Algorithms Example Sheet 5: Problems Solution Notes

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]

Let G be the DAG and G^R the graph with reversed edges. Assume G^R was not a DAG. So, there is some cycle $v_1 \to v_2 \to \ldots \to v_k \to v_1$. But then the cycle $v_1 \to v_k \to v_{k-1} \to \ldots \to v_2 \to v_1$ would be a cycle in G (contradiction). So G^R is acyclic (and of course directed).

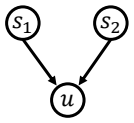
Exercise 5.P.2 Here are two buggy ways to code topological sort. For each, give an example to show why it's buggy.

(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]

(a) The problem is that there could be multiple vertices with no incoming edges. For example, consider the following graph:



Running a single DFS cannot explore (and so cannot add to the output) both s_1 and s_2, so the returned ordering will be wrong.

(b) Again, consider the graph above starting the DFS at s_1 and then from s_2. Then the returned order will be $[s_1, u, s_2]$ which is not a total order.

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

The following algorithm can be applied:

1. Find a topological ordering of the vertices.
2. Check if there is an edge between adjacent vertices.

We are searching for an ordering (i.e. a permutation) r of the vertices such that the edges $v_{r(i)} \to v_{r(i+1)}$ exist for every $i \in [n]$. Consider the topological ordering $t(\cdot)$, then $t(v_{r(i)}) < t(v_{r(i+1)})$ (because the edge $v_{r(i)} \to v_{r(i+1)}$ exists), so $t(v_{r(1)}) < \ldots < t(v_{r(n)})$. Hence, there will be such a path iff there is a edge between adjacent vertices in the topological ordering.

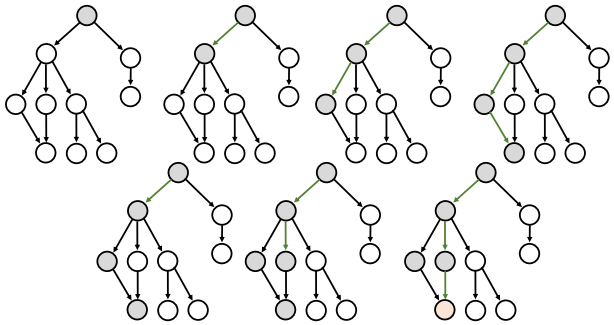
Exercise 5.P.4 Show that every DAG G has at least one vertex with no incoming edges and at least one vertex with no outgoing edges.

(Solution): We know that every DAG has a topological ordering. In this ordering the first vertex v should have no incoming edges. If it did, say (u, v), then it would mean that u is before it in the topological ordering, which would be a contradiction. Similarly, for the final vertex.

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]

