Introduction

This project includes a complete A* algorithm implementation with GUI representation. A* is one method for pathfinding within a graph or maze that uses informed decisions to dictate which path is created.

Our inspiration behind developing our own implementation of this specific algorithm revolves behind its applicability to video game development, and its intelligent decision making ability based upon passing information between three ADTs. A* relies upon deft manipulation of the ADTs we have studied in this course, and further utilizes a relatively simple idea to consistently make the best decision in a complex pathfinding environment. Beyond offering an interesting project for this class, A* is an algorithm we can easily add to and grow after the conclusion of the class. We discovered this algorithm during early research for this project, and were excited to see that it made excellent usage of the skills we have developed during this course.

Our project first displays the user with a graph that has start and end nodes colored in purple, with the fastest route identified by A* between the two represented by nodes colored blue. The graph additionally displays obstacles in black and nodes A* considered but did not visit in grey. The user is able to click nodes on the graph with the exception of the start and end nodes, converting the node they select to or from an obstacle, depending on its original state. A* then dynamically considers this addition to the graph, and replots the optimal route and nodes considered along the way.

How to Run

To run our A*, grab all three files (AStarDriver, AStarGUI, and AStarNode) and place them in a package called AStarMazeProject. Once that is complete, run the AStarDriver class if you want to run it as default.

Since the program is modular, if you wish to make certain changes to the behaviors you can. They are detailed below:

1. If you wish to make the graph larger, initialize WIDTHNODES or HEIGHTNODES to your new, different value at the top of AStarDriver. They are default at 20 each.
2. If you wish to change the starting location or destination location, modify the array value in initialization of those variables in the AStarDriver main method. They are default at [0,0] and [12,12] respectively.

3. If you want to add obstacles to run at start, you may do so in the AStarDriver main method. Simply follow the current “obstacles” patterns in (a) and (b). Keep in mind, however, that you may add obstacles within the GUI by clicking on them, explained later on.

```
a. nodeArr[1][1].setObstacle(true);
b. nodeArr[1][1].getButton().setBackground(Color.BLACK);
```

Design Decisions

A* has many implementations that vary in performance and purpose. The primary design choices are the selection of the appropriate heuristic equation for the graph being traversed, as well as the selection of the ADTs A* makes use of to traverse the graph.

All forms of A* include three ADTs; the types of which may vary. There is an ADT holding nodes that have already been considered (closed), one that holds nodes that can be moved to from the current node (open), and one that holds the neighbors of the current node(neighbors). Seeing as each ADT is used for a unique purpose in A*, there is room to optimize the algorithm with the selection of the ADTs. We based our selection of these ADTs on how we planned to use them, selecting one with the best runtime complexity for the task at hand.

CLOSED - Composed of all nodes already visited. Any node in the closed list was at one point in the open list. The closed list only performs two functions in the context of A*: add node and search for node. Of these two, A* searches more frequently than it adds to the closed list. As such, we opted for a HashMap for this list, as a HashMap has constant time complexity for search.

OPEN - The open list is a list of all nodes that are available to move to from the current node. Any node in this list has a priority value assigned to it based on the sum of cost to initial and estimated cost to final, and A* pulls only the node with the lowest value priority. We used a PriorityQueue with an overridden compare function that checks for the lowest priority of all nodes in the list, as this ADT has this as pre-built functionality.

NEIGHBORS - Meant to hold all the neighbors of the current node that are not obstacles, this list will not be queried for individual nodes, but instead will be filled and looped through for each node visited. We opted for a HashSet, as each node can be accessed and stored in constant time.

Beyond selection of the ADTs, the most important decision in any A* implementation is the heuristic equation used to calculate estimated distance from current node to end node. There are numerous ways of finding estimated distance to the end node, some of which perform better in certain situations than others. We considered three main heuristic equations:

1. Manhattan Distance: Best for use when 4 degrees of movement are allowed, in other words, no diagonal movement is allowed.
2. Diagonal Distance: Best for use when 8 degrees of movement are allowed.
3. Euclidean Distance: Best for when any degree of movement is allowed.

Since our graph is a 2D array that allows for 8 degrees of movement, we opted to use the diagonal distance equation. We used the equation found here for this purpose. While it did not arise in our implementation, tweaking the heuristic equation can result in a faster A* performance that does not guarantee the optimal path every run. Likewise, a nearly exact heuristic can be used that always finds the best route at considerable runtime cost. Our implementation of A* uses a diagonal distance equation, placing our runtime complexity squarely in between the two aforementioned extremes.

These are the primary decisions that impact the performance of the algorithm, however there were other decisions unique to the project. This includes our decision to use a 2D matrix as opposed to the more optimal graph, which we made based on our then unfamiliarity with graphs. A graph would result in a lower storage complexity, and as such we intend to pursue that path in future development. There is additionally the GUI, and the way in which we showed the user the path that A* calculated. There is much room to advance here as well, in terms of what we display and what we can allow the user to interact with.

