusion**: The $a^n b^m c^{n+m}$ PDA initially used only one stack symbol.

   Solution: Implemented separate A and B symbols to track a's and b's independently.

3. **Acceptance Condition**: Early versions accepted with non-empty stacks. Solution: Modified to require stack contains only start symbol at acceptance.

5 Conclusion

This project is an interactive web-based PDA simulator that successfully illustrates non-deterministic computation for context-free languages CFL through synchronized state diagrams, stack animations, and execution traces, to offer a extensive educational value while demonstrating accurate implementations of eight key languages.

5.1 Key Achievements

1. Implemented a reliable BFS-based simulation algorithm that handles non-deterministic transitions.
2. developed a user-friendly visual interface that can be accessed through web browsers without installation.
3. Verified the accuracy using comprehensive test cases for several language classes
4. Documented the differences between related languages (palindromes vs. mirror strings)

5.2 Limitations

Several limitations exist in the current implementation:

1. **Fixed PDA Set**: Users cannot define custom PDAs; only pre-programmed languages are available
2. **Performance**: Large inputs (¿100 characters) can cause slow visualization due to state space explosion
3. **Limited Feedback**: The system shows only whether input is accepted/rejected, not why certain paths fail

5.3 Future Work

In future we can further enhance this simulator by making it as a :

- **Custom PDA Editor**: Allow users to define transitions graphically or via formal notation
- **Multi-path Visualization**: Display all computation paths simultaneously in a tree structure
- **Conversion Tools**: Implementing the conversion of CFG to PDA and vice versa
- **Step Animation**: Add smooth transitions between states with configurable speed
- **Export Functionality**: Enable saving execution traces or diagrams as images or in the form of pdfs.
- **Educational Mode**: Include hints, explanations, and interactive quizzes

References

[1] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Boston, MA: Addison-Wesley, 2001. This foundational textbook is often called the "Cinderella Book" by students and has shaped automata theory education for decades. Available at: `http://infolab.stanford.edu/~ullman/ialc.html`

[2] M. Sipser, *Introduction to the Theory of Computation*, 3rd ed. Boston, MA: Cengage Learning, 2012. This widely-used textbook provides accessible explanations of computational theory with clear proofs and practical examples that were invaluable for understanding PDA acceptance conditions.

[3] S. H. Rodger and T. W. Finley, *JFLAP: An Interactive Formal Languages and Automata Package*. Sudbury, MA: Jones and Bartlett Publishers, 2006. JFLAP served as the primary inspiration for this project. Developed at Duke University since the early 1990s, it remains the most comprehensive automata simulator in education. Official website: `https://www.jflap.org`

[4] "JFLAP," Wikipedia, The Free Encyclopedia. Last edited December 2024. Accessed December 30, 2024. `https://en.wikipedia.org/wiki/JFLAP`. This article notes that a 2011 review called JFLAP "the most sophisticated tool for simulating automata" with "unparalleled effort" in its development.

[5] S. H. Rodger, M. Procopiuc, and O. Procopiuc, "Visualization and Interaction in the Computer Science Formal Languages Course with JFLAP," in *Proceedings of 1996 Frontiers in Education Conference*, Salt Lake City, UT, pp. 121-125, 1996. This early paper introduced the concept of visual automata simulation that influenced modern educational tools.

[6] "Automata Simulator Topics," GitHub. Accessed December 30, 2024. `https://github.com/topics/automata-simulator`. This GitHub topic page showcases dozens of modern web-based automata simulators, demonstrating the continued interest in browser-based educational tools that require no installation.

[7] U. Bhat, "Automata Simulator - A Web-Based Simulator," GitHub Repository, 2019. `https://github.com/bhatushar/automata-simulator`. This TypeScript/React implementation provided insights into modern architectural approaches for building interactive state diagrams using libraries like GoJS.

[8] P. Linz, *An Introduction to Formal Languages and Automata*, 5th ed. Burlington, MA: Jones & Bartlett Learning, 2011. Linz's textbook offers practical construction techniques for PDAs with clear step-by-step examples that informed the design patterns used in this simulator.

[9] C. A. Shaffer, T. L. Naps, and S. H. Rodger, "Algorithm Visualization: The State of the Field," *ACM Transactions on Computing Education*, vol. 10, no. 3, article 9, pp. 1-22, August 2010. This comprehensive survey demonstrates that interactive visualization significantly improves learning outcomes compared to static presentations.

[10] C. D. Hundhausen, S. A. Douglas, and J. T. Stasko, "A Meta-Study of Algorithm Visualization Effectiveness," *Journal of Visual Languages and Computing*, vol. 13, no. 3, pp. 259-290, June 2002. This meta-analysis of 24 experimental studies proved that active engagement with visualizations leads to better comprehension than passive viewing.

[11] P. Chakraborty, P. P. Saxena, and C. P. Katti, "Fifty Years of Automata Simulation: A Review," *ACM Inroads*, vol. 2, no. 4, pp. 51-58, December 2011. This historical review traces the evolution of automata simulators from early command-line tools to sophisticated graphical systems, highlighting JFLAP's dominance in the field.

[12] "Flask Documentation," Pallets Projects. Accessed December 2024. `https://flask.palletsprojects.com/`. Flask's lightweight architecture and minimal boilerplate made it ideal for this educational project, allowing rapid development of RESTful APIs without unnecessary complexity.