Daniel Ortega

CPSC 490 Senior Project Final Report

Advisor: Scott Petersen

Making Music with Pathfinding Algorithms

Abstract

Pathfinding, at its core, is finding a path between two points. More narrowly, it is finding the shortest route between two nodes of a graph. This is applicable in the context of mathematical graphs: whether they are weighted, directed, or otherwise. In computer science, many different pathfinding algorithms have been devised with varying runtimes and optimality.

When first learning how pathfinding algorithms work, many students can be confused how different algorithms search the sample space. This naturally has led to the creation of pathfinding visualization programs. While these visualizations often help people better understand the general concept of how each algorithm searches for a path, adding some other kind of representation might be able to further enhance that understanding.

As a result, I created a musical visualization of various pathfinding algorithms as they progress from start to finish. Within a given grid made up of cells, each cell contains a musical note which will be played when an algorithm "visits" that cell. This produces unique musical output for various pathfinding algorithms, distinguishing between them and possibly allowing students to better understand how each algorithm works.

The program's inputs include the following: which pathfinding algorithm is used, where walls are placed, the start node, the end node, and the pitch value of each node. With experimentation, I created grid configurations which, when played by a pathfinding algorithm, sound like music. Organizing pitches within the grid to create chords and higher-order structure resulted in pattern-like arpeggios which vary depending on the algorithm chosen.

Introduction

Pathfinding

Breadth-first search (BFS) and depth-first search (DFS) both are guaranteed to find a path between a start and end node, given that a path exists. These algorithms work by searching an entire graph, exhausting all possible nodes. This means that for a graph with V vertices and E edges, BFS and DFS will run linearly in $O(V + E)$ time. However, there exist more efficient algorithms, such as Dijkstra's algorithm and A*. Conceived by Edsger W. Dijkstra in 1956, Dijkstra's algorithm is guaranteed to find an optimally shortest path between two nodes. It does this via dynamic programming, finding the shortest path from the starting node to every node in the graph. Optimized, its runtime is $\Theta(E + V \log V)$.

A* is a variant of Dijkstra's algorithm which uses a heuristic to improve the pathfinding behavior. The heuristic estimates the cost of the cheapest path from the current node to the finish node. In the worst case, A* runs in $O(b^d)$, where b is the average number of successors per state factor and d is the depth of the shortest path. However, depending on the quality of the heuristic, the runtime can be greatly reduced via pruning the search space.

Musical Graphs

The Tonnetz is a specially designed graph where each node contains a musical tone, as depicted in Figure 1. First described by Euler in 1739, every node is connected to six others by edges representing musical relationships in just intonation: either a perfect fifth, major third, or minor third. Because of the nature of just intonation, a syntonic comma separates the tuning of each row in the Tonnetz.

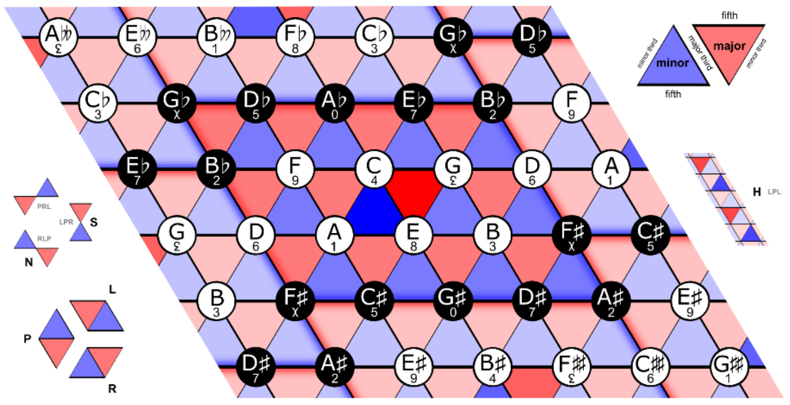


Figure 1: The Tonnetz

Combination of Disciplines

This project stems from the idea of combining musical graphs with the aforementioned pathfinding algorithms. For example, one might define a start node and end node on the Tonnetz, then traverse the graph with Dijkstra's algorithm. I wanted to try to create a program which accomplishes this musical pathfinding, but with room to customize and creatively experiment. In this paper, I will report on the process of developing a concrete vision, implementing, and planning for future work on the program.