Runtime Complexity Analysis

Let's assume the **worst** - our startNode is [0,0] and our destNode is [N,N]

1. Calculating each node's neighbors has constant runtime - each node can only have up to 8 neighbors, and accessing them is instant.
2. Ensuring our new node was not already considered as O(N) time, since we need to search our HashMap
3. Analyzing costs is constant time with the Diagonal Distance calcs
4. Adding and popping nodes to/from our PriorityQueue totals to O(log(N)) time, since inserting is the more intensive portion
5. We need to perform (1-4) for some N / constant, so O(N) times.

In total: (O(1) + O(N) + O(1) + O(log(N)) * O (N / C) =

Worst Runtime Complexity: O (N^2)

Let’s now assume the **best** - our startNode is [0,0] and our destNode is [N,N]

1. Calculating each node’s neighbors has constant runtime - each node can only have up to 8 neighbors, and accessing them is instant.
2. Ensuring our new node was not already considered is constant runtime, since we need to search our HashMap. In an ideal situation such as we are considering, each value added to the Hashmap has a unique hash value without any need for rebucketing. As such, the best-case scenario for searching our Hashmap is constant time.
3. Analyzing costs is constant time with the Diagonal Distance calcs.
4. Adding and popping nodes to/from our PriorityQueue totals to O(log(N)) time, since inserting is the more intensive portion
5. We need to perform (1-4) for some N / constant, so O(N) times.

In total: (O(1) + O(1) + O(1) + O(log(N)) * O (N / C) =

Best Runtime Complexity: O (Nlog(N))

Let’s assume the **average** - our startNode is [0,0] and our destNode is [N,N]

1. Calculating each node’s neighbors has constant runtime - each node can only have up to 8 neighbors, and accessing them is instant.
2. Ensuring our new node was not already considered is constant runtime, since we need to search our HashMap. In an average situation we’re considering, almost every value added to the Hashmap has a unique hash value and there will be very little need for rebucketing. Since our nodes are each unique, our hashing method should provide unique hash values for each one. As such, the average case for searching the map is very near the best case, constant time.
3. Analyzing costs is constant time with the Diagonal Distance calcs.
4. Adding and popping nodes too/from our PriorityQueue totals to O(log(N)) time on average, because of heap insertion.
5. We need to perform (1-4) for some N / constant, so O(N) times.

In total: (O(1) + O(1) + O(1) + log(N)) * O(N) =

Average Runtime Complexity: O (Nlog(N))

Space Complexity Analysis

Our implementation of A* utilizes three unique ADTs (a priority queue, a HashMap, and a HashSet), as well as a 2D array of user determined size.

1. Priority Queue = O(1), since we only allow up to eight nodes in it at a time
2. HashMap = O(N)
3. HashSet = O(1), since we only allow up to eight nodes in it at a time
4. 2D array = O(N^2)

In total: (O(1) + O(N) + O(1) + O(N^2)) =

Space Complexity : O(N^2)

GUI Use

The GUI has a particular color scheme that is important to understand before using it.

BLACK - OBSTACLE

-Obstacles cannot be traversed and block the path. You may create an obstacle by clicking on that particular button in the graph. You may also remove an obstacle if you click on an already black node. In removing an obstacle, the node will once again become available for the path.

-When placing or removing obstacles, certain locations will prompt A* to run again to create a new path. For example, if we block off our current path with an obstacle, A* will take that into consideration to create a new path. Conversely, if we open up/remove an obstacle, A* will also re-run to consider that node. However, simply adding an obstacle completely off the path will not prompt any reconsiderations.

WHITE - FREE SPACE

-Free spaces can be chosen and/or considered for the path. Free spaces can be traversed in eight directions, which include the standard left/right/up/down with diagonal additions.

-Free spaces can be turned into obstacles manually through GUI clicks or turned into considered/path/starting/destination based on the program automatically. \

