Algorithms and Datastructures

Assignment 4

Lucas van der Laan
s1047485

1

Step	Unvisited	0	1	2	3	4	5
1	{1,2,3,4,5}	0	X	X	X	X	X
2	{1,2,3,4,5}	0	30	40	X	X	X
3	{2,3,4,5}	0	30	40	50	80	X
4	{3,4,5}	0	30	40	50	70	X
5	{4,5}	0	30	40	50	70	X
6	{5}	0	30	40	50	70	110
7	{}	0	30	40	50	70	110

2

(a)

Algorithm 1 Feasible path s $\rightarrow$ t

```
1: Require:
2:     E - The set of all the roads
3:     C - The set of all the cities
4:     L - A positive integer
5:     d - List of distances
6:     s - The starting city
7:     t - The ending city
8: d ← [∞ · |C|]                          ▷ The distance to every city is infinity
9: d[s] ← 0
10: Q ← PriorityQueue
11: enQueue(Q, v, 0)
12: while Q not Empty do u ← deQueue()
13:     for each w in adjacent[u] do
14:         new ← d[u] + c(u, w)
15:         if new < d[w] then
16:             if w ≠ t then
17:                 if d[w] == ∞ then
18:                     enQueue(Q, w, new)
19:                 else
20:                     decreasePriority(Q, w, new)
21:                 end if
22:             else
23:                 if d[t] < L then
24:                     return True
25:                 end if
26:             end if
27:             d[w] ← new
28:         end if
29:     end for
30: end while
31: return False
```

(b)

Algorithm 2 Shortest path s $\rightarrow$ t

```
1: Require:
2:    E - The set of all the roads
3:    C - The set of all the cities
4:    L - A positive integer
5:    d - List of distances
6:    s - The starting city
7:    t - The ending city
8: d ← [∞ · |C|]                          ▷ The distance to every city is infinity
9: d[s] ← 0
10: Q ← PriorityQueue
11: enQueue(Q, v, 0)
12: while Q not Empty do u ← deQueue()
13:     for each w in adjacent[u] do
14:         new ← d[u] + c(u, w)
15:         if new < d[w] then
16:             if w ≠ t then
17:                 if d[w] == ∞ then
18:                     enQueue(Q, w, new)
19:                 else
20:                     decreasePriority(Q, w, new)
21:                 end if
22:             end if
23:             d[w] ← new
24:         end if
25:     end for
26: end while
```

At the end of the algorithm, we will have the shortest path to t and everything between s and t. It is correct because:

- We only need to enqueue items either linked to the start or the end. Thus in in initialization, we only need to enqueue v with a weight of 0, since that is our start.
- We go until Q is empty, if we do not find t while doing so, there is no path to t and thus we do not have to continue with other paths.
- We check the adjacency list for every adjacency at u. u will always be the item with the highest (lowest in number) priority, thus the shortest path. We then check if the new distance is shorter than what we had, if so we replace the distance, thus marking a newest shortest path to w.
- We only add vectors to the Queue if d[w] is equal to infinity, this is because only items that are linked to s will get added. If they have already been discovered, we will instead decrease its priority, as we do not want it in the Queue twice.

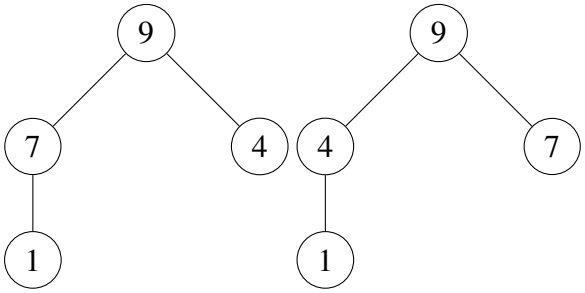
3

No, it does not work as intended.
Let's say we have 3 nodes (A, B, C) with Edges {(A,B,5), (B,C,-10), (A,C,2)}. If we add a constant c to every weight so that every weight is no longer negative, let's say 20, these edges would then change to {(A,B,25), (B,C,10), (A,C,22)}.
In the case of negative values, we would expect A $\rightarrow$ B $\rightarrow$ C to be -5, but with the added constant it is 35. This is because now 25 + 10 = 35, which results in A $\rightarrow$ C being shorter with 22.

