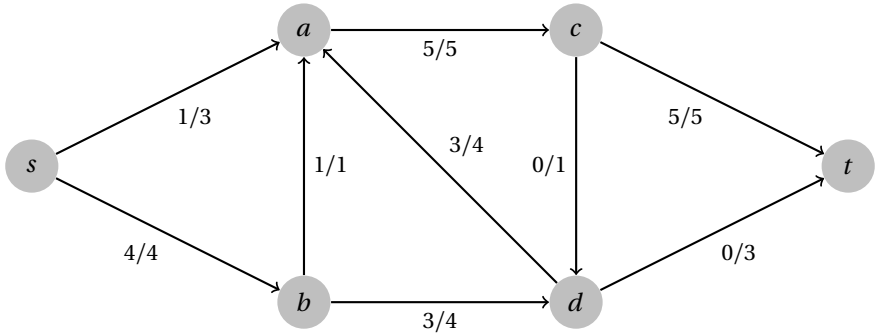
Week 12 Tutorial Sheet

(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.



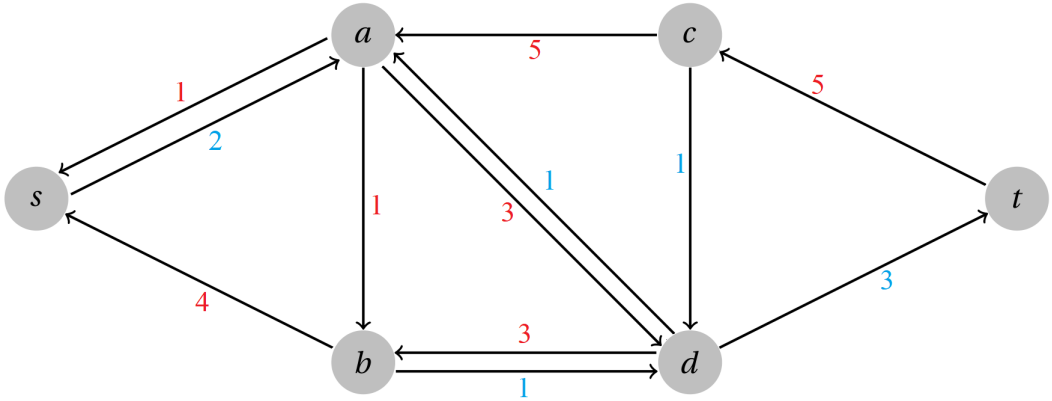
(a) Draw the corresponding residual network

(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

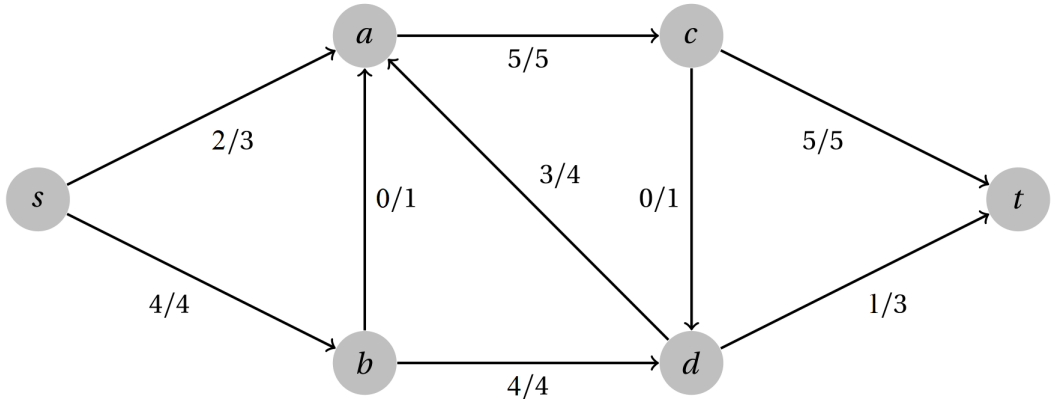
Solution

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

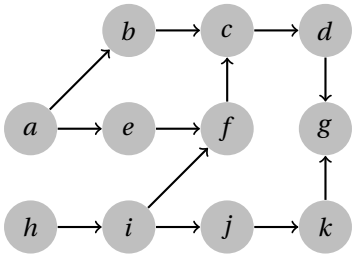


(b) One augmenting path is $s \to a \to b \to d \to t$ with capacity 1.

