**CPSC 481 Artificial Intelligence Final Project**Jason Meadows, Jarrod Burges, Jay Vang, Arturo Salazar

12-8-23

Department of Computer Science, California State University, Fullerton

**Problem Statement**

It can be difficult to conceptualize the workings of graph search algorithms. In all but the simplest cases, it is difficult and time consuming to trace their operation by hand. An interactive application can be an excellent way to visualize and experiment with the behavior of various graph search algorithms to ascertain their strengths and weaknesses. By implementing various algorithms in an interactive two-dimensional grid-based pathfinding program, a user can quickly visualize different techniques and select the best one for their own needs. By tracking various metrics, a quantitative evaluation technique can be applied to compare the relative performance of the algorithms.

**Approach**

Python was used to code each algorithm, and the Pygame library was used to visualize and animate their operation. First, the pathfinding algorithm agent can traverse the grid to find the goal by implementing a maze-like game environment where a grid of N x N size is created with a start point, end goal, and obstacles either placed by the user or randomly selected. Despite knowing the start and end goal, the pathfinding agent can only partially observe the playable maze grid, and can only see or act on the environment with single-cell movement in the cardinal directions in a deterministic action. Since the pathfinding algorithm is based on previous moves, this is also episodic, and the environment is static since the obstacles and goal do not move. To compare how algorithms can vary in usefulness and accuracy depending on the environment, multiple implementations were added to the software. These are the graph search algorithms A\*, Dijkstra, breadth-first search, and depth-first search.

**Description of Software and Evaluation**

We wanted our application to be user-friendly and use components familiar to users, so we opted to use a library on top of the game engine called Pygame GUI which offered some primitive user interface (UI) components such as buttons, text boxes, drop-down boxes, and a slider. The environment the pathfinding agent is required to traverse is an N x N grid with preset sizes of 20, 40, 80, 100, and 200 that are user-selectable. This central grid component is drawn entirely using built-in Pygame graphics functions. The user is able to select a unique start and end point and any number of obstacles using the drawing tools in the UI. The user must also select the desired algorithm for the agent to use with the provided drop-down box. Once the ‘Find Path’ button is pressed, a performance timer starts counting, allowing the application to display the pathfinding algorithm’s execution time and the total number of spaces traversed, which allows for direct comparisons between the algorithms. These metrics are shown on the bottom left of the UI in milliseconds and steps. The path length returned by the algorithm is also shown. The main grid area of the UI, visible in Figure 2, shows the pathfinding route taken in purple and grid spaces explored in teal, a darker shade of teal marks the spaces in the frontier set.

The entire program was created using six Python files, which contain respectively, the main game loop and UI, a node abstraction, the pathfinding algorithms, settings, grid functions, and utilities. The node and pathfinding algorithm files contain the main logic to implement the pathfinding algorithms. The utilities file contains an implementation of Bresenham’s method of line rasterization to solve UI issues we were experiencing when building the tools to allow the user to draw obstacles. Prior to the inclusion of this function, lines drawn would often have gaps if the cursor was moved quickly by the user.

**Implementation Challenges**

One of our important early goals was to separate out the drawing and rendering functionality from the pathfinding implementations. It would have been possible to write the pathfinding algorithms such that the functions responsible for the pathfinding were also directly drawing to the screen, but that would mean our runtime analysis would really be timing the drawing functions more than timing the pathfinding algorithm performance. By decoupling the rendering and drawing components from the pathfinding algorithms we were able to get a more realistic view of how long the pathfinding algorithms were taking in comparison to one another.

We initially started development with one large monolithic Python file containing everything but it became quickly clear that it was hard to collaborate with one giant block of code and so we split out various parts into their own Python modules. We worked in separate branches on a shared Github repository. This also led to a small issue when it came to dividing up the implementation of our pathfinding algorithms. We ended up using slightly different abstractions for the nodes/grid spaces and rendering logic had only been written for the newer version of the node class. We addressed this without substantially refactoring by creating a translation function to convert between the two formats.