GRAY - CONSIDERED NODE

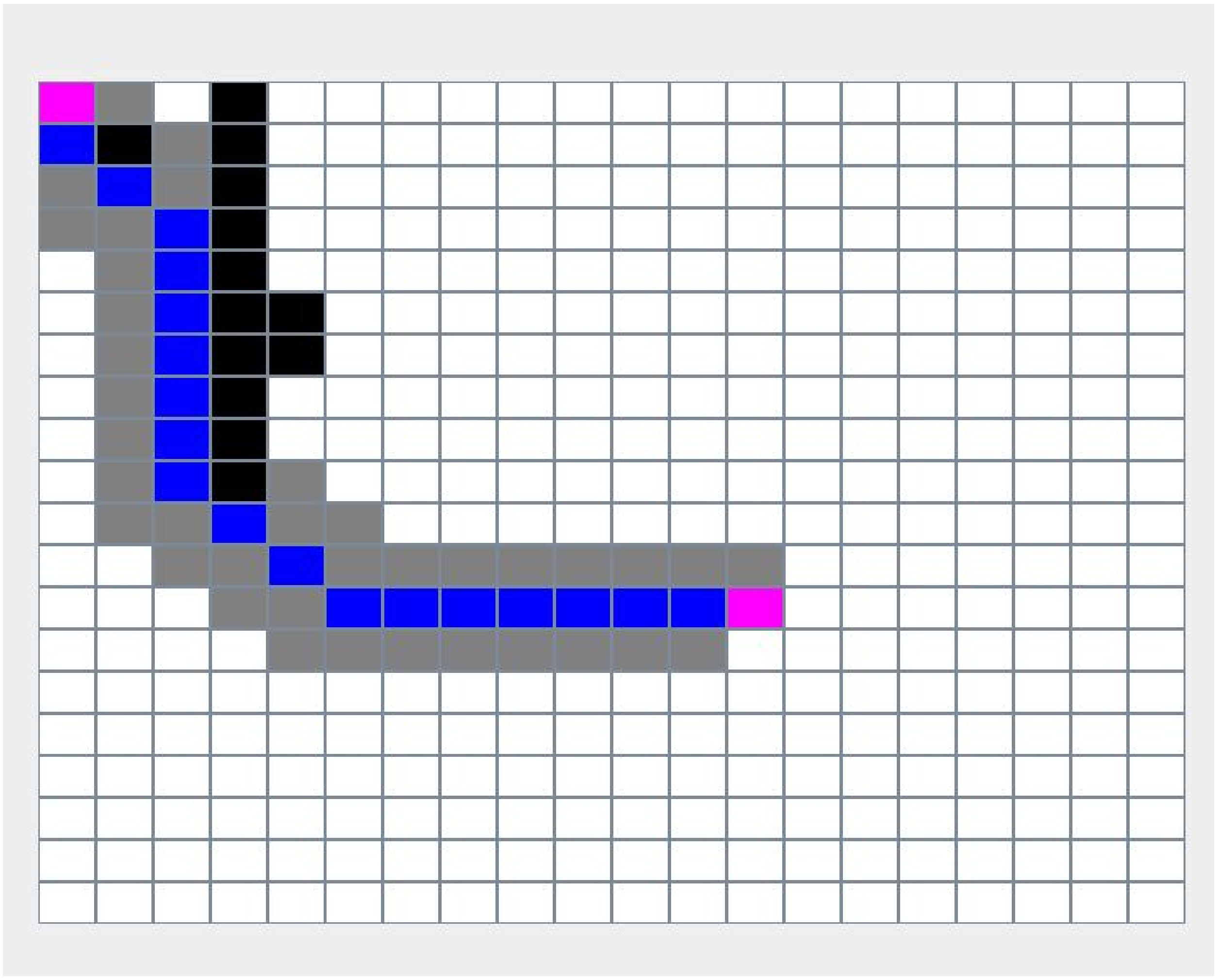
-Considered nodes are nodes that had costs calculated for them and were put into our PriorityQueue for comparison. Once a node is considered and *not* used, it may not be considered again without recalculating A*.

BLUE - FOUND PATH

-All nodes in the optimized route from the given start node to the given end node.

PURPLE - START/END NODE

-As mentioned above, the start and end nodes are modular in that someone could change their locations directly in the code in AStarDriver.



Roadmap

While this implementation is complete, there are a few improvements and functionalities we want to complete in the future:

1. Displaying the costs stored for each node in their GUI button representation.
2. Making the graph such that the user can entirely block A* from reaching the end goal.
3. Addition of DFS in determination of heuristic value, ideally such that the algorithm is able to see future obstacles that would make a current route decision unfeasible.
4. Transitioning graph from 2D node array to an adjacency list graph
5. Adding directionality and weight to the graph and A*'s traversal thereof.
6. Numerous routes displayed, each of which is found by different heuristic methods, enabling the user to see the unique benefit each affords in real-time.

Contributors

Nate Boldt
Jake Kaufman

Sources

- https://www.redblobgames.com/pathfinding/a-star/introduction.html
- https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html
- http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#S7
- https://www.educative.io/edpresso/what-is-the-a-star-algorithm
- https://stackoverflow.com/questions/43816484/finding-the-neighbors-of-2d-array
- https://stackoverflow.com/questions/9511118/is-a-the-best-pathfinding-algorithm
- https://en.wikipedia.org/wiki/Lifelong_Planning_A*
- https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/
- https://cs.stanford.edu/people/eroberts/courses/soco/projects/2003-04/intelligent-search/astar.html