Algorithms Example Sheet 5: Problems

Further DAG problems

Exercise 5.P.1 If we take a DAG and reverse all the edges, do we get a DAG? Justify your answer.

[Exercise 11 in Lecturer's handout]

Exercise 5.P.2 Here are two buggy ways to code topological sort. For each, give an example to show why it's buggy.

(a) Pick some vertex s with no incoming edges. Simply run `dfs_recurse`, starting at this node, and add an extra line `totalorder.prepend(v)` as we did in `toposort`.

(b) Run `dfs_recurse_all`, but order nodes in order of when they are visited, i.e. insert `totalorder.append(v)` immediately after the line that sets `v.visited=True`.

[Exercise 12 in Lecturer's handout]

Exercise 5.P.3 Given a DAG G, design an algorithm to determine if there is a path that includes each vertex exactly once.

Exercise 5.P.4 Show that every DAG G has at least one vertex with no incoming edges and at least one vertex with no outgoing edges.

Exercise 5.P.5 Give pseudocode for an algorithm that takes as input an arbitrary directed graph g, and returns a boolean indicating whether or not g is a DAG.

[Exercise 13 in Lecturer's handout]

Exercise 5.P.6 Give an example DAG with 9 vertices and 9 edges. Pick some vertex that has one or more edges coming in, and run through `toposort` starting on line 6 with this vertex. Mark each vertex with two numbers as you proceed: the discovery time (when the vertex is coloured grey) and the exit time (when the vertex is coloured black). Then draw a linearized DAG by arranging the vertices on a line in order of their finishing time, and reproducing the appropriate arrows between them. Do all the arrows go in the same direction?

[Exercise 17 in Lecturer's handout]

Exercise 5.P.7 The code for `toposort` is based on `dfs_recurse`. If we base it instead on the stack-based dfs from Section 5.2, and insert the line `totalorder.prepend(v)` on line 13 (after the iteration over v's neighbours), would we obtain a total order? If so, justify your answer. If not, give a counterexample, and pseudocode for a proper stack-based `toposort`.

[Exercise 20 in Lecturer's handout]

Exercise 5.P.8 [Approximate DAGs] (optional) Sometimes we want to impose a total order on a collection of objects, given a set of pairwise comparisons that can be thought of as a "DAG with noise'. For example, let vertices represent movies, and write $v_1 \to v_2$ to mean "The user has said she prefers v_1 to v_2." A user is likely to give answers that are by and large consistent, but with some exceptions. Discuss what properties you would like in an "approximate total order", and how you might go about finding it. *[This is an open-ended question, and a prelude to data science and machine learning courses.]*

[Exercise 18 in Lecturer's handout]

Exercise 5.P.9 Write out a formal proof of the correctness of the `toposort` algorithm, filling out all the details that are skipped over in the handout. Pay particular attention to the third case, "v_2 is coloured grey", where it is claimed "The call stack corresponds to a path in the graph from v_2 to v_1."

[Exercise 19 in Lecturer's handout]

Minimum Spanning Trees

Exercise 5.P.10 An engineer friend tells you "Prim's algorithm is based on Dijkstra's algorithm, which requires edge weights to be ≥ 0. If some edge weights are < 0, we should first add some constant weight c to each edge so that all weights are ≥ 0, then run Prim's algorithm."

(a) Your friend's algorithm will produce a MST for the modified graph. Is this an MST for the original graph?

(b) What would happen if you run Prim's algorithm on a graph where some weights are negative? Justify your answer.

[Exercise 9 in Lecturer's handout]

Exercise 5.P.11 [MST with updates]

(a) Attempt **[2015P1Q9 (c)]**.

(b) Design an algorithm to find the second-best MST (if it exists), i.e. the tree T with the smallest weight $w(T)$ such that $w(T) >$ **MST**.

Maximum flow

Exercise 5.P.12 We are given a directed graph, and a source vertex and a sink vertex. Each edge has a capacity $c_E(u \to v) \geq 0$, and each vertex (excluding the source and the sink) also has a capacity $c_V(v) \geq 0$. In addition to the usual flow constraints, we require that the total flow through a vertex be $\leq$ its capacity. We wish to find a maximum flow from source to sink. Explain how to translate this problem into a max-flow problem of the sort we studied in section 6.2.