One key component of the A\* algorithm is the use of a heuristic function and in our case we implemented it first using Euclidean distance as the heuristic but given that our approach only uses vertical and horizontal moves along the grid and does not include diagonals, the Euclidean distance tended to have some edge cases that led to sub-optimal pathing decisions, not reliably providing the shortest path. We then changed the heuristic function to Manhattan distance which then performed as we initially expected.

**Evaluation Metrics**

While the visualization alone provides a lot of insight into the strengths and weaknesses of each pathfinding algorithm, we wanted to track and display relevant quantitative metrics as well. To achieve this, we used Python's time module, specifically the perf\_counter() function within that module. Each time the "Find Path" button is clicked, the currently selected pathfinding algorithm from the drop-down box is run, when this happens we call time.perf\_counter() just before and just after the pathfinding function executes. This helped isolate the function's runtime, which was then printed to the user interface text box at the bottom of the user interface. Additionally, we tracked the path length returned and the number of steps performed by the algorithm. Tracking these values helped to clearly demonstrate the advantage of the A\* algorithm’s informed search approach with an appropriate heuristic. This was particularly evident on larger grids. On the larger grids, it was not uncommon to see A\* running in a quarter or less of the time that other algorithms took. Here is an example of the output for each algorithm on the same 200 x 200 grid with a large wall obstacle in the center:

Breadth First Search: Path length = 275, ran in 244.69 milliseconds, performed 71,691 steps.

Depth First Search: Path Length = 9961, ran in 2029.81 milliseconds, performed 41,271 steps..

A\*: Path length = 275, ran in 54.82 milliseconds, performed 33,074 steps.

Dijkstra's: Path length = 275, ran in 129.33 milliseconds, performed 72,062 steps.

As expected, BFS, A\*, and Dijkstra's all found the shortest path length of 275, with A\* being the fastest to execute and performing considerably fewer steps than the other algorithms. DFS performs quite poorly here, both in terms of the length of path returned and the runtime.

**Conclusion and Future Work**

The application works as intended and accurately displays the steps of each search algorithm as well as metrics to evaluate their relative performance. The user is given an engaging set of tools to rapidly create grid environments where the pros and cons of these algorithms can be observed and evaluated. We feel it is a helpful tool for quickly gaining an intuitive sense of the capabilities of the included pathfinding algorithms.

Additional features could be added that would allow this application to even better perform its intended role. One such feature is the ability to evaluate diagonal paths. This would allow other heuristics for the A\* algorithm, such as Euclidean distance, to be evaluated and compared. A tool to add weights to some nodes would help to better showcase aspects of Dykstra’s algorithm that are not demonstrated in the current version. Another helpful feature would be an alternate display mode that puts multiple versions of the same grid side by side, so that algorithms could be viewed running simultaneously.

**References**

PyGame. (n.d.). Python game development. Retrieved from <http://www.pygame.org>

PyGame GUI. (n.d.). User interface component library for PyGame. Retrieved from <https://pygame-gui.readthedocs.io/en/latest/>

Van Rossum, G., & Drake, F. L. (2009). Python 3 reference manual. Scotts Valley, CA: CreateSpace.

