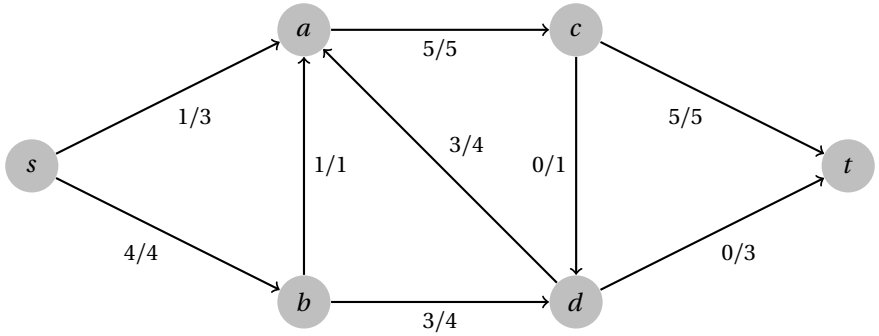
Week 12 Tutorial Sheet

(Solutions)

Useful advice: The following solutions pertain to the theoretical problems given in the tutorial classes. You are strongly advised to attempt the problems thoroughly before looking at these solutions. Simply reading the solutions without thinking about the problems will rob you of the practice required to be able to solve complicated problems on your own. You will perform poorly on the exam if you simply attempt to memorise solutions to the tutorial problems. Thinking about a problem, even if you do not solve it will greatly increase your understanding of the underlying concepts. Solutions are typically not provided for Python implementation questions. In some cases, psuedocode may be provided where it illustrates a particular useful concept.

Assessed Preparation

Problem 1. Consider the following flow network. Edge labels of the form f/c denote the current flow f and the total capacity c.



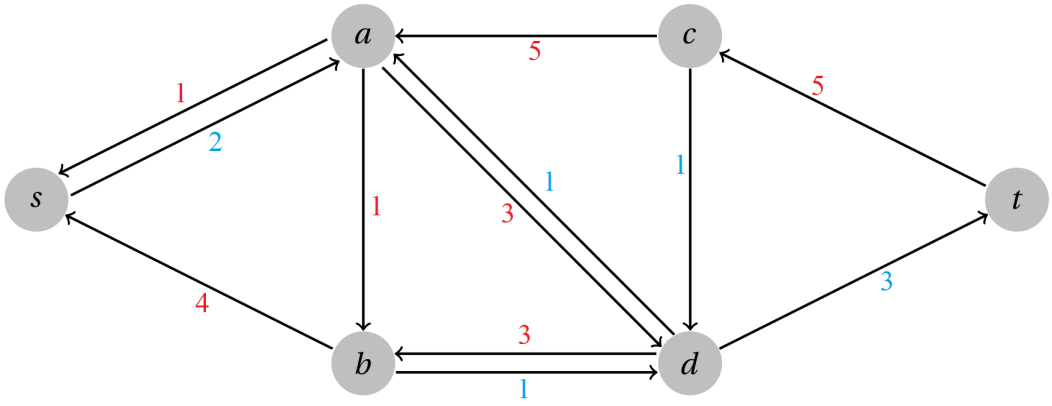
(a) Draw the corresponding residual network

(b) Identify an augmenting path in the residual network and state its capacity

(c) Augment the flow of the network along the augmenting path, showing the resulting flow network

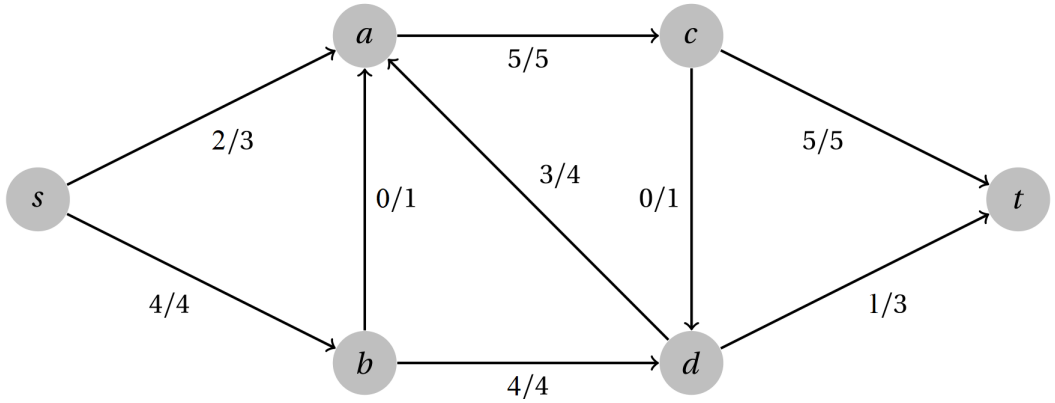
Solution

(a) Residual capacity is shown in blue, reversible flow is shown in red.