(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. Devise an algorithm for counting the number of paths between two given vertices s and t in a directed acyclic graph. Don't worry about the magnitude of the answer (i.e. assume that all arithmetic operations

take constant time, despite the fact that the answer might be exponential in the input size.)

Solution

We will approach this using a dynamic programming algorithm. Suppose we are at some vertex u. Then the number of different ways in which we could get to t involves trying all of our different edges (u, v) and then counting the number of resulting paths from all of the v. So let us define the following subproblems.

$$\texttt{DP}[u] = \{\text{The number of paths from } u \text{ to } t\}.$$

The recurrence is then given by

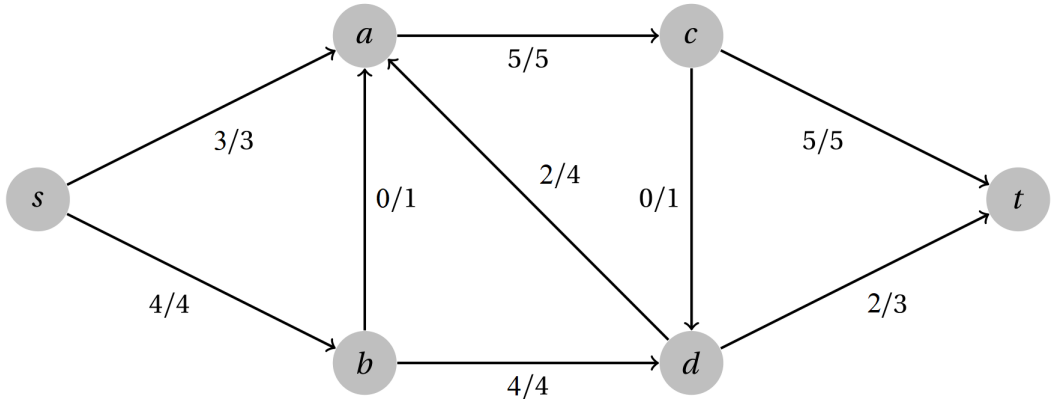
$$\texttt{DP}[u] = \begin{cases} 1 & \text{if } u = t, \\ \displaystyle\sum_{v \in \text{adj}[u]} \texttt{DP}[v] & \text{otherwise.} \end{cases}$$

The answer is the value of $\texttt{DP}[s]$. The subproblems are dependent in a reverse topological order, since in order to compute $\texttt{DP}[u]$ we must know the value of all of u's descendants.

```
1:  function COUNT_PATHS(G = (V, E), s, t)
2:      Set count[1..n] = 0
3:      count[t] = 1
4:      Set order = reverse(topological_sort(G))
5:      for each vertex u in order do
6:          for each edge (u, v) adjacent to u do
7:              count[u] = count[u] + count[v]
8:          end for
9:      end for
10:     return count[s]
11: end function
```

Problem 5. Complete the Ford-Fulkerson method for the network in Problem 1, showing the final flow network with a maximum flow.

Solution



Problem 6. Using your solution to Problem 5, 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 7. 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 8. 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 9. 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 10. 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.

Problem 11. Given a list of n integers $r_1, r_2, ..., r_n$ and m integers $c_1, c_2, ..., c_m$, we wish to determine whether there exists an $n \times m$ matrix consisting of zeros and ones whose row sums are $r_1, r_2, ..., r_n$ respectively and whose column sums are $c_1, c_2, ..., c_m$ respectively. Describe an algorithm for solving this problem by using a flow network.

Solution

Constuct a flow network as follows: Create two sets of vertices, A and B. Each vertex in A $(a_1, a_2, ...a_n)$ corresponds to one of the n rows, and each vertex in B $(b_1, b_2, ...b_m)$corresponds to one of the m columns. Add a source and a sink. For $1 \leq i \leq n$, add an edge from the source to a_i with capacity r_i. For each $1 \leq j \leq n$, add an edge from vertex b_j to the sink with capacity c_j. Add edges $e_{i,j}$ between every pair of vertices (a_i, b_j) with capacity 1. Determine the maximum flow in this network. If all the edges from the source are full, then we can construct such a matrix, and moreoever, the elements $A_{i,j}$ of the matrix are equal to the flows along $e_{i,j}$. If any edge from the source is not full, such a matrix does not exist.

To see why this method works, consider the flows into and out of the a_i vertices. We notice that a row has r_i 1s iff the vertex a_i has a total outflow of r_i (because each edge out of a_i corresponds to one of the values in column i, and all the edges from a_i have capacity 1), but this is only possible if the flow from the source to a_i is exactly r_i because the only edge that flows into a_i is from the source. These edges have capacity r_i, so they all need to be full.