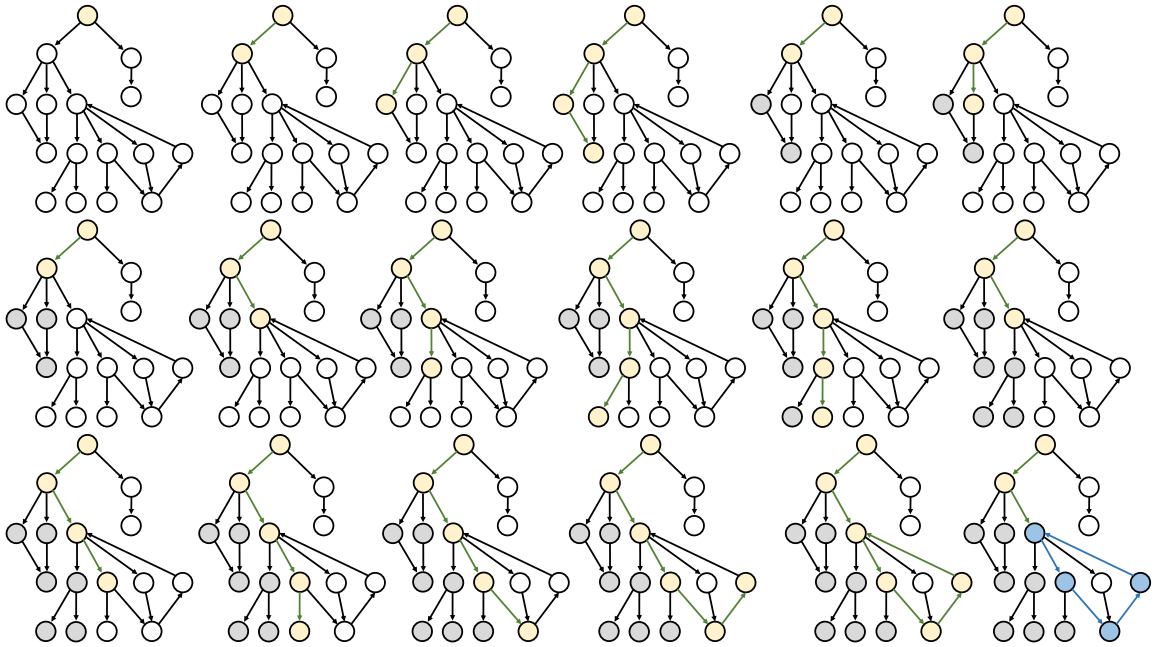
(Solution 1): Run `toposort(G)` to get an ordering t of the vertices in the graph (the algorithm will terminate in $\mathcal{O}(V+E)$ time even in the presence of cycles because it is just a modification of DFS). Then we can check if t is indeed a topological ordering by iterating over the edges (u, v) and verifying that $t(u) < t(v)$.
(Failed approach): Running DFS and checking if a visited vertex was encountered is wrong because the vertex may have been visited from a previous iteration (or may not form a cycle). Consider the execution in the following graph where a visited vertex is found but not cycle is formed:



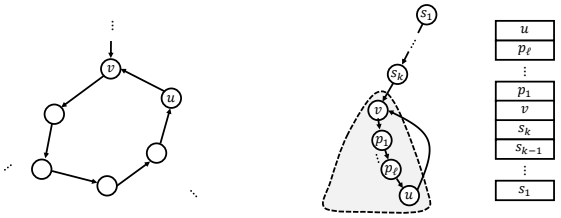
(Solution 2): However, we can modify DFS in the following way:

- Create three starts for vertices: visited, `unvisited`, `in-stack`.
- Each time we examine an edge (u, v), if v is `in-stack`, then it means that there is a cycle, namely the path consisting of all vertices from u's position on the stack to the top of stack and the final edge closes the cycle. (We are using the fact that in a DFS there is always a path from the bottom element of the stack to the top one)
- Return that the graph has a cycle iff such a back-edge is found.

Consider the execution in the following graph:



Now to argue that it always finds a cycle, consider a cycle in a graph G. Consider the first time that the DFS visits a vertex v of the cycle (and let u be the vertex before it). The existence of the path $v \rightsquigarrow u$ implies that u will be visited before terminating the DFS from v. Hence, at that point everything of the stack will consist of a path and hence the back-edge (v, u) will form a cycle (which might not be the original one).



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]

Yes, by definition the finishing times are sorted in reverse topological order, so the arrows should be going left.

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]

This version of DFS will not compute the correct order even for simple DAGs (e.g. one with two vertices and one edge $1 \to 2$).
To correct this, we need to insert v only once we have processed all of its neighbours. This can be done by having a vertex in two states. When pushed the first time it will be `visit`. When popped and it is in `visit` state it will be pushed back with state `done` and then all of its neighbours will be pushed back. When popped and it is in `done` state (which will happen after all of its neighbours are visited), it will be added to the topological ordering.
The lecturer gives the following code:

```
from algorithms.graph import *


def toposort_stack(g):
    totalorder = []

    def visit(s):
        toexplore = [(s, "visit")]
        s.seen = True
        while len(toexplore) > 0:
            v, todo = toexplore.pop()
            if todo == "visit":
                toexplore.append((v, "done"))
                for w in v.neighbours:
                    if not w.seen:
                        toexplore.append((w, "visit"))
                        w.seen = True
            else:  # todo=="done"
                totalorder.append(v)

    for v in g.vertices:
        v.seen = False
    for v in g.vertices:
        if not v.seen:
            visit(v)
```

```
    totalorder.reverse()
    return totalorder


order = toposort_stack(simple_graph_4)
for v in order:
    print(v)
```

