

**Capstone Project Phase B**

# **Asynchronous Hyperautomata Simulator**

**Nevo Gottlieb**

**Raz Kessel**

**Supervisor: Dr. Sarai Sheinvald**

**Git repository: <https://github.com/RazKessel/Capstone-Project---Hyperautomata-Simulator.git>**

## Table of contents

<b>1 General Description .....</b>	<b>3</b>
<b>1.1 Description of the Solution .....</b>	<b>4</b>
<b>1.2 Research and Development Process .....</b>	<b>4</b>
<b>1.3 Challenges and Solutions.....</b>	<b>5</b>
<b>1.4 Results and Conclusions .....</b>	<b>5</b>
<b>1.5 Lessons Learned .....</b>	<b>5</b>
<b>2 User Manual Guide for Hyperautomata Simulator .....</b>	<b>6</b>
<b>2.1 Login and Registration.....</b>	<b>6</b>
<b>2.2 Help and Tool Tip.....</b>	<b>7</b>
<b>2.3 Set a New Automata .....</b>	<b>8</b>
<b>2.3.1 Set Amount of Words to Accept .....</b>	<b>8</b>
<b>2.3.2 Set States.....</b>	<b>8</b>
<b>2.3.3 Set Transitions.....</b>	<b>8</b>
<b>2.3.4 Set Words.....</b>	<b>10</b>
<b>2.3.5 Edit States and Transitions .....</b>	<b>11</b>
<b>2.3.6 Move States Positions .....</b>	<b>11</b>
<b>2.3.7 Undo &amp; Redo .....</b>	<b>11</b>
<b>2.3.8 Setting Automata Result .....</b>	<b>11</b>
<b>2.4 Handle Simulation .....</b>	<b>12</b>
<b>2.4.1 Running the Simulation.....</b>	<b>12</b>
<b>2.4.2 Pause Run .....</b>	<b>13</b>
<b>2.4.3 Resume Running .....</b>	<b>13</b>
<b>2.4.4 Advancing "Step by Step .....</b>	<b>13</b>
<b>2.4.5 Reset Run.....</b>	<b>13</b>
<b>2.5 Reset the Application .....</b>	<b>13</b>
<b>2.6 Save and Load Runs.....</b>	<b>13</b>
<b>2.6.1 Save Run .....</b>	<b>13</b>
<b>2.6.2 Load Run .....</b>	<b>14</b>
<b>3 Maintenance Guide.....</b>	<b>15</b>
<b>3.1 System Environment .....</b>	<b>15</b>
<b>3.2 Installation Instructions .....</b>	<b>15</b>
<b>3.3 Key Files and Their Purposes .....</b>	<b>16</b>
<b>3.4 Common Maintenance Tasks.....</b>	<b>16</b>
<b>3.5 Challenges and Recommendations .....</b>	<b>17</b>
<b>3.6 Documentation References .....</b>	<b>17</b>

**Abstract.** The Asynchronous Hyperautomata Simulator is an advanced computational tool designed to explore and analyze the behavior of hyperautomata, a computational model extending traditional finite automata with asynchronous capabilities. Unlike conventional synchronous models, this simulator focuses on real-world scenarios where components progress at varying speeds, such as in distributed systems, robotics, and biological processes.

The simulator provides a comprehensive and user-friendly environment for researchers, educators, and developers. It features a graphical user interface (GUI) that enables intuitive definition of states, transitions, and configurations, along with step-by-step simulation controls. The simulator is built on an algorithmic backbone that employs a BFS-based state exploration algorithm for efficient and exhaustive analysis of all possible runs, supporting the asynchronous nature of transitions. Database integration ensures persistent storage of user sessions, simulation runs, and historical data for later reference.

The simulator's key functionalities include visualizing automata, simulating multiple asynchronous processes, and enabling users to interact dynamically by adding states, transitions, or input tapes during execution. With a modular software architecture, it balances performance, scalability, and usability, making it a versatile platform for advancing research and practical applications in asynchronous system design.

## 1 General Description

The project is a simulator designed for executing hyperautomata, an advanced computational model that extends finite automata with asynchronous capabilities. The simulator's primary purpose is to provide a user-friendly interface and backend for exploring the behavior and properties of hyperautomata. It targets researchers, educators, and developers working in fields like distributed systems, robotics, and security protocols. For all theoretical information about hyperautomatas, feel free to check our first phase project book.

Key features of the simulator include:

**User Interface:** A graphical interface enabling users to define automata, customize configurations, and control simulations step-by-step.

