NED University of Engineering and Technology

CS-323 - Artificial Intelligence

Final Project Report

Path Finding Algorithms Visualizer

With visualizations for 5 AI search algorithms

Sohaib Ahmed Abbasi
CS-19096

Ishaq Kamran
CS-18073

Table of Contents

Abstract

This Path Finding Visualizer is a GUI application for visualizing some famous path finding / search algorithms. It can be used for understanding the working of search algorithms and learn them in a fun and interactive way. The algorithms included are:

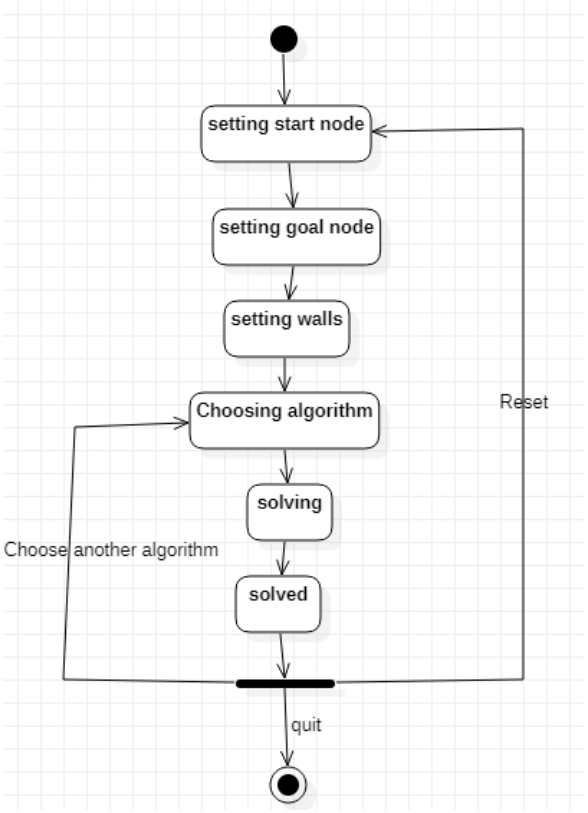
1. depth first search,
2. depth first search,
3. greedy best first search,
4. uniform cost search,
5. A star search.

Technologies / Languages Used

This application is developed using the python programming language. It specifically uses the pygame module of python for graphics/visualization.

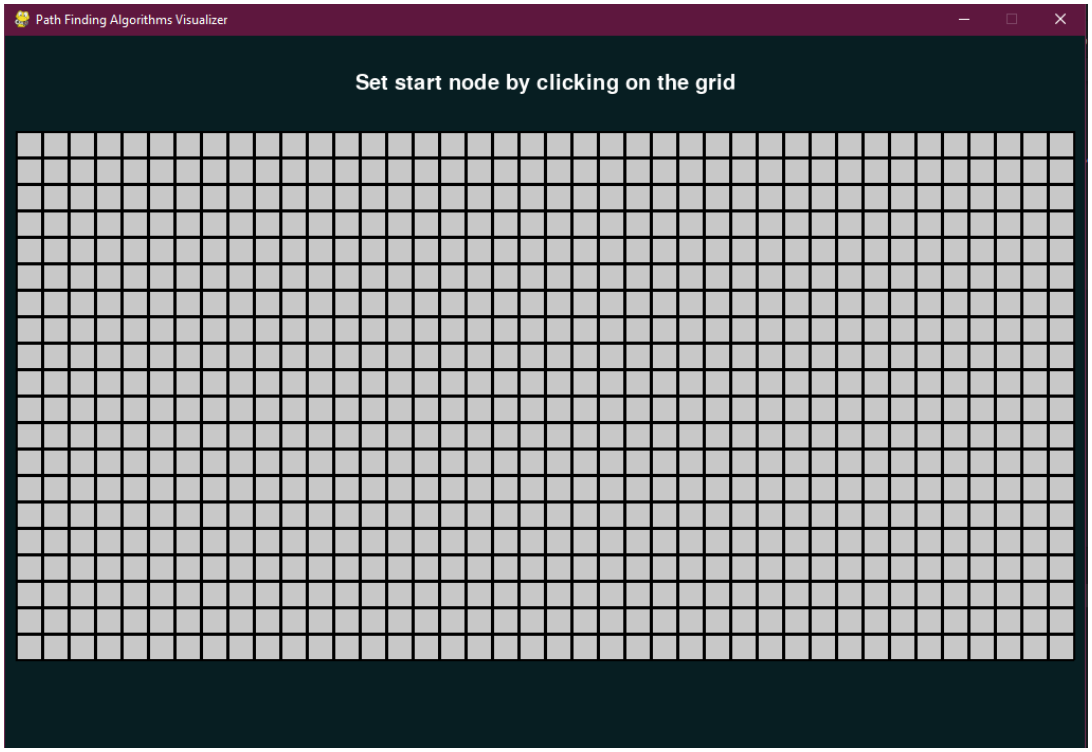
Project Flow (with screenshots)

The program transitions through some states, as shown by the state transition diagram below. To determine what state the application is in, Boolean variables like `setting_start, setting_goal, choosing_algo,` etc. These variables are present in main.py. Explanation for each state with screenshots of the application is given below.