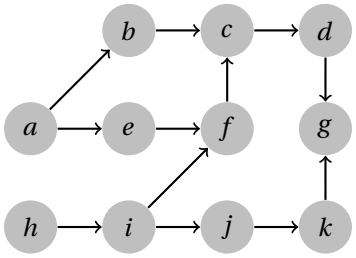


(b) One augmenting path is $s \to a \to b \to d \to t$ with capacity 1.

(c) Note that there are other possible augmenting paths and therefore other solutions.



Problem 2. Show the steps taken by Kahn's algorithm for computing a topological order of the following graph.



Solution

Kahn's algorithm initially adds vertices a and h into the queue since they have an indegree of zero. We add a to the topological order, which enqueues b and e since a was their only incoming edge. We next add h and enqueue i. We then add b, e, then i. This enqueues f and j. We add f which enqueues c. We add j which enqueues k. We add c which enqueues d. We add k and then d, which finally enqueues g, which we add. The final topological order is $a, h, b, e, i, f, j, c, k, d, g$. Note that other orders are possible. For example, we could have started with h instead of a.

Tutorial Problems

Problem 3. Devise an algorithm for determining whether a given directed acyclic graph has a unique topological ordering. That is, determine whether there is more than one valid topological ordering.

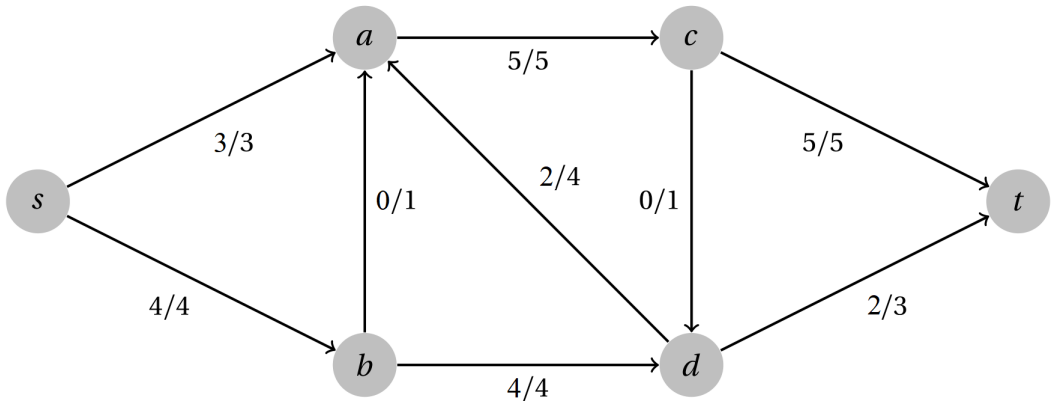
Solution

The easiest way to approach this is to modify Kahn's algorithm. Recall that the queue used by Kahn's contains vertices that have no dependencies left and hence could be added to the topological order. Therefore if the queue ever contains more than one element, the topological order is not unique. Otherwise, the topological order is unique.

Problem 4. Complete the Ford-Fulkerson method for the network in Problem 1, showing the final flow network

with a maximum flow.

Solution



Problem 5. Using your solution to Problem 4, list the vertices in the two components of a minimum $s-t$ cut in the network in Problem 1. Identify the edges that cross the cut and verify that their capacity adds up to the value of the maximum flow.

Solution

To find the minimum cut after running the Ford-Fulkerson algorithm, we simply identify all vertices reachable from the source in the residual graph. These vertices will be on one side of the cut, and all remaining vertices will be on the other. In this graph, there are no outgoing edges in the residual graph, since the two edges outgoing from s are full. So the source is on one side of the cut, and everything else is on the other.
Component 1: $\{s\}$
Component 2: $\{a,b,c,d,t\}$
The flow across this cut is indeed 7, as expected.

Problem 6. Let G be a flow network and let f be a valid flow on G. Prove that the net outflow out of s is equal to the net inflow into t.

Solution

Suppose the set of all vertices is V. We know that the flow of every cut is equal to the flow of the network. Therefore the capacity of the cut $(\{s\},\{V-\{s\}\}$ (which is the flow out of s) must equal the capacity of the cut $(\{V-\{t\}\},\{t\}$ (which is the capacity into t).

