# CSE366 AI Lab Simulator Manual

## Table of Contents



## Introduction

The **CSE366 AI Lab Simulator** is a Python project designed to simulate various Artificial Intelligence algorithms and concepts, particularly for educational purposes. It provides interactive visualizations of different AI algorithms, including search algorithms, genetic algorithms, and pathfinding in environments like mazes and grids.

This simulator is organized into modular components, making it extensible and easy to understand. It includes simulations for:

- **Robot Task Simulation**: A robot navigating a grid environment to complete tasks using different search algorithms.
- **8-Queens Genetic Algorithm Simulation**: Solving the 8-Queens problem using a genetic algorithm with visualization.

- **Maze Solver Simulation**: An agent solving mazes using various search algorithms, with adjustable maze complexity and density.

---

# Installation Instructions

To set up the **CSE366 AI Lab Simulator** on your local machine, follow these steps:

## Prerequisites

- **Python 3.6 or higher**: Ensure that Python is installed on your system.
- **pip**: Python package installer, typically included with Python installations.

## Clone the Repository

```
git clone https://github.com/EWU-MRAR-AI-ML-Courses/CSE366_AI_Lab_Simulator.git
```

## Navigate to the Project Directory

```
cd CSE366_AI_Lab_Simulator
```

## Create a Virtual Environment (Optional but Recommended)

```
python -m venv venv
```

Activate the virtual environment:

- On Windows:

  ```
  venv\Scripts\activate
  ```

- On macOS/Linux:

  ```
  source venv/bin/activate
  ```

## Install Required Packages

Install the dependencies listed in `requirements.txt`:

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

The primary dependencies are:

- **pygame**: For creating graphical interfaces and animations.

# Project Structure

The project is organized into several directories and files for better modularity and maintainability.

```
CSE366_AI_Lab_Simulator/
├── README.md
├── LICENSE
├── requirements.txt
├── modules/
│   ├── __init__.py
│   ├── agents/
│   │   ├── __init__.py
│   │   ├── robot_agent.py
│   │   └── maze_agent.py
│   ├── environments/
│   │   ├── __init__.py
│   │   ├── grid_environment.py
│   │   └── maze_environment.py
│   ├── search_algorithms/
│   │   ├── __init__.py
│   │   ├── uninformed_search.py
│   │   ├── informed_search.py
│   │   └── local_search.py
│   ├── simulations/
│   │   ├── __init__.py
│   │   ├── simulation_base.py
│   │   ├── search_simulation.py
│   │   └── maze_simulation.py
│   └── utils/
│       ├── __init__.py
│       └── constants.py
└── examples/
    ├── __init__.py
    ├── main.py
    ├── main_8_queen.py
    └── main_maze_solver.py
```

**Key Directories and Files:**

- **modules/**: Contains all the core modules, organized into subdirectories.
  - **agents/**: Agent classes that define the behavior of entities in simulations.
  - **environments/**: Environment classes that represent the world or context in which agents operate.
  - **search_algorithms/**: Implementations of various search algorithms.
  - **simulations/**: Simulation classes that manage the execution and visualization of simulations.
  - **utils/**: Utility modules, such as constants and helper functions.
- **examples/**: Contains scripts to run the simulations.

# Modules Overview

## Agents Module

**Location:** `modules/agents/`

This module contains agent classes that define the behavior and properties of entities within the simulations.

- **robot_agent.py**: Defines the `RobotAgent` class used in the robot task simulation.
- **maze_agent.py**: Defines the `MazeAgent` class used in the maze solver simulation.

**Key Classes:**

- `RobotAgent`: Represents a robot navigating a grid to complete tasks using search algorithms.
- `MazeAgent`: Represents an agent solving a maze using search algorithms.

## Environments Module

**Location:** `modules/environments/`

This module contains environment classes that define the world or context in which agents operate.

- **grid_environment.py**: Defines the `GridEnvironment` class for the robot task simulation.
- **maze_environment.py**: Defines the `MazeEnvironment` class for the maze solver simulation, including the maze generation with adjustable complexity and density.

**Key Classes:**

- `GridEnvironment`: Represents a grid with obstacles and tasks for the robot.
- `MazeEnvironment`: Generates and manages mazes with multiple paths, allowing adjustments to complexity and density.

## Search Algorithms Module

**Location:** `modules/search_algorithms/`

This module contains implementations of various search algorithms used by agents.

- **uninformed_search.py**: Contains implementations of uninformed search algorithms like DFS, BFS, and UCS.

- **informed_search.py**: Contains implementations of informed search algorithms like A*.
- **local_search.py**: Contains local search algorithms and genetic algorithm functions for the 8-Queens problem.

