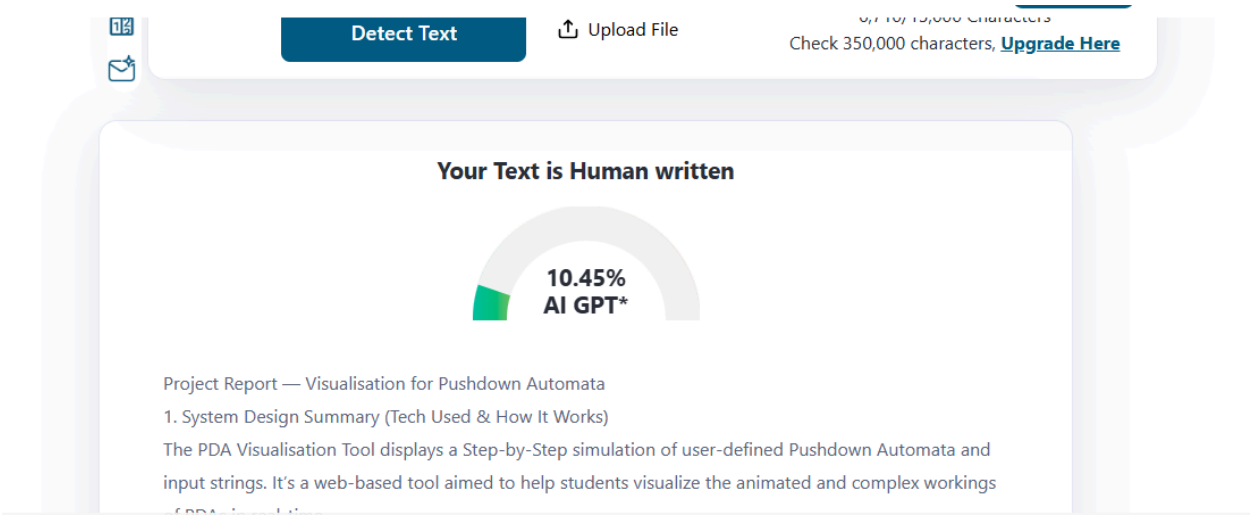

Project Report — Visualisation for Pushdown Automata

1. System Design Summary (Tech Used & How It Works)

The PDA Visualisation Tool displays a Step-by-Step simulation of user-defined Pushdown Automata and input strings. It's a web-based tool aimed to help students visualize the animated and complex workings of PDAs in real-time.

Tech Stack

- **Frontend:** React + TypeScript
- **State Management:** Redux
- **Styling:** Tailwind CSS
- **Visualization:** I'll be using D3.js / HTML Canvas (for transitions, stack growth, node rendering)
- **Backend (optional if client-side only):** Node.js + Express
- **Build Tools:** Next.js

How It Works

The tool allows the user to define the components of the PDA, including states, input alphabet, stack alphabet, transitions, start state, and accept states. Once the PDA is defined, the system converts this into an internal structured object that represents the machine.

When the user inputs a string, the engine simulates each transition while showing the following:

- The current state of the machine
- The unread input symbol
- The current content of the stack

The visualizations are updated live with animations, which include:

- Highlighting the active state
- Animating the push/pop actions on the stack
- Indicating whether the string is **accepted** or **rejected**

Future Adaptations

While this is the core functionality planned, I anticipate that we might adjust the implementation slightly based on user feedback, supervisor input, or future technical improvements.

2. Main Modules / Code Structure

/src/engine/pda.ts

This module contains the core PDA simulation logic:

- **PDAEngine class:** Initializes the machine and handles the core PDA logic.
- **step() function:** Executes a single transition of the PDA.
- **simulate() function:** Simulates the entire input string.
- **Stack operations:** Includes functions like `push`, `pop`, and `peek` for managing the stack.

- **Transition resolver:** Resolves which transition to follow based on the current state and input symbol.

/src/components/Editor/

This is the user interface (UI) module for entering the PDA components:

- **State creator:** Let the user define states.
- **Transition table:** Allows users to specify transitions between states.
- **Stack alphabet input:** Enables users to define stack alphabet symbols.
- **Accept-state selector:** Let users mark the accepting states.