Problem 7. Consider a variant of the maximum network flow problem in which we allow for multiple source vertices and multiple sink vertices. We retain all of the capacity and flow conservation constraints of the original maximum flow problem. As in the original problem, all of the sources and sinks are excluded from the flow conservation constraint. Describe a simple method for solving this problem.

Solution

We create a new graph by adding a "super-source" and "super-sink" vertex. The super source is connected to each source by an edge with capacity equal to the total capacities of all the outgoing edges from that source. Similarly, each sink is connected by an edge to the super sink, and the capacity of that edge is equal to the total capacities of all incoming edges to that sink. We then solve the flow problem on this

new graph as normal.

Problem 8. A Hamiltonian path in a graph $G = (V, E)$ is a path in G that visits every vertex $v \in V$ exactly once. On general graphs, computing Hamiltonian paths is NP-Hard. Describe an algorithm that finds a Hamiltonian path in a directed acyclic graph in $O(V + E)$ time or reports that one does not exist.

Solution

A Hamiltonian path must begin at some vertex and then travel through every other vertex without ever revisiting a previous one. The first thing to note is that the starting vertex must therefore be a vertex with no incoming edges, since such a vertex can never be visited in any other way by the path. This means that a Hamiltonian path will not exist if there are multiple such vertices. This should remind us of something similar, topological orderings!

Let's argue that a Hamiltonian path must be a topological ordering of a graph. Consider the sequence of vertices in a Hamiltonian path. If a vertex has an edge that travels to a previous vertex (meaning that the two are out of topological order), then this would produce a cycle in the graph, which is impossible since the graph is acyclic by assumption. Therefore a Hamiltonian path, if one exists, is a topological order of the graph.

Suppose that a Hamiltonian path exists. Then since every adjacent pair of vertices must be connected, the topological order of the graph must be unique, since every vertex has a dependency on the one before it. Therefore if the topological order is not unique, a Hamiltonian path will not exist. Finally, suppose that there is a unique topological order, then this implies that there are no pairs of vertices in the order that are not adjacent, since otherwise they could be swapped without breaking the dependency constraints. Since every pair of vertices in the order must be adjacent, this constitutes a Hamiltonian path.

We can conclude that a Hamiltonian path exists if and only if the graph has a unique topological order, and if so, the Hamiltonian path is the topological order.

Problem 9. Consider a directed acyclic graph representing the hierarchical structure of n employees at a company. Each employee may have one or many employees as their superior. The company has decided to give raises to m of the top employees. Unfortunately, you are not sure exactly how the company decides who is considered the top employees, but you do know for sure that a person will not receive a raise unless all of their superiors do.

Describe an algorithm that given the company DAG and the value of m, determines which employees are guaranteed to receive a raise, and which are guaranteed to not receive a raise. Your algorithm should run in $O(V^2 + VE)$ time.

Solution

We can rephrase this problem in terms of topological orderings. An employee will receive a raise if they are among the top m employees, which means that they must be in the first m elements of a topological order. However, we can not simply compute a topological order and check the first m elements since the order is not guaranteed to be unique. More concretely, an employee is guaranteed a raise if they are in the first m elements of **every** possible topological order. Similarly, an employee is guaranteed to not get a raise if they are never in the first m elements in **any** topological order.

In order to determine this, consider a particular vertex v in a directed acyclic graph. The vertices that must come after it in a topological ordering are all of the vertices that it can reach, since they are all of its dependants. Conversely, all of the vertices that must come before v are the ones that can reach v. All other vertices could go before or after v.

Therefore we want to count for each employee, the number of employees that are reachable in the company DAG, and conversely, the number of employees that can reach them in the company DAG. We can

count the number of vertices reachable from a particular employee in $O(V+E)$ time with a simple depth-first search. Similarly, to count the number of employees that reach them, we can perform a depth-first search in the company DAG with all of the edges reversed. Let the number of reachable employees be r, and the number of employees that reach them be s. An employee is guaranteed to be in the first m elements of any topological order if and only if

$$r \geq n - m.$$

Similarly, an employee is guaranteed to not be in the first m elements if and only if

$$s \geq m.$$

We can therefore run this procedure for every employee and output the results. We perform two depth-first searches per employee, taking $O(V+E)$ time each, and hence the total time complexity is $O(V^2+VE)$ as required.

