CSE 6140 Assignment 1
due Sept. 18, 2017 at 6pm on T-Square

Please upload:
1) a PDF cover letter indicating with whom you worked (if applicable), the sources you used, and if you wish, your impressions about the assignment (what was fun, what was difficult, why...);
2) a PDF with your solutions of Problems 1, 2 and 3;
3) a PDF of your report for Problem 4;
4) a single zip file of your code, README, results for Problem 4.
Each file name should start by `<GTusername>_HW1`.

Please type your answers in LaTeX. You may handwrite them if you wish, but if we cannot read your handwriting, you will not receive points for your answer.

Note that MSTs (Problem 4) will be discussed in class on Sept. 5 (see course schedule on T-Square).

1 Algorithm design and complexity

The problem consists of finding the lowest floor of a building from which a box would break when dropping it. The building has n floors, numbered from 1 to n, and we have k boxes. There is only one way to know whether dropping a box from a given floor will break it or not. Go to that floor and throw a box from the window of the building. If the box does not break, it can be collected at the bottom of the building and reused.

The goal is to design an algorithm that returns the index of the lowest floor from which dropping a box will break it. The algorithm returns $n + 1$ if a box does not break when thrown from the n-th floor. The cost of the algorithm, to be kept minimal, is expressed as the number of boxes that are thrown (note that re-use is allowed).

1. For $k \geq \lceil \log(n) \rceil$, design an algorithm with $O(\log(n))$ boxes thrown.
2. For $k < \lceil \log(n) \rceil$, design an algorithm with $O\left(k + \frac{n}{2^{k-1}}\right)$ boxes thrown.
3. For $k = 2$, design an algorithm with $O(\sqrt{n})$ boxes thrown.

2 Greedy 1

A 30 foot long water pipe in Bob's garden has sprung leaks at multiple spots along its seam. As useful as it might be as an irrigation tool, Bob wants to plug the leaks with thin strips of sealant. Each such strip is 9 inches in length and shockingly expensive, so laying down strip after strip in succession along the seam is not a smart idea; and Bob is most definitely a smart man.

How can Bob employ the most efficient greedy algorithm, such that all n leaks are plugged with the minimum number of strips? Assume each leak to be tiny in comparison to the sealing strip length. Also state the time complexity of the algorithm and prove its correctness using the concepts of Greedy Choice and Optimal Substructure Properties.

3 Greedy 2

You are on a hike along the Appalachian Trail when you happen upon an abandoned mine. Against your better judgement, you venture inside and find a large deposit of various precious minerals. Luckily, you have a bag and a pickaxe with you, but you can't carry everything. Therefore, you must figure out how much of each mineral to take to maximize the value of your bag.

Formally: Given a bag with weight limit L, and a list of n minerals, where mineral i has value v_i and weight w_i, design a greedy algorithm to maximize the value of items you pack in your bag. (Note: v_i is the value of the entierty of item i, so if you decide you only want to take half of item i you only get value $\frac{v_i}{2}$). Prove your algorithm gives the optimal result using an **exchange argument**.

4 Programming Assignment

You are to implement *either* Prim's or Kruskal's algorithm for finding a Minimum Spanning Tree (MST) of an undirected graph, and evaluate its running time performance on a set of graph instances. The 13 input graphs are RMAT graphs [1], which are synthetic graphs with power-law degree distributions and small-world characteristics.

4.1 Static Computation

The first part of the assignment entails coding either Prim's or Kruskal's algorithm to find the cost of an MST given a graph file. You may use the programming language of your choice (either C++, Java, or Python). We provide a wrapper function in all three languages to help you get started with the assignment. You may call your own functions inside the wrapper. We also have implemented a timer in the wrapper that records the running time of your algorithms. To implement these algorithms, you may make use of data structure implementations in the programming language of your choice; e.g. in python, the heapq library may be used for implementing priority queues and set operations may be used for implementing the union-find data structure. In Java, java.util.PriorityQueue may be used for implementing priority queues and

java.util.Set may be used for set operations. Use of advanced data structures as opposed to trivial ones for implementation will carry bonus marks.

The 'graph file' format is as follows:

> Line 1: N E (N = number of vertices, E = number of edges)
>
> Every subsequent line contains three integers: u v weight (u = source of edge, v = destination of edge, weight = weight of edge between u and v)

The MST calculation is to be implemented in the `computeMST` function, as indicated in the wrapper code.

4.2 Dynamic Recomputation

The next part of the assignment requires you to update the cost of the MST given new edges to be added to the graph. You are provided with a 'changes file' and the format is as follows:

> Line 1: N (N = number of changes/edges to be added)
>
> Every subsequent line contains three integers: u v weight (u = source of new edge to be added, v = destination of edge, weight = weight of edge between u and v)

You are to implement the function `recomputeMST` as indicated in the wrapper code that computes the new MST given the new edge to be added into the graph. You are responsible for maintaining the old MST before recomputing.

Note: it is very easy to complete this part of the assignment by simply adding the new edge and calling your old `computeMST` function from part 1 (Subsection 4.1). The objective is to minimize computation and efficiently recompute the cost of the MST.

4.3 Execution

The wrapper code is set up to require three command line arguments: <executable> <graph_file.gr> <change_file.extra> <output_file>. The <graph_file> is the one described in Subsection 4.1 and the <change_file> is the one described in Subsection 4.2.

4.4 Experiments

You are required to run your code for all 13 input graphs provided. The wrapper functions we provide in C++, Java, and Python describe the following procedure. For each graph:

- parse the edges (`parseEdges`): **to be implemented**
- compute the MST using either Prim's or Kruskal's algorithm (`computeMST`): **to be implemented**
- write the cost of the initial MST and time taken to compute it to the output file: **provided in wrapper code**
- For each line in the <change_file>,

- Parse the new edge to be added: **provided in wrapper code**
- Call the function `recomputeMST`: **to be implemented**
- Write to the output file the cost of the new MST and the time taken to compute it: **provided in wrapper code**

Name the output files as such: <graph_file>_output.txt and place them in the folder *results*.

4.5 Report

Write a brief report wherein you:

a) List which data structures you have used for your choice of algorithm (Prim's/Kruskal's). Explain the reasoning behind your choice and how that has influenced the running time of your algorithms, and the theoretical complexity (i.e., specify the big-Oh for your implementation of each of the two algorithms - computeMST and recomputeMST).

b) Plots: Plot the running time as the number of edges in the graph increases (across the 13 graphs you were given) for both the static and dynamic calculations, i.e. one plot showing on the x-axis the number of edges the graph has and the y-axis the time for the static MST calculation, and another plot where the y-xis the time needed to insert 100 edges (as given the changes files) using your recomputeMST code. Discuss the results you observe. How does the empirical scaling you observe match your big-Oh analysis? How does the behavior vary with the dynamic recomputation?

4.6 Deliverables

- code for initial, static MST implementation
- code for dynamic MST recomputation
- README, explaining how to run your code
- Report (Subsection 4.5)
- output files (13) within the *results* folder- one for each graph

References

[1] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Proc. 4th SIAM Intl. Conf. on Data Mining,* Florida, USA, April 2004.