Chapter 16

Network Flows and Linear Programming

16.1 The Steepest Ascent Hill Climbing Algorithm

We have all experienced that climbing a hill can take a long time if you wind back and forth barely increasing your height at all. In contrast, you get there much faster if energetically you head straight up the hill. This method, which is call the method of *steepest ascent*, is to always take the step that increases your height the most. If you already know that the hill climbing algorithm in which you take any step up the hill works, then this new more specific algorithm also works. However, if you are lucky it finds the optimal solution faster.

In our network flow algorithm, the choice of what step to take next involves choosing which path in the augmenting graph to take. The amount the flow increases is the smallest augmentation capacity of any edge in this path. It follows that the choice that would give us the biggest improvement is the path whose smallest edge is the largest for any path from s to t. Our steepest ascent network flow algorithm will augment such a best path each iteration. What remains to be done is to give an algorithm that finds such a path and to prove that this finds a maximum flow is found within a polynomial number of iterations.

Finding The Augmenting Path With The Biggest Smallest Edge: The input consists of a directed graph with positive edge weights and with special nodes s and t. The output consists of a path from s to t through this graph whose smallest weighted edge is as big as possible.

Easier Problem: Before attempting to develop an algorithm for this, let us consider an easier but related problem. In addition to the directed graph, the input to the easier problem provides a weight denoted w_{min}. It either outputs a path from s to t whose smallest weighted edge is at least as big as w_{min} or states that no such path exists.

Using the Easier Problem: Assuming that we can solve this easier problem, we solve the original problem by running the first algorithm with w_{min} being every edge weight in the graph, until we find the weight for which there is a path with such a smallest weight, but there is not a path with a bigger smallest weight. This is our answer. (See Exercise 16.1.1.)

Solving the Easier Problem: A path whose smallest weighted edge is at least as big as w_{min} will obviously not contain any edge whose weight is smaller than w_{min}. Hence, the

answer to this easier problem will not change if we delete from the graph all edges whose weight is smaller. Any path from s to t in the remaining graph will meet our needs. If there is no such path then we also know there is no such path in our original graph. This solves the problem.

Implementation Details: In order to find a path from s to t in a graph, the algorithm branches out from s using breadth-first or depth-first search marking every node reachable from s with the predecessor of the node in the path to it from s. If in the process t is marked, then we have our path. (See Section **??**.) It seems a waste of time to have to redo this work for each w_{min} so let's use an iterative algorithm. The loop invariant will be that the work for the previous w_{min} has been done and is stored in a useful way. The main loop will then complete the work for the current w_{min} reusing as much of the previous work as possible. This can be implemented as follows. Sort the edges from biggest to smallest (breaking ties arbitrarily). Consider them one at a time. When considering w_i, we must construct the graph formed by deleting all the edges with weights smaller than w_i. Denote this G_{w_i}. We must mark every node reachable from s in this graph. Suppose that we have already done these things in the graph $G_{w_{i-1}}$. We form G_{w_i} from $G_{w_{i-1}}$ by adding the single edge with weight w_i. Let $\langle u, v\rangle$ denote this edge. Nodes are reachable from s in G_{w_i} that were not reachable in $G_{w_{i-1}}$ only if u was reachable and v was not. This new edge then allows v to be reachable. Unmarked nodes now reachable from s via v can all be marked reachable by starting a depth first search from v. The algorithm will stop at the first edge that allows t to be reached. The edge with the smallest weight in this path to t will be the edge with weight w_i added during this iteration. There is not a path from s to t in the input graph with a larger smallest weighted edge because t was not reachable when only the larger edges were added. Hence, this path is a path to t in the graph whose smallest weighted edge is the largest. This is the required output of this subroutine.

Running Time: Even though the algorithm for finding the path with the largest smallest edge runs depth-first search for each weight w_i, because the work done before is reused, no node in the process is marked reached more than once and hence no edge is traversed more than once. It follows that this process requires only $\mathcal{O}(m)$ time, where m is the number of edges. This time, however, is dominated by the time $\mathcal{O}(m \log m)$ to sort the edges.

Code:

algorithm $LargestShortestWeight\,(G, s, t)$

$\langle$***pre−cond***$\rangle$**:** G is a weighted directed (augmenting) graph. s is the source node. t is the sink.

