Solutions to Homework Practice Problems

[DPV] 3.3 Topological Ordering Example

Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.

(a) Indicate the pre- and post-numbers of the nodes.

Running DFS gives the following pre- and post-numbers:

Node	A	B	C	D	E	F	G	H
pre	1	15	2	3	11	4	5	7
post	14	16	13	10	12	9	6	8

(b) What are the sources and sinks of the graph?

The graph has two sources (A and B) and two sinks (G and H).

(c) What topological ordering is found by the algorithm?

Th topological ordering of the graph is found by reading the post-numbers in decreasing order: B, A, C, E, D, F, H, G.

(d) How many topological orderings does this graph have?

Any topological ordering of the graph will be of the form $[AB]C[DE]F[GH]$, with the ordering of the pairs in brackets arbitrary (for example, $ABCEDFHG$ is valid). Each bracketed pair can be organized in 2 different ways, so there are $2 \cdot 2 \cdot 2 = 8$ different topological orderings for this graph.

[DPV] 3.4 SCC Algorithm Example

Run the strongly connected components algorithm on the following directed graphs G. When doing DFS on G^R: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.

(a) In what order are the strongly connected components (SCCs) found?

(i)
The SCCs are found in the following order:

$$\{C, D, F, J\}, \{G, H, I\}, \{A\}, \{E\}, \{B\}$$

(ii)
The SCCs are found in the following order:

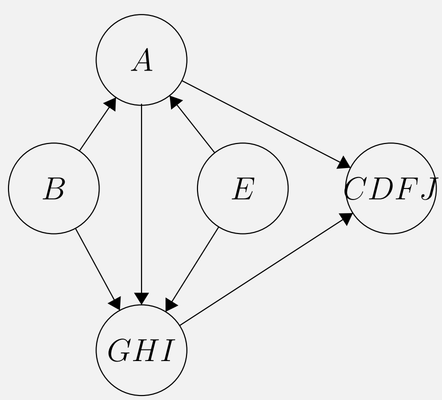
$$\{D, F, G, H, I\}, \{C\}, \{A, B, E\}$$

(b) Which are source SCCs and which are sink SCCs?

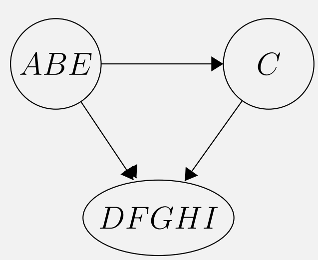
(i)
The source SCCs are $\{E\}$ and $\{B\}$. The sink SCC is $\{C, D, F, J\}$.
(ii)
The source SCC is $\{A, B, E\}$. The sink SCC is $\{D, F, G, H, I\}$.

(c) Draw the "metagraph" (each meta-node is an SCC of G).

(i)



(ii)



(d) What is the minimum number of edges you must add to this graph to make it strongly connected?

(i)
Two edges must be added to make the entire graph strongly connected: one from any vertex in $\{C, D, F, J\}$ to B, and one from any vertex in $\{C, D, F, J\}$ to E.
(ii)
One edge must be added to make the entire graph strongly connected: from any vertex in $\{D, F, G, H, I\}$ to any vertex in $\{A, B, E\}$.

[DPV] 3.5 Reverse of Graph

The reverse of a directed graph $G = (V, E)$ is another directed graph $G^R = (V, E^R)$ on the same vertex set, but with all edges reversed; that is, $E^R = \{(v, u) : (u, v) \in E\}$. Give a linear-time algorithm for computing the reverse of a graph in adjacency list format.

(a) Algorithm

First, initialize an empty adjacency list formatted graph for V vertices. Then, for each $u \in V$, go through the adjacency list of neighbors of u. For each neighbor v of u in G, add u as a neighbor of v in the new adjacency list for G^R.

(b) Correctness

The reverse of a graph is one where the vertices are the same, but the direction of the edges are flipped. By adding edge $\overrightarrow{(v, u)}$ for each edge $\overrightarrow{(u, v)}$, we accomplish the objective.

(c) Runtime Analysis

Creating the new adjacency list takes $O(n)$ or $O(|V|)$ time. Adding a single edge is a constant time $O(1)$ operation; we traverse the original graph and add all m edges, which takes $O(n + m)$ time. Our overall run time is $O(n) + O(n + m) = O(n + m)$ time.

[DPV] Problem 3.15 Computopia

The police department in the city of Computopia has made all streets one-way. The mayor contends that there is still a way to drive legally from any intersection in the city to any other intersection, but the opposition is not convinced. A computer program is needed to determine whether the mayor is right. However, the city elections are coming up soon, and there is just enough time to run a *linear-time* algorithm.

Part (a): Formulate this problem graph-theoretically, and explain why it can indeed be solved in linear time.

(a) Algorithm

We will represent the city in this problem as a directed graph $G = (V; E)$. The vertices in V represent the intersections in the city, and the directed edges in E represent the one-way streets of the city between intersections. Then, the problem is to determine whether a path from u to v exists for all $u, v \in V$, and to do so in linear time.

We can solve this problem by running the SCC algorithm and checking if there is a single SCC. If the entire graph G is itself a single strongly connected component, then we report the mayor's claim is true. If there is more than one SCC, we report that the mayor's clam is false.

