INDIAN STATISTICAL INSTITUTE

MTech(CS) I year 2025-2026

Subject: Computing Laboratory

Practice Problem Set III

Q1. **Cycle detection in a graph using union-find.** You are given an undirected graph $G(V, E)$ in the following format. If $V = \{v_0, v_1, \ldots, v_k\}$, the input will consist of $k+1$ (the number of vertices), followed by a list of edges. Any edge $e = (v_i, v_j)$ will be specified as a pair of non-negative integers i, j. The number of edges will not be known to you in advance.

(a) Use the simplest Union-Find strategy (see Slide 12 of `union-find.pdf`, and Problem 1 of `practice-problem-set-2.pdf`) to determine whether G has a cycle. You should construct an array of size `num_vertices` (i.e., $k+1$), and read each edge (u, v) one by one. If `find(u) == find(v)`, report a cycle. Otherwise do `union(u,v)`. To the extent possible, reuse the existing `GRAPH` structure defined for you.

(b) Next, assume the graph is weighted (but undirected). Each edge will now be specified by **3 numbers:** 2 integers (as above), along with a floating point number, representing the weight of the edge.

(i) Define a structure type named `EDGE` consisting of the above 3 fields.

(ii) Add an `edge_list` field (and any other necessary fields) to the `GRAPH` structure. This field will expliclity keep track of the list of all edges in the graph. (This results in duplication of information, but we will ignore that for now.) Also add a `print_edge` field, analogous to the `print_vertex` field.

(iii) Write a function to print the edges, sorted in increasing order by weight, for a weighted graph.

(c) Use the idea in Q1.(b)iii (appropriately modified) to construct the adjacency list representation of an undirected graph from a list of all the edges. Also add the "inverse" function that constructs a list of all the **distinct** edges from the adjacency list.

At the end of this exercise, you should be able to very easily add the following functions (with hopefully self-explanatory names) to the provided implementation of graphs:

- `read_graph_as_adj_list()`
- `read_graph_as_edge_list()`
- `print_graph_as_adj_list()`
- `print_graph_as_edge_list()`
- `convert_graph_adj_list_to_edge_list()`
- `convert_graph_edge_list_to_adj_list()`

Q2. Add the `rank(key)` and `rangeCount(low, high)` operations on top of your BST (see Slides 51 and 57 of red-black-trees.pdf).

Q3. We are given a Trie with a set of strings stored in it. Write a program to suggest all possible completions of a prefix (string) typed by the user. If the user does not choose an existing completion, your program should add/store the new string to the Trie.

Q4. Given an array of `n` coin-denominations (all distinct positive integers), and a target value `sum`, write a program to find the minimum number of coins required to obtain `sum`. Assume that you have an infinite supply of coins of each denomination. If it is not possible to form the sum using the given coins, return -1.

Examples:

Input: `coins[] = {25, 10, 5}, sum = 30`
Output: `2`
Explanation: Minimum 2 coins needed, 25 and 5

Input: `coins[] = {9, 6, 5, 1}, sum = 19`
Output: `3`
Explanation: $19 = 9 + 9 + 1$

Input: `coins[] = {5, 1}, sum = 0`
Output: 0
Explanation: For 0 sum, we do not need a coin.

Input: `coins[] = {4, 6, 2}, sum = 5`
Output: `-1`
Explanation: Not possible to make the given sum.