Project Overview

I conceived this project to fulfill the senior requirement of CPSC 490 at Yale. My advisor for this project was Dr. Scott Petersen. At the beginning of the Spring 2022 academic semester, I asked Dr. Petersen what project I might work on, to which he recommended I take on a project I personally enjoy. For the first two weeks of the semester, I spent time brainstorming ideas, eventually settling on the project as it is presented the *Goal* section below.

As a Computer Science major, I undertook this project because I was interested in developing my skills in game design, version control, and documentation for professional development. More personally, I wanted to better understand how two of my interests, computer science and music, might intersect. As a result, my brainstorming led me to read research pertaining to pathfinding algorithms and musical graphs. This progressed into my idea for a project, which was to assign a musical note to each node within a graph, and play each note as a pathfinding algorithm visits each node. While there are countless visualizations of pathfinding algorithms on the internet, I could not find one which worked exactly like what I envisioned, so I decided to stick with the project idea.

After more research, I realized that my program could serve as an adjacent alternative to pathfinding visualizers, providing a new perspective for budding computer scientists who might be confused by the initial concept of pathfinding. The auditory aspect of my program gives a new dimension by which people might understand pathfinding algorithms better. This gave good cause to commit to working on the project.

Goal

To guide the development of the program, I outlined a minimum viable product (MVP) in my project proposal three weeks into the semester. This was a great help in deciding how to approach the project. An original mock of the MVP from February 2022 appears in Figure 2, and the outline of the MVP is as follows.

The MVP will be a program with a default grid of notes. The user will select a certain pathfinding algorithm from a menu, select start and end cells, and optionally draw walls into the grid for the pathfinding algorithm to avoid. Upon starting the algorithm, the program should play a note whenever it visits a cell as a candidate in any way, as opposed to after it has found a particular path. This should be accompanied by some sort of visual indication that that cell was visited in real time, such as a change in color of the cell. After a path is found, or one is impossible, the user might edit the start cell, end cell, walls, and algorithm, and then begin the program again.

Furthermore, besides programming the MVP, a secondary goal during the semester was to create grid configurations which sound like music, rather than random notes played in succession.

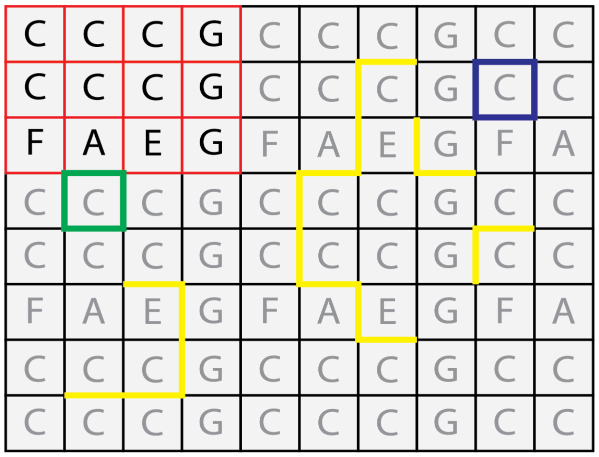


Figure 2: MVP mock. Red = tiled repetition of cells, Yellow = walls, Green = start cell, Blue = end cell

Implementation of Musical Pathfinding Program

With Dr. Petersen's advice, I chose to implement the musical pathfinding program using Python's pygame module and SuperCollider. SuperCollider is an environment and programming language used for audio synthesis and algorithmic composition, and I was familiar with the language because of taking Dr. Petersen's courses in the Computer Science department which used SuperCollider. This would allow for a workflow where Python took care of the program's UI and logic, while SuperCollider produced sound output in real time according to what was happening in Python.

To begin the actual implementation of the program, I searched among pre-existing pathfinding visualization tools online. This was a phenomenal baseline for rudimentary functionality, including the pathfinding algorithms themselves, visualized grid, and presence of walls. However, there were many more features which I needed to code. I will detail the general order in which I implemented these additional program features, and then elaborate on the design decisions behind each feature in the subsequent section. A less analytical, but more complete timeline of my work is available in the documentation files of the project's GitHub repository.

Design Decisions

MVP

First, I needed to add a method for the user to draw and remove walls. This suggested an implementation where each node stores a state in an IntEnum which defines it as either normal, start, end, or wall. To draw and remove walls, I tracked mouse-up and mouse-down events in pygame, calculated which node the mouse was hovering over, and then updated the relevant node's state.