$\langle$***post−cond***$\rangle$**:** P specifies a path from s to t whose smallest edge weight is as large as possible. $\langle u, v\rangle$ is its smallest weighted edge.

begin
  Sort the edges by weight from largest to smallest
  $G' =$ graph with no edges
  mark s reachable
  loop
    $\langle$***loop−invariant***$\rangle$**:** Every node reachable from s in G' is marked reachable.
    exit when t is reachable
    $\langle u, v\rangle =$ the next largest weighted edge in G

Add $\langle u, v \rangle$ to G'
if(u is marked reachable and v is not) then
Do a depth-first search from v marking all reachable nodes not marked before.
end if
end loop
P = path from s to t in G'
return(P, $\langle u, v \rangle$)
end algorithm

Running Time of Steepest Ascent: How many times must the network flow algorithm augment the flow in a path when the path chosen is that whose augmentation capacity is the largest possible?

Decreasing the Remaining Distance by Constant Factor: The flow starts out as zero and may need to increase be as large as $\mathcal{O}(m \cdot 2^{\ell})$ when there are m edges with ℓ bit capacities. We would like the number of steps to be not exponential but linear in ℓ. One way to achieve this is to ensure that the current flow doubles each iteration. This, however, is likely not to happen. Another possibility is to turn the measure of progress around. After the i^{th} iteration, let R_i denote the remaining amount that the flow must increase. More formally, suppose that the maximum flow is $rate_{max}$ and that the rate of the current flow is $rate(F)$. The remaining distance is then $R_i = rate_{max} - rate(F)$. We will show that the amount w_{min} by which the flow increases is at least some constant fraction of R_i.

Bounding The Remaining Distance: The funny thing about this measure of progress, is that the algorithm does not know what the maximum flow $rate_{max}$ is. It is only needed as part of the analysis. We must bound how big the remaining distance, $R_i = rate_{max} - rate(F)$, is. Recall that the augmentation graph for the current flow is constructed so that the augmentation capacity of each edge gives the amount that the flow through this edge can be increased by. Hence, just as the sum of the capacities of the edges across any cut $C = \langle U, V \rangle$ in the network, acts as an upper bound to the total flow possible, the sum of the augmentation capacities of the edges across any cut $C = \langle U, V \rangle$ in the augmentation graph, acts as an upper bound to the total amount that the current flow can be increased.

Choosing a Cut: We need to choose which cut we will use. (This is not part of the algorithm.) As before, the natural cut to use comes out of the algorithm that finds the path from s to t. Let $w_{min} = w_i$ denote the smallest augmentation capacity in the path whose smallest augmentation capacity is largest. Let $G_{w_{i-1}}$ be the graph created from the augmenting graph by deleting all edges whose augmentation capacities are smaller or equal to w_{min}. This is the last graph that the algorithm which finds the augmenting path considers before adding the edge with weight w_{min} that connects s and t. We know that there is not a path from s to t in $G_{w_{i-1}}$ or else there would be an path in the augmenting graph whose smallest augmenting capacity was larger then w_{min}. Form the cut $C = \langle U, V \rangle$ by letting U be the set of all the nodes reachable from s in $G_{w_{i-1}}$ and letting V be those that are not. Now consider any edge in the augmenting graph that crosses this cut. This edge cannot be in the graph $G_{w_{i-1}}$ or else it would be crossing from a node in U that is reachable from s to a node that is not reachable from s, which is a contradiction. Because this edge has been deleted in $G_{w_{i-1}}$, we know that its aug-

mentation capacity is at most w_{min}. The number of edges across this cut is at most the number of edges in the network, which has be denoted by m. It follows that the sum of the augmentation capacities of the edges across this cut $C = \langle U, V \rangle$ is at most $m \cdot w_{min}$.

Bounding The Increase, $w_{min} \geq \frac{1}{m} R_i$: We have determined that the remaining amount that the flow needs to be increased, $R_i = rate_{max} - rate(F)$, is at most the sum of the augmentation capacities across the cut C, which is at most $m \cdot w_{min}$, that is, $R_i \leq m \cdot w_{min}$. Rearranging this gives that $w_{min} \geq \frac{1}{m} R_i$.

