CPSC 320 2024W1: Assignment 2 Solutions

2 Logarithmic functions grow more slowly than "polynomial" functions

The textbook (2.8, page 41) provides the following useful fact, stating roughly that logarithmic functions are big-O-upper-bounded by simple "polynomial" functions, specifically functions that are n to some constant power:

> **Fact:** *For every $b > 1$ and every $x > 0$, we have $\log_b n = O(n^x)$.*

1. (4 points) Use the fact above to prove the stronger assertion that logarithmic functions of n grow strictly more slowly, in the little-o sense, than functions that are n to some constant power:

   For every $b > 1$ and every $x > 0$, we have $\log_b n = o(n^x)$.

   Fix $b > 1$ and $x > 0$. Using the fact, and since $x/2 > 0$, it must be the case that $\log_b n = O(n^{x/2})$. By the definition of big-O, we have that

   $$\exists c > 0 \exists n_0 \in \mathbb{N} \text{ such that } \log_b n \leq c n^{x/2}.$$

   As a result,

   $$\lim_{n\to\infty} \frac{\log_b n}{n^x} \leq \lim_{n\to\infty} \frac{cn^{x/2}}{n^x} = \lim_{n\to\infty} \frac{c}{n^{x/2}} = 0.$$

2. (4 points) Now use the fact of part 1 to show that $\sqrt{n} = o(n/\log^3 n)$. Here the log is to the base 10, and $\log^3 n = (\log n)^3$.

   $$\lim_{n\to\infty} \frac{\sqrt{n}}{(n/\log^3 n)} = \lim_{n\to\infty} \frac{\log^3 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{\log^3 n}{n^{1/2}}.$$

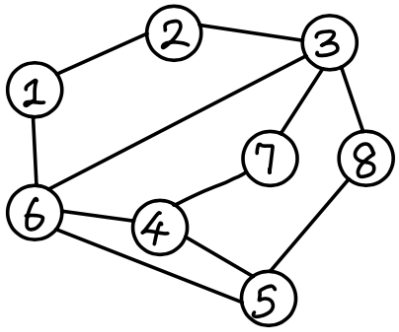
   Recall that we interpret $\log n$ as $\log_{10} n$. From part 1 above, choosing $b = 10$ and $x = 1/6$, we have that $\lim_{n\to\infty} \frac{\log n}{n^{1/6}} = 0$. So it is also the case that

   $$\lim_{n\to\infty} \left(\frac{\log n}{n^{1/6}}\right)^3 = \lim_{n\to\infty} \frac{\log^3 n}{n^{1/2}} = 0.$$

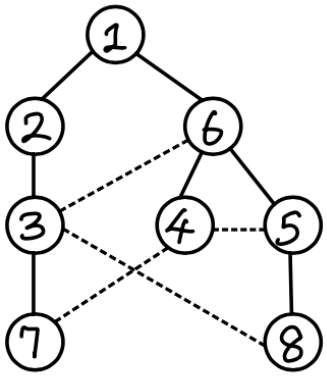
3 Counting Shortest Paths

Let $G = (V, E)$ be an undirected, unweighted, connected graph with node set $\{1, 2, \ldots, n\}$, where $n \geq 1$. For any node v, let $c(1, v)$ be the total number of shortest paths (i.e., paths with the minimum number of edges) from 1 to v.

Example: The following graph has one shortest path from 1 to 6. Also there are three shortest paths from 1 to 8. Two of these, namely path 1,2,3,8, and path 1,6,3,8, go through node 3, while one, namely path 1,6,5,8 goes through node 5.



1. (2 points) Draw a breadth first search tree rooted at 1 for the graph above. Include all dashed edges as well as tree edges. (A scanned hand-drawn figure is fine as long as it is clear.)



2. (2 points) How many shortest paths are there from node 1 to node 7? That is, what is the value of $c(1, 7)$? Give a list of the paths.

   $c(1, 7) = 3$, and the three paths are 12,3,7; 1,6,3,7; and 1,6,4,7.