Next, the user had to be able to switch between editing walls and editing the placement of start and end nodes. I chose to implement different "draw modes" for this purpose. The draw mode was stored as an IntEnum in Python to keep track of what the user wanted to do when clicking on the grid. Once this was done, the process of updating the start and end node was straightforward due to its similarity to how walls were updated with mouse-up and mouse-down events.

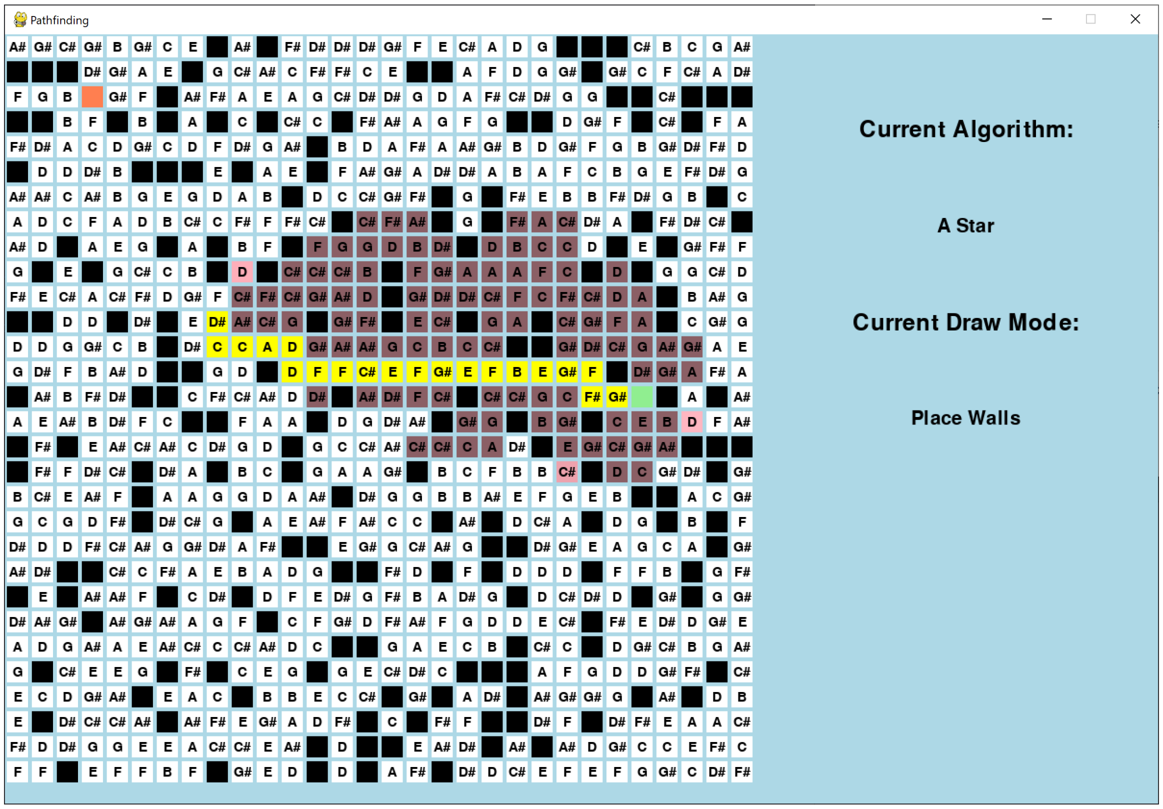
The most essential part of the program, then, was generating sound output from SuperCollider. This was accomplished by sending Open Sound Control (OSC) messages from Python whenever a node was visited by the pathfinding algorithm. These OSC messages, containing the desired note value, would be received by an OSCFunc in SuperCollider, which then triggered a predefined Synth to play the correct tone.

Because the MVP was completed in about four weeks, there was room for additional functionality to be added. Figure 3 depicts the MVP with randomized walls and pitches. I had outlined stretch goals in my project proposal, so over the next three weeks, I implemented these features which would allow for more creative experimentation by users of the program. This would eventually allow for the creation of more pleasant-sounding, less random note sequences.

Additional Features

As the next desirable feature, I wanted the user to be able to customize the notes in the grid. The pre-existing setup of the draw modes in the program allowed for a natural extension which would give the user customization options. By adding two new draw modes, I added the functionality of selecting and deselecting groups of nodes. This required adding an additional value of "selected" to the IntEnum which tracks a node's state. Once a group of nodes was selected, I tracked keystrokes in pygame so that the user could type a pitch name to change all selected nodes to that pitch. For sharps and flats, I bound the up and down arrow keys to transpose all selected pitches up and down, respectively.