Supplementary Problems

Problem 10. Consider a variant of the maximum network flow problem in which vertices also have capacities. That is for each vertex except s and t, there is a maximum amount of flow that can enter and leave it. Describe a simple transformation that can be made to such a flow network so that this problem can be solved using an ordinary maximum flow algorithm[1].

Solution

For each non-source non-sink vertex v, do the following: Create two new vertices v_{in} and v_{out}. Take all the inflowing edges to v and replace v with v_{in}. Take all the outflowing edges from v and replace v with v_{out}. Create an edge from v_{in} to v_{out} with capacity equal to the capacity of v. Delete v from the network.

Now we can solve for the maximum flow in this graph, and the solution will be the maximum flow in the original graph as well.

Problem 11. Two paths in a graph are *edge disjoint* if they have no edges in common. Given a directed network, we would like to determine the maximum number of edge-disjoint paths from vertex s to vertex t.

(a) Describe how to determine the maximum number of edge-disjoint $s-t$ paths.

(b) What is the time complexity of this approach?

(c) How could we modify this approach to find *vertex-disjoint paths*, i.e. paths with no vertices in common

Solution

(a) Given a directed graph G, create a flow network G' from G as follows: Let s be the source, and t be the sink. Give every edge a capacity of 1. Now we can run the maximum flow algorithm, and the resulting flow is the number of *edge-disjoint* paths.

This works because the number of *edge-disjoint* paths that pass through a vertex v is at most $\min(\text{indegree}(v), \text{outdegree}(v))$. The maximum flow through a vertex v is the same value, because all edges are capacity 1.

[1] Do not try to modify the Ford-Fulkerson algorithm. In general, it is always safer when solving a problem to reduce the problem to another known problem by transforming the input, rather than modifying the algorithm for the related problem.

(b) The time complexity of Ford-Fulkerson is bounded by $O(Ef)$ where E is the number of edges and f is the maximum flow. Here, the maximum flow is the outdegree of s, so the complexity is $O(E \times \text{outdegree}(s))$ which is bounded by $O(EV)$.

(c) Create the flow network as specified in part a), but add capacities for each vertex, and then perform the transform described in the solution to problem 10. This ensures that the flow through each vertex is at most 1, wheras before the flow through a vertex was equal to the number of paths through it.

Problem 12. (Advanced) A useful application of maximum flow to people interested in sports is the *baseball elimination* problem. Consider a season of baseball in which some games have already been played, and the schedule for all of the remaining games is known. We wish to determine whether a particular team can possibly end up with the most wins. For example, consider the following stats.

	Wins	Games Left
Team 1	30	5
Team 2	28	10
Team 3	26	8
Team 4	20	9

It is trivial to determine that Team 4 has no chance of winning, since even if they win all 9 of their remaining games, they will be at least one game behind Team 1. However, things get more interesting if Team 4 can win enough games to reach the current top score.

	Wins	Games Left
Team 1	30	5
Team 2	28	10
Team 3	29	8
Team 4	20	11

In this case, Team 4 can reach 31 wins, but it doesn't matter since the other teams have enough games left that one of them must reach 32 wins. We can determine the answer with certainty if we know not just the number of games remaining, but the exact teams that will play in each of them. A complete schedule consists of the number of wins of each team, the number of games remaining, and for each remaining game, which team they will be playing against. An example schedule might look like the following.

		Games Remaining				
	Wins	Total	vs T1	vs T2	vs T3	vs T4
Team 1	29	5	0	2	1	2
Team 2	28	10	2	0	4	4
Team 3	28	8	1	4	0	3
Team 4	25	9	2	4	3	0

Describe an algorithm for determining whether a given team can possibly end up with the most wins. Your algorithm should make use of maximum flow.

Solution

Assume that the given teams wins all of their remaining games since this is the best they can do.

Each remaining game will assign a point to one of the two teams that plays in it, so make a graph with games on the left and teams on the right. For each game, add an edge from s with capacity 1, and you add edges from the game to each of the teams that play, so the points from that game (the flow) flow into one of the two teams that played it. For efficiency, you can merge duplicate games played between the same pair of teams into one vertex and make the capacity from s to that game equal to the number of such games, instead of 1.

We'd like to prevent any team from getting more points than our given team, so cap the capacity of that team to the sink at our favourite team's score - 1. If our favourite team can win, the points can be distributed to all of the teams without them overtaking us. This happens if the maximum flow of the graph is the number of games remaining (ie. the source edges are saturated). Otherwise, it was not possible to assign all of the points and hence our favourite team can not win.