Problem 12. Consider the problem of allocating a set of jobs to one of two supercomputers. Each job must be allocated to exactly one of the two computers. The two computers are configured slightly differently, so for each job, you know how much it will cost on each of the computers. There is an additional constraint. Some of the jobs are related, and it would be preferable, but not required, to run them on the same computer. For each pair of related jobs, you know how much more it will cost if they are run on separate computers. Give an efficient algorithm for determining the optimal way to allocate the jobs to the computers, where your goal is to minimise the total cost.

Solution

Create a graph with (bidirectional) edges between related jobs, whose capacities are the cost to separate them. Add edges from s to all jobs with capacity cost to run on computer 1, and edges from all jobs to t with capacity cost to run on computer 2. The minimum cut will be the best total cost since you are paying to either put the job on computer 1 or 2, and paying for related jobs that are cut from each other when assigned to different computers.

Supplementary Problems

Problem 13. 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 14. 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.

[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.

(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(indegree(v), outdegree(v)). The maximum flow through a vertex v is the same value, because all edges are capacity 1.

(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 $O(E \times$ 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 13. 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 15. You are a radio station host and are in charge of scheduling the songs for the coming week. Every song can be classified as being from a particular era, and a particular genre. Your boss has given you a strict set of requirements that the songs must conform to. Among the songs chosen, for each of the n eras $1 \leq i \leq n$, you must play exactly n_i songs of that era, and for each of the m genres $1 \leq j \leq m$, you must play exactly m_j songs from that genre. Of course, $\sum n_i = \sum m_j$. Devise an algorithm that given a list of all of the songs you could choose from, their era and their genre, determines a subset of them that satisfies the requirements, or determines that it is not possible.

Solution

Make a bipartite graph with vertices $u_1 \ldots u_i$ corresponding to eras, and $v_1 \ldots v_j$ corresponding to genres. Each song is represented by an edge from its genre to its era (of capacity 1 if you can only play each song once. How would you allow for each song to be played some other number of times at most?). Add a source with outgoing edges to each genre vertex u_k with capacity n_k (where $1 \leq k \leq n$). Add a sink vertex, with an edge from each vertex v_l to the sink with capacity m_l (where $1 \leq l \leq n$). Run Ford-Fulkerson on this graph, and if a flow of capacity $\sum n_i = \sum m_j$ is achieved, then it is possible to play songs in the required combination. Otherwise it is not. The flow along song edges tells you how many times to play each song.

Problem 16. Recall the concept of a directed acyclic graph (DAG). A *path cover* of a DAG is a set of paths that include every vertex **exactly once** (multiple paths can not touch the same vertex). A minimum path cover is a path cover consisting of as few paths as possible. Devise an efficient algorithm for finding a minimum path cover of a DAG. [Hint: Use bipartite matching]

Solution

Create a bipartite graph where each side is a copy of V. For each edge in the graph, put that edge (u, v), add an edge in the bipartite graph (u on the left, v on the right).

Solve for the maximum cardinality matching (using flow). I claim that any unmatched vertex on the left is the beginning of a path, hence the answer is $|V|$ - #successful matches. To see why this is true, note

that the matching is essentially choosing for each vertex, who do I go to next in my path? Each vertex can be in one path only, hence it can be matched to one vertex, and vice versa, at most one vertex can match to it. The unmatched vertices on the left must therefore be the beginnings of paths, and the unmatched vertices on the right are the ends of paths. In fact, if you imagine traversing the matching in the bipartite graph, the paths you traverse will be precisely the path cover.

Fun fact: extend this to non-acyclic graphs and run this algorithm, and you are determining a cycle cover, a set of disjoint cycles that cover the graph, by the same logic.

Problem 17. (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.

Problem 18. (Advanced) Consider a directed acyclic graph G where each edge is labelled with a character from some finite alphabet A. Given a string S over the alphabet A, count the number of paths in G whose edge labels spell out the string S. Your algorithm should run in $O((V+E)n)$, where n is the length of the string S.