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]

We proved as part of Exercise **??** the property that if a vertex is "grey" (i.e. it is on the stack) and there is a back-edge to it, then there is a loop. So when terminating the DFS from a vertex u, all vertices reachable from u have been added to the reverse topological order and for any vertex v that was visited during the DFS, there exists a path from u (because u was on the stack at the same time as v).

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]

(a) Yes, it is. Consider a tree T in the original graph G and the tree consisting of the same edges in the modified graph G', Then $w'(T) = w(T) + (|V| - 1) \cdot c$. Hence, the relative ordering between the trees remains the same (adding a constant to all elements of a sorted array keeps the tree sorted).

(b) No part in the proof of Prim's algorithm requires that the edge weights are ≥ 0. Hence, it works on negative weights as well.

Exercise 5.P.11 [MST with updates]

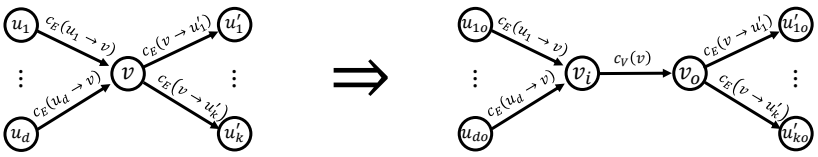
(a) Attempt **[2015P1Q9 (c)]**.

(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.

We can transform all vertices v in the input graph as follows:

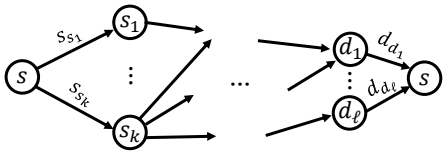


where u_i are the vertices incoming to v and u_i' are the vertices outgoing from v. Therefore the flow through the edge $v_i \to v_o$ corresponds to the flow through the vertex v in the original graph. Since we set $c_E(v_i \to v_o) = c_V(v)$, any valid flow will respect the vertex capacities.

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]

We can model this by inserting a source s and connecting to all vertices s_i with a positive supply s_{s_i} through an edge with capacity s_{s_i}. We also insert a sink t and connect all vertices d_i with a positive demand d_{d_i} to it, through an edge with capacity d_{d_i}.



Then we run the maximum flow algorithm from s to t. If the total capacity is equal to $\sum_u d_u$ then it means that all demands where reached, because the only possible way for flow to reach t is through its incident edges.

Exercise 5.P.14 In the context of Exercise **??**, 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]

This exercise is a bit ambiguous. The way we formulated the problem satisfies A. We could allow for the demand vertices to have an surplus flow by adding some edges with ∞ capacity back to the source vertex. This would satisfy B, as any surplus flow would be routed back to the source. However, by Exercise **??**, it is always possible to remove any flow through this extra edge.

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.

Consider the following algorithm:

1. Repeat while $|f| > 0$:
2. Consider the auxiliary graph H consisting of all edges with $f(u \to v) > 0$.
3. Since the flow is positive, there is a path from s to t in H.
4. Find such a path p_i using e.g. BFS.
5. Let λ_i be the minimum edge on the path.
6. $f := f - \lambda_i$.
7. For each edge e in p_i, set $f(e) := f(e) - \lambda_i$.

The algorithm will terminate after $\mathcal{O}(E)$ iterations since each iteration sets at least one new edge to 0. There will exist such a path in H, because the invariant is that if we can reach a vertex v from s with positive flow, then this flow should also go out of v (by flow conservation) and at some point reach t (otherwise the balance condition does not hold).

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.

