ROB 456
Homework # 4
Aaron Rito & RJ Lockard
11/13/2017

1. **Web sites you used:**

   See bibliography at end of document.

2. **People you worked with:**

   Aaron Rito and RJ Lockard.

3. **The final path**

   See “Results” and “Analysis” sections

4. **Your heuristic function (in English)**

   In our code we used the Manhattan and Euclidean heuristic functions, which are the estimate of the shortest path to goal from current node. In the case of the 4 connected graph, Manhattan is used and written as the absolute value of $\left|((Y_1 - Y_0) + (X_1 - X_0))\right|$ (to compute the shortest cardinal path to the goal for each node.) In the case of a 8 connected graph, the Euclidean heuristic is most accurate and written as $\sqrt{((y_1 - y_0)^2 + (x_1 - x_0)^2)}$. Our algorithm can also be run with no heuristic, which directly converts it to Dijkstra’s algorithm.

   Using the A* star algorithm with its heuristic consideration, we were able to achieve better performance when finding the shortest path compared to when we strictly used Dijkstra’s algorithm. Since the A* algorithm is an extension of Dijkstra’s algorithm, it is using the idea of best-first search and looks for the path with the lowest expected total cost of the source to goal distance. The path cost value that is computed is a sum of two functions, the past path cost for the node and the future path cost to the goal.

5. **How you implemented the graph and priority queue**

   This program takes a n x n square graph (hard-coded in a numpy array) and finds the shortest path form a start node to and goal node by recursively searching through a list of open nodes. To make the open list, a starting node is added to the open node list. Then, the algorithm searches the neighboring nodes recursively. Each open neighboring node is then added to the open list. The starting node is then added to a visited list. For each node, we keep track of the node's parent. The parent is set as the neighboring node that has the lowest g value where (f = g + h, where g is the cost of the path through the preceding parents (back to start), and h is the heuristic distance to the goal.) This continues until each node in the list has been visited. Using the python

heapq library, the open nodes are popped into a dictionary type array with their f values as the key. The heap automatically sorts the key values in a binary search tree, making the search through the parents faster. This automatically implements the **priority queue using a binary heap for open list, and a simple set for closed list** (as we don't need the f data anymore, just the parent). Once all the nodes have been visited, and thereby popped on to the visited set, then the set is iterated through and saved (backwards) in a path array. We reverse the path, and print the results. According to wiki "The time complexity is polynomial when the search space is a tree, there is a single goal state, and the heuristic function *h* meets the following condition:

$$|h(x) - h'^{*}(x)| = O(\log(h'^{*}(x)))$$

where h^* is the optimal heuristic, the exact cost to get from x to the goal. In other words, the error of h will not grow faster than the logarithm of the "perfect heuristic" h^* that returns the true distance from x to the goal" (citation link)

6. **Any known bugs/issues**

When we first started this assignment, we we're exactly sure exactly what to do so we began by just implementing a few lines of code and hard coding in our world matrix. After we had our file set up, we actually spent several hours researching and watching YouTube lectures for additional information relating to the A* algorithm. After we had a good grasp of the concept and drafted some pseudocode, we attempted to brute force our entire process, creating multiple matrices to keep track of values, a dictionary containing lists for each node, an endless amount of nested for loops cycling through arrays, it just got to complicated. We then researched recursive implementations of this algorithm and found a few sources online, and were able to implement a recursive solution that works with nxn size square maps, and explores all the heuristic options, comparing them to pure Dijkstra's algorithm. We had trouble trying to import the csv directly into the numpy array, and ran out of time, so we ended up just hardcoding the array.

RESULTS:

These are the results of the optimal path analysis from the map array supplied in world.csv.

NOTE: Optimal path may not be the actual shortest path.

