CPSC 320 2024W1: Assignment 2

Hansel Poe

October 1, 2024

This assignment is due **Friday, October 4 at 7 PM**. Late submissions will not be accepted. All the submission and formatting rules for Assignment 1 apply to this assignment as well.

1 List of names of group members (as listed on Canvas)

Provide the list here. This is worth 1 mark. Include student numbers as a secondary failsafe if you wish.

- Hansel Poe - 82673492

2 Statement on collaboration and use of resources

To develop good practices in doing homeworks, citing resources and acknowledging input from others, please complete the following. This question is worth 2 marks.

1. All group members have read and followed the guidelines for groupwork on assignments given in the Syllabus).
   ◉ Yes ○ No
2. We used the following resources (list books, online sources, etc. that you consulted):
3. One or more of us consulted with course staff during office hours.
   ○ Yes ◉ No
4. One or more of us collaborated with other CPSC 320 students; none of us took written notes during our consultations and we took at least a half-hour break afterwards.
   ○ Yes ◉ No

   If yes, please list their name(s) here:
5. One or more of us collaborated with or consulted others outside of CPSC 320; none of us took written notes during our consultations and we took at least a half-hour break afterwards.
   ○ Yes ◉ No

   If yes, please list their name(s) here:

3 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 $\log_b n = o(n^x)$.

   We will use the following fact: If $lim_{x\to\infty} \frac{f(n)}{g(n} = \infty$,then $g(n) \in of(n)$.
   Fix $b > 1$ and $x > 0$. Now, consider $\lim_{n\to\infty} \frac{n^x}{log_b n}$. Using L'Hopital rule, we take the derivative of both the numerator and denominator, we get $\lim_{n\to\infty} \frac{xn^{x-1}}{\frac{1}{nln(b)}} = \lim_{n\to\infty} xln(b)n^x = \infty$. Therefore, $log_b n \in o(n^x)$. $Q.E.D$

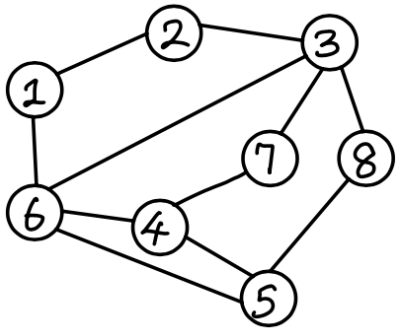
2. (4 points) Now use the fact of part 1 to show that $\sqrt{n} = o(n/\log^3 n)$. Here the log is to the base 10, and $\log^3 n = (\log n)^3$.

From the fact above, we can let $b = 10$ and $x = \frac{1}{6}$ such that for all positive real number c, there is n_0 such that for all $n \geq n_0$, $\log n \leq cn^{\frac{1}{6}}$ taking the cube of both sides, we get $\log^3 n \leq c^3\sqrt{n}$, divide both sides by $\log^3 n$ and then multiply by $\sqrt{n}$ to get $\sqrt{n} \leq c^3 n/\log^3 n$we can let $n_0' = n_0$ and $c' = c^3$ so that for all positive real number c' there is n_0' such that, for all $n \geq n_0'$, $\sqrt{n} \leq c'n/\log^3 n$, and by definition of little o, $\sqrt{n} \in o(n/\log^3 n)$

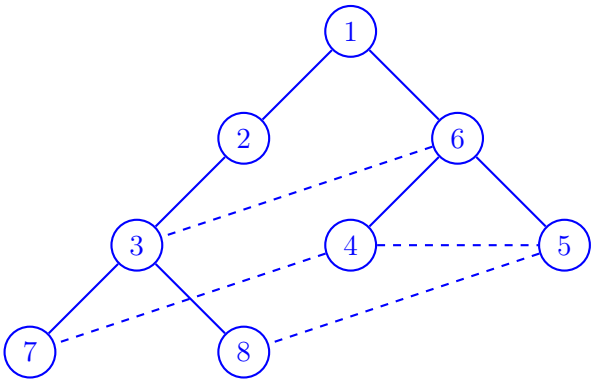
4 Counting Shortest Paths

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. There are three solutions : $(1, 2, 3, 7)$, $(1, 6, 3, 7)$, $(1, 6, 4, 7)$

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.

procedure COUNT-SHORTEST-PATHS(G.1)
▷ G is an undirected, connected graph with nodes $\{1, 2, \ldots, n\}$, where $n \geq 1$
▷ 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** $v \in V$:
c(1,1) = 1
for each $v > 1$ **do**
c(1,v) = 0;
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
$d[1] \leftarrow 0$ ▷ node 1 is at level 0
for all nodes $v > 1$ **do**
$d[v] \leftarrow \infty$ ▷ v is not yet in the tree
end for
$d \leftarrow 1$
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 $d[v] == \infty$ **then** ▷ v has not yet been added to the tree
$d[v] \leftarrow d$ ▷ 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**
$c(1, v) = c(1, u) + c(1, v)$;
end if

end for
end for
$d \leftarrow d + 1$
end while
end procedure

5 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:

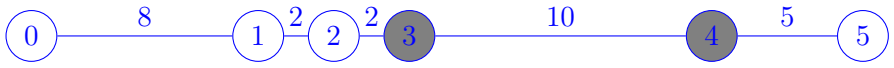
- $R[1..k]$: inter-town distances along the planned route, with $k \geq 1$. That is, there are $k+1$ towns along the route with town 0 being the start and town k being the destination; and $R[i]$ is the distance (in km) from town $i-1$ to town i for $1 \leq i \leq k$.

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 town



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.