Exercise 5.P.13 The Russian mathematician A.N. Tolstoy introduced the following problem in 1930. Consider a directed graph with edge capacities, representing the rail network. There are three types of vertex: supplies, demands, and ordinary interconnection points. There is a single type of cargo we wish to carry. Each demand vertex v has a requirement $d_v > 0$. Each supply vertex v has a maximum amount it can produce $s_v > 0$. Tolstoy asked: can the demands be met, given the supplies and graph and capacities, and if so then what flow will achieve this?
Explain how to translate Tolstoy's problem into a max-flow problem of the sort we studied in section 6.2.

[Exercise 5 in Lecturer's handout]

Exercise 5.P.14 In the context of Exercise 13, a dispute has arisen in the central planning committee. Comrade A who oversees the factories insists that each demand vertex must receive precisely d_v, no more and no less. Comrade B who oversees the trains insists that each demand vertex v must be prepared to receive a surplus flow, more than d_v, so as not to constrain the flows on the train system any more than necessary. Does your solution satisfy Comrade A or Comrade B? How would you satisfy the other?

[Exercise 7 in Lecturer's handout]

Exercise 5.P.15 Devise an algorithm that takes as input a flow f on a network, and produces as output a decomposition $[(\lambda_1, p_1), \ldots, (\lambda_n, p_n)]$ where each p_i is a path from the source to the sink, and each λ_i is a positive number. The decomposition must satisfy $f = \sum_i \lambda_i p_i$, by which we mean "put flow λ_i along path p_i, and add together all these flows-along-paths, and the answer must be equal to f". Explain why your algorithm works.

Exercise 5.P.16 [Edge-disjoint paths] Two paths are edge disjoint if they do not share an edge. Given a directed graph G find the maximum number of edge disjoint paths from s to t.

Exercise 5.P.17 [Baseball elimination problem] You are given the points w_x that each team x has in the league. There are $g_{xy} = g_{yx}$ remaining games between teams x and y. You would like to determine if there is a possible outcome so that team z finishes first (or tied first).
For example, if there are 4 teams and the league table is as follows:

Team	Points
x	12
y	12
z	10
w	7

and the remaining games are $g_{xy} = 3$, $g_{zw} = 3$ and $g_{wx} = g_{wy} = 1$. Team z can reach at most 13 points by winning all games, but either team x or team y will win at least two games (from those that they play against each other), so they will reach 14 points. Hence, it is not possible for z to finish first.

Exercise 5.P.18 In the London tube system (including DLR and Overground), there are occasional signal failures that prevent travel in either direction between a pair of adjacent stations. We would like to know the minimum number of such failures that will prevent travel between Kings Cross and Embankment.

(a) Explain how the tube map g can be translated into a suitable flow network g', with Kings Cross the source and Embankment the sink, such that a set of signal failures preventing travel in g is translated into a cut in g'. [*Hint:* Remember that a cut is a partition of the vertices, not an arbitrary selection of edges.]

(b) Explain how a cut in g' can be translated into a set of travel-preventing signal failures in g, such that the number of signal failures is $\leq$ the capacity of the cut.

(c) Suppose we take the minimum cut in g' and translate it into a set of travel-preventing signal failures in g. Show that this is the minimum number of travel-preventing signal failures.

(d) Find a maximum flow on your directed graph. Hence state the minimum number of signal failures that will prevent travel. [*Hint:* You should run Ford-Fulkerson by hand, with sensibly-chosen augmenting paths.]

[Exercise 6 in Lecturer's handout]

Exercise 5.P.19 [Hall's Theorem] Consider a bipartite graph, in which edges go between the left vertex set L and the right vertex set R. A matching is called *complete* if every vertex in L is matched to a vertex in R, and vice versa. For a complete matching to exist, we obviously need $|L| = |R|$. The following result is known as Hall's Theorem:
A complete matching exists if and only if, for every subset $X \subseteq L$, the set of vertices in R connected to a vertex in X is at least as big as X.
Prove Hall's Theorem, using a max-flow formulation. *[Hint: Use the same construction as we used in lectures, except with capacity ∞ on the edges between L and R. In this graph, some cuts have infinite capacity, and some cuts have finite capacity. If a cut has finite capacity, what can you deduce about its capacity?]*