Week 6 Applied Sheet

Objectives: The applied sessions, in general, give practice in problem solving, in analysis of algorithms and data structures, and in mathematics and logic useful in the above.

Instructions to the class: You should actively participate in the class. The preparation problems are not assessed. We strongly recommend that everyone tries to solve the preparation problems before the applied session as you will benefit the most from the applied session if you come properly prepared. However, you can still attend the applied session if you have not solved those problems beforehand.

Instructions to Tutors: The purpose of the applied session is not to solve the practical exercises! The purpose is to check answers, and to discuss particular sticking points, not to simply make answers available.

Supplementary problems: The supplementary problems provide additional practice for you to complete after your applied session, or as pre-exam revision. Problems that are marked as **(Advanced)** difficulty are beyond the difficulty that you would be expected to complete in the exam, but are nonetheless useful practice problems as they will teach you skills and concepts that you can apply to other problems.

Implementation checklist

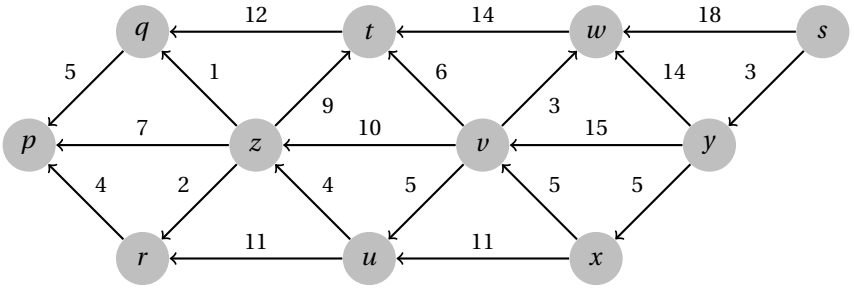
It will be most beneficial for your learning if you have completed this checklist **before** the tutorial.

By the end of week 6, write Python code for:

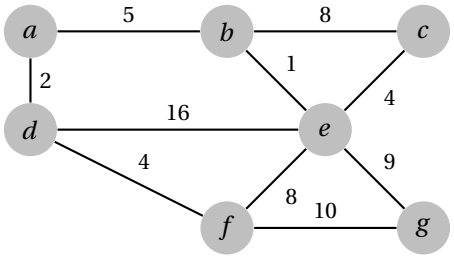
- Dijkstra's algorithm for single source shortest paths.
- Prim's algorithm for computing a minimum spanning tree.

Problems

Problem 1. **(Preparation)** Use Dijkstra's algorithm to determine the shortest paths from vertex s to all other vertices in this graph. You should clearly indicate the order in which the vertices are visited by the algorithm, the resulting distances, and the shortest path tree produced.



Problem 2. **(Preparation)** Show the steps taken by Prim's and Kruskal's algorithms for computing a minimum spanning tree of the following graph. Use vertex a as the root vertex for Prim's algorithm. Make sure that you indicate the order in which edges are selected, not just the final answer.



Problem 3. **(Preparation)** Consider the following state of a union-find data structure, using the union by size heuristic, as described in lectures.

ID	0	1	2	3	4	5	6	7	8
Parent	-1	2	-3	4	7	-1	7	-4	2

Determine the state of the array after each the following operations are executed (in order). If two trees are the same size, assume the first argument to `union()` will be chosen as the root:

1. `union(3, 6)`
2. `union(0, 5)`
3. `union(5, 3)`

Problem 4. Consider a buggy implementation of Dijkstra's algorithm in which the distance estimate to a vertex is only updated the first time the vertex is discovered, and not in any subsequent relaxations. Give a graph on which this bug will cause the algorithm to not produce the correct answer.