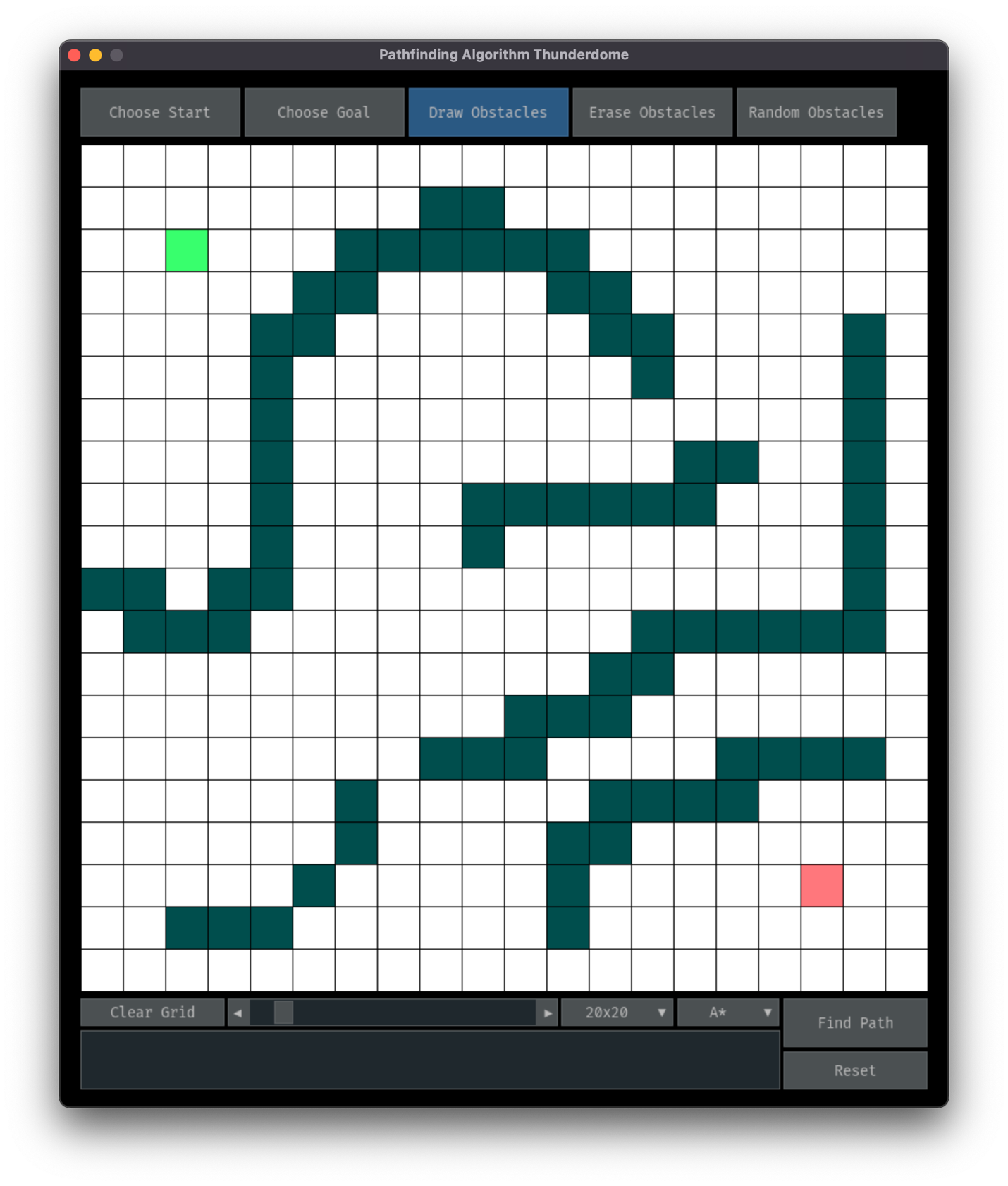


Figure 1 Example main user interface seen by the user while interacting with the program.