**Database Integration:** Persistent storage for user sessions, historical runs, and simulation data.

**Algorithmic Efficiency:** An optimized BFS-based exploration algorithm to ensure thorough and efficient state exploration.

## 1.1 Description of the Solution

The simulator is structured around modular components that separately handle simulation logic, user interaction, and database integration:

### 1. Software Architecture:

The project uses a component-based design to separate responsibilities, ensuring modularity and scalability.

The main components include:

- **State Management:** Modules like `state.py` and `transition.py` define the logic for states and transitions.
- **UI Components:** `drawing_board.py` and the `panels/` folder handle graphical elements and user input.
- **Database Integration:** `db_integration.py` connects the application to a database for tracking user data and simulations.

### 2. Algorithm:

The simulator employs BFS to explore all possible simulations (nodes) of the hyperautomata:

- Nodes represent simulation configurations (state, tapes, positions).
- Edges correspond to transitions defined in the automata.

A visited set ensures no state is revisited, preventing redundant computations.

The queue manages the order of state exploration, ensuring systematic traversal of all configurations.

Complexity:  $O(V + E)$ , where  $V$  is the number of all unique simulations and  $E$  is all possible transitions.

## 1.2 Research and Development Process

Research:

- Extensive study of hyperautomata models and their applications.
- Comparison of potential algorithms for state exploration, leading to the selection of BFS for its simplicity and completeness.

Development:

- Incremental implementation, starting with backend simulation logic, followed by UI integration and database support.
- Prototyping the BFS algorithm and refining it to handle asynchronous transitions effectively.

Tools:

- Development Environment: PyCharm and Git for version control.
- Programming Language: Python for its flexibility and extensive libraries.
- Database: SQLite for lightweight and efficient data storage.
- Testing Methods: Unit tests, integration tests, system tests (PyTest).

Client Interaction:

- Regular feedback sessions with potential users (researchers) to refine features and usability.
- Iterative updates to ensure the tool met user expectations and research needs.

### **1.3 Challenges and Solutions**

Challenge: Avoiding redundant state exploration during BFS.

- Solution: Implementation of a visited set to track explored nodes efficiently.

Challenge: Managing the complexity of asynchronous transitions.

- Solution: Modularizing the transition logic into reusable components.

Challenge: Designing a user-friendly interface for defining automata.

- Solution: Creation of interactive graphical components for visualizing states and transitions.

Challenge: Balancing performance and completeness.

- Solution: BFS provided a complete exploration while maintaining a manageable runtime.

### **1.4 Results and Conclusions**

Achievement of Goals:

The simulator successfully meets its objectives, providing an efficient, scalable, and user-friendly tool for hyperautomata exploration.

The BFS algorithm ensures exhaustive and efficient state exploration, while the database allows users to save and resume simulations.

Decision-Making:

The choice of BFS was validated by its ability to handle asynchronous automata while maintaining simplicity.

The modular design proved effective for both development and future scalability.

### **1.5 Lessons Learned**

Positive Practices:

- Incremental development and regular testing ensured stability and progress.
- Client feedback improved usability and functionality.

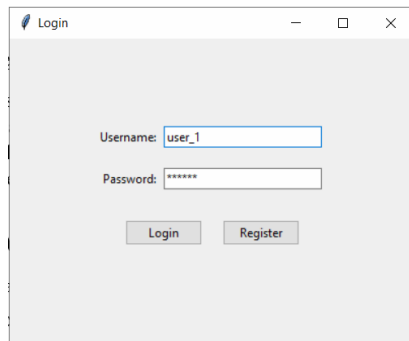
Improvements:

- Database Selection: In hindsight, using a more robust database like PostgreSQL could have provided better scalability.
- Algorithm Optimization: Exploring heuristics for transition prioritization could improve performance in large automata.

## 2 User Manual Guide for Hyperautomata Simulator

### 2.1 Login and Registration

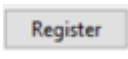
Upon launching the application, the user is greeted with the Login Window:

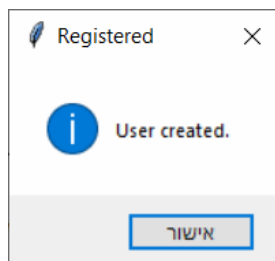


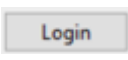
Username and Password: Enter your credentials to log in.