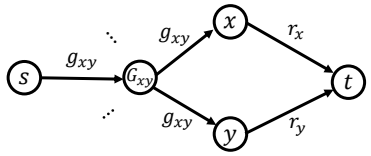
Run the maximum flow algorithm on the graph assuming that each edge has unit capacity. The proof of correctness (and a way to retrieve the disjoint paths) is by using Exercise **??**, since the flow decomposition returns a collection of unit length s-t flows, so there are f edge-disjoint paths. The existence of a cut of size f (by the min-cut theorem), tells you that there is no way that there could be more than k distinct edges from component of s to the component of t, so if there were $> k$ paths connecting s and t, they would have to share one of this edges (hence not disjoint).
See also the 6 paths in the London underground/overground (Exercise **??**).

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:

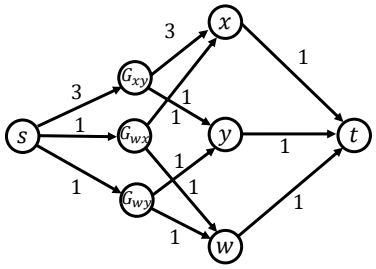
Team	Points
x	12
y	12
z	10
w	7

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.

We are looking for the best possible scenario for z, so we should assume that z wins all of their games. Hence, we know that in the end z will have $w'_z = w_z + \sum_u g_{uz}$. Hence, we are looking for an assignments of the rest of the games so that no team will have more than w'_z points. This means that team x can win at most $r_x := w'_z - w_x$ games (if negative then we know that z finish first). We encode this constraint by having a common sink t, and connect every team x to t with capacity r_x.
Now we also need to encode the constraint that each of the g_{xy} games has to be won by either x or y. This can be done by connecting the source s to vertex G_{xy} with capacity g_{xy} and this vertex connect it to x and y, with capacity G_{xy}. So each team can win at most g_{xy} of the games and each game is won by a single team.



In this graph we compute the maximum flow. The maximum flow will tell us the maximum number of games that can be played with all teams having at most w'_z points. *Why?* Because given a valid flow we can convert it into a sequence of outcomes to some of the games. Also, given a valid sequence of games, we can convert it to a flow.
Consider the example above, the graph looks as follows. The maximum flow is 4, which means that not all 5 games can be played.



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]

(a) Note that the graph is undirected. We convert it into a directed graph where each bi-directional edge (u, v) is converted into two directional edges (u, v) and (v, u) with unit capacities. We compute the maximum flow f in the directed graph and this also gives a cut of size f. Now, we need to argue that the corresponding cut will not have both (u, v) and (v, u) edges, because this would mean that we have double-counted an edge.

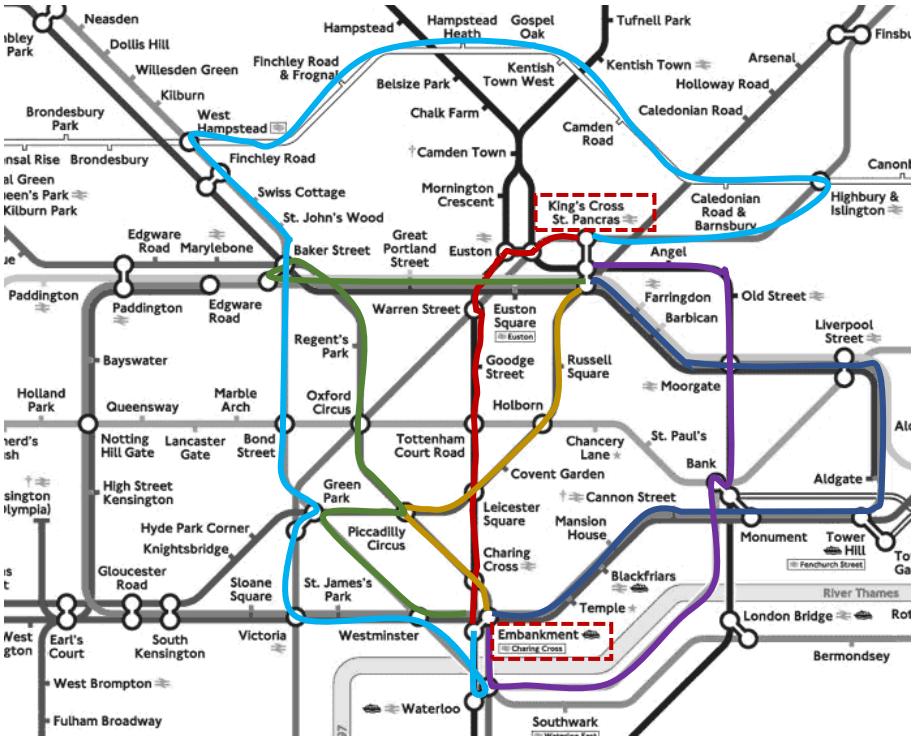
To see this, consider the two components S and T of the cut. Then for (u, v) to be in the cut it means that $u \in S$ and $v \in T$. If we had (v, u) in the cut it would mean that $v \in S$ (and $u \in T$) which cannot be the case since by definition S and T are disjoint. (An alternative way to see it, is that we can always eliminate a cycle from a flow and it will still remain a flow. In this case the cycle would be $u \to v \to u$.).