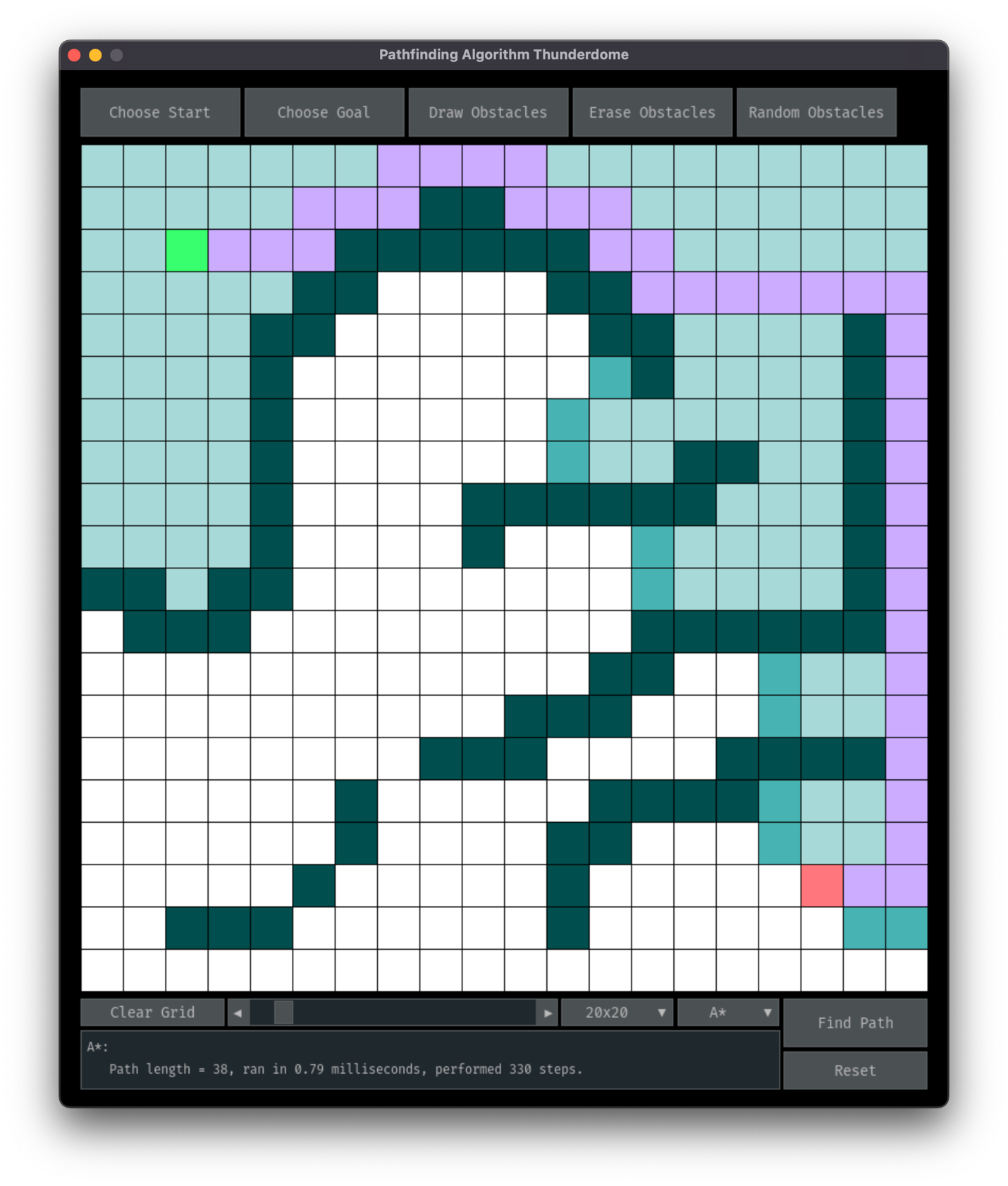


Figure 2 Example user interface seen by the user after selecting the A\* algorithm and clicking ‘Find Path’. The text box in the lower left displays the evaluation criteria.