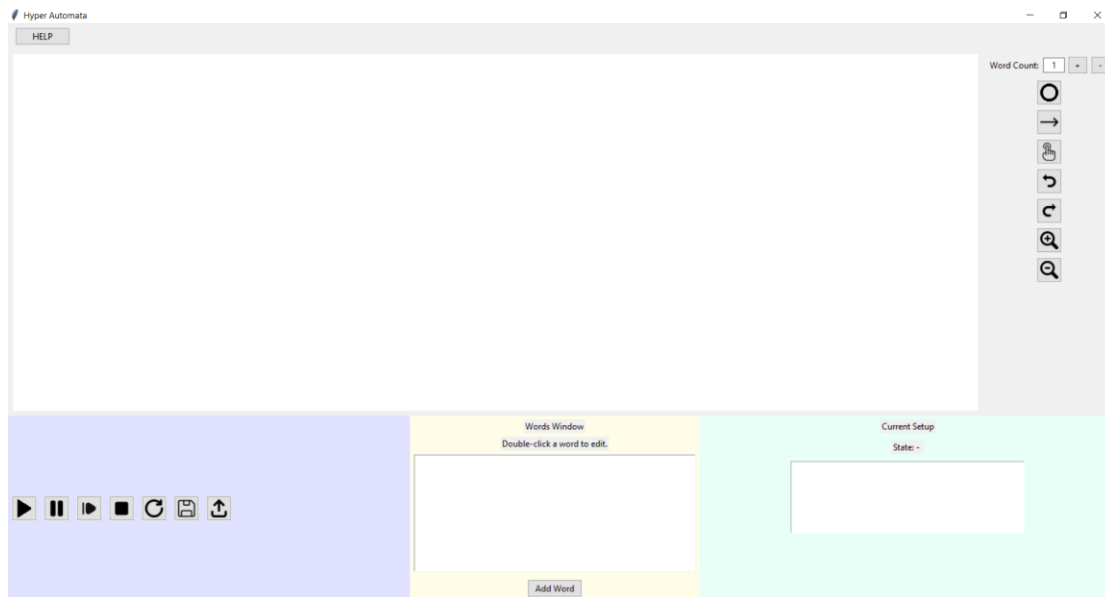
Login Button: Click to access the simulator if your credentials are valid.

Register Button: For new users, click "Register" to create an account.


By clicking  it will pop a confirmation for user creation:

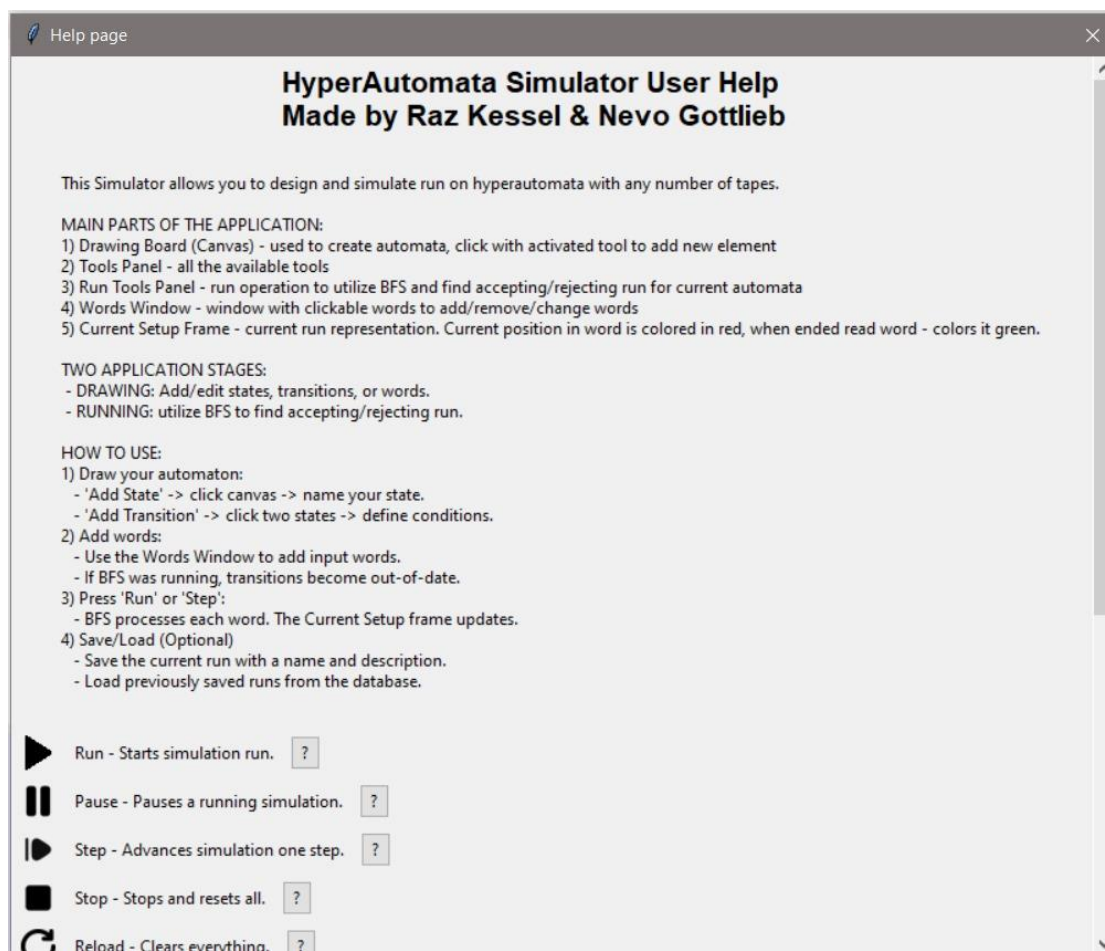


By clicking  it will open the main application window:



## 2.2 Help and Tool Tip

By clicking  it will pop a detailed tool tip:




## 2.3 Set a New Automata

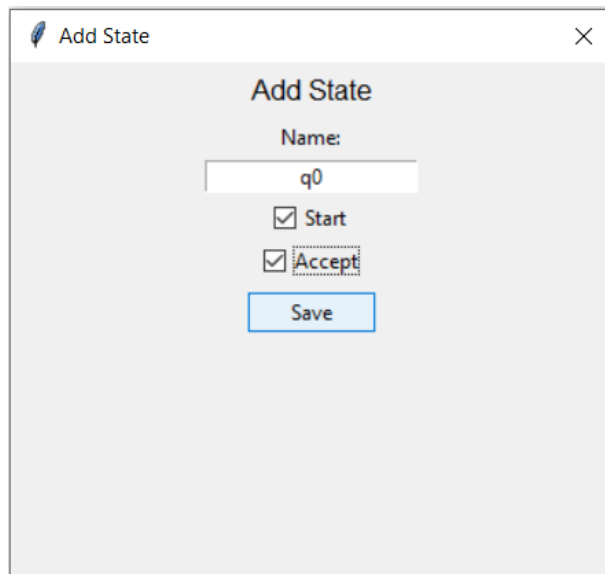
### 2.3.1 Set Amount of Words to Accept

In the main application window set amount of words the automata should accept:

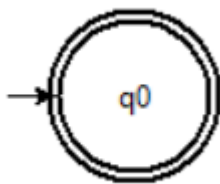


### 2.3.2 Set States

In the main application window click on  (Add State), It will pop a window where you can define state name, decide if it's a start state or accept state.




After clicking Save a visualized state will appear on the canvas:

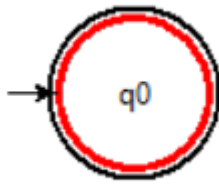


You can continue adding states as much you desire.

### 2.3.3 Set Transitions

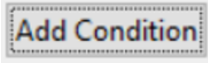
By clicking on  (Add Transition) you will be able to set a transition between two states by first clicking on desired from state, the chosen state will colored red

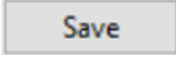


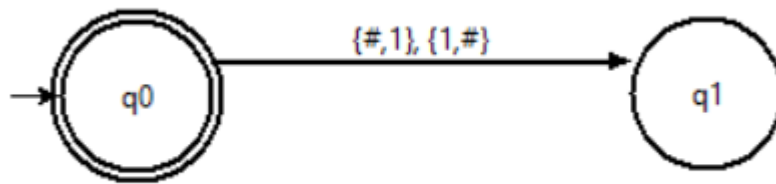


Then clicking on the target state. It will pop a window where you can set the transition vector.

\*notice that the amount of vector to set is depends on the word amount that defined in 2.3.1, but it possible to add more if desired and it will change the word amount accordingly.

By clicking  you can add conditions as much as desired.