Setting Start Node

Click anywhere on the node to set the start node, represented by red color.

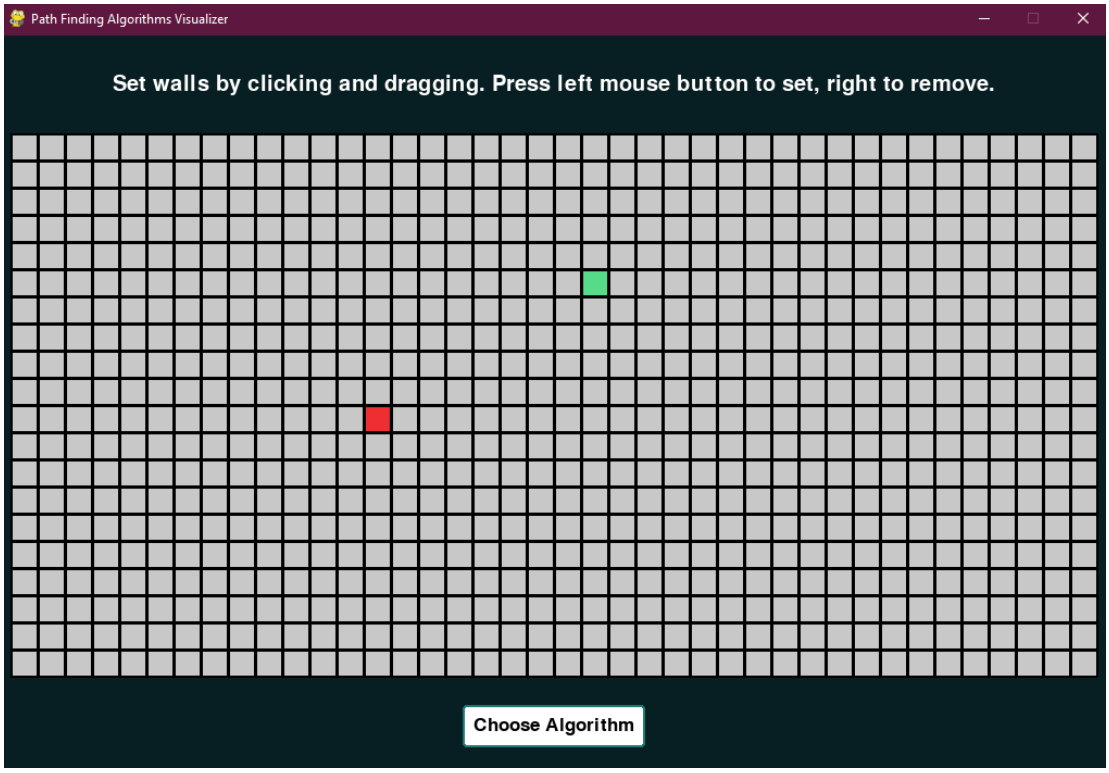


Setting Goal Node

Click anywhere on the grid to set the goal node, represented by green color.