The last feature that I added was saving and loading grids from a text file. When creatively experimenting with different note and wall configurations within the grid, it was very difficult to make any progress without the ability to save the state of the grid in some way. As a result, I implemented the saving and loading functionality earlier than anticipated so as to more easily hear the effects of changing note and wall configurations in the grid in comparison to past grid states. Regarding implementation, this took much more time than anticipated, but it worked in the end by using Python's JSON library to read/write the grid's data from/to a .txt file.



Current Algorithm:

A Star

Current Draw Mode:

Place Walls

Figure 3: Musical pathfinding program with randomized walls and pitches

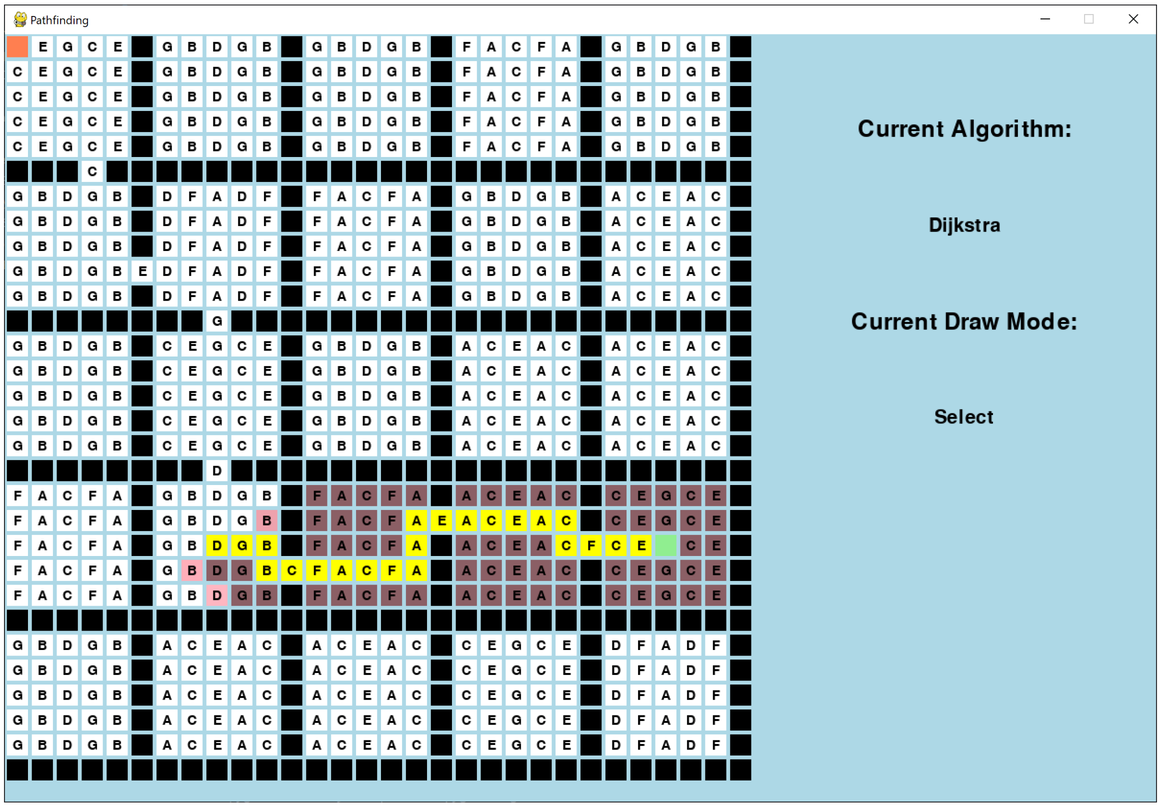
Challenges Faced and Solutions

Over the course of the semester, I met with Dr. Petersen in weekly progress check-ins, where we would address the current state of the project and take care of any roadblocks or necessary adaptations. While working on this project, the stretch goals set out in my project proposal at the beginning of the semester were modified due to unforeseen challenges, time constraints, and variance in judgment.

One challenge was the startup time of learning how to make and edit game UI in Python. To combat this, I completed hours of short online game development courses to gain familiarity with pygame. This allowed me to better understand how to manipulate the grid of my program, display text, and update visuals based on keystrokes of the user.

