CS 3510 Section B: Design & Analysis of Algorithms September 12th 2023

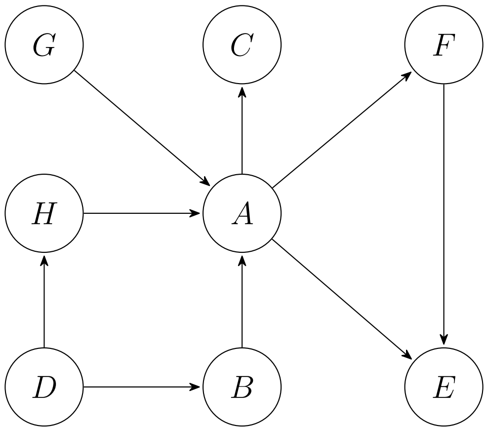
Problem Set 3: Graphs

YOUR NAME HERE Due: September 20th 2023

- Please type your solutions using LaTeX or any other software. Handwritten solutions will not be accepted.
- Your algorithms must be in plain English & mathematical expressions, and the pseudo-code is optional. Pseudo-code, without sufficient explanation, will receive no credit.
- Unless otherwise stated, all logarithms are to base two.
- If we ask for a specific running time, a correct solution achieving it will receive full credit even if a faster solution exists.

Problem 1: Topological Sort

(20 points)



Using the above graph G, answer the following questions.

(a) Processing nodes in alphabetical order, perform a Depth-First Search on G. Break ties alphabetically. Write the pre and post numbers of all vertices (starting from 1).

Solution:

G [1, 10],
A [2, 9],
C [3, 4],
E [5, 6],
F [7, 8],
B [11, 12],
D [13, 16],
H [14, 15]

(b) What are the strongly connected components of G?

Solution: Every Node is a separate strongly connected component, so if you are asking the scc of G, it would just be G.

(c) Give a topological ordering of G if one exists.

Solution: G,A,C,E,F,B,D,H

Problem 2: Multiple Monkeys

(20 points)

Given a set of monkeys $M \subseteq V$ and a set of bananas $B \subseteq V$ as vertices in an unweighted, undirected graph G. Find the shortest path in the graph (with respect to the number of edges) from any monkey to any banana in linear time. Formally, design an algorithm to find the shortest path in G such that it starts at some monkey $m \in M$ and finishes at any banana $b \in B$. Provide an explanation on correctness and runtime.

$$\min_{m \in M, b \in B} dist(m, b)$$

Note: not all vertices are labelled as monkeys or bananas!

Solution:

- *Algorithm: Initialize a Distance Matrix: Create a 2D array dist where dist[i][j] will hold the distance from node m to node b. The size of this matrix will be M $\times$ B. Run BFS through for every node of M, initialize a queue and a distance array to keep track of distances from u to all other nodes. Set the distance to the starting node u as 0 and enqueue it. While the queue is not empty, dequeue a node v. For each neighbor of v, if neighbor is not visited, update distance to be dist[u][v]+1 and enqueue the neighbor. Store the distance in the dist matrix. Then just find the shortest distance between any m and any b through the matrix.*
- *Correctness: BFS guarantees the shortest path in an unweighted graphs, and is in linear time. This algorithm just does BFS M amount of times and compares for the shortest distance.*
- *Time Complexity Analysis: Since BFS is $O(|V| + |E|)$, this algorithm would be $O(V * (|V| + |E|)$*

Problem 3: Truth Finder

(20 points)

In propositional logic, a 2-clause is an OR ($\vee$) of exactly two propositional variables or negated propositional variables. Some examples may include: $(p \vee q), (p \vee \neg q), (\neg p \vee q), (\neg p \vee \neg q)$. Suppose you are given a proposition which is the AND ($\wedge$) of many 2- clauses, such as

$$(x \vee y) \wedge (\neg a \vee \neg x) \wedge ... \wedge (\neg z \; orw)$$

Design an algorithm which takes as input a proposition of this form. Your algorithm should return a TRUE if there is an assignment to the propositional variables to make the proposition true and FALSE otherwise. You may suppose the input has n propositional variables and your runtime should be polynomial in n. Provide an explanation of runtime and correctness.

Example One:
Input: $(x \vee \neg y) \wedge (y \vee \neg z) \wedge (\neg z \vee \neg x)$
Output: TRUE
Explanation: The assignment $x = T, y = T, z = F$ makes the proposition true.

Example Two:
Input: $(\neg x \vee \neg y) \wedge (x \vee \neg y) \wedge (\neg x \vee y) \wedge (x \vee y)$
Explanation: There is no assignment of the propositional variables that makes the proposition true.

- (Hint 1): $\neg p \vee q \equiv p \implies q$
- (Hint 2): Such a formula has no solution if and only if there is any propositional variable p such that $p \iff \neg p$.

Solution:

- *Algorithm:*
  1. Construct the directed graph G with 2n nodes (one for each variable and one for its negation) and 2m edges(For each clause a OR b, there is an edge from $\neg a$ to b, and an edge from $\neg b$ to a.).
  2. Find the SCC of G using the algorithm given in class, which run in linear time.
  3. Check if any SCC contains both a literal and its negation. If so, G has no satisfying assignment (according to hint 2).
  4. If not, return true since all the formula will be guaranteed to be satisfied.
- *Correctness: The algorithm correctly functions by constructing an implication graph, where each clause becomes two implications. It then finds SCC in this graph. The key insight is that if a variable and its negation are in the same SCC, it leads to a contradiction, making the formula unsatisfied. Otherwise, a satisfying assignment can be constructed by assigning truth values to SCCs in topological order, ensuring all implications are respected.*
- *Time Complexity Analysis: Since the entire SCC algorithm is done in linear time,*

$O(V + E)$, $O(2(n + m))$

Problem 4: Navigating Atlanta

(20 points)

The City of Atlanta, Georgia includes the most extensive pedestrian skyway system in the world. It contains over 15 kilometers of heated pedestrian walkways to connect local landmarks and shield pedestrians from the city's harsh winter climate. Unfortunately, residents have started complaining that this system is difficult to navigate. You are a software engineer working on Hooli Maps, a navigational application, and have decided to tackle this problem by calculating directions.