The solution is valid if the distance between town 0 and the first town in L, any two towns in L, and the last town in L and the destination are all $\leq dp$ or in the trivial case, when $k = 1$ and L is empty.

In the trivial case, where $k = 1$, line 3 ensures L starts out empty. In the first iteration of the outer while loop, we have $j = 1 \not< k$ (because of line 6) which fails the check at line 12, so nothing is added to L. line 15 then sets $i = 1$ and the next check at line 4 fails, terminating the loop. This ensures L remains empty by the end of the algorithm

In non trivial case, the check at line 8 ensures that j will keep advancing until j is already the destination or until $d' + R[j + 1] \leq dp$. This means that when j is added to L at line 13, if it is the first element in L, then distance between town 0 and j is $\leq dp$, if not, then the distance between the previous town in L (town with largest number in L that is less than j) and j is $\leq dp$. The outer while loop ensures that we will keep adding new towns to L until we reach an i where the distance between town i and the destination is $\leq dp$. If we reach such a case, then i must be the last town added to L. For this iteration of i, we must have $j = k$ by the end of the inner for loop and the check at line 12 fails, nothing is added to L and line 15 updates i to $j = k$ and the outer for loop terminates. Confirming the fact that the distance between the final element in L (the town with largest number) and the destination is $\leq dp$

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 is big-o of k. The sum of number of outer while loop and inner while loop iterations determines the running time. Two lines are responsible for incrementing j, line 6 and 9. Execution of one of these lines corresponds to one iteration of either an inner or outer for loop and also corresponds to one value of j. Line 15 updates the value of j to i (meaning the next outer for loop starts out where the previous one left of) and so j will assume the values of all the towns throughout the execution of this algorithm (except for 0) and this represents the total number of inner and outer for loop iterations (with possible constant difference). There are k towns, so the running time is big-o of 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 $(d, p, R[1..k])$.

   We know that $i_1^* \leq i_1$. Replacing i_1^* with i_1 in L', we do not change the number of towns in L', keeping it optimal. But also, $\sum_{n=1}^{i_1'} R[n] = \sum_{n=1}^{i_1} R[n] \leq dp$ and $\sum_{n=i_1'+1}^{i_2'} = \sum_{n=i_1+1}^{i_2'} \leq \sum_{n=i_1^*+1}^{i_2'} \leq dp$ because $i_1^* \leq i_1$. So solution L' stays valid.

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 only two towns: 0 and 1, and one distance $R[1] \leq dp$. so L' is empty.

   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 ...

   Assume the negation of our statement, that is, for all optimal solution, it doesn't start with i_1. Now consider an instance where $k > 1, \sum_{n=1}^{k} R[n] > dp$. From our inductive hypothesis, for $k \geq 1$, the greedy algorithm will output an optimal solution for our instance. But then, our solution is empty, because it doesn't start with i_1 and the distance between town 0 and k exceeds dp, with no town with provisions in between, Thus our solution is invalid. By contradiction, There must exist an optimal solution that starts with i_1.

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

   For any $j \geq 1$; If we have i_j in our solution S, then the solution is optimal for the $(r, p, R[1 \dots i_j])$ instance with $i_j + 1$ towns. We can let i_j be our starting point and use the greedy algorithm from our hypothesis to output an optimal solution S' for the instance $(d, p, R[i_j \dots i_{j+1}])$. We then do $S = S \cup S'$ to get the solution for the instance $(d, p, R[1 \dots i_{j+1}])$

   Once i_1 is added to the list. By above, the greedy algorithm will then continue to chose optimal solutions for the remaining solutions.

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

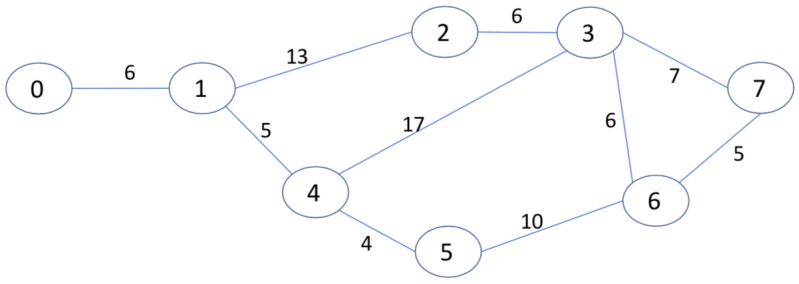
6 Route planning

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	6	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.

   Define a new boolean structure $p[v]$ that is 1 if v can be reached and 0 if not. We initialize $p[v] = 0$ for all $v > 0$ and $p[1] = 1$

   Now, in the bfs, for each visited node v , when iterating through all its adjacent vertices u, we add this piece of code:

if $w(e = (u, v)) \leq dp$ and $p[v] == 1$ **then**
  $p[u] \leftarrow 1$;
end if

At the end of the bfs, we return $p[n-1]$, 1 means a feasible route exists, 0 means it doesn't.

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 assume the the modified DFS returns 1 if a feasible route exists, 0 if not.

   Now, for each visited node v in the dfs,

   if $v == n - 1$ **then**
     return 1;
   end if

   Because a feasible route exists if we can visit $n - 1$.

   Then, when iterating through each adjacent node u, let $e = (u, v)$, if $e > dp$, then, do not visit u, we simply proceed onto the next adjacent vertex, visiting a vertex u only if it is unvisited and $e \leq dp$

   Because DFS uses a stack, it can backtrack if we reached a dead end (a node where we can't proceed because all adjacent u is visited or $e > dp$).

   If no feasible solution exists, then we will backtrack all the way to vertex 0, that is, the stack will pop all of its elements before it has the chance to return 1.

   So we add

   return 0

   at the end of our algorithm to indicate that no feasible route exists.