CS 5150/6150 Graduate (Advanced) Algorithms
Fall 2022
Assignment 6: Graph Algorithms
Total Points: 75

Submission notes

- Due at 11:59 pm on **Wednesday, November 23, 2022**.
- Solutions must be typeset (not hand-written or scanned).
- You should strive to write clear, concise solutions to each problem, ideally fitting on a page or less. The easier your work is to read & follow, the easier it is for the TAs to award you points.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for problem sets, *collaboration with other students must be limited to a high-level discussion of solution strategies.* If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration at the top of your homework submission. **You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.**

IMPORTANT Instructions for Problems 1 and 3

For these problems, which are often called *graph reduction problems*, you will want to transform the input into a graph and apply any of the following standard graph algorithms that we've covered in class or are assumed as prerequisites as subroutines: search (BFS, DFS), shortest paths (Dijkstra, Bellman-Ford, Floyd-Warshall, Johnson), topological sort, max-flow/min-cut (Ford-Fulkerson, Edmonds-Karp), and minimum spanning tree (Prim, Kruskal, Boruvka). You may want to run more than one standard algorithm, or use the same one multiple times - both are acceptable.

Whenever you use a standard graph algorithm, you **must provide the following information**. (I recommend actually using a bulleted list.)

- What are the vertices? What does each vertex represent?
- What are the edges? Are they directed or undirected?
- If the vertices and/or edges have associated values, what are they?
- What problem do you need to solve on this graph?
- What standard algorithm are you using to solve that problem? Explain how its solution solves your problem.
- What is the running time of your entire algorithm, including the time to build the graph if needed, as a function of the original input parameters?

Instructions on How to Answer Algorithm Design Questions

Please read the following guidelines on how to answer algorithm design questions *(you may lose points if you do not follow the instructions).*

1. **Algorithm Description** You are required to clearly describe the main idea of your algorithm.
2. **Pseudocode** Providing pseudocode for your algorithm is optional. However, pseudocode is very helpful for explaining algorithms clearly, and you are *strongly encouraged* to provide pseudocode for your algorithms. Additionally, having pseudocode available often leads to more partial credit for incorrect solutions, as the TAs can more easily identify correct ideas in your solution.

   Note: copying code from an implementation (e.g. python/java/c++) as "pseudocode" does not satisfy this requirement, as grading it is difficult, and key steps may be difficult to identify. In principle, pseudocode should be relatively easy to read while keeping all relevant operations of the algorithms. *You are encouraged to do your best to write good pseudocode, following examples given in class.*
3. **Correctness** You also need to explain/argue why your algorithm is correct, and handles all necessary cases – especially when the correctness of your algorithm is not obvious.
4. **Time Analysis** You are required to analyze the time complexity of your algorithm (i.e., provide a written justification for the claimed asymptotic runtime in terms of the algorithm's operations, not just state the time complexity). You may refer to your pseudocode in this analysis, as it often simplifies the descriptors (e.g. "the `for` loop on line 2").

1. **(25 points)**

   Please read the instructions for graph reduction problems on pg. 2 before beginning.

   It's now avalanche season in the Wasatch, and time to be (much) more cautious in choosing your hiking trails. Unfortunately, some of the new grad students saw one too many amazing photos on social media, and headed up into the mountains for a dayhike. They made it up to one of the smaller summits, but just called you terrified that the route they took up would be extremely unsafe to descend. Fortunately, using years of information from the Utah Avalanche Center's forecasts and reports[1], an unnamed faculty member has painstakingly constructed a winter trail-map annotated with the probabilities of encountering a slide on an arbitrary late-November day in the mountains. The map shows a network of trails, where vertices represent trailheads (which are all accessible by vehicle), trail junctions, and landmarks (e.g. lakes, summits) and edges are well-established routes connecting them. The trails have been curated so that they do not cross except at the locations corresponding to vertices, and luckily your friends are at one of the marked sites. There is at most one edge (route) between any two vertices (sites). Each edge (route) *and* each vertex (site) on the map has a probability that you will encounter an avalanche while traversing/visiting it, and you can assume that you will never encounter more than one avalanche at a given site (it's too cold to stick around that long in one spot) or while traversing a single route (edge).

   Describe and analyze an efficient algorithm to find the path from your friends' location to any trailhead (where you can meet them with a vehicle) *with the smallest number of expected avalanches*. Note that in winter, it's a really bad choice to stray from known trails without extensive backcountry training and equipment (which your friends don't have), so you cannot recommend any unmarked route or shortcuts.

[1] UAC is an awesome resource everyone should know about: `https://utahavalanchecenter.org`

2. **(25 points)**

Let $G = (V, E)$ be an undirected graph with edge-weights given by $w : E \to \mathbb{R}$. Assume that $w(e) \neq w(f)$ whenever e, f are distinct edges of G. We say that an edge is *treacherous* if it is the maximum weight edge for some cycle of G. On the other hand, an edge is *reliable* if it is not contained in any cycle of G.

(a) (9 points) Prove that the minimum spanning tree of G contains every reliable edge.

(b) (9 points) Prove that the MST of G does not contain any treacherous edge.

(c) (7 points) Describe and analyze an efficient algorithm for finding the minimum spanning tree by repeatedly removing the highest-weight treacherous edge of G.

3. **(25 points)**

Please read the instructions for graph reduction problems on pg. 2 before beginning.

While in many families, Thanksgiving traditions revolve around food and football, I think board games are an under-appreciated holiday classic which often channel minor irritation from crowded gatherings into healthy competition. This season, you're introducing your cousins to a game called Crusade. The board[2] consists of a map consisting of n (contiguous) regions outlined with black borders; regions that share any non-empty segment of border are called *neighboring*. Players *control* regions by having at least one meeple stationed there, and a region is called *vulnerable* if it borders at least one other region controlled by another player.

At the start of each turn, every region is controlled by exactly one player, and the current player's first action is to *migrate their meeple*. Each meeple may either stay where it is, or move from its current region to a neighboring one that the player also controls. The movements are performed one by one, in an order of the player's choice, but note that at all times, every region must contain at least one meeple (you cannot abandon regions) and a meeple can never migrate into an opponent-controlled region (taking new land is a separate phase of play).

A key strategic component of gameplay is defending the regions[3] you currently control from invading neighbors. Because of this, you want to move your meeples so that the *minimum number of meeple in any of your vulnerable regions is maximized*. In other words, you want to make your weakest vulnerable region as strong as possible (measured by the number of meeple there after migration).

Design and analyze an efficient algorithm that, given the state of the board at the beginning of your turn, determines the **maximum number of meeple in your weakest vulnerable region after migration**. You may assume that at the start of the turn, you control at least one region *and* that at least one region you control is vulnerable.

Hint: binary search may help reduce your runtime.

[2] In the 2D plane.
[3] Not necessarily connected.