Solution

This problem can be solved very similarly to the ordinary path-counting problem on a DAG (Problem 4). We will use dynamic programming. Let us define the subproblems

$$\texttt{DP}[u,i] = \{\text{The number of paths from } u \text{ that spell out } S[i..n]\}$$

If we are at a particular vertex u and we want to write $S[i..n]$, then we need to find all of our outgoing edges (u,v) that are labelled $S[i]$, and then try to write $S[i+1..n]$ from vertex v. A recurrence can therefore be written as follows.

$$\texttt{DP}[u,i] = \begin{cases} 1 & \text{if } i = n+1, \\ \displaystyle\sum_{\substack{v \in \text{adj}[u] \\ l(u,v)=S[i]}} \texttt{DP}[v,i+1] & \text{otherwise} \end{cases}$$

where $l(u,v)$ denotes the label of the edge (u,v). The solution to the problem is then the sum of $\texttt{DP}[u,1]$ for all vertices $u \in V$. We have Vn subproblems, and for each edge in the graph, it gets processed at most once per value of i, so the total time complexity is $O(Vn+En) = O((V+E)n)$ as required.

Problem 19. (Advanced) Describe how an instance of the unbounded knapsack problem can be converted into a corresponding directed acyclic graph. Which graph problem correctly models the unbounded knapsack problem on this graph?

Solution

Create a graph with $C+1$ vertices, labelled 0 to C, where C is the capacity of the knapsack. For each item available with weight w and value v, create an edge from every vertex $c \geq w$ to the vertex $c-w$ with weight v. The solution to the unbounded knapsack problem is the critical path of this graph.

Problem 20. (Advanced) A problem closely related to the transitive closure problem mentioned in lectures is the *transitive reduction*. In a sense, the transitive reduction is the opposite of the transitive closure. It is a directed graph with the fewest possible edges that has the same reachability as the original graph. In other words, for all pairs of vertices u and v, there is a path between u and v in the transitive reduction if and only if there is a path between u and v in the original graph. Give an algorithm for computing the transitive reduction of a directed acyclic graph. Your algorithm should run in $O(V^2+VE)$ time.

Solution

As usual, we will assume that G is a simple graph. If it is not, we can simply remove duplicate edges and loops at the beginning since they contribute nothing to the reachability. Consider each edge (u,v) in the graph G. If (u,v) is the only way to travel from u to v, then clearly we must keep it. Otherwise, if there is some other way to get from u to v, since the graph is simple, there must be a path consisting of multiple edges that leads from u to v. If this is the case, removing (u,v) does not harm the reachability, and since

the other vertices on the $u \rightsquigarrow v$ path must maintain their respective reachabilities, we should remove (u, v), rather than removing anything on that path.

Given this, our algorithm is as follows. Let's use the critical path dynamic programming algorithm to compute the longest paths between every pair of vertices in the graph. Computing the longest paths that start at a particular vertex takes $O(V + E)$, so doing this from every vertex takes $O(V^2 + VE)$ time. Then, for each edge (u, v), we decide to keep it if the longest path from u to v is length 1 (i.e. just this edge), otherwise we delete it. This algorithm takes $O(V^2 + VE)$ time in total.

Problem 21. (Advanced) You are in charge of projects at a large company and need to decide for this year, which projects the company will undertake. Each project has a profit value associated with it. Some projects are worth positive profit, meaning you earn money from completing them. Some projects are worth negative profit, meaning they cost more money than they make. Projects have prerequisites with other projects, meaning that a project can only be completed once all of its prerequisites have been completed. The goal is to determine which projects to complete in order to make the maximum amount of profit, while ensuring that all prerequisite relationships are satisfied. This problem can be solved by reducing it to a minimum cut problem as follows:

- Create a flow network with a source vertex s and a sink vertex t
- Create a vertex for each project
- For each project x with positive profit p, add a directed edge from s to x whose weight is p
- For each project x with negative profit $-p$, add a directed edge from x to t whose weight is p
- For each project x that has y as a prerequisite, add a directed edge from x to y with weight ∞

The minimum cut of this network can be used to determine the optimal set of projects to complete.

(a) Explain what the components of the minimum $s-t$ cut correspond to in the optimal solution

(b) What is the purpose of the infinite capacity edges?

(c) Explain how to compute maximum profit from the capacity of the minimum $s-t$ cut

Solution

Let's think about the structure of the graph. The source vertex connects to all of the profitable projects, while the unprofitable projects get connected to the sink vertex. Suppose that some particular profitable project x remains in the S component of the minimum cut. Then because of the infinite capacity edges connecting that project to its prerequisites, they too must be in the S component, since disconnecting them would cost ∞, which would not be a minimum cut. This means that we must pay the cost of disconnecting any unprofitable prerequisites from the sink vertex, which is the total cost of undertaking said unprofitable projects. Suppose instead that a profitable project is not in the S component. This means that the edge from the source to it must be disconnected, which costs the value of the project. In this case, it is not required to disconnect the unprofitable prerequisites since they are not connected to s directly, but only via their dependants. From these observations, we can deduce the following answers.