**Key Functions:**

- `dfs()`: Depth-First Search algorithm.
- `bfs()`: Breadth-First Search algorithm.
- `ucs()`: Uniform-Cost Search algorithm.
- `astar()`: A* Search algorithm.
- `hill_climbing()`, `simulated_annealing()`: Local search algorithms.
- Genetic algorithm functions for the 8-Queens problem (`generate_individual()`, `fitness()`, etc.).

## Simulations Module

**Location:** `modules/simulations/`

This module contains simulation classes that manage the execution and visualization of simulations.

- **simulation_base.py**: Base class for simulations.
- **search_simulation.py**: Manages the robot task simulation.
- **maze_simulation.py**: Manages the maze solver simulation, including UI elements and animation control.

**Key Classes:**

- `SimulationBase`: Base class providing common functionality for simulations.
- `SearchSimulation`: Handles the robot task simulation, including the environment and agent interactions.
- `MazeSimulation`: Handles the maze solver simulation, including maze generation, agent movement, and UI.

## Utils Module

**Location:** `modules/utils/`

This module contains utility modules used across the project.

- **constants.py**: Defines constants such as colors, dimensions, and default values used in the simulations.

# Running the Simulations

All simulations are run from scripts located in the `examples/` directory.

## Robot Task Simulation

**Script:** `examples/main.py`

**Description:**

Simulates a robot navigating a grid environment to complete tasks using different search algorithms. The environment includes obstacles and tasks that the robot must navigate to and complete.

**Running the Simulation:**

```
cd examples
python main.py
```

**Features:**

- Select different search algorithms (DFS, BFS, UCS, A*).
- Customize the grid size and task positions.
- Visualize the robot's path and task completion.

## 8-Queens Genetic Algorithm Simulation

**Script:** `examples/main_8_queen.py`

**Description:**

Solves the 8-Queens problem using a genetic algorithm, visualizing the evolution of the population over generations.

**Running the Simulation:**

```
cd examples
python main_8_queen.py
```

**Features:**

- Adjustable maximum generations.
- Control the delay between generations to observe the algorithm's progress.
- Visual representation of the chessboard and queen positions.

### Maze Solver Simulation

**Script:** `examples/main_maze_solver.py`

**Description:**

Simulates an agent solving a maze using various search algorithms. The maze has multiple paths from start to end, and you can adjust its complexity and density.

**Running the Simulation:**

```
cd examples
python main_maze_solver.py
```

**Features:**

- Select different search algorithms (DFS, BFS, UCS, A*).
- Adjust maze dimensions, complexity, and density.
- Visualize the agent's path through the maze.
- UI elements including Start and Reset buttons, and information display.

---

# Command-Line Arguments and Usage Examples

Each simulation script accepts command-line arguments to customize the simulation.

### Robot Task Simulation Arguments

**Script:** `main.py`

**Available Arguments:**

- `--algorithm`: Select the search algorithm (`dfs`, `bfs`, `ucs`, `astar`, `hill_climbing`, `simulated_annealing`).
  - **Default:** `astar`
- `--grid_size`: Set the size of the grid.
  - **Default:** `16`
- `--task_positions`: Specify task positions in the format `[(x1,y1),(x2,y2),...]`. If not provided, tasks are placed randomly.

**Usage Examples:**

- Run with BFS algorithm:

  ```
  python main.py --algorithm bfs
  ```

- Set grid size to 20:

  ```
  python main.py --grid_size 20
  ```

- Specify task positions:

  ```
  python main.py --task_positions "[(3,3),(5,5),(7,7)]"
  ```

## 8-Queens Simulation Arguments

**Script:** `main_8_queen.py`

**Available Arguments:**

- `--max_generations`: Set the maximum number of generations for the genetic algorithm.
  - **Default:** `1000`
- `--delay`: Set the delay between generations in seconds.
  - **Default:** `0.1`

**Usage Examples:**

- Run with maximum 500 generations:

  ```
  python main_8_queen.py --max_generations 500
  ```

- Set delay between generations to 0.2 seconds:

  ```
  python main_8_queen.py --delay 0.2
  ```

## Maze Solver Simulation Arguments

**Script:** `main_maze_solver.py`

**Available Arguments:**

- `--algorithm`: Select the search algorithm (`dfs`, `bfs`, `ucs`, `astar`).
  - **Default:** `dfs`
- `--maze_width`: Set the width of the maze (odd number).
  - **Default:** `21`
- `--maze_height`: Set the height of the maze (odd number).
  - **Default:** `21`
- `--complexity`: Adjust the maze complexity (0.0 to 1.0).
  - **Default:** `0.75`
- `--density`: Adjust the maze density (0.0 to 1.0).
  - **Default:** `0.75`

**Usage Examples:**

- Run with A* algorithm:

```
python main_maze_solver.py --algorithm astar
```

- Set maze dimensions to 31x31:

```
python main_maze_solver.py --maze_width 31 --maze_height 31
```

- Adjust maze complexity and density:

```
python main_maze_solver.py --complexity 0.9 --density 0.9
```

---

# Extending and Modifying the Code

The modular structure of the **CSE366 AI Lab Simulator** allows for easy extension and modification. Below are some guidelines on how you can enhance the project.

