Abdulrahman Abdulrahman, Brian Moran, Fares Easa, Khalid Masuod

CS440

10/20/20

Assignment 1: Heuristic Search

Objective: For this project we were tasked with creating a 2D discretized terrain grid that will support an A\* path finding Algorithm. The A\* algorithm will be run with three different heuristics in mind: uniform cost search, normal A\* search algorithm, and weighted A\* search.

Submission:

Assign1.zip containing:

* Algorithm.java
* AlternateAlgorithm.java
* CS440 Readme.docx is the current Read Me file
* DrawGrid.java
* Exev.java is the execution file that creates the grid object and runs the algorithm
* GridGenerator.java has the Grid object file, has constructors to create grid, and can write the file to grid
* Node.java is the node object file that creates and holds the nodes to be placed in the grid.
* Pair.java

Program Design:

Grid Generator Description:

Grid class has a 2D array of Nodes, from NodeClass

* + An array of 2 integers that represents the start Nodes
  + An array of 2 integers representing the end nodes;
  + Cells that are hard to traverse;

5 Kinds of constructors:

* + One Constructor that generates the complete grid with start and end points
  + The file name is the file to save the grid, rand is to separate the constructor from others
  + One Constructor that takes in arrays with a start and end location;
  + Reads in a files grid and creates a new start and stop location
  + One Constructor that takes in a string that is a file name to import the data;

Functions for Manipulation:

* + Print to stream the values within the grid - Warning! Huge output for the stream.

traverse()

* + Prints to the stream the grids designated start and end point;

stats()

* + Generates start and stop and ensures the file

generate()

* + Generates start for the path process, in array format

genStart()

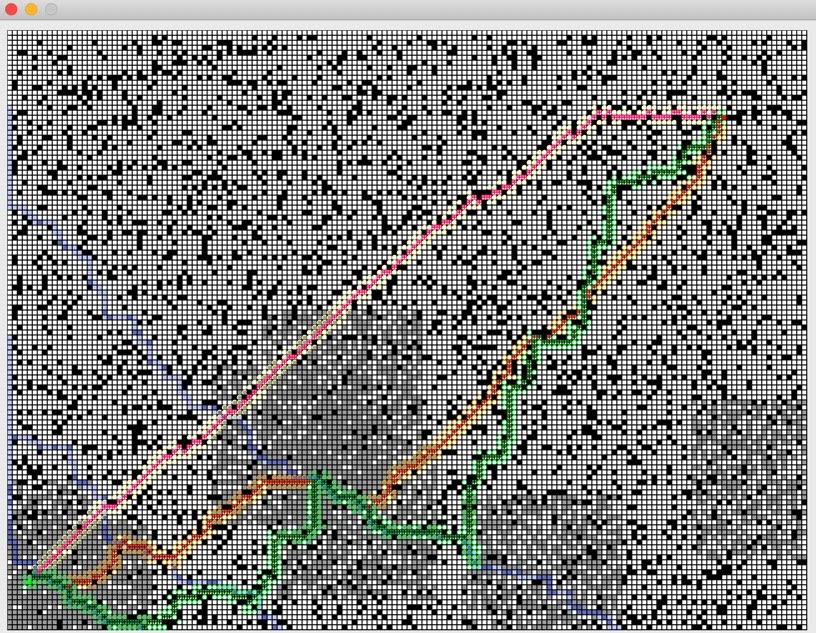
* + Generates Goals for the path process, in array format

genEnd()

* + Generates the Harder to traverse cells
    - hardCells()
  + Generates the blocked cells that you cannot traverse
    - blockedCells()
  + Generates the Normal cells within the grid
    - normalCells(){
  + Generates 4 highways within the map
    - highways()
  + Creates a new file with the first two lines being the start and end points and then 160 rows of data;
    - writeToFile()

Deliverables:

* Due to time constraints, we did not create a gui of any kind to visualize different maps of varying start and ends as well as different obstacles. However, we have all of the components required to do so. The grid can be generated from a file, as well as saved to a file, the start and end points can be chosen or randomized.
* All three algorithms were able to be completed.