/src/components/Simulation/

This module handles the interactive visual components:

- **Animated state graph (D3.js):** Displays the graph of states and transitions.
- **Stack visualizer:** Animates the stack's push/pop actions during the simulation.
- **Input tape visualizer:** Shows the input string and highlights the unread symbols.
- **Controls:** Play, Pause, Step, and Reset buttons for controlling the simulation flow.

/src/utils/parser.ts

This utility module converts the raw user input into a machine-readable format that the PDA engine can use.

/src/pages/

Defines the main pages of the application:

- **Home page:** Landing page of the tool.
- **PDA Editor:** The page where users can define and edit the PDA.
- **Simulator Page:** Where users run the simulation and view the output.

Note on Code Structure (Important for Approval)

The module structure presented above reflects the initial plan for organizing the project. However, **depending on feedback or evolving requirements**, the exact folder names, module separation, and internal class structure may vary slightly during development. I may adjust things to:

- Improve code readability and maintainability.
- Optimize performance where needed.
- Simplify user workflows or enhance the visual experience.

These changes will always be made to ensure a seamless and user-friendly experience in terms of project implementation and development.

3. Screenshots / Demo Video Requirements

Once the UI is implemented, the following visuals will be included within this report with a screenshot or a recording:

- **Main dashboard:** Overview of the tool's interface.
- **PDA editor page:** Where users define their PDAs.
- **Transition visualizer:** Displays state transitions during simulation.
- **Run simulation screen:** Showing input processing and results.
- **Accept/reject animation:** Visualization indicating whether the input string is accepted or rejected by the PDA.

4. Example PDAs Tested

Example 1 – Balanced Parentheses

Language:

- `L = { w | w has balanced parentheses }`

Key transitions:

- On encountering `(` → push `(` onto the stack.
- On encountering → pop `(` from the stack.
- Accept when the stack is empty and the input is fully processed.

Example 2 – Palindrome (Odd length)

Language:

- `L = { w | w is a palindrome over {a, b}, with odd length }`

Typical behavior:

- Push the first half of the string onto the stack.
- Skip the middle symbol.
- Pop and compare the second half with the stack.

Example 3 – $a^n b^n$

Language:

- $L = \{ a^n b^n \mid n \geq 1 \}$

Key transitions:

- Push `a` onto the stack.
- Pop `a` for each `b` encountered.
- Accept if the stack is empty after processing all input.

5. Challenges Faced

Ambiguous transitions

Some PDAs allow multiple possible transitions for a single input symbol. This requires handling **nondeterministic** behavior in the simulation, which I had to carefully manage.

Stack visualization

The biggest difficulty was making the stack operations (push/pop) animation smooth. I concentrated on creating a better user experience by improving the performance of the animations.

Large PDAs create cluttered graphs

In PDAs, visualizations can be cluttered when there are too many states However, I resolved this issue by implementing a force-directed graph layout which allows users to optimize the viewport by zooming, panning, and repositioning the visualization.

User-friendly editor

Ensuring the tool is easy for students to use without requiring them to understand formal PDA notation was a challenge. I've built a simple, intuitive interface to help with this, along with input validation.

6. Possible Improvements

- **Support for NFA → PDA conversion**
  This would allow users to convert a nondeterministic finite automaton (NFA) into an equivalent PDA.

- **Export machine as JSON or XML**
  Users could save their PDAs and load them again for later use.

- **Add step history and rewind**
  Allow users to view a history of steps taken during the simulation and go back if needed.

- **Support for multiple stacks**
  Implementing multi-stack PDAs to simulate more complex languages.

- **Add PDA library templates**
  Include predefined PDAs for common patterns (balanced parentheses, $a^n b^n$, etc.), making it easier for users to get started.

7. Source Code & ReadMe

The complete project will be available at:

GitHub Repository:
https://github.com/SyedaIlmaAli/pda-visualizer

The **README** includes:

- Setup instructions
- Running instructions
- Folder structure
- Example PDAs