(a) Each project in the S component is a project that we will complete. Each project in the T component are the projects that we will not complete. This makes sense since any profitable project in S forces all of its prerequisites to also be in S.

(b) The infinite capacity edges ensure that the prerequisite relationships are satisfied. You can not disconnected an infinite capacity edge since that would make the cost of the cut infinity, which will never be a minimum cut!

(c) From the discussion above, when we decide to complete a profitable project, we subsequently pay the cost of its unprofitable prerequisites by disconnecting them from t. Alternatively, when we decide to not complete a profitable project, we pay the value of that project, i.e. we pay the profit

that we are choosing to give up. This implies that the capacity of the cut is

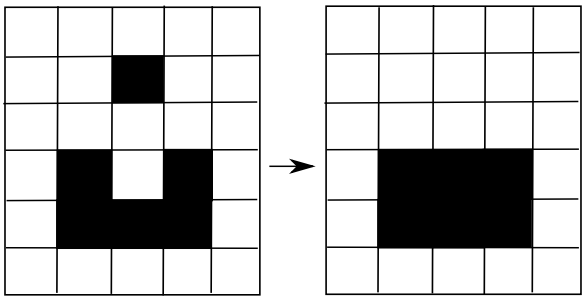
$$\begin{aligned} c(S,T) = &\text{ The cost of the unprofitable projects we are forced to do} \\ &+ \text{ The cost of the profitable projects we choose not to do} \end{aligned}$$

From this, it is easy to compute the maximum possible profit by noticing that

$$\text{Max Profit} = \text{Total profit of all profitable projects} - c(S,T),$$

which makes sense since we are seeking the **minimum** value of $c(S,T)$, which will therefore maximise the profit.

Problem 22. (Advanced) You are the owner of a plot of land that can be described as an $n \times m$ grid of unit-square-sized cells. Each cell of land is either filled in, or a hole. The government has to decided to crack down on safety regulations and requires you to fence off the holes in the ground. Each unit of fencing will cost you $\$c_{\text{fence}}$. In order to reduce the amount of fencing required, you have the option to fill in some of the holes, or even dig some additional holes. It will cost you $\$c_{\text{dig}}$ to dig a new hole, or $\$c_{\text{fill}}$ to fill in an existing hole. For safety, there are no holes on the boundary cells of your land, and you are not permitted to make any. For example, in the following case (where black represents a hole), with $c_{\text{fence}} = 10$, $c_{\text{dig}} = 10$, and $c_{\text{fill}} = 30$, the cost to fence the land initially would be \$160. The optimal solution is to dig out the middle cell and fill in the topmost hole, yielding a total cost of \$140.



Describe how to determine the minimum cost to make your land safe by reducing it to a minimum cut problem.

Solution

In order to model this as a minimum cut problem, we need to decide what the components of the cut will represent, i.e. what two things are we trying to separate? The answer is reasonable obvious, we are paying a cost to separate holes from non-holes, so the components of the cut will be the cells that end up as holes, and the cells that end up as non-holes. Two adjacent cells must pay a cost to be disconnected (the cost $\$c_{\text{fence}}$), or we can pay to change something from a hole to a non-hole or vice versa.

So, let's make a graph where each vertex represents one of the cells of land, and add two special vertices, the sink and source, s and t. Let's say that we want the S component to represent the non-holes, and the T component to represent the holes. First we will add edges from s to all of the boundary cells with capacity ∞ because we are not allowed to change these into holes. For the rest of the non-holes, we add a vertex from s to their respectively vertices with capacity c_{dig}, because it costs $\$c_{\text{dig}}$ to remove this cell from the non-holes component and make it a part of the holes component. Correspondingly for each cell that is currently a hole, we add an edge from its corresponding vertex to t with capacity c_{fill}, since it costs $\$c_{\text{fill}}$ to make this cell part of the non-holes component.

Finally, we must deal with the fencing cost. It costs $\$c_{\text{fence}}$ to have two adjacent cells be in different components, so for every pair of adjacent cells, we add edges in both directions with capacity c_{fence}. This ensures that if the two cells are in different components, we will have to pay $\$c_{\text{fence}}$ to keep them apart,

but if they are in the same component, we pay nothing. Note that we must add the edge in both directions since we do not know in advance which cell will be in which component, but the cut will never double count the cost since the capacity of a cut only measures the edges going $S \to T$, but not those in the other direction.

