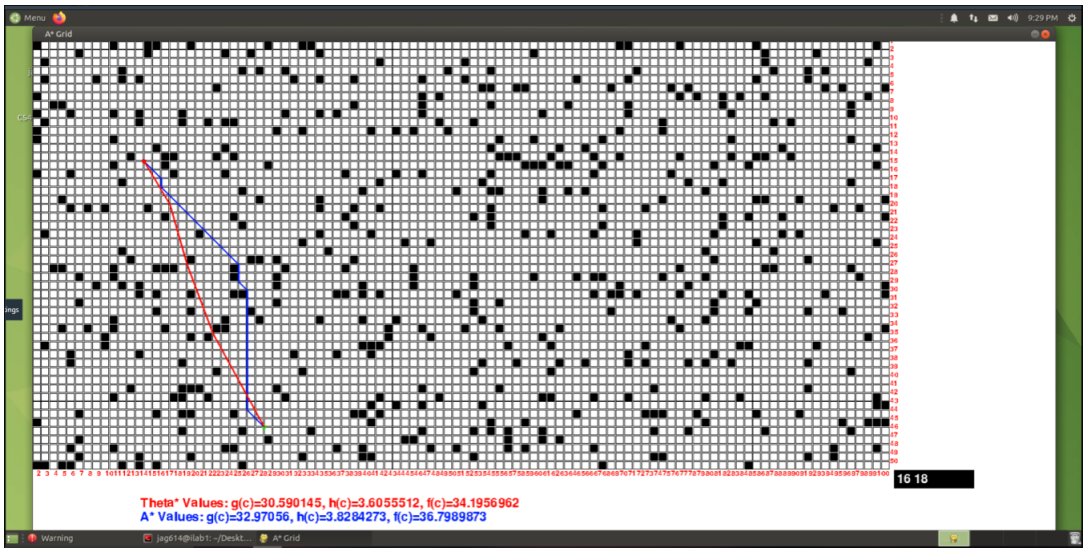
A* vs. Theta* Analysis

Jason Gizo

August 30, 2022

1 Visualization

The visualization for this project was written in Python, and utilizes Pygame as the primary package, as well as PyQt5 for needed screen measurements.
The Python script takes in a text file containing the grid information as given, as well as the designated vertices for each path generated using $A*$ and $Theta*$. The script will then put together a grid using Pygame, and store the path created from $A*$ and $Theta*$ search. Once the GUI is generated, the user has the option to press the "A" key and "T" key to display the $A*$ and $Theta*$ paths, which are colored blue and red respectively. The user can also press the "DELETE" key to restore the grid to its original state without any paths. This way, if desired one could view the $A*$ path, the $Theta*$ path, or both.
There is a black text entry box located below the grid. This box is meant for the user to input the desired vertex, and the corresponding h, g, and f values are displayed for both $A*$ and $Theta$. The user will input the coordinate of the vertex, i.e.: "1 2" for $(1, 2)$, and press the "ENTER" key. Note, not every vertex has these values attached to it, as not every vertex is visited during the search. The user is notified if it is attempted to access such vertex.
The script does not need both paths generated in order to function properly. Suppose only $A*$ was ran, and its respective path was output, the GUI would only display that path and throw no errors with the missing $Theta*$ path.
The main code to perform the search algorithms was written in Java, and the GUI in Python. Below is a screenshot of the GUI.



2 A* and Theta* Implementation

The implementation of $A*$ and $Theta*$ were both in Java.

For $A*$, the code operates as follows: Depending on the command line argument size, either a file is read from a designated location, or a random grid is generated. Whichever of the two, this grid is stored in a 2D-array of type int. In said 2d array, 0 indicates a free cell, and a 1 indicates a blocked cell. Then, I implemented my own Point class to store the x and y coordinates, g and h-values of the point, and the parent of the point.
Note: since the example grid starts at vertex (1,1), this value is manually decreased to match the indices of the 2d array. However, this is carefully observed throughout calculations and performed operations.
Upon creation, the point's constructor defaults the h and g values to $+\infty$ and $-\infty$, and the parent is set to NULL. For the open list/fringe, I use a heap priority queue. For the closed list/explored list, I implemented a simple array list of type Point.

For $Theta*$, the code operates very similarly. I used the same data structures described above for Theta*. The same Point class is used for vertices, and the int-2D array for the grid. The priority heap queue that was developed for the open list/fringe is also used here, but here an array list is used for the closed list/explored list. I paid additional attention to checking if a path is blocked in the line of sight function. This can be tricky with the method of storing vertices at its $i-1th$ index, so it was assured no errors arose.