3. (4 points) Here is an inductive definition for $c(1, v)$: The base case is when $v = 1$, in which case we define $c(1, 1)$ to be 1, since there is exactly one shortest path from 1 to itself (with no edges). When $v > 1$, let $d[v]$ be the depth of any node v in the bfs tree of G rooted at 1. Then

$$c(1, v) = \sum_{\substack{u \mid (u,v) \in E, \text{ and} \\ d[u] = d[v] - 1}} c(1, u).$$

Intuitively, on the right hand side we are summing up the number of shortest paths from node 1 to all nodes u at level $d[v] - 1$, such that there is an edge of G (either a tree edge or a dashed edge) from u to v. In our example above, $c(1, 8) = c(1, 3) + c(1, 5)$.

Provide code in the spaces indicated below, that leverages this inductive definition to obtain an algorithm that computes $c(1, v)$ for all nodes v. This code first initializes $c(1, v)$ for all v, and then calls a modified version of breadth first search that you will flesh out.

In the following code, term $c(1, u)$ of the sum given in the inductive definition above is added to $c(1, v)$ when edge (u, v) is examined during the call to MODIFIED-BFS(1).

```
procedure COUNT-SHORTEST-PATHS(G.1)
    ▷ G is an undirected, connected graph with nodes {1, 2, ..., n}, where n ≥ 1
    ▷ compute c(1, v), the number of shortest paths, from 1 to v, for all nodes v
    ▷ add code here to initialize c(1, v) for all v ∈ V:
    c(1, 1) ← 1
    for all v ∈ V, v ≠ 1 do
        c(1, v) ← 0                                   ▷ no shortest paths to v found yet
    end for
    call MODIFIED-BFS(G)
end procedure

procedure MODIFIED-BFS(G)
    ▷ Assume that this procedure can access and update the variables c(1, v)
    add node 1 as the root of the bfs tree
    d[1] ← 0                                                  ▷ node 1 is at level 0
    for all nodes v > 1 do
        d[v] ← ∞                                          ▷ v is not yet in the tree
    end for
    d ← 1
    while not all nodes are added to the tree do
        for each node u in the tree with d[u] = d − 1 do
            for each v adjacent to u do
                if d[v] == ∞ then                     ▷ v has not yet been added to the tree
                    d[v] ← d                  ▷ put node v at level d of the tree (as a child of u)
                end if
                ▷ add your code here to update c(1, v):
                if d[v] == d then                 ▷ there is either a dashed or tree edge from u to v
                    c(1, v) ← c(1, v) + c(1, u)
                end if
            end for
        end for
        d ← d + 1
    end while
```

```
end procedure
```

4 Provision planning

You run a business to provide provisions to individuals with plans for long-distance hikes. An individual requesting your help tells you:

- d: The distance (in km) that the individual can hike per day, where d is a positive integer;
- p: How many days of provisions (food, water, etc.) they can carry, where p is a positive integer.

In addition you have access to:

- $R[1..k]$: inter-town distances along the planned route, with $k \geq 1$. That is, there are $k+1$ towns along the route with town 0 being the start and town k being the destination; and $R[i]$ is the distance (in km) from town $i-1$ to town i for $1 \leq i \leq k$.

You need to store provisions in towns along the way, that the hiker will pick up en route. For this problem, you need only concern yourself with instances $(d, p, R[1..k])$ that have valid solutions. A *valid solution* is a list of towns where provisions can be placed, so as to ensure that the hiker will not run out of provisions. In a valid solution, the distance traveled between the starting town and the first town in the solution, or between a consecutive pair of towns in the solution, or between the last town in the solution and the destination, is $\leq dp$ (where d and p are defined above). You never need to provide provisions at town 0 or town k (the hiker has their own provisions at the start of the trip and does not need them once the destination is reached). You want to find an *optimal solution*, that is, a valid solution of minimum length.