(b) Correctness

Why does this work? In a SCC, there is a path from every vertex to every other vertex in the same SCC. If the graph has a single SCC then every intersection has a route to every other intersection. If there is more than one SCC, at least one is a source and one is a sink, and the vertices (intersections) of the sink SCC will not have a path to the vertices (intersections) of the source SCC.

(c) Run Time

Creating the graph representation of the city takes $O(n + m)$ time to model the n intersections and m roads. The SCC algorithm takes linear time $O(n + m)$ for its two runs of DFS, as required.

Part (b): Suppose it now turns out that the mayor's original claim is false. She next claims something weaker: if you start driving from town hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town hall. Formulate this weaker property as a graph-theoretic problem, and carefully show how it too can be checked in linear time.

(a) Algorithm

The algorithm requires computing the SCCs (again, in $O(n+m)$ time) and checking if there are any edges out of the SCC containing the town hall – that is, is the town hall in a *sink SCC*. We can do this by examining either the DAG of the meta-graph of the SCC or each vertex which lies in the same SCC S as the town hall to see if they have any outgoing edges to vertices not in SCC S.

(b) Justification

The weaker claim requires that the town hall resides in a sink SCC. Why? If it lies in a sink SCC S then from the town hall we can reach every other intersection in S and from every other intersection in S we can get to the town hall. And, if S is not a sink SCC then there are edges out of it, and therefore there are intersections that can be reached from the town hall but cannot get back to the town hall.

(c) Run Time

We saw in part (a) that it takes linear $O(n + m)$ time to find the SCCs of this graph. The examination of the SCC where the town hall resides also takes $O(n + m)$ time, so the total running time of this algorithm is $O(n + m)$, which is linear time.

[DPV] 4.14 shortest path through a given vertex

> You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $v_0 \in V$. Give an efficient algorithm for finding shortest paths between *all pairs of nodes*, with the one restriction that these paths must all pass through v_0.

(a) Algorithm

Run Dijkstra's algorithm from v_0 to get all distances from v_0 to all vertices in V. Reverse the graph and run Dijkstra's again from v_0 - the output corresponds to the distances of the shortest path from all other vertices to v_0 in the original graph. For any pair of vertices $u, w \in V$, the shortest distance from u to w using a path through v_0 will be the sum of the two distances, u to v_0 in the reverse graph and v_0 to w in the original graph. Any single path $u \rightsquigarrow v_0 \rightsquigarrow w$ may be recovered using the $prev[]$ arrays which result from each run of Dijkstra.

(b) Justification

Why this works: our graph is strongly connected, meaning that a path exists between every pair of vertices. We also know that Dijkstra's, given a starting vertex, will find the shortest path from that starting point to every other vertex. When we reverse a directed graph we are essentially reversing the direction of the path between any pair of vertices. So the shortest path from u to w which includes v_0 is the combination of the shortest path from u to v_0 in the reversed graph plus the shortest path from v_0 to w in the original graph.

(c) Run Time

Each round of Dijkstra's takes $O((m+n)\log(n))$ - this runtime can be simplified to $O(m \log n)$ since the graph is strongly connected. Building the reverse graph takes $O(n + m)$ time. Explicitly calculating the pairwise shortest distances between each pair of vertices would take $O(n^2)$ time, resulting in an overall runtime of $O(n^2 + (m+n)log(n))$, or $O(n^2 + m \log n)$ (if simplified).

[DPV] Problems 5.1, 5.2 (Practice fundamentals of MST designs)

5.1

(a) the cost is 19
(b) there are 2 possible MSTs
(c)

Edge included	Cut
AE	{A} & {B,C,D,E,F,G,H}
EF	{A,E} & {B,C,D,F,G,H}
BE	{A,E,F} & {B,C,D,G,H}
FG	{A,B,E,F} & {C,D,G,H}
GH	{A,B,E,F,G} & {C,D,H}
CG	{A,B,E,F,G,H} & {C,D}
GD	{A,B,C,E,F,G,H} & {D}

5.2 (a)

Vertex included	Edge included	Cost
A		0
B	AB	1
C	BC	3
G	CG	5
D	GD	6
F	GF	7
H	GH	8
E	AE	12

5.2 (b)

Here are the values for the parent pointer π at each iteration of Kruskals. From this you should be able to deduce the disjoint-sets.

Union	Values of π for each vertex
Start	[A, B, C, D, E, F, G, H]
(A,B)	[B, B, C, D, E, F, G, H]
(F,G)	[B, B, C, D, E, G, G, H]
(D,G)	[B, B, C, G, E, G, G, H]
(G,H)	[B, B, C, G, E, G, G, G]
(C,G)	[B, B, G, G, E, G, G, G]
(B,C)	[B, G, G, G, E, G, G, G]
(A,E)	[G, G, G, G, G, G, G, G]

[DPV] Problem 5.9

(a) **False**. Consider a graph where a vertex is adjacent to a single edge
(b) **True**. Consider the order in which edges would be processed by Kruskal's
(c) **True**. A minimum weight edge would be a candidate for at least one possible MST
(d) **True**. The *Cut Property* assures this