Introduction to Intelligent Systems (CSCI-331-02)
Professor David Garcia-Muñoz
Nick Greenquist and Alex Hedges
Due 2017-03-28

Project 1 Write Up

A. **Give a high-level summary of how you implemented the algorithms required. What differences did you note in the performance of each one of them?**
   a. **BFS:** This algorithm was implemented in the manner recommended in CLRS. Each vertex was initialized to have a distance of 0 and to be unvisited. Starting from the given start vertex, it determined all neighbors of the current vertex. It then set the distances, predecessors, and visited status of all them. Each neighbor was then added to the queue. After going through all neighbors, the front vertex was dequeued from the queue, and the above operations are repeated on it until the goal vertex is found. At this point, the algorithm terminates.
   b. **Backtracking:** This algorithm was implemented in the manner recommended in CLRS for DFS. Each vertex was initialized to have a distance of 0 and to be unvisited. Starting from the given start vertex, it marked the vertex as visited and ran DFS on each neighbor to that vertex. If the goal state is found, the algorithm terminates.
   c. **A*:** To implement A*, we used a variety of data structures to handle the data. A priority queue was used to store available nodes. This was sorted by their final costs. Next, a while loop runs until the goal node has been expanded. Inside this loop, we dequeue the lowest final cost node from the priority queue and expand it. Expanding a node means finding all neighbor nodes from this node. The program assigns all the necessary values to the node such as cost, final cost, parent, and base cost. If the node has not been seen before (and therefore is not in the priority queue), we enqueue it. If the node is in the queue, we update the final cost if the final cost from our current path is less than what we have seen before. Finally, once the goal node has been expanded, we loop backwards from its parents and add up the base costs to get the final cost from the start to the end, and we print out the names of the nodes that make up the path.
   d. **Differences:** We did much logging in the output of our methods to ensure they were working. Additionally, this allowed us to observe the performance of the algorithms. Because the graph is so small, run time was not useful for us. Instead, we counted how many nodes were expanded. Expanding too many nodes that were not eventually used in the final path is considered bad performance. A* had the best performance in both finding the correct path and minimizing how many nodes are checked. BFS and backtracking are the slowest because the worst case is they expand every node and the goal node is the last to be expanded. A* is usually going to be faster because it is guided by a greedy algorithm in addition to heuristics. It will rarely expand nodes needlessly. In terms of finding the correct shortest path, again A* is better. It will always find a shortest

path if the heuristic is admissible. BFS is better than backtracking in finding the shortest path but it has one discrepancy when run on `RitMap.txt`. If you travel from BOO, BFS will select the path going towards WAL. The correct path is to go up to VIG. This happens because BFS will find the path with the least amount of nodes to go through, instead of the path with the least cost. Backtracking rarely finds the shortest path and it not a good option for finding a shortest path.

B. If you gave to your code a map which was too large to fit into main memory, which would likely run faster: BFS, Backtracking, or A*?
Backtracking would run fastest, as it requires that one only keep track of a stack of the current path and a record of the nodes already visited. Meanwhile, BFS and A* both require a queue of vertices left to check and a record of the nodes already visited. A* additionally requires its queue to be a priority queue, a record of the current path cost and the estimated total path cost, and the heuristic values for each vertex.

C. In what ways are BFS and A* similar?
BFS is a special case of A* where the heuristic is always 0 and the cost for each edge is 1. They therefore both have some similarities in code structure, such as the use of a queue and the use of the informal strategy of “nearest-first search.” However, as BFS is a special case, it can be written much more succinctly without the need to keep track of the additional features of A*.

D. Which search algorithm (if any) are guaranteed to find the shortest path to the goal? What does it depend on?

a. **BFS:** No, as pure BFS is not designed to find shortest paths
b. **Backtracking:** No
c. **A*:** Yes, if the heuristic is admissible. Even if the heuristic values are not admissible, A* will usually find the shortest path.

E. According to the results obtained by the different algorithms implemented, how do they perform when compared to the way humans would do in the real world (not necessarily moving from building to building)?

a. **BFS:** Humans behave like BFS in that they often check all available paths from where they are currently. After checking all available paths, humans will then move on to an available new building from where they are. Once there, they will again check all their options. Now, like in BFS, a human might go back to where they were previously and check an available building from their previous building before moving further with any new paths. This mimics BFS in that the human will always make sure to expand all available paths in their close vicinity before moving too far along in one direction (limiting the depth of their search until all available buildings have been expanded).
b. Backtracking: Humans behave like backtracking in that they turn back around and make another attempt at a path after reaching a dead end. They will often try to continue from a point if they have not yet reached the goal, but if they go too

far without reaching a goal, they usually act like depth-limited search and turn themselves around.

c. **A*:** Humans behave like A* in that they usually have some heuristic or “intuition” that guides them alone a more efficient path than a random search. Humans normally never make decisions “blind” like how backtracking and BFS behave. A* is a combination of best-first search together with a ‘human-like’ intuition. Humans will most likely expand (travel to) the nearest building from where they are instead of blindly going to the next available one. This mimics the best-first nature of A*. In addition to selecting shorter building paths from where they are, humans also are guided by instinct. This instinct represents the heuristic value for the building they are looking to go to. One example might be cardinal directions. If you know Golisano is on the west side of campus, a heuristic value for deciding which new building to go to would be if it is to the west of you. A building to the east in a human's mind will likely not be the best way to go so they would assign a worse heuristic value. In addition, A* keeps available neighbors in memory and expands the lowest cost one. A human also remembers options from the past. So, like A*, they might travel far in one direction and decide that paths they remember from past nodes they were at are in fact better. This mimics how A* only expands the best value nodes.

F. **How would you define the same problem in order to represent the way humans would move through the map?**
Humans do not view the buildings on a campus as a collection of vertices in a graph connected by perfectly straight edges. Instead, humans use walkways that are adjacent to buildings. Instead of walking in straight lines from building to building, humans usually walk along a walkway without entering buildings until reaching the final destination. In addition, multiple paths along walkways often connect two points on campus, and people have preferences regarding scenery, weather, traffic, and other features of these walkways, all of which can change over time. Therefore, a human likely would not come up with the same path as a standard graph search algorithm would.

G. **Heuristic Comparison:**

a. **Second Heuristic for A*:** 20 times the minimum number of edges between the given vertex and GOL

b. **Comparison of Heuristic 1 (H1) and Heuristic 2 (H2):** H2 returns the same path for every node with GOL as the end node that H1 does. However, there a few differences between H1 and H2. H2 is not as optimal as H1. The best way to see this is by setting BOO as the start node. When comparing the two heuristics, the program output every node that was expanded as part of the algorithm. Because the heuristic overestimates the cost for the node above BOO, which is UNI, the algorithm expands too far in the wrong direction until finally coming back to UNI. The heuristic is also not admissible. The true cost to get to GOL from UNI is 137, but H2 estimates it as 140. Although this estimation is not admissible, the correct shortest path is still found with H2. With a more complex graph however, the simple H2 might not lead to optimal paths in all cases.