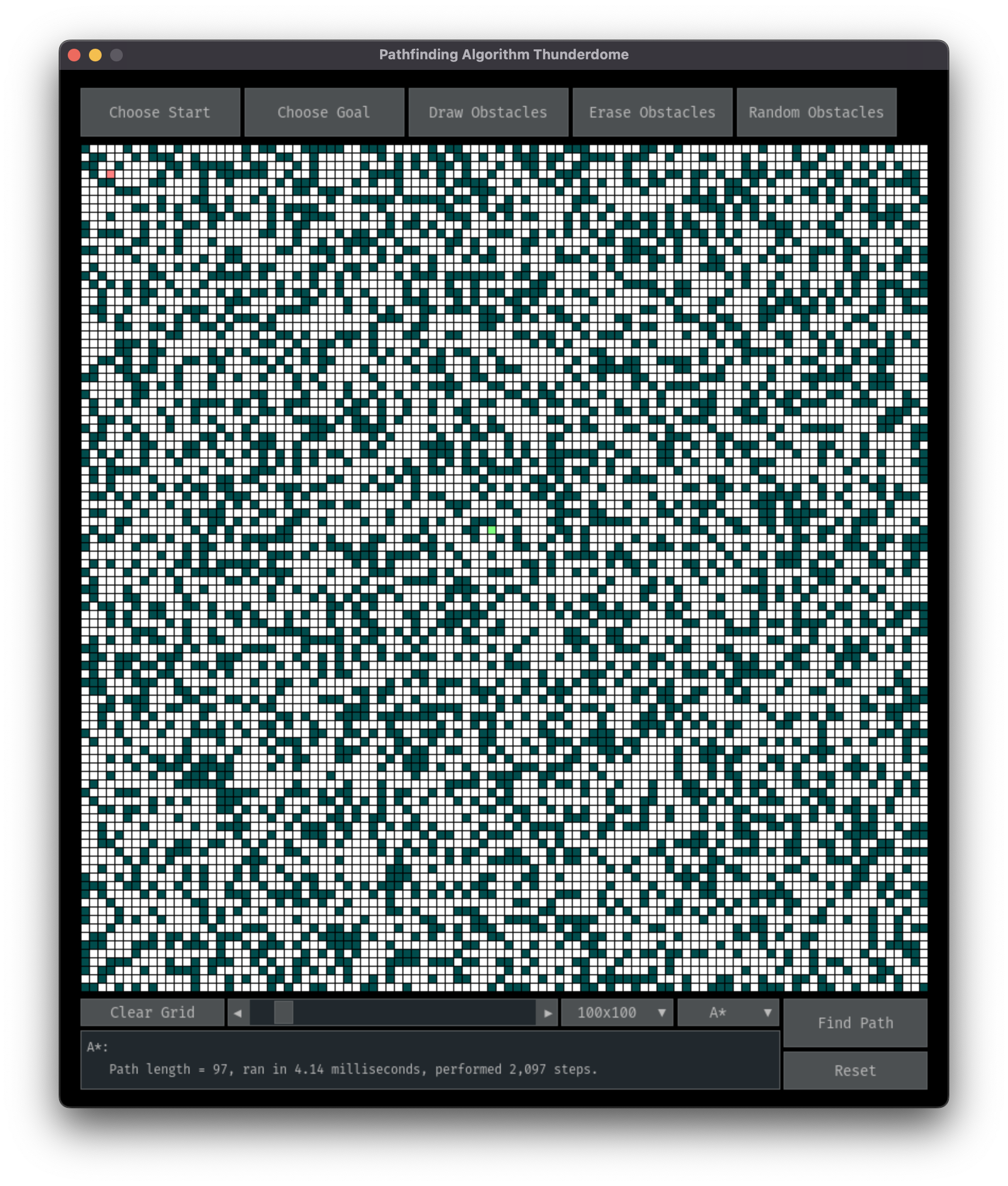


Figure 3 Example of the 100 x 100 grid size and clicking the ‘Random Obstacles’ button