Example: A hiker plans to hike for at most 7km per day, and can carry provisions that last for two days, so $d = 7$ and $p = 2$. Also, $k = 5$ and $R[1..5] = [8, 2, 2, 10, 5]$, so distances between towns are:

One valid solution is the list of towns 1, 3, 4:

1. (3 points) In the example above, the solution is not optimal. Give *three different optimal solutions*, which have only two towns in each.

   Three optimal solutions are the list 1,4; the list 2,4; and the list 3,4.

2. (2 points) Consider the following greedy algorithm.

```
1: procedure GREEDY-PROVISIONS(d, p, R[1..k])
2:     i ← 0                                              ▷ 0 is the starting town
3:     L ← empty list
4:     while i < k do
5:             ▷ find the town j ≤ k that's furthest away from town i, among those of distance ≤ dp
6:         j ← i + 1
7:         d' ← R[j]
8:         while (j < k) and (d' + R[j + 1] ≤ dp) do
9:             j ← j + 1
10:            d' ← d' + R[j]
11:        end while
12:        if j < k then
13:            L ← L, j                                   ▷ append j to the solution L
14:        end if
15:        i ← j                                          ▷ update i
16:    end while
17:    output the list L
18: end procedure
```

Explain why the output L is a valid solution, i.e., one in which the hiker will not run out of provisions, given that instance $(d, p, R[1..k])$ is guaranteed to have a valid solution.

Given a town i (initially 0) that is not the destination, lines 5-11 find the town j that's furthest away from i but within distance dp. So provisions will last from i to j, ensuring validity. If this town j is not the destination, then it is added to the list L (lines 12, 13). Then i is updated to j and the process is repeated, until i is the destination.

3. (3 points) Give a big-O bound on the running time of the greedy algorithm as a function of k, the number of towns, and justify your answer. (The lines of pseudocode above are numbered so that you can refer to specific lines in your reasoning.)

The algorithm runs in $O(k)$ time. Lines 2 and 3 take time $O(1)$. The **while** loops iterate through through the $k + 1$ towns in increasing order, taking constant time per town (lines 6,7, lines 9, 10 and lines 12-14). Although there are nested **while** loops, each town is considered just once. Line 17 takes time $O(k)$. So the total is $O(1) + O(k) + O(k) = O(k)$.

4. (3 points) Let L be the output of the greedy algorithm, and let i_1 be the first town in list L. Let L^* be an optimal solution for instance $(d, p, R[1..k])$, and let i_1^* be the first town in list L^*. Let L' be the list obtained from L^* by replacing i_1^* by i_1. Explain why L' is also an optimal solution for instance $(d, p, R[1..k])$.

   Provisions last from town 0 to town i_1, since solution L is valid (by part 2). Also, since i_1 is chosen to be the town furthest away from town 0 but still within distance dp, it must be that $i_1^* \leq i_1$. Let i_2^* be the second town in solution L' (or L^*). Since L^* is valid, and $i_1^* \leq i_1$, we have that the distance from i_1 to i_2^* is also at most dp, and so provisions last between the towns i_1 and i_2^* of L'. The rest of L' is the same as L^*, and so L' is valid, and since it has the same length as L^*, it is also optimal.

5. (3 points) Complete the following argument that uses induction on k to show that the greedy algorithm outputs an optimal solution on any instance $(r, p, R[1..k])$ with a valid solution. You need to fill in the details for the base case, and parts (i) and (ii) of the inductive step. (The inductive hypothesis is done for you.)

   Base case: If $k = 1$ then...

   ...there are just two towns, and the destination has distance at most dp from the starting town (since the given instance is valid). So L is the empty list, which is optimal.

   Inductive hypothesis: Let $k \geq 1$. The greedy algorithm outputs an optimal solution for any instance with $k + 1$ towns (assuming that the instance has a valid solution).

   Inductive step: Show that the greedy algorithm outputs an optimal solution when there are $k + 2$ towns along the route. (Refer back to earlier parts of this problem.)

   (i) There is an optimal solution that starts with i_1 because ...

   ...the solution L' constructed in part 4 is such a solution.