3 Binary Heap Implementation

I decided to implement my own binary heap for this project. This heap features many of the standard functions required for a binary heap, such as *enqueue* and *dequeue*, and can be created of type *Point*, the class created to store grid information. The heap also contains a "remove()" method, which removes an element from the heap. This differs from dequeue, since dequeue pops off the top of the queue. My written heap prioritizes the greatest element of the queue, whereas Java's built in PriorityQueue prioritizes the least element. My heap has a ".contains()" method, which had plenty of use throughout the $A*$ and $Theta*$ algorithms. This contains() method operates in $O(n)$ time worst case, which is on par with Java's built in priority queue. Note: This runtime can be cut down had all elements been popped into a sorted array. Then any future search can be done in $O(logn)$ time. This binary heap also has post-order and pre-order traversal.
Note: The written binary heap was only used for the open list, not the closed list in either search algorithm.

4 Rundown of Code

There is a README file included in the .zip package. Whilst briefly discussed earlier, this section is to include a more involved explanation of how the code works together. Firstly, when unzipped, there is a "pathfinding" folder, a "solutions" folder, and GridVisualizer.py. The "pathfinding" folder contains all of the Java files, all of which take care of the $A*$ and $Theta*$ search. These files must all be compiled using
"javac *.java"
After, the main file, *PathFinding* is ready to be ran. This file takes command line input of two arguments, the output file location, and the input file location. The input file location is optional. Had this been left blank, the program will randomly generate a 100x50 grid to be operated on. The output file location should be set to the "solutions" directory upon running. This output file is a text file containing all of the information needed to be sent to the Python script in order to display the GUI. An example of running this file may look like,
"java PathFinding ../solutions/ input_file_1.txt".
Now, a file named "\solution.txt" has been output to the "solutions" directory. This file can now be fed to the Python script "GridVisualizer.py". This file takes in one argument, which is the text file output from the Java program. Ideally, this is in the "solutions" directory, but you may have set a different directory.
You can run the Python script by doing:
"python3 GridVisualizer.py solution_output_location"
An example may look like
"python3 GridVisualizer.py solutions/\solution.txt"
The Python script will then run, and may take a few moments upon first run.

This script uses Pygame, and while not initially on iLab, a command is set to install it inside of the script. The script will then produce a GUI using all the information output from the Java files. Press "A" to display the $A*$ path, "T" to display the $Theta*$ path, and "DELETE" to delete the paths from the grid. A black box is in the lower right corner, and after inputting your desired coordinate in the form "x y", the corresponding information of costs is displayed.

5 Experimentation on A* and Theta*

This experimentation was done on 50 randomly generated grids, pre-optimization of the algorithms, ran on iLab to standardize all results. The spreadsheet containing the gathered experimental data can be found at the end of this PDF.

The experimental data showed that the average running time of A* was 19 ms, and the average running time of Theta* was 3.92 ms. The average path length of A* was 41.187 units, and the average path length for Theta* was 39.156 units. It is important to note that running time did increase with path length as to be expected.

Additionally, A* took noticeably more time for longer paths than Theta*. Through these results, it is clear that in my implementation, Theta* runs in a much more optimal time, and results in a shorter path length. Shorter Theta* path length was also to be expected, as Theta* is able to minimize the distance the distance between two vertices with its calculation. A* weaves diagonal lines of length $\sqrt{2}$ in between its horizontal and vertical, whereas Theta* attempts the longest unblocked diagonal between the two points.

The run time difference can be attributed to the difference in heuristics. Theta* relies on the longest straight line distance before blockage, where as A* cannot go by the longest straight line distance, but rather a more computationally expensive heuristic. In the experimental data, it shows that Theta* results in smaller h-values, which is to be expected since the Theta* heuristic is standard euclidean distance.

The heuristic values for each search were chosen for a randomly chosen shared explored point between A* and Theta*. For certain unique paths, such as paths that were the exact same for both searches, or mirrored paths, the running time for A* was nearly $3x$ that of Theta*. It should be noted that both heuristics are consistent for both algorithms. The findings showed that $\mathrm{h}_{A*}(n) \geq \mathrm{h}_{Theta*}(n)$. It is to Theta*'s advantage that its heuristic is shorter. If these two algorithms shared the same heuristic, since A* looks at close neighbors rather than attempting the longest straight line between two points, A* would end up calculating closer to the true shortest path. The h-values for A* would be smaller than that of Theta* since the diagonal cost would be closer to the grid edge travel cost,

compared to Theta* far away vertex straight line distance calculation.

6 Spreadsheet

For the spreadsheet containing the experimentation results:
`docs.google.com/spreadsheets/d/1P7nw32AOVfRakw4xAlMuoMvD2QN1kSch2UvtM5QQME0/edit?usp=sharing`