I also had some initial trouble with using OSC in SuperCollider. For example, at one point in the project, I changed the sound output from one long tone which varied in pitch over time into a pluck which would be triggered by every OSC message. At first, my implementation either created a new synth even though the previous synth was not freed, or would not trigger the gate argument in the synth. In the end, closely reading SuperCollider and OSC documentation allowed me to figure out how to make the Python program repeatedly trigger a singular synth with various pitches via OSC.

Then, the idea of implementing alternate grid layouts turned out to be more ambitious than I had initially thought. While I worked on a Python implementation which would allow for a variable number of edges to be associated with a given graph node, there was difficulty in finding a way to consistently display the layout via pygame. For instance, the actual search algorithms could be modified so that they search between six neighbors, resulting in a hexagonal grid. However, visualizing this in pygame would require a complete rework of the pygame display every time the program was run, which was undesirable.



Current Algorithm:

Dijkstra

Current Draw Mode:

Select

Figure 4: Musical pathfinding program with embedded chord areas

Finally, one area of difficulty was the thought process of constructing a grid which sounded musically "good," because playing random notes in a grid would not enhance anyone's understanding of pathfinding algorithms. I had to think about how to control the notes played by the program at a high level while also giving enough room for the pathfinding algorithm to make significant musical choices. So, I researched music theory and patterns which I might be able to translate and embed into the program's grid. Eventually, I settled on creating different areas of the grid which contain notes within a single chord. For example, in Figure 4, the pathfinding algorithm starts in C major, then moves to A minor, F major, and so on as it progresses through different areas of the grid. This gives the user a more concrete feeling of how the pathfinding algorithm moves through different sections of the grid.

Results

After experimenting with embedding chord structures into the grid, the program produced what some people might consider music. It is largely what I expected, which was something close to, but not quite, an arpeggiated chord progression. Nevertheless, this was much better than the output when the grid was randomly generated. Prior to adding certain features, such as customizing the pitch of specific nodes and saving the state of the grid, it did not sound like music, but rather a conglomeration of pitches.

Sample output from various grids is available in the documentation files of the project's GitHub repository, along with video demos of the program as it produces the sound.

Potential for Future Research

Although I successfully reached my goal of implementing the MVP, there is room for the program to be further built upon in the future. One basic functionality might be to export the sound from SuperCollider to an audio file or recording the note values played in the grid to a MIDI file. Still, there are even more interesting features which could allow for more musical creativity.

For example, it is possible to embed other aspects about each note being played in the grid. The user could affect the duration, instrument, timbre, and volume of each note, to name only a few qualities. The added complexity allows for much more variability in the sounds played by the pathfinding algorithm. Furthermore, one might even add additional, separate grids which run simultaneously and determine, for instance, pitch and duration based on separate pathfinding algorithms.

One more extension of the project could be adding a machine learning program which takes a MIDI file as input and constructs a grid with note distributions similar to the original MIDI file. This could result in random variations of the original MIDI which maintain the same musical feel but are not exactly the same.

Conclusion

The work that I have done for my senior project has been satisfying and exploratory. I still have much to learn about the relationship between pathfinding and algorithmically generating music, and the program itself has room to grow. However, I am pleased with what I have done in accomplishing the goal set at the beginning of the semester: completing a working program which generates music. As a result, I have greatly enjoyed working on my senior project.

Acknowledgements

Many thanks to my advisor, Dr. Scott Petersen, for guidance, instruction, and oft-unappreciated comic relief throughout my senior year at Yale. Thank you to my family: Mom, Dad, and Sophia, for the endless support given throughout my entire education and lifetime.

Resources

Papers

Collins, Tom, David Meredith, and Anja Volk. "Mathematics and Computation in Music." *Proceedings of 5th International Conference*, MCM. Vol. 9110. 2015.

Conklin, Darrell. "Music generation from statistical models." *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences*. 2003.

Lee, Sangkeun, et al. "Random walk based entity ranking on graph for multidimensional recommendation." *Proceedings of the fifth ACM conference on Recommender systems*. 2011.

Tymoczko, Dmitri. "The generalized tonnetz." *Journal of Music Theory* (2012): 1-52.

Wang, Mengsha, et al. "RNDM: A random walk method for music recommendation by considering novelty, diversity, and mainstream." *2018 IEEE 30th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2018.

Models and Examples

- https://supercollider.github.io/
- https://github.com/reddragonnm/pathfinding-music
- https://qiao.github.io/PathFinding.js/visual/