Dijkstra Priority Queue Report

Emily Elzinga

September 2022

1 Checklist

20 Correctly implement Dijkstra's algorithm and the functionality discussed above. Include a copy of your (well-documented) code in your submission to the TA.

20 Correctly implement both versions of a priority queue, one using an array with worst case O(1), O(1) and O(|V|) operations and one using a heap with worst case O(log|V|) operations. For each operation (insert, delete-min, and decrease-key) convince us (refer to your included code) that the complexity is what is required here.
Check, mostly. There are some kinks to work out for larger graphs.

10 Explain the time and space complexity of both implementations of the algorithm by showing and summing up the complexity of each subsection of your code.
Check. The overall complexity for the binary heap implementation is $O((|V|+3)log_n)$, and the overall complexity for the array implementation is $O(V^2)$ because the $deleteMin$ method gets called |V| times.

20 For Random seed 42 - Size 20, Random Seed 123 - Size 200 and Random Seed 312 - Size 500, submit a screenshot showing the shortest path (if one exists) for each of the three source-destination pairs, as shown in the images below.
For Random seed 42 - Size 20, use node 7 (the left-most node) as the source and node 1 (on the bottom toward the right) as the destination, as in the first image below.
For Random seed 123 - Size 200, use node 94 (near the upper left) as the source and node 3 (near the lower right) as the destination, as in the second image below.
For Random seed 312 - Size 500, use node 2 (near the lower left) as the source and node 8 (near the upper right) as the destination, as in the third image below

20 For different numbers of nodes (100, 1000, 10000, 100000, 1000000), compare the empirical time complexity for Array vs. Heap, and give your best estimate of the difference (for 1000000 nodes, run only the heap version and then estimate how long you might expect your array version to run based on your other results). For each number of nodes do at least 5 tests with different random seeds, and average the results. Redo any case where the destination is unreachable. Each time, start with nodes approximately in opposite corners of the network. Graph your results and also give a table of your raw data (data

for each of the runs); in both graph and table, include your one estimated runtime (array implementation for 1000000 points). Discuss the results and give your best explanations of why they turned out as they did.

2 Pseudocode

Algorithm 1: Builds a heap implementation of a priority queue for the given vertices and distances

```
function makeQueue ($vertices, priorities$);
Input  : Vertices V with $dist$ array as Priorities
Output: A Priority Queue of vertices in heap format
bHeap = [inf]*(vertices size);
pointers = [-1]*(vertices size);
priorities = priorities;
for each index b of bHeap do
    pointer[bHeap[b]] = b
end
```

```
function deleteMin ($H$);
Input  : The Priority Queue H
Output: The vertex with the lowest priority
min = bHeap[0];
bHeap[0] = bheap[size - 1];
pointer[0] = (pointer(bHeap[size-1]));
bHeap = delete(bHeap[size-1]);
pointer = delete(pointers[bHeap[size-1]];
function decreaseKey ($node, newPriority$);
Input  : A node v in the graph and its new priority
Output: the PQ with the given node distance decreased
j = pointers[node];
priorities[node-1] = newPriority;
parentj = floor(($j - 1$)/2);
while $newPriority < priorities[parentj]$ do
    temp = bHeap[j];
    bHeap[j] = bHeap[parentj];
    bHeap[parentj] = temp;
    temp = pointers[bHeap[j]];
    pointers[bHeap[j]] = pointers[bHeap[j];
    pointers[bHeap[parentj]] = temp;
    j = parentj;
    parentj = floor((j)/2);
end
```