## Adding New Search Algorithms

1. **Implement the Algorithm:**
   - Create a new function in the appropriate module (`uninformed_search.py` or `informed_search.py`).
   - Ensure it follows the same interface as existing algorithms.
2. **Integrate with Agents:**
   - Update agent classes (`robot_agent.py`, `maze_agent.py`) to include the new algorithm.
   - Add a case in the `find_path` method to handle the new algorithm.
3. **Update Simulations:**
   - Modify the simulation scripts to accept the new algorithm as a command-line argument.
   - Ensure the UI reflects the inclusion of the new algorithm.

## Creating New Environments

1. **Define the Environment Class:**
   - Create a new file in `modules/environments/` with your environment class.
   - Define the environment's properties and behaviors.
2. **Implement Environment Logic:**
   - Include methods for generating the environment, managing obstacles, tasks, etc.
3. **Integrate with Agents:**
   - Ensure agents can interact with the new environment.
   - Modify agent classes if necessary to accommodate new environment features.

### Adding New Simulations

1. **Create a Simulation Class:**
   - In `modules/simulations/`, create a new simulation class inheriting from `SimulationBase`.
   - Implement the necessary methods (`handle_events`, `update`, `draw`, etc.).
2. **Design the Simulation:**
   - Define how agents and environments interact in your simulation.
   - Incorporate UI elements as needed.
3. **Create a Main Script:**
   - Add a new script in `examples/` to run your simulation.
   - Handle command-line arguments and initialize the simulation.

### Adjusting Visualization and UI

- **Customize Colors and Styles:**
  - Update `constants.py` with new color constants.
  - Modify drawing methods in simulations to use new styles.
- **Add UI Elements:**
  - Incorporate new buttons, panels, or information displays in the simulations.
  - Ensure they are responsive and update appropriately during the simulation.

### Improving Performance

- **Optimize Algorithms:**
  - Analyze the existing search algorithms for efficiency.
  - Implement optimizations such as heuristic improvements in A*.
- **Manage Resources:**
  - Ensure proper handling of resources, such as closing Pygame windows and quitting gracefully.

---

# Troubleshooting and FAQs

### 1. Pygame Window Not Displaying Properly

- **Solution:** Ensure that Pygame is correctly installed. Update Pygame to the latest version:

  ```
  pip install --upgrade pygame
  ```

- **Check for Errors:** Run the script from the command line to view any error messages.

### 2. Import Errors or Module Not Found

- **Solution:** Verify that the directory structure is correct and that `__init__.py` files are present in all module directories.
- **Python Path:** Ensure that the script adjusts the Python path to include the parent directory:

```
current_dir = os.path.dirname(os.path.abspath(__file__))
parent_dir = os.path.dirname(current_dir)
sys.path.append(parent_dir)
```

### 3. Simulation Closes Immediately

- **Solution:** Check for exceptions or errors printed in the console.
- **Event Loop:** Ensure that the simulation's main loop is correctly implemented and that event handling is properly set up.

### 4. Agent Not Finding a Path in Maze Solver

- **Solution:** Adjust the maze's complexity and density to create solvable mazes.
- **Algorithm Limitations:** Some algorithms like DFS may not find a path in complex mazes due to their nature.

### 5. Slow Performance in Large Mazes

- **Solution:** Reduce the maze dimensions or complexity/density to improve performance.
- **Optimize Code:** Look for sections in the code that can be optimized for better performance.

---

# Contributing

Contributions to the **CSE366 AI Lab Simulator** are welcome! If you have ideas for new features, improvements, or bug fixes, please follow these steps:

1. **Fork the Repository:**
    - Create your own fork of the project on GitHub.
2. **Create a New Branch:**
    - Work on a new branch specific to your changes.
3. **Implement Your Changes:**
    - Follow the project's coding style and conventions.
    - Ensure your code is well-documented.
4. **Test Your Changes:**
    - Run existing simulations to ensure your changes don't break anything.
    - Add new tests or simulations if applicable.
5. **Submit a Pull Request:**
    - Provide a clear description of your changes and the motivation behind them.

- The project maintainers will review your pull request.