   (ii) Once i_1 is added to the list, the remaining solution chosen by the Greedy algorithm is optimal because...

   by the Inductive hypothesis, since once i_1 is added to the list, the Greedy algorithm solves the smaller problem between town i_1 and the destination, which involves at most $k + 1$ towns.

   [NOTE: We are using strong induction here, so relying on a stronger hypothesis than that stated, namely that the greedy algorithm outputs an optimal solution for any instance with *at most* $k + 1$ towns.]

   Combining (i) and (ii), we can conclude that our algorithm finds an optimal solution for instance $(d, p, R[1..k])$.

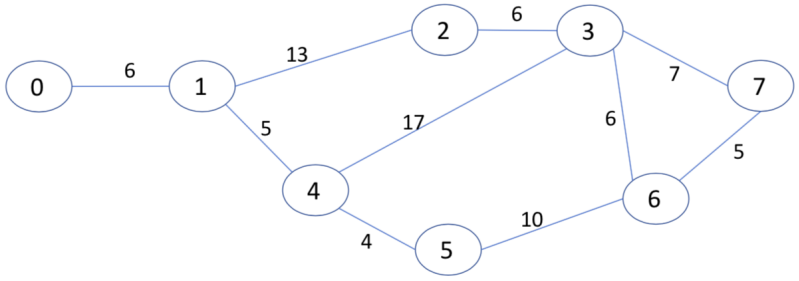
5 Route planning

You want to expand your hiker support business, so that if multiple routes are possible from the starting to destination town, you can determine whether at least one of these routes is *feasible*.

For this problem, the possible routes are represented as an undirected graph $G = (V, E)$, with nodes numbered 0 to $n-1$ representing towns. We assume that the starting and destination towns are nodes 0 and $n-1$, respectively. There is an edge $e = (u, v)$ between any pair of towns u and v that are directly connected by a trail, and the weight $w(e)$ of such an edge is the distance between the towns u and v. Let m be the number of edges of the graph.

A *route* is a simple path (a path in which no node repeats) from town 0 to town $n-1$. Given d (max distance traveled per day) and p (number of days of provisions that the hiker can carry), a route $0 = t_0, t_1, \ldots, t_k = n-1$ is *feasible* if for all edges (t_{i-1}, t_i), $1 \leq i \leq k$, it is the case that $w(t_{i-1}, t_i) \leq dp$. (This ensures that if the hiker stocks up on provisions in any town along the route, the hiker won't run out of provisions before getting to the next town.)

Example: Let $d = 6$, $p = 2$, $n = 8$, and let the graph be:



In this case route $0, 1, 2, 3, 7$ is not feasible, since even if the hiker gets provisions in town 1, the hiker cannot make it to town 2 before running out of provisions. However, route $0, 1, 4, 5, 6, 3, 7$ is feasible.

1. (1 point) For the above example, list one more feasible route, with six towns, including the start and destination. (No justification needed.)

0	1	4	5	6	7

2. (2 points) For the above example, list *two* more routes with at most six towns that are *not* feasible. (No justification needed.)

0	1	2	3	7	

0	1	4	3	6	7

3. (2 points) Given $(d, p, G = (V, E), w())$ where each edge e of E has weight $w(e)$, explain how to use breadth first search to determine, in $O(n + m)$ time, whether there is a feasible route.

   Let $G' = (V, E')$ be the graph obtained from G by removing all edges with $w(e) > dp$, and then making the remaining edges unweighted. Graph G' can be constructed in $O(n + m)$ time from G. We can run BFS on G' from the starting town, and if the destination town is reached we know there is a feasible route.

4. (2 points) Given $(d, p, G = (V, E), w())$ where each edge e of E has weight $w(e)$, can we use depth first search to determine, in $O(n + m)$ time, whether there is a feasible route?

   Yes, we can use the same approach as part 3, except use DFS instead of BFS. The path returned may not be optimal, but it will be feasible since any pair of consecutive towns on the path are within distance at most dp of each other.