The minimum cost to make your land safe will be the capacity of the minimum cut of this graph, i.e. the value of its maximum flow.

Problem 23. **(Extra Advanced)** Consider a variant of the maximum flow problem in which we enforce lower bounds on the edge flows, i.e. we can require that an edge has at least a certain amount of flow. Formally, each edge now has a capacity c, and a demand d, and we require that $d(u,v) \leq f(u,v) \leq c(u,v)$. This problem can be solved by modifying the network and using an ordinary maximum flow algorithm.

(a) Describe how to determine whether a feasible flow exists, i.e. a flow the satisfies the demand

(b) Describe how to determine a maximum feasible flow, i.e. a maximum flow satisfying the demands

(c) Describe how to determine a minimum feasible flow, i.e. a minimum flow satisfying the demands

Solution

Finding a feasible flow

We begin by describing how to check whether a feasible flow exists, i.e. whether a flow exists that satisfies the demands. We pay no attention to attempting to maximise the flow at this point. Intuitively, for each edge (u,v), our goal to make the vertex u prove that it has sufficient incoming flow to satisfy the demand $d(u,v)$. To do so, we create a modified network G' consisting of the same vertices, but with new source and sink vertices s' and t'. (The old source and sink are **not** removed).

Given a particular edge (u,v), in order for the vertex u to prove that is has enough flow to satisfy the demand, we will add an edge from u to t' with capacity $d(u,v)$. Since this flow should ultimately have ended up at v, we add an edge from s' to v with capacity $d(u,v)$. This edge "refunds" the flow that we took from u as proof of satisfying the demand. Then, we add the edge between u and v, but with capacity $c(u,v)-d(u,v)$, since the $d(u,v)$ units of flow are now supplied via s'.

Finally, in order to ensure that flow can actually move around the network, we add an edge from t to s with capacity ∞. This is required since otherwise no flow could ever flow into t (as it is no longer the "real" sink vertex), and it would be incorrect to add an edge from t to t' since we would no longer be able to measure the demands being sent in as proof. In order to make this more efficient, we should then merge all edges that travel between the same pairs of vertices, since this construction may have added many such edges. An example transformation is shown below.

In order to determine whether a feasible flow exists, we need to check whether a flow exists in G' that satisfies all of the demands, in other words, it saturates all of the edges into t'. Such a flow must be a maximum flow, so we find a maximum flow f' in G' and check whether $|f'|$ is equal to $\sum_e d(e)$. If it is, then we have found a feasible flow for G. If it does not, then a feasible flow does not exist. We can recover the feasible flow f by removing s' and t' and adding back $d(u,v)$ to the flow of every edge (u,v).

Finding a maximum feasible flow

We begin with a feasible flow f found in part (a). In order to maximise the feasible flow that we have found, it would be tempting to simply run Ford-Fulkerson from s to t starting with the flow f. However, this might accidentally cancel out some of the flow that is necessary to satisfy the demands, so we need to be proactive and prevent this. In order to do this, for every edge $(u,v) \in E$, we simply subtract $d(u,v)$ from $c(u,v)$ and $f(u,v)$ before running Ford-Fulkerson. This "hides" the flows necessary to satisfy the demands so that they are not undone by an augmenting path. After running Ford-Fulkerson from s to t, we can add $d(u,v)$ back to the flow on each edge (u,v) to obtain a final maximum feasible flow. We

could also save a step by noticing that G' contains precisely edges and flows that are short by $d(u,v)$, so we could simply run Ford-Fulkerson from s to t in G' before recovering the feasible flow.

Finding a minimum feasible flow

Lastly, we consider the problem of finding a minimum feasible flow. Without lower bound constraints, minimum flows are quite boring, since the zero flow is always feasible. With lower bounds they actually become interesting though. To find one, we begin with a feasible flow f found in part (a), and build a network of "unnecessary flow". This new network has the same vertices and edges as G, but we set the capacity of each edge (u,v) to be $f(u,v)-d(u,v)$, i.e. the amount of flow over the demand. We then run Ford-Fulkerson to find a maximum (s,t) flow f' in this graph, which will correspond to the maximum amount of flow that we can remove from f. Our minimum feasible flow is then given by $f-f'$.