CS333 Homework 3

Jeffrey Luo (jl834) Nathan Kim (njk24)

1.
   a. Using Dijkstra's algorithm, the order starting from node s would be:
      s->c->a->e->d->f->b->t
      While using the priority queue, we keep track of the minimal path it takes to get to a node from the starting node. We put s->c, s->a, and s->d into our priority queue first and then select the path with the shortest distance. In this case, it would be c, so first s is removed, then c. From here, we add all paths starting from c into the priority queue, which would include c->b and c->e (these distances are added onto the previous path's distances, so c->b would be 3.5). Then from the current paths in the priority queue, we choose the shortest distance, which would be from s->a, meaning a is removed from the queue. We continue adding and removing nodes in this way, removing the node with the shortest distance while adding nodes that are connected to the node removed last.
   b. Using the A* search algorithm, the order starting from node s would be:
      s->a->c->e->f->t
      The total path to reach the goal is the cost to reach a certain node + the heuristic value for the individual node. From our starting node s, we check all neighboring paths, where s->a is 4, s->c is 4.5, and s->d is 6. We choose the shortest path, which is the path to a, so s and a are the first two removed from our priority queue. Then similar to Dijkstra's algorithm, we add into the queue the neighboring nodes to the node that we just removed: s->a->b, where f(n) = 5. Our shortest f(n) in our queue is s->c, so we continue on this path. We continue in this way, which follows the order listed above, which has f(n) = 4.5, which is the reason why we don't remove s->d (6) or s->a->b (5).

2.
   a. Updating a node's priority value can be updated by removing the node and re-adding it with an updated priority value when using a binary heap. If we use a hashmap of the nodes and their indices in the binary heap array, then we can remove and re-adding the node in O(logn) time. This happens to a vertex only once for each neighbor it has so it would only occur m amount of times. Therefore the total time it takes to update the priority values would be O(mlog(n)). Because each node is removed exactly once and not re-added (besides updating priority values), we only remove n nodes with each removal taking O(log(n)) time. Therefore, the

total runtime for removing all nodes would be O(nlog(n)). This gives us a total running time using a binary heap implementation to be O((n+m) log(n)).

   b. Although the (n+m) term would drastically affect the runtime drastically, log(n) would largely remain unaffected by having n^2 entries. This is because O(log(n^2)) would simply be O(2log(n)) = O(log(n)) as it is a logarithmic function.

3. See code.
4.
   a.
      i. 1.2986736018470506 miles
         First 3 nodes: 5904215824, 5904215825, 5904215826
         Last 3 nodes: 297651717, 297651716, 297651715
      ii. 0.9846262453810296 miles
         First 3 nodes: 6119788404, 541197808, 541197649
         Last 3 nodes: 4727263055, 4727263035, 6766454532
      iii. 3.912336858718287 miles
         First 3 nodes: 2865075396, 2865075435, 554881692
         Last 3 nodes: 6504101801, 344481045, 344481044
   b.
      i. Dijkstra: 13289, A-star: 5222
      ii. Dijkstra: 10982, A-star: 3115
      iii. Dijkstra: 46141, A-star: 25402
         On average, Dijkstra visits 2.63 times more nodes than A-star
   c.
      i. Dijkstra: 0.4426732063293457, A-star: 0.3348360061645508
      ii. Dijkstra: 0.3361778259277344, A-star: 0.3044133186340332
      iii. Dijkstra: 0.645392894744873, A-star: 0.5196950435638428
         On average, Dijkstra takes 1.22 times more seconds than A-star
   d. Runtime and nodes visited don't have a one to one relationship as the implementation of the priority queue and calculations of the best possible path are negligible when using A-star and Dijkstra on the same graph. Because the runtime for Dijkstra is logarithmic, the time it takes to go through all the nodes is ultimately lessened than if the implementation were O(n). Comparing this to the amount of nodes that are visited, Dijkstra still has to go through many more paths as it doesn't use the heuristic function whereas A-star with the use of a good heuristic function, will drastically reduce the amount of nodes that need to be visited. This difference is also emphasized by the fact that A-star search also has to calculate the heuristic, which can (relatively) alter the runtime. This

calculation of heuristics is also what drastically reduces the amount of nodes visited.