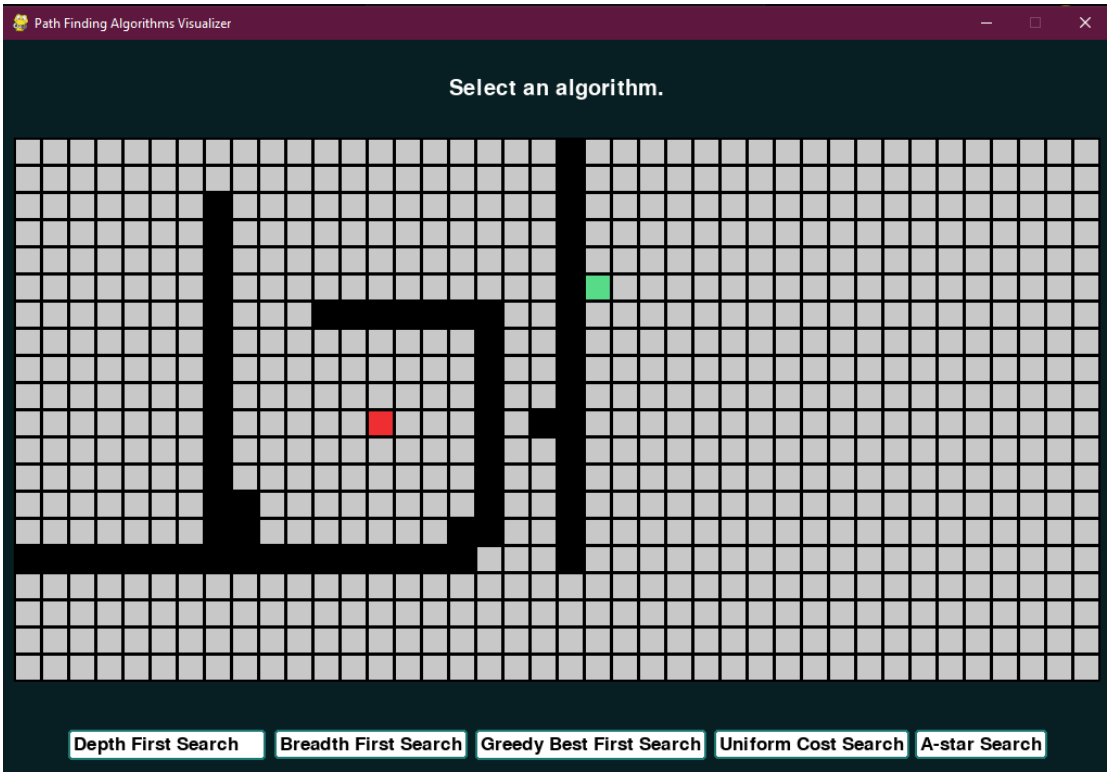
Setting walls

Click and drag with left mouse button to set walls and right mouse button to remove existing walls. Walls are represented by black color. After setting desired walls, click on choose algorithm button below the grid.



Choosing Algorithm

Choose an algorithm from the ones given below the grid. Clicking on any algorithm's button starts visualization of search from start to goal node using that algorithm.

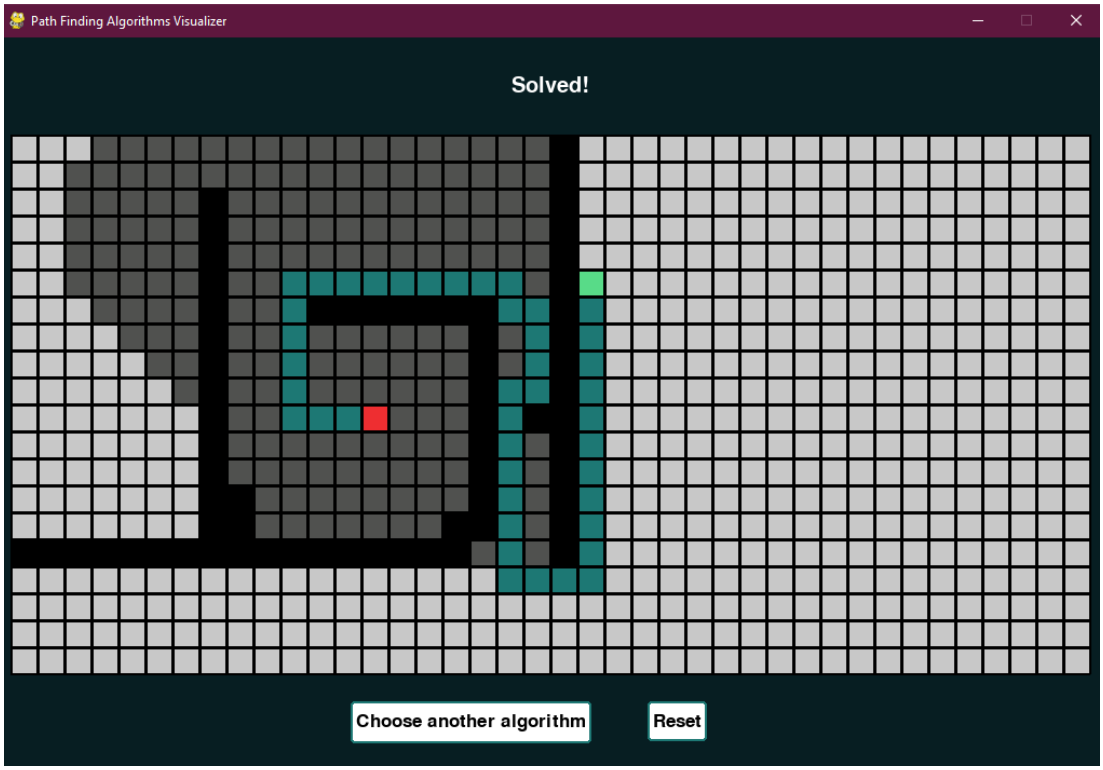


Solving

Visualization of algorithm as it searches path from start to goal is shown. Visited/explored nodes are shown in grey color.