The Number of Iterations: If the flow increases each iteration by at least $\frac{1}{m}$ times the remaining amount R_i, it decreases the remaining amount, giving that $R_{i+1} \leq R_i - \frac{1}{m} R_i$. You might think that it follows that the maximum flow is obtained in only m iterations. This would be true if $R_{i+1} \leq R_i - \frac{1}{m} R_0$. However, it is not because the smaller R_i gets, the smaller it decreases by. One way to bound the number of iterations needed is to note that $R_i \leq (1 - \frac{1}{m})^i R_0$ and then to either bound logarithms base $(1 - \frac{1}{m})$ or to know that $\lim_{m \to \infty} (1 - \frac{1}{m})^m = \frac{1}{e} \approx \frac{1}{2.17}$. However, I prefer the following method. As long as R_i is big, we know that it decreases by a lot. After some I^{th} iteration, say that R_i is still relatively big when it is still at least $\frac{1}{2} R_I$. As long as this is the case, R_i decrease by at least $\frac{1}{m} R_i \geq \frac{1}{2m} R_I$. After m such iterations, R_i would decrease from R_I to $\frac{1}{2} R_I$. The only reason that it would not continue to decrease this fast is if it already had decreased this much. Either way, we know that every m iterations, R_i decreases by a factor of two.

This process may make you think of what is known as zeno's paradox. If you cut the remaining distance in half and then in half again and so on, then though you get very close very fast, you never actually get there. However, if all the capacities are integers then all values will be integers and hence when R_i decreases to be less that one, it must in fact be zero, giving us the maximum flow.

Initially, the remaining amount $R_i = rate_{max} - rate(F)$ is at most $\mathcal{O}(m \cdot 2^\ell)$. Hence, if it decreases by at least a factor of two each m iterations, then after mj iterations, this amount is at most $\mathcal{O}(\frac{m \cdot 2^\ell}{2^j})$. This reaches one when $j = \mathcal{O}(\log_2(m \cdot 2^\ell)) = \mathcal{O}(\ell + \log m)$ or $\mathcal{O}(m\ell + m \log m)$ iterations. If your capacities are real numbers, then you will be able to approximate the maximum flow to within ℓ' bits of accuracy in another $m\ell'$ iterations.

Bounding the Running Time: We have determined that each iteration takes $m \log m$ time and that only $\mathcal{O}(m\ell + m \log m)$ iterations are required. It follows that this steepest ascent network flow algorithm runs in time $\mathcal{O}(\ell m^2 \log m + m^2 \log^2 m)$.

Fully Polynomial Time: A lot of work was spent finding an algorithm that is what is known as *fully polynomial*. This requires that the number of iterations be polynomial in the number of values and does not depend at all on the values themselves. Hence, if you charge only one time step for addition and subtraction, even if the capacities are strange things like $\sqrt{2}$, then the algorithm gives the exact answer (at least symbolically) in polynomial time. My father, Jack Edmonds, and a colleague, Richard Karp, developed such an algorithm in 1972. It is a version of the original Ford-Fulkerson algorithm. However, in this, each iteration, the path from s to t in the augmenting graph with the smallest number of edges is augmented. This algorithm iterates at most $\mathcal{O}(nm)$ times, where n is the number of nodes and m the number of edges. In practice, this is slower than the $\mathcal{O}(m\ell)$ time steepest ascent algorithm.

Exercise 16.1.1 *Could we use binary search on the weights w_{min} to find the critical weight (see Section* **??***) and if so would it be faster? Why?*

16.2 Linear Programming

When I was an undergraduate, I had a summer job with a food company. Our goal was to make cheap hot dogs. Every morning we got the prices of thousands of ingredients: pig hearts, sawdust, etc. Each ingredient has an associated variable indicating how much of it to add to the hot dogs. There are thousands of linear constraints on these variables: so much meat, so much moisture, and so on. Together these constraints specify which combinations of ingredients constitute a "hot dog." The cost of the hot dog is a linear function of what you put into it and their costs. The goal is to determine what to put into the hot dogs that day to minimize the cost. This is an example of a general class of problems referred to as *linear programs*.

Formal Specification: A linear program is an optimization problem whose constraints and objective functions are linear functions. The goal is to find a setting of variables that optimizes the objective function, while respecting all of the constraints.

Precondition: We are given one of the following instances.

Instances: An input instance consists of (1) a set of linear constraints on a set of variables and (2) a linear objective function.

Postcondition: The output is a solution with minimum cost and the cost of that solution.

Solutions for Instance: A solution for the instance is a setting of all the variables that satisfies the constraints.