Using cardinal coordinates and Manhattan heuristic,
the optimal path was 44 steps. The path taken was:

[(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (1, 8), (2, 8), (3, 8), (4, 8), (5, 8), (6, 8), (7, 8), (8, 8), (9, 8), (9, 7), (10, 7), (10, 6), (10, 5), (11, 5), (12, 5), (13, 5), (13, 6), (13, 7), (13, 8), (13, 9), (13, 10), (13, 11), (13, 12), (13, 13), (13, 14), (13, 15), (13, 16), (13, 17), (13, 18), (13, 19), (14, 19), (15, 19), (16, 19), (17, 19), (18, 19), (19, 19)]

Using cardinal coordinates and pure Dijkstra's, no heuristic,
the optimal path was 70 steps. The path taken was:

[(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (1, 8), (2, 8), (3, 8), (4, 8), (5, 8), (6,

8), (7, 8), (7, 7), (7, 6), (7, 5), (8, 5), (9, 5), (10, 5), (11, 5), (12, 5), (13, 5), (13, 6), (13, 7), (13, 8), (13, 9), (13, 10), (13, 11), (12, 11), (12, 12), (12, 13), (12, 14), (12, 15), (11, 15), (10, 15), (9, 15), (8, 15), (7, 15), (6, 15), (5, 15), (4, 15), (3, 15), (2, 15), (1, 15), (0, 15), (0, 16), (0, 17), (0, 18), (0, 19), (1, 19), (2, 19), (3, 19), (4, 19), (5, 19), (6, 19), (7, 19), (8, 19), (9, 19), (10, 19), (11, 19), (12, 19), (13, 19), (14, 19), (15, 19), (16, 19), (17, 19), (18, 19), (19, 19)]

Using diagonals and Euclidean heuristic,
the optimal path was 34 steps. The path taken was:

[(0, 1), (1, 2), (2, 3), (2, 4), (2, 5), (2, 6), (2, 7), (3, 8), (4, 8), (5, 8), (6, 8), (7, 8), (8, 8), (9, 7), (10, 6), (11, 5), (12, 5), (13, 6), (13, 7), (13, 8), (13, 9), (13, 10), (13, 11), (13, 12), (13, 13), (13, 14), (13, 15), (13, 16), (14, 17), (15, 17), (16, 18), (17, 18), (18, 18), (19, 19)]

Using diagonals and pure Dijkstra's, no heuristic,
the optimal path was 35 steps. The path taken was:

[(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 6), (2, 7), (3, 8), (4, 8), (5, 8), (6, 8), (7, 7), (7, 6), (8, 5), (9, 5), (10, 5), (11, 5), (12, 5), (13, 6), (13, 7), (13, 8), (13, 9), (13, 10), (12, 11), (12, 12), (12, 13), (12, 14), (12, 15), (13, 16), (14, 17), (15, 17), (16, 16), (17, 17), (18, 18), (19, 19)]

ANALYSIS: The shortest path of 34 steps was found by using the Euclidean heuristic and the diagonally connected graph. The Manhattan heuristic had a significant performance over Dijkstra's when using a 4 connected graph. In contrast, Dijkstra's was almost as good as A* in the diagonally connected graph, missing the shortest found path by 1 step.

BIBLIOGRAPHY:

[1]
"5. Data Structures — Python 3.6.3 documentation." [Online]. Available: https://docs.python.org/3/tutorial/datastructures.html. [Accessed: 14-Nov-2017].

[2]
"8.4. heapq — Heap queue algorithm — Python 2.7.14 documentation." [Online]. Available: https://docs.python.org/2/library/heapq.html. [Accessed: 14-Nov-2017].

[3]
"A* Search Algorithm," *GeeksforGeeks*, 16-Jun-2016. .

[4]
"A* search algorithm," *Wikipedia*. 10-Nov-2017.

[5]
"How can I reverse a list in python? - Stack Overflow." [Online]. Available: https://stackoverflow.com/questions/3940128/how-can-i-reverse-a-list-in-python. [Accessed: 14-Nov-2017].

[6]

“numpy.array — NumPy v1.13 Manual.” [Online]. Available: https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.array.html. [Accessed: 14-Nov-2017].

[7] “numpy - Shape of array python - Stack Overflow.” [Online]. Available: https://stackoverflow.com/questions/15668380/shape-of-array-python. [Accessed: 14-Nov-2017].

[8] “numpy.sqrt — NumPy v1.13 Manual.” [Online]. Available: https://docs.scipy.org/doc/numpy-1.13.0/reference/generated/numpy.sqrt.html. [Accessed: 14-Nov-2017].

[9] “Python A* Pathfinding (With Binary Heap) « Python recipes « ActiveState Code.” [Online]. Available: http://code.activestate.com/recipes/578919-python-a-pathfinding-with-binary-heap/. [Accessed: 14-Nov-2017].