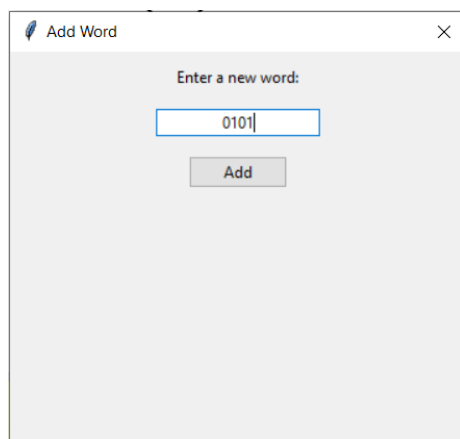
By clicking  it will define the transition, and show it visually on the canvas:



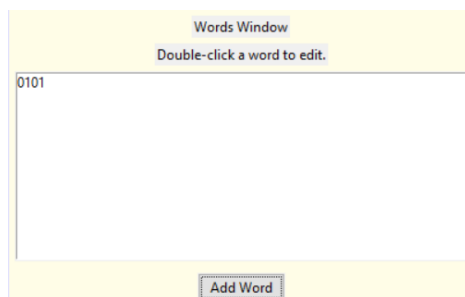
You can continue add transitions between states as much as you desire.

### 2.3.4 Set Words

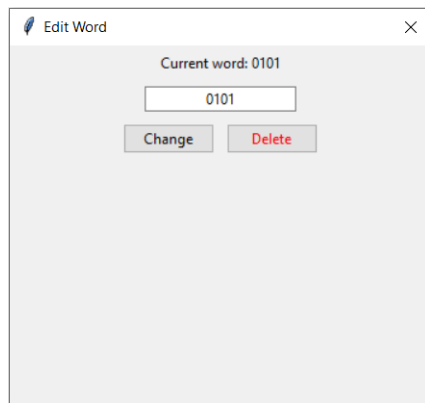
On the main application window, click on **Add Word** to add word. It will pop a window where you should write the desired word:

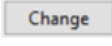



By clicking **Add**, it will set the word and show it on the words window:




Double click on the word itself, will open a pop up window where you can edit or delete the word:



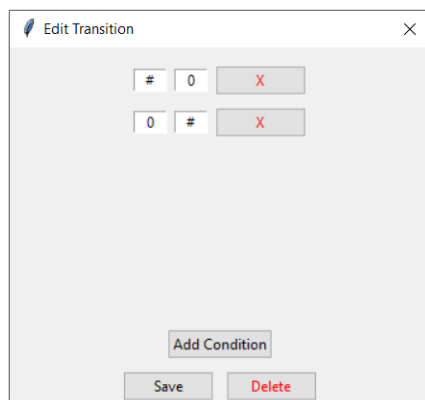
By clicking , changes will be saved and the pop up window will be closed.

By clicking , the word will be deleted and will no longer appear in the words window.


### 2.3.5 Edit States and Transitions

By clicking on  (Selection) you will be able to edit states and transitions by left clicking on them.


Left click will pop up window to edit the desired object as you see earlier, for example:




### 2.3.6 Move States Positions

By clicking on  (Selection) you will be able to move states positions by right click + drag to the desired place on the canvas.