Measure Of Success: The cost or value of a solutions is given by the objective function.

Example 16.2.1:

maximize

$$7x_1 - 6x_2 + 5x_3 + 7x_4$$

subject to

$$3x_1 + 7x_2 + 2x_3 + 9x_4 \leq 258$$
$$6x_1 + 3x_2 + 9x_3 - 6x_4 \leq 721$$
$$2x_1 + 1x_2 + 5x_3 + 5x_4 \leq 524$$
$$3x_1 + 6x_2 + 2x_3 + 3x_4 \leq 411$$
$$4x_1 - 8x_2 - 4x_3 + 4x_4 \leq 685$$

Matrix Representation: A linear program can be expressed very compactly using matrix algebra. Let n denote the number of variables and m the number of constraints. Let a denote the row of n coefficients in the objective function, M denote the matrix with m rows and n columns of coefficients on the left hand side of the constraints, let b denote the column of m coefficients on the right hand side of the constraints, and finally let x denote the column of n variables. Then the goal of the linear program is to maximize $a \cdot x$ subject to $M \cdot x \leq b$.

Network Flows: The network flows problem can be expressed as instances of linear programming. See Exercise 16.2.1.

The Euclidean Space Interpretation: Each possible solution, giving values to the variables $x_1, \ldots, x_n$, can be viewed as a point in n dimensional space. This space is easiest to view when there are only two or three dimensions, but the same ideas hold for any number of solutions.

Constraints: Each constraint specifies a boundary in space, on one side of which a valid solution must lie. When $n = 2$, this constraint is a one-dimensional line. See Figure 16.1. When $n = 3$, it is a two-dimensional plane, like the side of a box. In general, it is an $n-1$ dimensional space. The space bounded by all of the constraints is called a *polyhedral*.

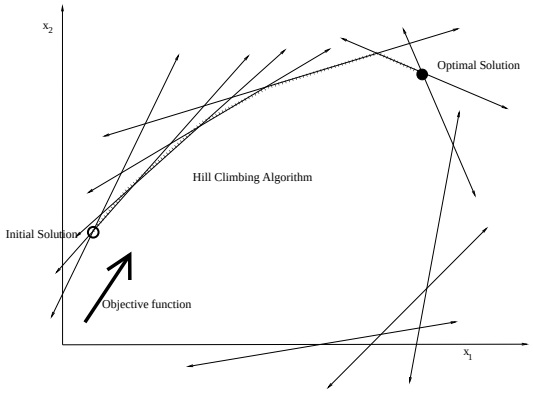


Figure 16.1: The Euclidean space representation of a linear program with $n = 2$.

Vertexes: The boundary of the polyhedral is defined by many *vertexes* where a number of constraints intersect. When $n = 2$, pairs of line-constraints intersect at a vertex. See Figure 16.1. For $n = 3$, three sides of a box define a vertex (corner). In general, it requires n constraints to intersect to define a single vertex. This is because saying that the solution is on the constraint, says that the linear equation meets with equality and not with "less than or equal to." Then recall that n linear equations with n unknowns are sufficient to specify a unique solution.

The Objective Function: The objective function gives a direction in Euclidean space. The goal is to find a point in the bounded polyhedral that is the furthest in this direction. The best way to visualize this is to rotate the Euclidean space so that the objective function points straight up. The goal is to find a point in the bounded polyhedral that is as high as possible.

A Vertex is an Optimal Solution: As you can imagine from looking at Figure 16.1, if there is a unique solution, it will be at a vertex where n constraints meet. If there is a whole region of equivalently optimal solutions, then at least one of them will be a vertex. Our search for an optimal solution will focus on these vertices.

The Hill Climbing Algorithm: The obvious algorithm simply climbs the hill formed by the outside of the bounded polyhedral until the top is reached. In defining a hill climbing algorithm for linear programming we just need to devise a way to find an initial valid solution and to define what constitutes a "step" to a better solution.

A Step: Suppose by the loop invariant, we have a solution that in addition to being valid, it is also a vertex of the bounding polyhedral. More formally, the solution satisfies all of the constraints and meets n of the constraints with equality. A step will involve climbing along the edge (one dimensional line) between two adjacent vertices. This involves *relaxing* one of the constraints that is met with equality so that it no longer is met with equality and *tightening* one of the constraints that was not met with equality so that it now is met with equality. This is called is called *pivoting* out one equation

and in another. The new solution will be the unique solution that satisfies with equality the n presently selected equations. Of course, each iteration such a step can be take only if it continues to satisfy all of the constraints and improves the objective function. There are fast ways of finding a good step to take. However, even if you do not know these, there are only $n \cdot m$ choices of "steps" to try, when there are n variables and m equations.