We can also look at, what if we normalize the negative value to 0, so we make c 10 instead. This would result in {(A,B,15), (B,C,0), (A,C,12)}, which shows the exact same thing as with c being 20. A $\rightarrow$ B $\rightarrow$ C is still bigger than A $\rightarrow$ C.

4

(a)



(b)

Algorithm 3 D-Ary-Parent(i)

Require:
    i - The index of the child
    d - How many children every parent has
return $\leftarrow \lceil \frac{i}{d} \rceil - 1$

This is assuming that i is 1-based, if it is 0-based we need to add another 1 to k.

Algorithm 4 D-Ary-Child(i, k)

Require:
    i - The parent of k
    k - The index that we want
    H - The heap as an array
    d - How many children every parent has
return $\leftarrow$ H$[d \cdot i + k]$

(c)

The height of a d-ary heap is $\lfloor \log_d(n) \rfloor + 1$, just like in b. we can just scale the binary heap to a $d-$arry heap. This is correct, because every row of heap elements has d^i elements, where i is

the index of the row. The first row is row 0, and has $d^0 = 1$ element, the second row is $d^1 = d$, the third is $d^2 = d^2$ etc... This results in us being capable of reversing this and using the log of d to get which i we are at. Because our index is 0-based instead of 1-based, we add 1 to the result and get the total height.

5

(a)

Algorithm 5 MaxHeapify(H, d, i)

```
1:  Require:
2:      H - The heap as an array
3:      d - How many children every parent has
4:      i - The index of the parent
5:  dd ← d
6:  if Size(H) − 1 < d · i + d then
7:      dd ← (d · i + d) − (Size(H) − 1)
8:  end if
9:  biggest ← 0
10: biggestIndex ← 0
11: for let j = 0..d do
12:     child ← D-Ary-Child(i, j + 1)
13:     if child > biggest then
14:         biggest ← child
15:         biggestIndex d · i + (j + 1)
16:     end if
17: end for
18: H[] ← H[i]
19: H[i] ← biggest
20: if d ≠ dd then
21:     return H
22: else
23:     return MaxHeapify(H, d, biggestIndex)
24: end if
```

Algorithm 6 Extract-Max-d-ary(H, d)

```
1: Require:
2:     H - The heap as an array
3:     d - How many children every parent has
4: returnValue ← H[0]
5: n ← Size(H)
6: H[0] = H[n − 1]
7: H ← MaxHeapify(H, d, 0)
8: return returnValue
```

$\mathcal{O}(\log(n) \cdot d)$, because $\mathcal{O}(d)$ happens for every iteration of the MaxHeapify function, which for the worst case happens for every row, which is the height. The height is $\lfloor \log_d(n) \rfloor + 1$, so for

every $\lfloor \log_d(n) \rfloor + 1$ we do d iterations of a loop. We can simplify this from $\mathcal{O}((\lfloor \log_d(n) \rfloor + 1) \cdot d)$ to $\mathcal{O}(\log(n) \cdot d)$.

(b)

As I was running out of time, I am just writing it down in words/steps, as it is quite similar to Extract-Max.

- Insert the element into the last position of the Heap (H$[n]$)
- Max heapify the parent of n using D-Ary-Parent(n) from 4b.
  - This is done using a single step from the Max-Heapify from 5a, as we do not need it recursively here.
- If the parent changed, max heapify the parent of the parent
- Repeat the last step until either the parent hasn't changed, or the last parent that we max-heapified was the root.

The running time is then also the exact same as Extract-Max, as in the worst case we go through the entire height of the heap and we have to do Max-Heapify again, which involves the loop of d for the entire height. Thus the running time is $\mathcal{O}(\log(n) \cdot d)$.

(c)

- If k < A[i] then thrown an error else A[i] = k
- The rest of the algorithm is the same as insert, but then from wherever i is.
  - This means that we keep on swapping the parent and child upwards as long as the child is bigger than the parent.

Because i can be the literal last element of the Heap, the worst case is the same as Insert again, which is $\mathcal{O}(\log(n) \cdot d)$.