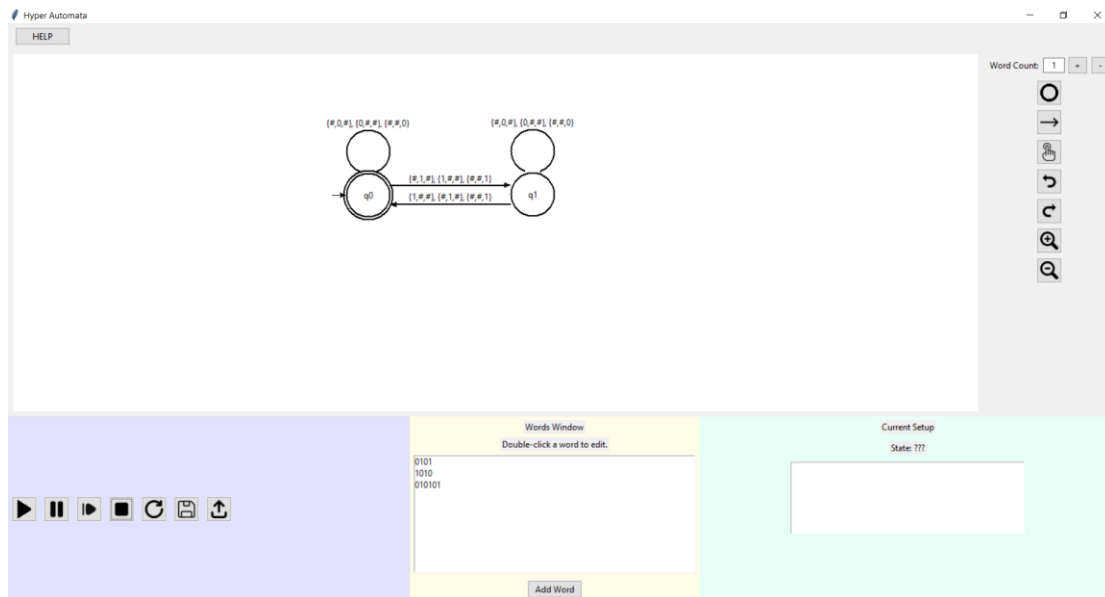
### 2.3.7 Undo & Redo

By clicking , you will be able to Undo the last action if regret.

By clicking , you will be able to Redo the Undo action.


### 2.3.8 Setting Automata Result

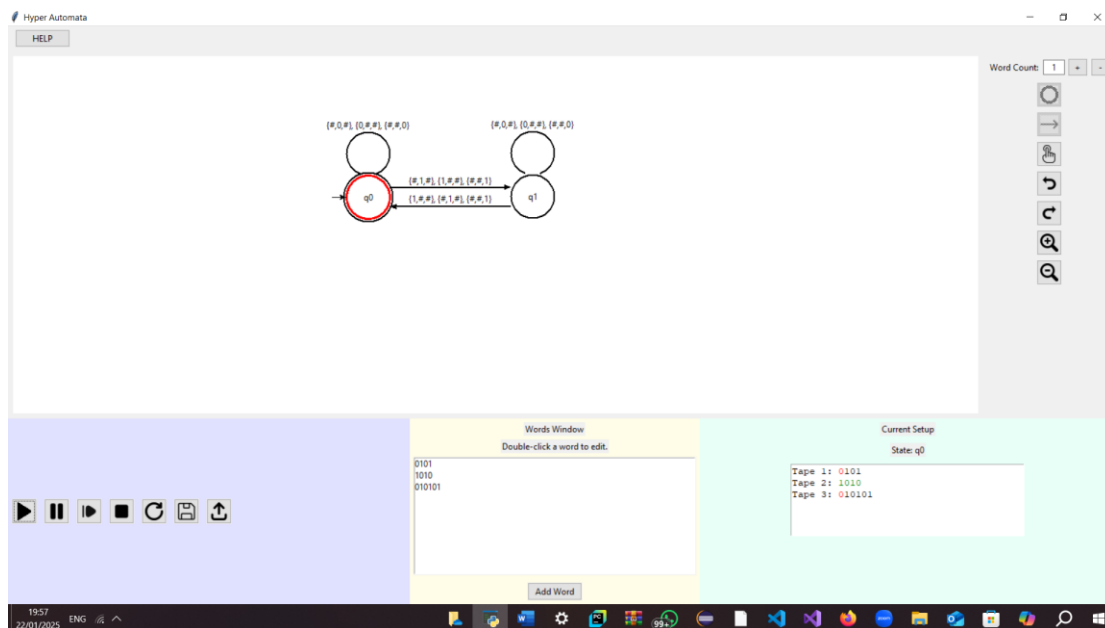
After the steps of 2.3, you should see on the canvas the automata with all states and transitions you defined, and the words on the words window below:



## 2.4 Handle Simulation

### 2.4.1 Running the Simulation

By clicking on the  (Run) the program will visualize an accepting run if exists, else run randomly.




The Red circle marks the state current state of the running.

The current Setup (right down corner) shows live the current states, and the tapes (words) positions.


Red colored char is a read char.

Tape that is fully colored in green is consumed.


### 2.4.2 Pause Run

By clicking on the  (Pause), the run will pause at the moment of clicking.


### 2.4.3 Resume Running

By clicking on  (Run), running present process will be resume.


### 2.4.4 Advancing "Step by Step"

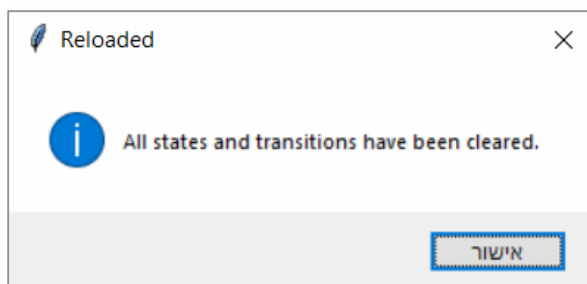
Each click on  (Step), it will be forward the state of the automata and the tapes positions accordingly, to the next step.

