DFS/BFS/Dijkstra
Problem Set 3 – CS6515 (Spring 2025)

- This problem set is due on **Thursday February 6th**.
- Submission is via Gradescope.
- Your solution must be a typed pdf (e.g., via LaTeX, Markdown, etc. Anything that allows you to type math notation) – no handwritten solutions.
- Please try to make your solutions as concise and readable as possible. Most problems will have solutions that are no more than a page long. Consider using bullet points and adding space to break up large paragraphs into smaller chunks.
- There are 3 problems. Each problem is graded with 20p. **There is +1p bonus per problem** for stating (i) how long it took you to solve that problem, and (ii) how long it took you to type the answer.

Remark on algorithm descriptions To make things easier for the TAs to grade, please use a combination of plain English and mathematical notation. Do not write code (C, Java, etc.). For example, you can say something like "$x := \max_{i=0,\ldots,n-1}$ `A[i]`" or "Find the maximum value in the array A" instead of writing a for-loop that computes the maximum.

7 One-Way Streets

We have a street network and want to convert all streets into one-way streets. When assigning directions to the streets, it's important that drivers will never get trapped, and can always drive from any point A to any other point B in the network.

Formally, we say a directed graph $G = (V, E)$ is "strongly connected" if for every pair $u, v \in V$ there exists a path from u to v.

Problem: Given an undirected connected graph $G = (V, E)$, we want to assign each edge a direction, such that the new graph is strongly connected. If that is not possible, then the algorithm should return "invalid."

1. Prove the following: When running DFS on an undirected graph, then there are no forward and no crossing edges. (Hint: Compare pre and post numbers and use them to argue that DFS should have used the edge in the opposite direction.)

2. Prove the following: We can assign valid directions if and only if: After running DFS, for every tree edge (u, v) there is a back-edge from some descendant of v to an ancestor of u. (Hint: (i) $\rightarrow$: If there is no backedge then... (ii) $\leftarrow$: Argue, if there is a back-edge then from v we can move "up" in the DFS tree. Why does that allow us to reach every other vertex?)

3. Give an algorithm that solves the problem, and analyze the time complexity. (No need to prove correctness, as it follows from point 2.)

For full points, the algorithm should run in $O(|E|)$ time.

8 Dijkstra With Delays

You are walking on a weighted directed graph $G = (V, E)$ with n vertices and m bridges. Each bridge $e \in E$ is assigned 3 numbers W_e, S_e, R_e. It takes W_e minutes to walk across bridge e, and the bridge is only open between time S_e to R_e every day ($0 \leq S_e < R_e \leq T, S_e + W_e \leq R_e$) where $T = 1440$ is the number of minutes in a day. You must exit the bridge before it closes. You are allowed to wait at a vertex for an adjacent bridge to open and you can also wait until the next day for the bridge to open again.

What is the minimum time needed to get from vertex 0 to $n - 1$?

Problem: Give an algorithm that solves the above problem and analyze its time complexity. Also give a brief explanation why the algorithm is correct. (You do not need to write a full proof. The explanation is only there to help the TAs understand your algorithm.)

For full points, the algorithm should be as fast as possible.

8.1 Programming Option

Instead of submitting a pdf for the question above, you can write a Python program.

Grading: Gradescope will automatically test your submission. You can resubmit as often as you want (until the due date) and you can see on gradescope how many test cases you currently passed. You will likely receive 16 (of 20) points for passing the first 11 test cases, and 20p for passing all test cases, but the final grading decision is made by a TA. (This is to make sure your submission is actually solving the problem described above. For example, hardcoding the output is obviously not a valid solution.) We need this rule because all test cases are public (on Piazza) to help you test/debug your program.

Input and Output: A template is provided on Piazza to help you get started. The template handles how to read the input and write the output, so you can just focus on implementing the algorithm. If you do not want to use those templates, we have a description on how to read the input and write the output below.

The program reads the input from standard input and prints its output to standard output. The input format is as follows: The first line provides two numbers n (number of nodes) and m (number of edges). The next m lines list 5 numbers separated by space: u v W_e S_e R_e, representing the directed edge (u, v) with positive integer weights W_e S_e R_e. Example:

```
5 5
0 1 720 240 1200
0 2 420 0 420
1 4 30 120 480
2 3 30 360 1320
3 4 240 0 240
```

Here we can go from vertex 0 to vertex 4 in 1590 minutes using the following path:
$0 \to 1$, we wait 240min until the bridge is open, then we spend 720 minutes to cross the bridge.
$1 \to 4$, first waiting 600min (wait $1440 - 960$ until midnight, then wait 120 until bridge opens, so 600min in total) because the bridge is closed and we must wait until the bridge opens again. Then spend 30min to cross the bridge.
Total time: 240+720+600+30 = 1590.

9 Minimum Fancy-Weight Spanning Tree

We are given a connected undirected weighted graph $G = (V, E)$ (with positive edge weights $w_e \in \mathbb{R}_{>0}$ for $e \in E$) and a vertex $r \in V$ to be used as root. We want to construct a minimum fancy-weight spanning tree, which is a spanning tree that minimizes the sum of the fancy-weights of the edges. The fancy-weight of an edge is not fixed and depends on the number of descendants in the tree. In particular, for a spanning tree T, the fancy-weight of an edge $\{u, v\} \in T$, where v is descendant of u, is

$$w_{uv} \cdot (\text{\# descendants of } v \text{ in tree } T)$$

When counting the number of descendants we include v. So the number of descendants of a leaf would be 1 and the number of descendants of the root would be $|V|$.

1. Give an algorithm that solves the above problem.
2. Give a brief explanation why the algorithm works.
3. Analyze your algorithm's time complexity.

For full points, the algorithm should be $O(|E| \log |E|)$. You may assume the graph is connected (ie, a spanning tree exists).

Hint: Consider the partial tree T that your algorithm constructed so far. If you add an edge e to it, by how much does the weight of the tree increase?