Network Routing Report

# 1. Correct Implementation of Dijkstra’s. See appendix for full code

# 2. Correct Implementation of Priority Queues. See appendix for code

1. Binary Heap Operations
   1. **Insert O(1)**

Simple python set.add call. Is O(1) time

See code line 14

* 1. **deleteMin O(V)**

Loops over each element in the set O(V) and saves the location of the minimum element according to the distance values which are passed in. This element is then removed from the set using pop O(1) and returned. Since each element in the set is visited, it is O(V) time.

See code line 18-25

* 1. **decreaseKey O(1)**

Passes as distance values are not stored internally. Thus is O(1) time

See code line 29

1. Unsorted Array Operations
   1. **Insert O(1) / O(logV)**

Inserts the new node index with inf distance into the tree at the first empty spot. Tree is implemented as an array so adding to it is append with O(1) time. Bubbling up is not necessary here as inserts are all called on nodes with infinite distance so they will already be in the correct position. As it is a simple append, this is O(1). It would be worst case O(logV) if bubbling up was necessary.

See code lines 44-50

* 1. **deleteMin O(logV)**

Removes the first element in the tree which is guaranteed to be the min and returns it. This is O(1) time. The last element is then moved into the first position. The pointer for the last element is updated and bubble down is called on the first index to correct the ordering of the tree. Bubble down could potentially move the new top node all the way back down to the bottom which is a worse case of O(logV) as there are logV levels in the tree. This means the overall worst case for deleteMin is O(logV).

See code lines 52-68 and 98-134 (bubble down)

* 1. **decreaseKey O(logV)**

The location of the given node is found in the pointers array O(1) and used to change the value of the node in the tree at that location to the new value passed in. Bubble up is then called on that node (as the value of the Node will only be less than it was before, never more). Bubbling up is a worst case of O(logV) time because it could in the worst case move up every layer in the tree which is logV layers. This is done by comparing the value of the node with the value of it’s parent and swapping if it is less. This makes the overall worst case of decreaseKey O(logV) time.

See code lines 71-77 and 80-95 (bubble up)

# 3. Time and Space of both implementations

**Pseudo Code for Dijkstra’s algorithm:**

Create priority queue O(1)

For each node in network: O(V)

Set distance to node to infinity

Set previous node to null

Insert node into queue

decreaseKey for start node to 0 set O(1) / heap O(logV)

while queue is not empty: O(V)

min = deleteMin from queue set O(V) / heap O(logV)

if distance to min == inf: break

for edge (u,v) from min: O(3) because 3 edges from each vertex

if distance to v is less:

decreaseKey for v set O(1) / heap O(logV)

**Time Complexity:**

From this pseudo code we can see that regardless of the priority queue implementation, the time complexity of Dijkstra’s algorithm is at least O(V) because it loops over each node twice. The number of edges in the graphs we were using was fixed at 3 edges per vertex so this constant factor is ignored. In different graphs, this value might be included as a time complexity of O(V + E).

Using the unsorted array (set) implementation, insert and decrease key are both O(1) and so don’t contribute to the overall time complexity. The deleteMin operation is, however, O(V) time so when using an unsorted array implementation of the priority queue, the overall algorithm is O(V^2) time complexity.

Using the binary heap implementation, insert, decrease, and deleteMin operations all have a worst case of O(logV). While this is slower than the O(1) time for insert and decrease, it is faster for deleteMin. The time complexity after dropping constant factors is dominated by the V calls to delete min giving a overall time complexity of O(VlogV) or O((V+E)logV) if the graph did not have a fixed number of edges per vertex.

**Space Complexity:**

Space complexity for the two algorithms is the same despite implementation of the priority queue. Each node is stored just once along with it’s distance value, previous pointer, and outgoing edges. This complete representation is of size V or V+E when edge number is not fixed. Since the algorithm does not create an additional memory, the space complexity is O(V).

# 4. Example Screenshots

a) Seed: 42, Size 20

A picture containing chart

Description automatically generated

b) Seed: 123, Size 200

Chart, scatter chart

Description automatically generated

c) Seed: 312, Size: 500

Chart

Description automatically generated

# 5. Empirical Analysis

Table

Description automatically generated

Table

Description automatically generated

This shows the number of times faster the Heap is at that level than the unsorted array implementationTable

Description automatically generated

Here is a graph of the results for both implementations with the heap in orange.Chart, line chart

Description automatically generated

I was unable to run the algorithm for 1000000 points because my laptop could not handle even generating that number of points. I estimated values for the unsorted array at 100,000 points and 1,000,000 points based on the observed rate of increase in the previous values (about 67x). In the heap, times appeared to be increasing by about 15x each time so I used this estimate for the 1,000,000 points as I couldn’t run it.