### 2.4.5 Reset Run

By clicking on  (Stop), it will be reset the steps to the beginning.

## 2.5 Reset the Application

By clicking  (Reloaded) , it will be reset the automata completely includes, words and will pop up a window to confirm it:

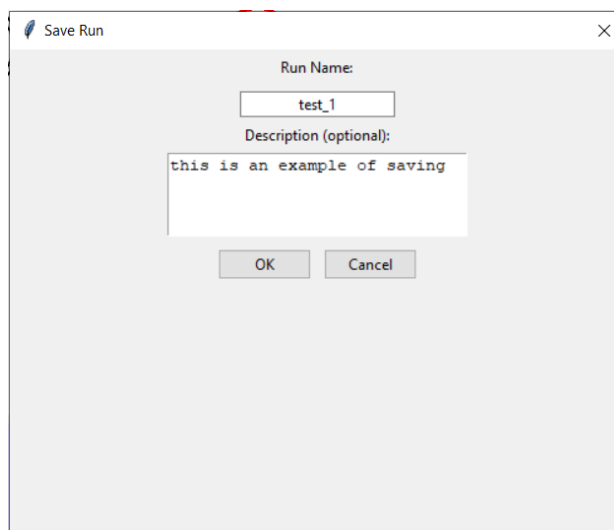


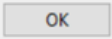
## 2.6 Save and Load Runs

### 2.6.1 Save Run

At any time you can click on  (Save), to save your work.

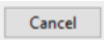
It will pop up a window where you can name the run and give it description:




On clicking  it will be saved, and will be able to be load in the future.

A pop up window will confirm the run saved successfully.

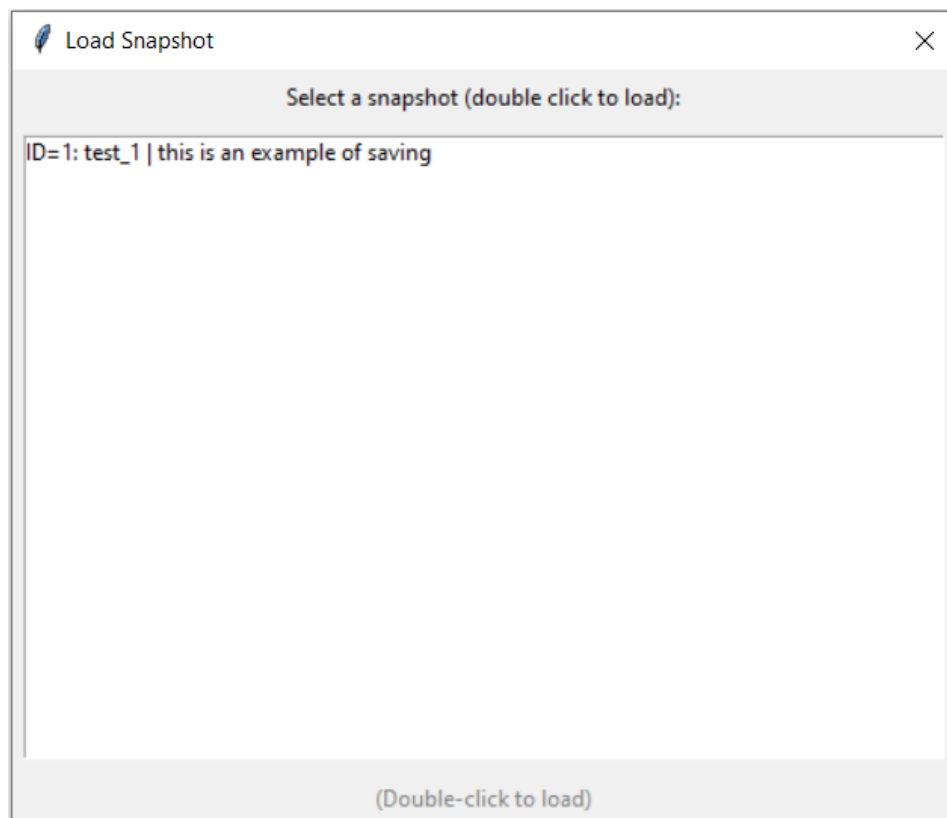


On clicking , the pop up window will be closed and nothing will be saved.

