Dangerously Smart Academics

Theodore Jaffee, Jaiden Magnan, Nikhil Iyer

Github: https://github.com/Nikhil-42/dsa-final-project

Youtube: https://youtu.be/V1wPiNXH1xA

DSA Final Project Report

Problem and Motivation

The 3D graph algorithm visualizer is trying to solve how an enemy agent could locate the player in a maze. The problem arises when the enemy is trying to search for the shortest-path over varying weights.

Features, Algorithms, and Description of Data

The program randomly generates a 317 x 317 2D maze with different colors representing different terrains. The edge weight of each terrain is determined by the color saturation of the pixels. The walls of the maze are generated with a DFS to ensure that every odd index of the maze has a path to every other odd index. The maze is output as a png file, then traversed over to generate an adjacency list. The following algorithms were then performed on this adjacency list with video output:

- Breadth First Search
- Depth First Search
- Dijkstra’s Algorithm
- A* Algorithm

The user can input command-line arguments to specify which algorithms they want to output in the video. Rotations are performed on these algorithms to ensure that starting position does not create bias in the comparisons. The shortest path was then backtracked and traced over with black.

At this point there is a generated mpg video showcasing the execution of each rotation, path visualization, and terrain. Furthermore, there is the original black / white png saved to generate the 3D version of the maze. To integrate these items into 3 dimensions we used Panda3D, an open-sourced framework for rendering and games. The original image served as a heightmap to represent the walls of the maze. The colored video was then used as a texture over the maze to show the coloring of each algorithm. These features essentially created a 3D visualization which could be moved through in the first-person.

Tools Used

- Panda3D: an open source rendering software used to transform the 2D maze.
- Numba: a JIT compiler for python and NumPy to speed up maze generation.
- NumPy: a general-purpose mathematical library which we used to store our adjacency list and other computational tasks
- FFmpeg: a video formatting tool we used to visualize the algorithms over our maze.
- Pillow: a Python imaging library to generate our maze as an image.
- Python: the programming language of choice for the project.
- Blender: a 3D rendering software to create the player.

Distribution of Responsibility

Nikhil focused on the original maze generation and video output using FFmpeg. He generated the maze with Numba, to enable ASM caching and decrease the time taken to output the maze image. He also determined the weights for each terrain and performed most of the mathematical calculations necessary for the algorithms. Nikhil worked with Jaiden to implement each of the algorithms and simplify / clean-up the code. Nikhil also created the assets for the player running across the maze in Blender. Panda3D camera controls and player animations were also completed.

Jaiden emphasized the implementation of the 3D rendering of the maze. He used the maze image as a heightmap and rendered the terrain in Panda3D. However, this was eventually changed from a heightmap to an array of triangle vectors. He then laid the video generated by Nikhil over this terrain to visualize the colors. Furthermore, Jaiden randomly generated the colors for the terrains and implemented the rotations of the algorithms. Presentation components were also heavily managed by Jaiden including the Github and Report.

Theo came up with several ideas for the project, including the original idea for a 3D visualization and which assets to use for the maze. He worked on the original maze generation in Blender before we decided to switch to Panda3D. He implemented the depth-first search algorithm, and debugged throughout. Finally, he assisted with the creation of the final report.

Analysis

Between the project proposal and the finished project, we made a number of important changes. We originally planned to generate the entire maze in Blender, but we realized that it would be more easily implemented in Panda3D. We originally planned to have an immovable, top-down simulation, but realized that it would not be visually appealing with such a large maze, so we switched to a movable, "game-like" simulation. We originally planned to visualize two algorithms in an adversarial way (i.e. whichever one reached the center first "wins"). Later, we added three more algorithms and let them all run to completion, rather than ending when one reached the center.

We used five algorithms: breadth-first search, depth-first search, Dijkstra's algorithm, and A* with the Manhattan distance heuristic. Both breadth-first search and depth-first search have a time complexity of $O(V+E) = O(E) = O(V)^{\dagger}$, where V is the number of vertices and E is the number of edges. Dijkstra's algorithm was implemented with a priority queue and has a time complexity of $O(V \log V)$. A* also has a time complexity of $O(V \log V)$, as it is a generalization of Dijkstra's, although it is typically faster in practice.

We also used a variety of functions throughout the project to complete core tasks. generate_maze() procedurally generates a maze from a NumPy array using depth-first search until all nodes have been visited, so it has a time complexity of $O(V)^{\dagger}$. add_terrain() adds color and texture to the maze and weights to the underlying graph. Its time complexity is $O(L^2)$, where L is the side length of the maze. build_adjacency_list generates an adjacency list from a maze, which takes O(V). search(), our pathfinding function, accepts any algorithm as a lambda, so its worst-case time complexity is O(S) where S is the complexity of the algorithm that is running. All of our other code, including writing files and other boilerplate code, has a constant time complexity in terms of the edges and vertices of our graph.

Because the maze is represented as an image and there are at most 4 edges for each pixel E is O(V) and vice versa.

Reflection

The experience overall was a learning experience focused towards implementing the theories and concepts we learned in class. It was incredibly important to choose the correct libraries and tools necessary to receive a desired output. I also learned that it is sometimes better to change something entirely than to force it to work smoothly.

The biggest example of this was rendering and animating the algorithms inside of Blender. It was slow and took over five minutes to generate the maze each time it was needed. It was determined that the software was too slow and too complicated to be practical. We searched for a game engine that would better serve our purpose and came across Panda3D. Panda3D was easy to figure out and worked almost instantaneously with just a few lines of code.

Another change throughout the project was with the video output. We originally used OpenCV2 to create an mp4 but it ended up cropping the video. Nikhil then determined that we should use FFmpeg instead which produced a more true video output.

Jaiden: I learned about visual output and 3D rendering, specifically how to use a heightmap (which we eventually changed). Nikhil taught me how to use iterators to visually display each step of an algorithm. Furthermore, I feel I could use all of the tools we used pretty effectively which I never touched before, including but not limited to NumPy, Numba, Panda3D, and FFMpeg.

Theo: I learned a lot about the software development workflow in general—using Git and GitHub for version control, using IDEs together with the terminal, collaborating with others, writing clean code, and iterating to get the best result. I also learned a lot about specific tools and frameworks we used: Blender, Panda3D, and FFMPEG among them.

Nikhil: Similar to Jaiden I learned a lot about video encoding and 3D rendering (did you know most video encoders require and even resolution?). I learned about working code-only in 3D with Panda3D, which was a unique experience. This project was a learning experience especially with respect to integrating several tools into a complete application that performs a function we defined.

<u>References</u>

- Maze Generation Logic: https://inventwithpython.com/recursion/chapter11.html
- Panda3D Documentation: https://docs.panda3d.org/1.10/python/index
- COP3530 Slides for graphing algorithms