Pink - Uniform Search

Red - A\*

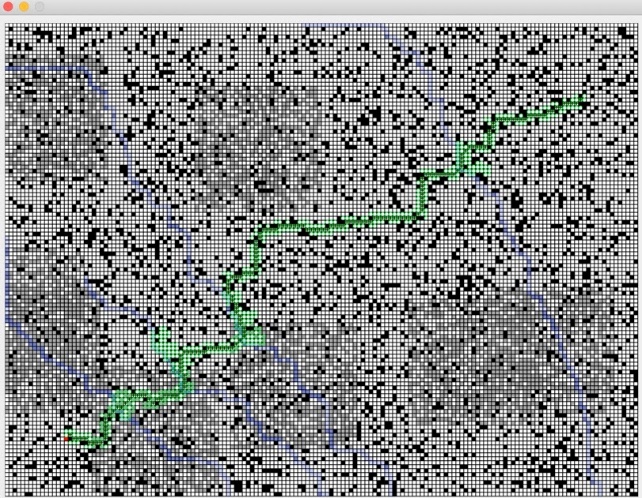
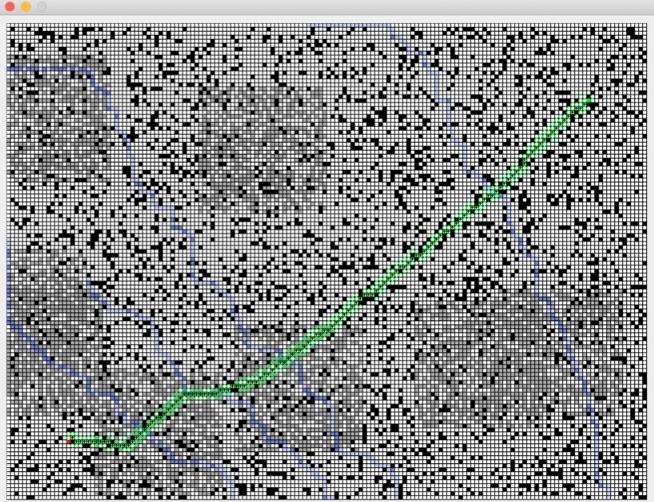
Green - Weighted A\* (w = .5)

Uniform search strictly uses the euclidean distance to get to the end meaning it will take the most direct path since each block moved is just one.

A\* uses the euclidean distance as well as keeps track of the total cost so will attempt to go routes that don’t cost as much as others around it.

Weighted A\* works exactly as the A\* does but multiplies the heuristic value by w, meaning when w < 1, the algorithm will care more about minimizing cost and check more paths on the way to the end so to find a more cost effective route.

When w > 1 the algorithm will blend more with uniform search and just get to the end faster instead of checking more paths. In the end, on average the algorithm works as, w > 1; faster run-time; longer cost-path, w < 1; longer run-time; shorter cost-path.



[Weighted A\* (w = 3) :COST = 61.4924]          [Weighted A\* (w = .3): COST = 49.6309]

* To optimize the points for the algorithm, we used separate programs for finding the different paths for the highways. By separating them, we were able to have loops for rejecting and accepting highways within different loops. In general, doing the original search for a fully implementable program, would be expensive, however we were able to cut it down by at most 4 times.
* Depending on how you change the heuristic value by editing the weight of the Weighted A\* algorithm, it changes drastically the path chosen as well as run-time of the algorithm.
* The best heuristic for COST EFFECTIVENESS that we had found through observation was when (w = .75), this made the algorithm mostly search for coordinates that were low cost as opposed to closer to the end, ultimately leading in a very low-cost path with not that much extra runtime.
* The best heuristic for RUN-TIME EFFECTIVENESS that we found was a value of 2, any more than that, the algorithm would act like Uniform Search and less it would be just as slow as A\*
* Our program is efficient in a sense that we were able to break the program into segments that can be run on a need basis. Our program can generate the grids based on inputted start and end locations. By using Java, the actual compilation and run of the executable file in terms of speed, Java is faster than Python as it is a compiled language. It takes less time to execute a code. The grid is going to already be generated into a 2D array, making it easier to run the algorithm on it. In terms of the search algorithm, the different implementations are run with their separate speed values.