(b) Removing all edges from the cut will leave no s-t path (otherwise $t \in S$).

(c) Given a set K of travel-preventing signal failures, for each edge in K, remove both of its directed counterparts. Then find all vertices reachable from s and create the set S and the rest let them be T. For each edge (u, v) in K with $u \in S$, we can re-insert the directed edge (v, u) since it will not be counted in the directed cut. Hence, the undirected cut is a valid directed cut.

(d) To find the minimum cut, we use Ford-Fulkerson's algorithm and the minimum-cut theorem.

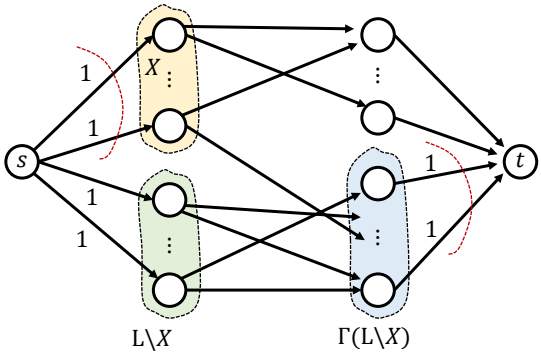
The intuition for the London underground map is that the more we allow the passenger to travel away from Embankment station the more stations she/he will be possible to reach. Hence, we should cut the edges around either King's cross station or around Embankment. To support this claim, we need to find six paths that do not share an edge between these two stations.



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?]*

($\Rightarrow$) Assume that a complete matching m exists. Then for any set $X \subseteq L$, let $Y = \{m(x) : x \in X\}$, then $|Y| = |X|$ since m is injective (there are not two x_1 and x_2 such that $m(x_1) = m(x_2)$). Hence the neighbours $\Gamma(X)$ of X are at least as many as $|Y| = |X|$.
($\Leftarrow$) Assume that for every $X \subseteq L$ we have $\Gamma(X) \geq |X|$. We will show that any cut C has capacity at least $|L|$. Note that cutting the edges from the source, creates a cut of capacity $|L|$. Because we set the middle edges to have ∞ capacity, no cut will contain any of these. So, all cuts will consist of some edges $s \to x$ (for x in some $X \subseteq L$) and some edges $y \to t$ (for y in some $Y \subseteq R$). The cuts in Y should stop all possible paths through $L \setminus X$, which lead to at least $\Gamma(L \setminus X)$ vertices in R. Hence,

$$\textit{capacity} = |X| + |Y| \geq |X| + |\Gamma(L \setminus X)| \geq |X| + |L \setminus X| = |L|.$$


1
1
1
1
X
s
t
1
1
L\X
Γ(L\X)