Solved

Result of the search is shown. Grey nodes represent visited nodes and blue nodes represent the path found. Depending on whether there is a possible path or not, a message is displayed on top of grid. Below the grid, options to choose another algorithm (to run with same position of start, goal and walls) or reset (to reset the grid) are given.



Explanation of the AI algorithms

The AI algorithms used in this project are related to “Search”, i.e., they are algorithms that search for some solution. In this application, the thing that the AI algorithm is searching for is a path from start node to goal node.

Generally, AI search algorithms working can be explained with the following pseudocode:

```
Repeat Forever:
    1. If the frontier is empty,
        Stop. There is no solution to the problem.
    2. Remove a node from the frontier. This is the node that will be considered.
    3. If the node contains the goal state,
        Return the solution. Stop.
       Else,
        - Expand the node (find all the new nodes that could be reached from this
        node) and add resulting nodes to the frontier.
        - Add the current node to the visited set.
```

The difference between different AI search algorithms is due to the way that they do step 2 in the above algorithm, i.e. which node they choose to remove from the frontier. This has a big impact on the amount of time spent, space used, and optimality of the path found.

Depth First Search

Uses the stack data structure as frontier. i.e., the node that will be removed at any point will be the one at top of stack. Stacks follow last-in-first-out principle. Depth first search selects one path and exhausts it. If goal is reached on that path, it returns it, otherwise it backtracks and chooses another path and then repeats the process.

Breadth First Search

Uses the queue data structure as frontier. i.e., the node that will be removed at any point will be the one at front of queue. Queues follow first-in-first-out principle. Breadth first search visits nodes that are at the same distance / depth “x” from the start node before visiting nodes that are at a distance / depth of “x+1”.

Greedy Best First Search

Uses a priority queue ordered by a heuristic function. i.e., at any point, the node removed from the frontier will be the one with lowest value of the heuristic function. A heuristic function is an estimate on the effort required to reach goal node from current node. The heuristic function used in this application is Manhattan distance heuristic, which means that heuristic of any node is the Manhattan distance between it and the goal node. Greedy best first search "greedily" moves towards the goal which makes it very fast in cases where there are no walls but it may get slower when there are walls.

Uniform Cost Search

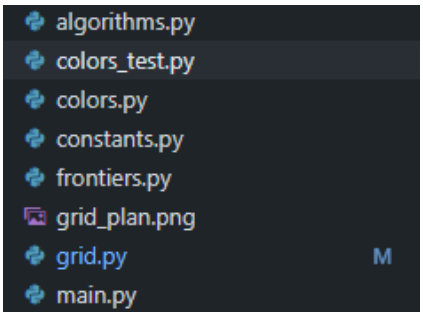
Uses a priority queue ordered by the cumulative path cost from start to the node under consideration. i.e. at any point it removes the node with least path cost. Uniform cost search maintains no visited list but it has a closed list, in which nodes that are visited and sure to not be goal are added.

A-Star Search

Uses a priority queue ordered by sum of a heuristic function and cumulative path cost from start to the node under consideration. i.e. at any point it removes the node with least sum of heuristic and path cost. A-star search just like uniform cost search maintains no visited list but it has a closed list.

Project Structure

All of the project files are in the root folder of the project.



algorithms.py

Contains logic for the sorting all the sorting algorithms. It has a class for all the algorithms. A base class `SearchAlgorithm` is made, from which all the other algorithms' classes inherit.

colors_test.py

A testing file, not used for the main application.

colors.py

Assigns RGB color codes to the color variables used throughout the application.

constants.py

Defines some constant values used throughout the application.

frontiers.py

Contains classes for frontiers used by search algorithms in the `algorithms.py` file.

grid.py

Contains a `Node` class representing each node in the grid / search space and a `Grid` class representing the entire grid. The `Grid` class contains `Node` class.

main.py

Contains main logic of the application and handles all the states and transitions between them. This file is to be run to execute the application.

Class Diagram