Algorithms

Example Sheet 4

Petar Veličković

Lent Term 2017

Warm-up

1. Propose an $O(V)$ algorithm to determine whether an undirected graph $G = (V, E)$ contains a cycle.

2. Compare and contrast the various shortest-path algorithms covered in the course, highlighting when you would use each one:
   - Dijkstra's algorithm;
   - Bellman-Ford algorithm;
   - Johnson's algorithm;
   - Matrix multiplication-based algorithm.

3. Consider the problem of finding shortest paths in a directed, weighted graph with negative edges, such that $-W$ is the smallest negative edge. Prove that adding W to all edge weights is not a sensible thing to do.

4. In the context of matrix multiplication-based shortest path algorithm, what are the entries of $\mathbf{M}^{(0)}$? Provide an informal justification for your decision (both from an intuitive and a practical standpoint).

5. We want to compute the minimum spanning tree (MST) of a graph with V vertices and E edges. Which algorithm would you choose in each of the following scenarios and why?
   (a) $E = \Theta(V^{1.5})$;
   (b) $E = \Theta(V \log^2 V)$, and you have been provided with a list of edges sorted by their weights;
   (c) $E = \Theta(V)$, and the edges are integers in the set $\{0, 1, \ldots, \lfloor \log V \rfloor\}$.

   You may assume that you have a Fibonacci heap and an optimised disjoint-set data structure at your disposal.

Exercises

1. Propose an algorithm to find the longest path (without repeated vertices) in an unweighted tree.

2. Given a directed acyclic graph (DAG) $G = (V, E)$, and two vertices $s, t \in V$, provide an algorithm for computing the number of *simple paths* (paths without repeated vertices) from s to t. Analyse its complexity.

3. Solve *Exercise 6* from the pre-sheet. [*Hint:* The desirable complexity is $O((N + M + K) \log M)$.]

4. Consider the task of finding single-source shortest path distances within a weighted, directed graph, from a source vertex s. Also, assume that all of the edges in this graph are nonnegative, except potentially outbound edges from s. Will Dijkstra's algorithm suffice for this task? Provide a proof of correctness if so, or a counterexample if not.

5. Considering Johnson's algorithm, do we really need to introduce a new vertex? Either prove that it is appropriate to choose a vertex from the original graph, or provide a counterexample.

6. Consider a weighted undirected graph $G = (V, E)$, and let T be its MST. Let $C \subset V$ be a cut of the graph, and $e \in E$ be the lightest edge crossing the cut. Assume $e \notin T$. Let $f \in E$ be an edge crossing the cut such that $f \in T$ (one must exist—why?). Is $T \cup \{e\} \setminus \{f\}$ a tree? Either provide a proof or a counterexample.

Implementation

As exemplified by topological sorting (and several other problems on this sheet), the humble *depth-first search* (DFS) can be a surprisingly powerful tool in one's algorithmic arsenal, allowing one to solve a wide variety of seemingly more complex graph problems. In this week's implementation exercise, you will be applying DFS to arrive at a solution for the very important task of detecting *strongly-connected components* (SCCs) of a given directed graph $G = (V, E)$.

A strongly-connected component of a graph is a subset of its vertices, $S \subseteq V$, such that, for all pairs of vertices $x, y \in S$, there exists a path from x to y as well as a path from y to x[1].

Here you will be implementing *Kosaraju's algorithm*[2] for SCC detection, which

[1]You may note that this, in a way, extends the notion of *cycles* from undirected graphs to directed ones. Indeed, an SCC implies the existence of a cycle that spans all of its vertices, but there may be *many* such cycles within a single SCC.

[2]`https://en.wikipedia.org/wiki/Kosaraju's_algorithm`

leverages two DFS passes to obtain the optimal[3] $O(V + E)$ time complexity. The main idea is: *if we were to reverse the direction of all edges in the graph, it would have exactly the same SCCs!* (Can you prove this?). The algorithm proceeds as follows:

1. Perform a depth-first search of G, creating a list L of vertices ordered descendingly by finishing time.
2. Perform a second depth-first search on the graph obtained by reversing all the edges in G. At each step, we explore starting from the first element in L not to have been visited so far. Let this vertex be v. All vertices reachable from v, and not visited by a previous step, will be in v's SCC.

To informally prove that this algorithm will always place all vertices that share an SCC in the same component:

- Consider two vertices, u and v, located in the same SCC.
- Regardless of the order in which we do our DFS steps, these two vertices will always be visited in the same step (why?).
- Therefore, no matter how we perform our DFS passes, if we put all visited nodes at one step into the same component, two vertices sharing an SCC will necessarily be in the same component.

Conversely, to informally prove that this algorithm will *not* place two vertices that do not share an SCC in the same component:

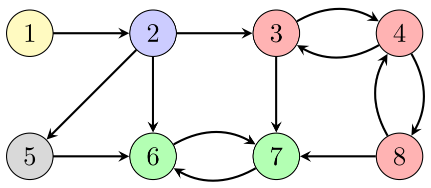
- Consider two vertices, u and v, located in separate SCCs. Assume without loss of generality that there exists a path from u to v in G[4].
- It must hold that u occurs before v in L (why?).
- This means that u (or any vertex in u's SCC) will be considered before v as a candidate to start expanding from in the opposite direction. However, as there cannot be a path from v to u in G, there cannot be a path from u to v in reversed G—therefore, this DFS pass will *not* include v.

Your implementation objective for today is to implement Kosaraju's algorithm, and then leverage it to determine *the minimal number of edges* one needs to add to the graph to make it strongly connected (i.e. that all vertices share the same SCC). Proper choice of data structures for G is critical!

In order to verify your implementation, please run it on the following example (wherein the SCCs have been highlighted, and only *one* edge needs to be added to make the graph strongly connected—e.g. one such edge is $6 \to 1$):

[3] There exists a better-optimised algorithm, *Tarjan's SCC algorithm* (which performs *only one DFS pass*), but it is far less intuitive, and offers no asymptotic gains.

[4] If the nodes are disconnected, then they can never be visited in the same DFS step, so this case is trivially irrelevant.



Job interviews

1. How would you compute the number of *triangles* (cycles of length 3) in a given undirected graph? What is its complexity? Provide the tightest bound possible, potentially inventing some new variables that describe the graph's topology.

2. What is the smallest number of edges you need to add to a given directed graph $G = (V, E)$ such that it is possible to make a cycle which uses each edge *exactly once* (*Eulerian cycle*)?