Your Data Analysis team gives you a list of lists of city blocks (represented as integers from 1 to n) that each particular skyway covers. For example, your data analysis team may offer $[[1, 2, 3], [2, 4, 5]]$, indicating that skyway 1 covers blocks 1, 2, and 3, and skyway 2 covers blocks 2, 4, and 5. The walk between any two city blocks takes far too long and would expose pedestrians to hypothermia, but if multiple skyways both cover the same city block, then pedestrians can exit the system at this block and transfer to any of the other skyways covering it. The pedestrian will have to be outside for the duration of the transfer.

Because your mapping software works outside these skyways, you have an $\mathcal{O}(1)$ blackbox that tells you how much time it takes to transfer, given the indices of the two skyways in the transfer, and the integer corresponding to the block at which to complete that transfer. Note that the result will not change if the ordering of the skyways are changed.

Given two integer city blocks b_0, b_1, and a list of s skyways, each represented as an array of city block integers from 1 to n, design an efficient algorithm to calculate the minimum amount of time a pedestrian has to be outside to navigate between the two blocks. Provide an explanation on runtime and correctness.

Solution:

```
function min_transfer_time(b0, b1, skyways):
// 1. Build graph
graph = empty dictionary
for each skyway with index i in skyways:
    for each block in skyway:
        if block not in graph:
            graph[block] = empty list
        add i to graph[block]

// 2. Initialize for Dijkstra's
min_heap = priority queue with (0, b0)  // (time, block)
visited = empty set
time_spent_outside = empty dictionary

// 3. Dijkstra's algorithm
while min_heap is not empty:
    current_time, current_block = pop from min_heap

    if current_block is b1:
            return current_time

        if current_block is in visited:
            continue
        add current_block to visited

        for each skyway_index in graph[current_block]:
            for each next_block in skyways[skyway_index]:
                if next_block is not in visited:
                    new_time = current_time


                    if current_block is in time_spent_outside:
                        for each prev_skyway_index, transfer_time in time_spent_outsi
                            if prev_skyway_index is not skyway_index:
                                new_time = minimum of new_time and transfer_time

                    if next_block not in time_spent_outside:
                        time_spent_outside[next_block] = empty dictionary
                    time_spent_outside[next_block][skyway_index] = new_time + transfe

                    push (new_time, next_block) to min_heap

    return -1
```

- *Correctness: Dijkstra's algorithm guarantees finding the shortest path in terms of time spent outside. We carefully consider all possible transfers at each block and update the times accordingly, ensuring that we find the optimal path. The use of the visited set prevents revisiting blocks and ensures termination. If no path exists between b0 and b1, the algorithm returns -1*
- *Time Complexity Analysis: The runtime is dominated by Dijkstra's algorithm, which has a complexity of $O((|V| + |E|)log|V|)$, where $|V|$ is the number of blocks (nodes) and $|E|$ is the number of connections within skyways (edges). In the worst case, $|V|$ is 'n' (the total number of blocks) and $|E|$ is 'n * s' (each skyway can cover all blocks). Additionally, we have the overhead of building the graph, which takes O(n * s) time. Therefore, the overall runtime is O(n * s log n)*

Problem 5: Badminton

(20 points)

A badminton club in Ladhaland has students from several universities. You are given a list of pairs of player IDs, where each pair is made up of players from the same university. However, there is no information provided on what university each pair is from. What you do know is that each university has exactly one designated badminton captain and that their player ID is in at least one of the pairs. You don't know what player IDs are captains.

a) Design an algorithm to find the number of **non captain** players in the badminton club. You can assume that every player appears in the provided list of pairs of player IDs and that there are at least 2 players from each university in the city that are a part of the badminton club. Your algorithm should run in worst case O(E) complexity, where E is the number of given pairs. Provide an explanation of correctness and runtime.

Solution:

```
def find_non_captain_players(pairs):

graph = defaultdict(list)
for u, v in pairs:
    graph[u].append(v)
    graph[v].append(u)

visited = set()
non_captain_count = 0

def bfs(start):
    queue = deque([start])
    visited.add(start)
    component_size = 0

    while queue:
        node = queue.popleft()
        component_size += 1
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)

    return component_size

for player in graph:
    if player not in visited:
        component_size = bfs(player)
```

```
                non_captain_count += (component_size - 1)

        return non_captain_count
```

- *Correctness: Each player is a node, and each pair is an edge, correctly modeling the problem as a graph. Each connected component corresponds to a university, ensuring that we correctly group players by university. By subtracting one (the captain) from each component's size, we accurately count non-captain players.*
- *Time Complexity Analysis: Building the graph takes O(E) time, where E is the number of pairs.*
  Each node and edge is visited once during the BFS traversal, resulting in O(V + E) time complexity. Since V less than or equal to 2E, this simplifies to O(E), thus overall time complexity is O(E) .