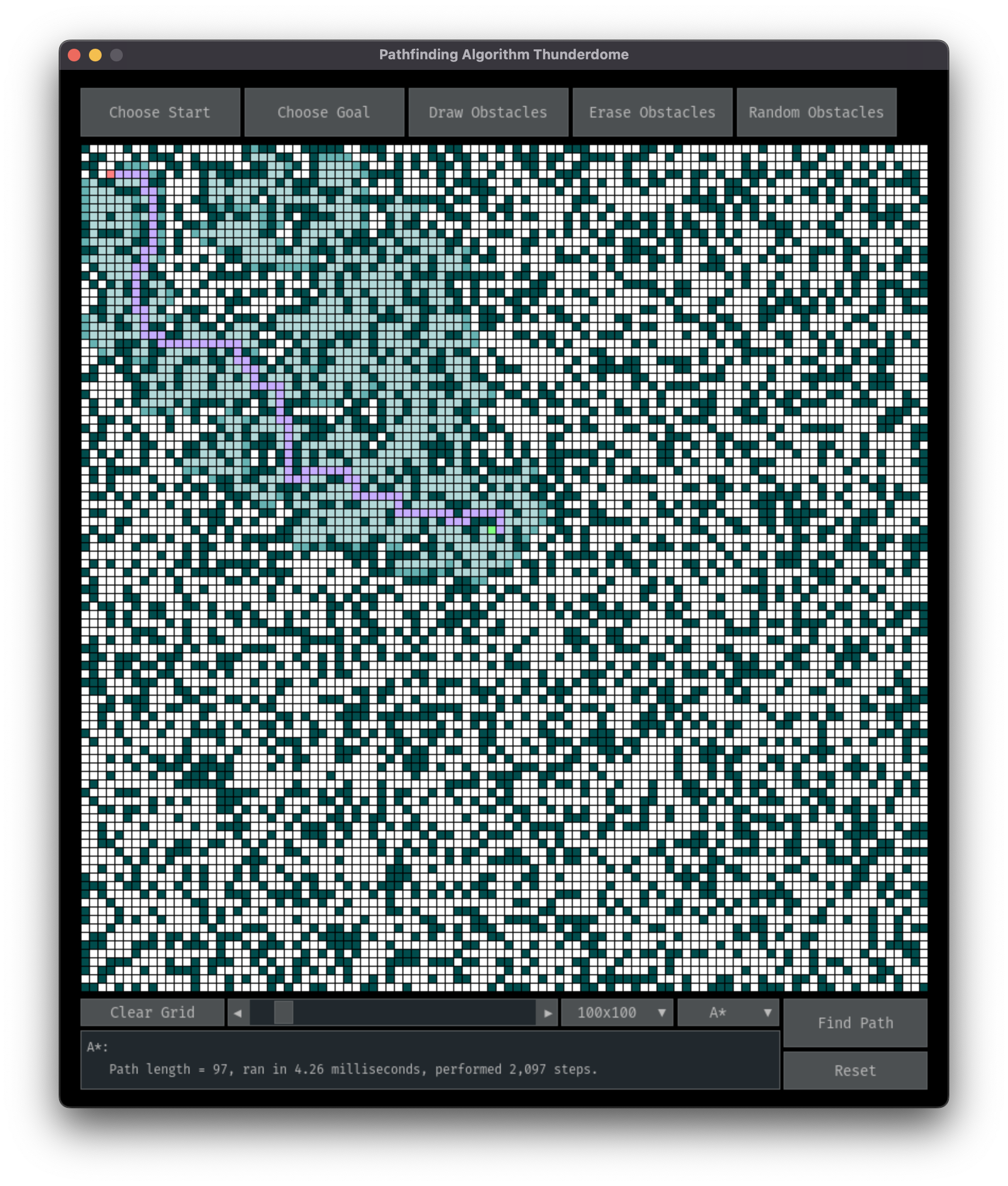


Figure 4 Example of the 100 x 100 grid size with ‘Random Obstacles’ after clicking ‘Find Path’. The text box in the lower left displays the evaluation metrics.