## 2.6.2 Load Run

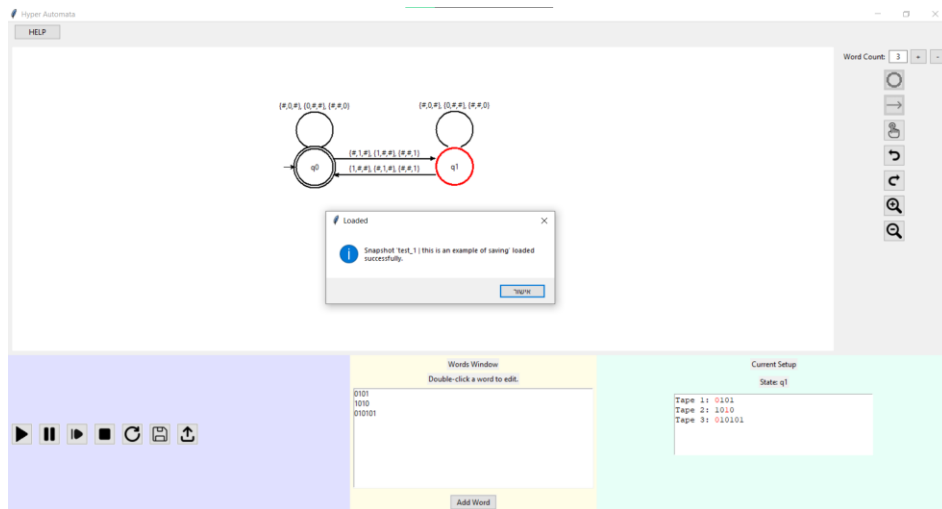
At any time you can click on  (Load) and load a run you saved in the past.

It will pop up a window where you can see all saved runs.



Double click on the desired run will load it and will be set in the same positions saved.

A pop up window will confirm that the run loaded successfully.



## Conclusion

The Hyperautomata Simulator provides an intuitive platform to design, visualize, and analyze hyperautomata. By following this guide, users can efficiently explore automata behavior, define configurations, and analyze simulation results.

## 3 Maintenance Guide

The Maintenance Guide ensures continued use and development of the Hyperautomata Simulator after the project completion. It includes guidelines for implementing changes, updates, and improvements to extend the system's lifespan.

### 3.1 System Environment

#### Software Requirements:

- Python Version: Python 3.8 or higher.

#### Hardware Requirements:

Minimum 4GB RAM.

500MB free disk space.

### 3.2 Installation Instructions

#### Clone the Repository:

- Use Git to clone the project repository:  

```
git clone https://github.com/RazKessel/Capstone-Project---Hyperautomata-Simulator.git
```

#### Set Up Virtual Environment:

- Create and activate a virtual environment:

```
python -m venv venv
source venv/bin/activate # On Windows: venv\Scripts\activate
```

#### Install Dependencies:

- Install required libraries using the requirements.txt file:

```
pip install -r requirements.txt
```

#### Run the Application:

- Navigate to the project directory and run the main file:

```
python main.py
```

### **3.3 Key Files and Their Purposes**

main.py: Entry point for the application.

components/: Contains the core modules for state management, transitions, and UI.

db\_integration.py: Manages database operations for storing simulations and user data.

requirements.txt: Lists all dependencies required for the project.

### **3.4 Common Maintenance Tasks**

#### Updating Dependencies:

- If new libraries are added, update the requirements.txt file:

```
pip freeze > requirements.txt
```

#### Adding Features:

- Add new UI components in the components/ folder.
- Extend state or transition logic in state.py or transition.py.

#### Bug Fixes:

- Review error logs in error\_log.log and app\_log.log files and debug using standard Python debugging tools.
- Test thoroughly after implementing fixes.

#### Database management:

- The database is presented in file automata.db.
- Its recommended to install SimpleSqliteBrowser plugin (usually, PyCharm suggest you to do so) for a friendly view.



### 3.5 Challenges and Recommendations

#### UI Updates:

- Ensure consistency in design by following existing component patterns.

#### Algorithm Improvements:

- Maintain BFS efficiency while implementing changes.

### 3.6 Documentation References

Tkinter: Used for creating the graphical user interface (GUI). Refer to the official documentation for advanced customization:

- **Tkinter Documentation**

Pillow: Utilized for handling and rendering images in the application:

- **Pillow Documentation**

SQLAlchemy: Facilitates database operations for managing user data and simulations:

- **SQLAlchemy Documentation**