Problem 5. Consider the problem of planning a cross-country road trip. You have a map of Australia consisting of the locations of towns, each of which has a petrol station, with the corresponding petrol prices (which may be different at each town). You are currently at a particular town s and would like to travel to town t. Your car has a fuel capacity of C litres, and for each road on the map, you know the amount of petrol it will take to travel along it. Your tank can only contain non-negative integer amounts of petrol, and all roads cost an integer amount of petrol to travel along. You cannot travel along a road if you do not have enough petrol to make it all the way. You may refuel at any petrol station whether your tank is empty or not (but only to integer values), and you are not required to fill your tank. Assuming that your tank is initially empty, describe an algorithm for determining the cheapest way to travel to city t. [Hint: You should model the problem as a shortest path problem and use Dijkstra's algorithm.]

Problem 6. Consider a variant of the shortest path problem where instead of finding paths that minimise the total weight of all edges, we instead wish to minimise the weight of the largest edge appearing on the path. Let's refer to such a path as a *bottleneck path*.

(a) Give an example of a graph in which a shortest path between some pair of vertices is not a bottleneck (and where the graph also contain a bottleneck path which is not a shortest path).

(b) Prove that all of the paths in a minimum spanning tree M of a graph G are bottleneck paths in G.

(c) Prove that not all bottleneck paths of G are paths in a minimum spanning tree M of G.

Problem 7. Consider the following algorithm quite similar to Kruskal's algorithm for producing a minimum spanning tree

```
1: function MINIMUM_SPANNING_TREE(G = (V, E))
2:     sort(E, descending, key((u,v)) = w(u, v))    // Sort edges in order from heaviest to lightest
3:     T = E
4:     for each edge e in nonincreasing order do
```

```
5:          if T - {e} is connected then
6:              T = T - {e}
7:          end if
8:      end for
9:      return T
10: end function
```

Instead of adding edges in order of weight (lightest first) and keeping the graph acyclic, we remove edges in order of weight (heaviest first) while keeping the graph from becoming disconnected.

(a) Identify a useful invariant maintained by this algorithm.

(b) Prove that this invariant is maintained by the algorithm.

(c) Deduce that this algorithm correctly produces a minimum spanning tree.

Problem 8. Can Prim's and Kruskal's algorithms be applied on a graph with negative weights to compute a minimum spanning tree? Why or why not?

Problem 9. Recall from lectures that breadth-first search can be used to find single-source shortest paths on unweighted graphs, or equivalently, graphs where all edges have weight one. Consider the similar problem where instead of only having weight one, edges are now allowed to have weight zero or one. We call this the *zero-one* shortest path problem. Write pseudocode for an algorithm for solving this problem. Your solution should run in $O(V+E)$ time (this means that you cannot use Dijkstra's algorithm!) [Hint: Combine ideas from breadth-first search and Dijkstra's algorithm]

Problem 10. In the union-find data structure, when performing a union operation, we always append the set with fewer elements to the set with more elements. This heuristic is called union by size heuristic. Prove that in the worst case we can perform all union operations in $O(V \log(V))$ time when using union by size.

Supplementary Problems

Problem 11. You are the manager of a super computer and receive a set of requests for allocating time frames in that computer. The i-th request specifies the starting time s_i and the finishing time f_i. A subset of requests is compatible if there is no time overlap between any requests in that subset. Develop an algorithm to choose a compatible subset of maximal size (i.e., you want to accept as many requests as feasible, but you cannot select incompatible requests). [Hint: Develop a greedy algorithm.]

Problem 12. Implement Dijkstra's algorithm and test your code's correctness by solving the following Codeforces problem: `http://codeforces.com/problemset/problem/20/C`.

Problem 13. Implement Kruskal's algorithm.

Problem 14. Suppose that we wish to solve the single-source shortest path problem for graphs with nonnegative bounded edge weights, i.e. there is a constant c such that for all edges (u,v) it holds that $0 \leq w(u,v) \leq c$. Explain how we can modify Dijkstra's algorithm to run in $O(V+E)$ time in this case. [Hint: Improve the priority queue]

Problem 15. Consider the following supposed invariant for Kruskal's algorithm: At each iteration, each connected component in the current spanning forest is a minimum spanning tree of the vertices in that component.

(a) Prove or disprove whether Kruskal's algorithm maintains this invariant.

(b) Can this invariant be used to prove that Kruskal's algorithm is correct?