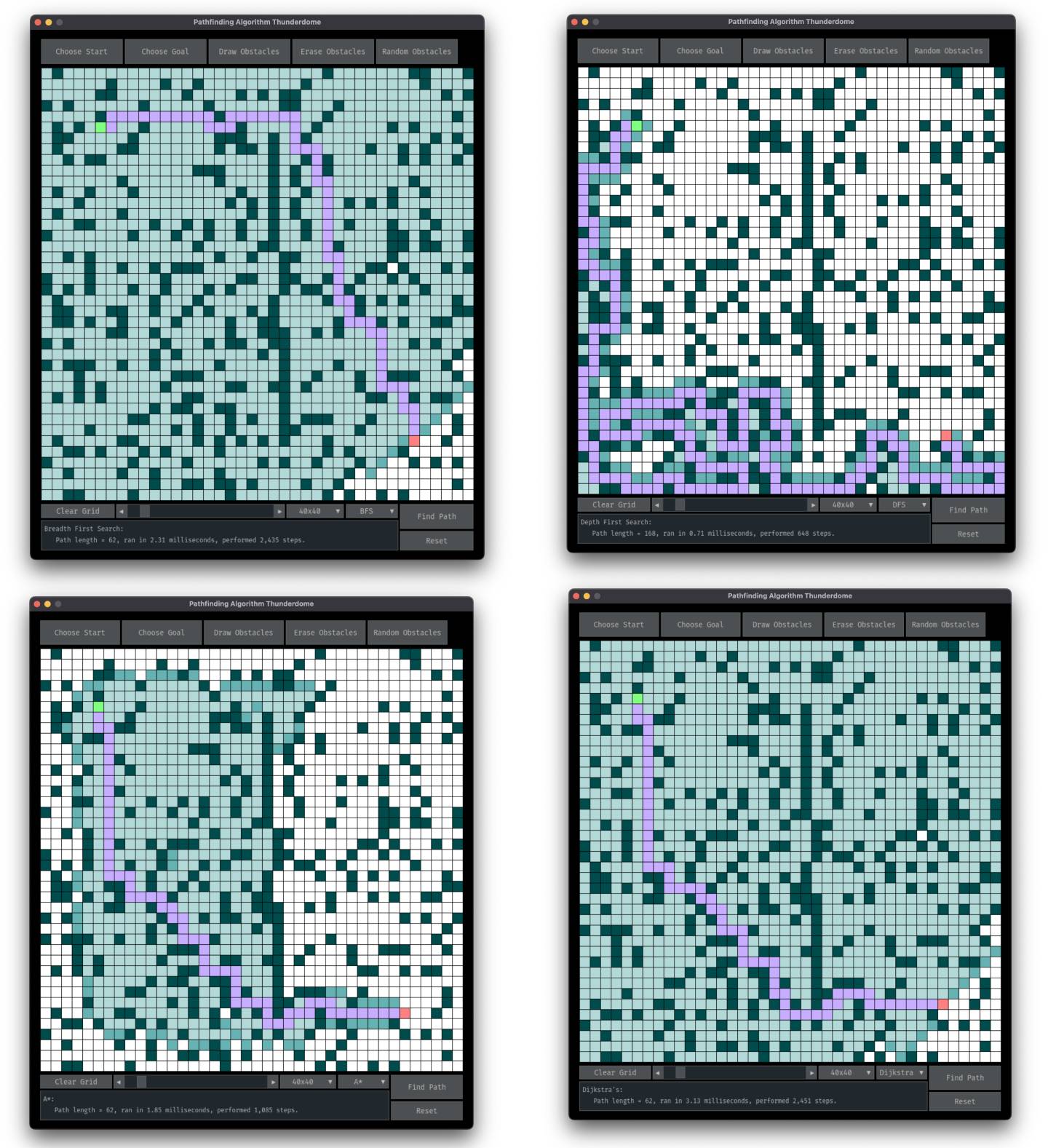


Figure 5 The output of each of the four pathfinding algorithms on a 40 x 40 grid of random obstacles. Top Left, BFS. Top Right, DFS. Bottom Left, A\*. Bottom Right, Dykstra's Algorithm. Note that while the optimal shortest length path is returned by all except DFS, the specific path of that length that is chosen is not always the same.