Week 8 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

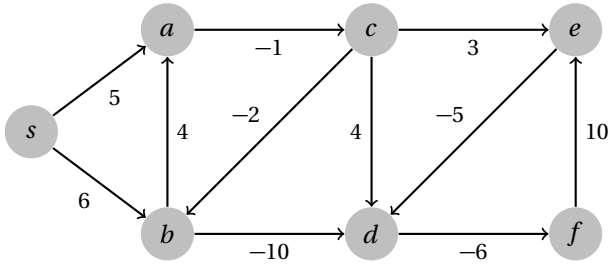
It will be most beneficial for your learning if you have completed this checklist **before** the tutorial.

By the end of week 8, write Python code for:

- Bellman-Ford single source shortest paths.
- Floyd-Warshall all-pairs shortest paths.

Problems

Problem 1. (Preparation) Use the space-efficient version of Bellman-Ford to determine the shortest paths from vertex s to all other vertices in this graph. Afterwards, indicate to which vertices s has a well defined shortest path, and which do not by indicating the distance as $-\infty$. Draw the resulting shortest path tree containing the vertices with well defined shortest paths. For consistency, you should relax the edges in the following order: $s \to a$, $s \to b$, $a \to c$, $b \to a$, $b \to d$, $c \to b$, $c \to d$, $c \to e$, $d \to f$, $e \to d$ and $f \to e$.



Problem 2. Consider the following algorithm for single-source shortest paths on a graph with negative weights:

- Find the minimum weight edge in the graph, say it has weight w
- Subtract w from the weight of every edge in the graph. The graph now has no negative weights
- Run Dijkstra's algorithm on the modified graph
- Add w back to the weight of the edges and compute the lengths of the resulting shortest paths

Prove by giving a counterexample that this algorithm is incorrect.

Problem 3. Describe an algorithm that given a graph G determines whether or not G contains a negative cycle. Your algorithm should run in $O(VE)$ time.

Problem 4. Given a graph that contains a negative weight cycle, give an algorithm to determine the vertices of one such cycle. Your algorithm should run in $O(VE)$ time.

Problem 5. Improve the Floyd-Warshall algorithm so that you can also reconstruct the shortest paths in addition to the distances. Your improvement should not worsen the time complexity of Floyd-Warshall, and you should be able to reconstruct a path of length k in $O(k)$ time.

Problem 6. Add a post-processing step to Floyd-Warshall that sets $\text{dist}[u][v] = -\infty$ if there is an arbitrarily short path between vertex u and vertex v, i.e. if u can travel to v via a negative weight cycle. Your post processing should run in $O(V^3)$ time or better, i.e. it should not worsen the complexity of the overall algorithm.

Problem 7. Arbitrage is the process of exploiting conversion rates between commodities to make a profit. Arbitrage may occur over many steps. For example, one could purchase US dollars, convert it into Great British Pounds, and then back into Australian dollars. If the prices were right, this could result in a profit. Given a list of currencies and the best conversion rate between each pair (i.e., how much of the first currency you need to spend to get 1 unit of the second currency), devise an algorithm that determines whether arbitrage is possible, i.e. whether or not you could make a profit. Your algorithm should run in $O(n^3)$ where n is the number of currencies.

Supplementary Problems

Problem 8. Implement Bellman-Ford and test your code's correctness by solving the following problem on UVA Online Judge: `https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=499`.

Problem 9. Devise an algorithm for counting the number of paths between two given vertices s and t in a directed acyclic graph which runs in $O(V+E)$. Don't worry about the magnitude of the answer (i.e. assume that all arithmetic operations take constant time, despite the fact that the answer might be exponential in the input size.)

Problem 10. Consider a directed acyclic graph G where each edge is labelled with a character from some finite alphabet A. Given a string S over the alphabet A, count the number of paths in G whose edge labels spell out the string S. Your algorithm should run in $O((V+E)n)$, where n is the length of the string S.

Problem 11. (Advanced) An improvement to the Bellman-Ford algorithm is the so-called "Shortest Paths Faster Algorithm", or SPFA. SPFA works like Bellman-Ford, relaxing edges until no more improvements are made, but instead of relaxing every edge $|V|-1$ times, we maintain a queue of vertices that have been relaxed. At each iteration, we take an item from the queue, relax each of its outgoing edges, and add any vertices whose distances changed to the queue if they are not in it already. The queue initially contains the source vertex s.

(a) Reason why SPFA is correct if the graph contains no negative-weight cycles.

(b) The algorithm described above will cycle forever in the presence of a negative-weight cycle. Add a simple condition to the algorithm to fix this and detect the presence of negative cycle if encountered.

(c) What is the worst-case time complexity of SPFA?

Problem 12. (Advanced) In this problem, we will derive a fast algorithm for the all-pairs shortest path problem on sparse graphs with negative weights[1]. We will assume for simplicity that the graph has no negative cycles, but

[1] This algorithm is known as Johnson's algorithm.

the techniques here can easily be adapted to handle them. The key ingredient of the algorithm is based on the idea of *potentials*.

(a) Let $d[v] : v \in V$ be the distances to each vertex $v \in V$ from an arbitrary source vertex. Prove that for each edge $(u, v) \in E$

$$d[u] - d[v] + w(u, v) \geq 0.$$

We define the quantity $p[u] = d[u]$ as the *potential* of the vertex u.

(b) Give a simple algorithm to compute a set of valid potentials. Potentials should not be infinity for any vertex.

Suppose now that we modify the weight of every edge (u, v) in the graph to

$$w'(u, v) = p[u] - p[v] + w(u, v)$$

(c) Prove that a shortest path in the modified graph was also a shortest path in the original, unmodified graph.

(d) Deduce that we can solve the all-pairs shortest path problem on sparse graphs with negative weights in $O(V^2 \log(V))$ time.

Problem 13. (Advanced) Describe how an instance of the unbounded knapsack problem can be converted into a corresponding directed acyclic graph. Which graph problem correctly models the unbounded knapsack problem on this graph?

Problem 14. (Advanced) A problem closely related to the transitive closure problem mentioned in lectures is the *transitive reduction*. In a sense, the transitive reduction is the opposite of the transitive closure. It is a directed graph with the fewest possible edges that has the same reachability as the original graph. In other words, for all pairs of vertices u and v, there is a path between u and v in the transitive reduction if and only if there is a path between u and v in the original graph. Give an algorithm for computing the transitive reduction of a directed acyclic graph. Your algorithm should run in $O(V^2 + VE)$ time.