As can be seen, the times for both implementations are clearly linear in the logarithmic scale meaning they are showing exponential growth as we expected. The heap implementation is significantly faster especially at large values where the nlogn term is significantly smaller than the n^2 term. At smaller values, the speeds are more similar as the overhead of running the algorithm and the constant factors have a larger impact.

These results show obvious n^2 and nlogn growth in runtimes which is what we expected from the theoretical analysis. The constant factors of course have an impact but especially with the larger values, the trend is more obvious.

# Appendix: Full Code

1 #!/usr/bin/python3

2

3

4 from CS312Graph import \*

5 import time

6

7 # Unsorted Array implementation of a priority queue

8 class PriorityQueueArray:

9 def \_\_init\_\_(self):

10 self.set = set()

11

12 # Insert a node into the array

13 def insertNode(self, index):

14 self.set.add(index)

15

16 # return node with the smallest distance, remove from list

17 def deleteMin(self, dist):

18 minIndex = next(iter(self.set)) # gets first element from set

19 for num in self.set:

20 if dist[num] == float('inf'):

21 continue

22 elif dist[minIndex] is float('inf') or dist[num] < dist[minIndex]:

23 minIndex = num

24 self.set.remove(minIndex)

25 return minIndex

26

27 # decrease the dist value at that index

28 def decreaseKey(self, index, newDist):

29 pass

30

31 # Check if tree has more items

32 def isEmpty(self):

33 if len(self.set) == 0:

34 return True

35 else:

36 return False

37

38

39 class PriorityQueueHeap:

40 def \_\_init\_\_(self):

41 self.tree = []

42 self.pointers = []

43

44 # Insert a new node

45 def insertNode(self, node\_id):

46 # Put new node into next stop

47 loc = len(self.tree)

48 self.pointers.append(loc)

49 self.tree.append((node\_id, float('inf')))

50 # Bubble up not necessary here as all inserts will have inf distance

51

52 # return and remove the minimum node

53 def deleteMin(self, dist):

54 # Catch error case where only 1 item in tree

55 if len(self.tree) == 1:

56 topNode\_id, topNode\_dist = self.tree.pop()

57 return topNode\_id

58 # Get top node and remove from tree

59 topNode\_id, topNode\_dist = self.tree[0]

60 self.pointers[topNode\_id] = None

61 # move last node to top

62 self.tree[0] = self.tree[-1]

63 self.tree.pop()

64 # Bubble top node down

65 bottomNode\_id, bottomNode\_dist = self.tree[0]

66 self.pointers[bottomNode\_id] = 0

67 self.bubbleDown(bottomNode\_id)

68 return topNode\_id

69

70 # lower the value of a node

71 def decreaseKey(self, node\_id, newDist):

72 # get location from pointers

73 loc = self.pointers[node\_id]

74 # Change value at that location

75 self.tree[loc] = (node\_id, newDist)

76 # Bubble up

77 self.bubbleUp(node\_id)

78

79 # Bubble a value up in the tree

80 def bubbleUp(self, node\_id):

81 cur\_id = node\_id

82 while True:

83 cur\_loc = self.pointers[cur\_id]

84 if cur\_loc == 0: # Check for top

85 break

86 parent\_loc = (cur\_loc - 1) // 2

87 cur\_id, cur\_value = self.tree[cur\_loc]

88 parent\_id, parent\_value = self.tree[parent\_loc]

89 if cur\_value < parent\_value: # swap child with parent if less

90 self.tree[parent\_loc] = self.tree[cur\_loc]

91 self.pointers[cur\_id] = parent\_loc

92 self.tree[cur\_loc] = (parent\_id, parent\_value)

93 self.pointers[parent\_id] = cur\_loc

94 else:

95 break

96

97 # Bubble a value down in the tree

98 def bubbleDown(self, node\_id):

99 cur\_id = node\_id

100 while True:

101 cur\_loc = self.pointers[cur\_id]

102 first\_child\_loc = round((cur\_loc + 0.5) \* 2)

103 second\_child\_loc = (cur\_loc + 1) \* 2

104 cur\_id, cur\_value = self.tree[cur\_loc]

105 # Check if children are valid

106 maxLoc = len(self.tree) - 1

107 if first\_child\_loc > maxLoc and second\_child\_loc > maxLoc:

108 break

109 else:

110 if first\_child\_loc > maxLoc:

111 first\_child\_value = float('inf')

112 second\_child\_id, second\_child\_value = self.tree[second\_child\_loc]

113 elif second\_child\_loc > maxLoc:

114 first\_child\_id, first\_child\_value = self.tree[first\_child\_loc]

115 second\_child\_value = float('inf')

116 else:

117 first\_child\_id, first\_child\_value = self.tree[first\_child\_loc]

118 second\_child\_id, second\_child\_value = self.tree[second\_child\_loc]

119

120 if first\_child\_value <= second\_child\_value:

121 child\_id = first\_child\_id

122 child\_value = first\_child\_value

123 child\_loc = first\_child\_loc

124 else:

125 child\_id = second\_child\_id

126 child\_value = second\_child\_value

127 child\_loc = second\_child\_loc

128 if cur\_value > child\_value: # swap child with parent if less

129 self.tree[child\_loc] = self.tree[cur\_loc]

130 self.pointers[cur\_id] = child\_loc

131 self.tree[cur\_loc] = (child\_id, child\_value)

132 self.pointers[child\_id] = cur\_loc

133 else:

134 break

135

136 # Check if tree has more items

137 def isEmpty(self):

138 if len(self.tree) == 0:

139 return True

140 else:

141 return False

142

143

144 class NetworkRoutingSolver:

145 def \_\_init\_\_(self):

146 self.dist = None

147 self.prev = None

148 self.source = None

149 self.dest = None

150 self.network = None

151

152 def initializeNetwork(self, network):

153 assert (type(network) == CS312Graph)

154 self.network = network

155

156 def getShortestPath(self, destIndex):

157 self.dest = destIndex

158 path\_edges = []

159 total\_length = self.dist[self.dest]

160 index = self.dest

161

162 # Check for unreachable

163 if self.prev[destIndex] is None:

164 return {'cost': float('inf'),

165 'path': path\_edges}

166

167 # search backwards using prev pointers to find all edges used

168 while self.prev[index] is not None:

169 previous = self.prev[index]

170 prevNode = self.network.nodes[previous]

171 for edge in prevNode.neighbors:

172 if edge.dest.node\_id == index:

173 path\_edges.append((edge.src.loc, edge.dest.loc, '{:.0f}'.format(edge.length)))

174 index = previous

175

176 return {'cost': total\_length, 'path': path\_edges}

177

178 def computeShortestPaths(self, srcIndex, use\_heap=False):

179 self.source = srcIndex

180 t1 = time.time()

181

182 # Choose which heap implementation to use

183 if use\_heap:

184 print("Using heap implementation")

185 Q = PriorityQueueHeap()

186 else:

187 print("Using array implementation")

188 Q = PriorityQueueArray()

189

190 dist = []

191 prev = []

192 # load the queue with all the points

193 for i in range(len(self.network.nodes)):

194 dist.insert(i, float('inf'))

195 prev.insert(i, None)

196 Q.insertNode(i)

197 dist[srcIndex] = 0

198 Q.decreaseKey(srcIndex, 0)

199

200 # loop until queue is empty

201 while not Q.isEmpty():

202 # get minimum node from queue

203 uInd = Q.deleteMin(dist)

204

205 # check if queue is still returning reachable nodes

206 if dist[uInd] == float('inf'):

207 print("All reachable nodes searched")

208 break

209

210 # update values for each neighboring node

211 for edge in self.network.nodes[uInd].neighbors:

212 vInd = edge.dest.node\_id

213 newDist = dist[uInd] + edge.length

214 if dist[vInd] == float('inf') or newDist < dist[vInd]:

215 dist[vInd] = newDist

216 prev[vInd] = uInd

217 Q.decreaseKey(vInd, newDist)

218

219 # set values for use in getPath function

220 self.dist = dist

221 self.prev = prev

222 t2 = time.time()

223 return t2 - t1

224