Finding an Initial Valid Solution: If we are lucky, the origin is a valid solution. However, in general finding some valid solution is itself a challenging problem. Our algorithm to do so will be an iterative algorithm that includes the constraints one at a time. Suppose we have vertex solution that satisfies all of the constraints in Example 16.2.1 except the last one. We will then treat the negative of this next constraint as the objective function, namely $-4x_1 + 8x_2 + 4x_3 - 4x_4$. We will run our hill climbing algorithm, starting with the vertex we have until, we have a vertex solution that maximizes this new objective function subject to the first i equations. This is equivalent to minimizing the objective $4x_1 - 8x_2 - 4x_3 + 4x_4$. If this minimum is less that 685, then we have found a vertex solution that satisfies the first $i + 1$ equation. If not, then we determined that no such solution exists.

No Small Local Maximum: To prove that the algorithm eventually finds a global maximum, we must prove that it will not get stuck in a small local maximum.

Convex: Because the bounded polyhedral is the intersection of straight cuts, it is what we call *convex*. More formally, this means that the line between any two points in the polyhedral are also in the polyhedral. This means that there cannot be two local maximum points, because between these two hills there would need to be a valley and a line between two points across this valley would be outside the polyhedral.

The Primal-Dual Method: The primal dual method formally proves that a global maximum will be found. Given any linear program, defined by an optimization function and a set constraints, there is a way of forming its *dual* minimization linear program. Each solution to this dual acts as a roof or upper bound on how high the primal solution can be. Then each iteration either finds a better solution for the primal or providing a solution for the dual linear program with a matching value. This dual solution witnesses the fact no primal solution is bigger.

Forming the Dual: If the primal linear program is to maximize $a \cdot x$ subject to $Mx \leq b$, then the dual is to minimize $b^T \cdot y$ subject to $M^T \cdot y \geq a^T$. Where b^T, M^T, and a are the transposes formed by flipping the vector or matrix along the diagonal. The dual of example 16.2.1 is

minimize

$$258 + 721y_2 + 524y_3 + 411y_4 + 685y_5$$

subject to

$$3y_1 + 6y_2 + 2y_3 + 3y_4 + 4y_5 \geq 7$$
$$7y_1 + 3y_2 + 1y_3 + 6y_4 - 8y_5 \geq -6$$
$$2y_1 + 9y_2 + 5y_3 + 2y_4 - 4y_5 \geq 5$$
$$9y_1 - 6y_2 + 5y_3 + 3y_4 + 4y_5 \geq 7$$

The dual will have a variable for each constraint in the primal and a constraint for each of its variables. The coefficients of the objective function become the numbers on the right hand side of the inequalities and the numbers on the right hand side

of the inequalities become the coefficients of the objective function. Finally, the maximize becomes a minimize. The dual is the same as the original primal.

Upper Bound: We prove that the value of any solution to the primal linear program is at most the value of any solution to the dual linear program as flows. The value of the primal solution x is $a \cdot x$. The constraints $M^T \cdot y \geq a^T$ can be turned around to give $a \leq y^T \cdot M$. This gives that $a \cdot x \leq y^T \cdot M \cdot x$. Using the constraints $Mx \leq b$, this is at most $y^T \cdot b$. This can be turned around to give $b^T \cdot y$, which is value of the dual solution y.

Running Time: The primal-dual hill climbing algorithm is guaranteed to find the optimal solution. In practice, it works quickly (though for my summer job, the computers would crank for hours.) However, there is no known hill climbing algorithm that is guaranteed to run in polynomial time.

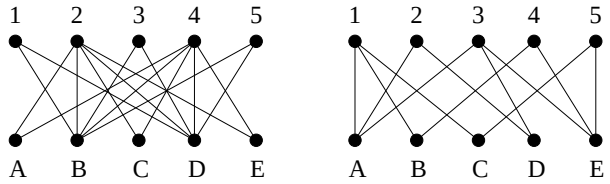
There is another algorithm that solves this problem, called the *Ellipsoid Method*. Practically, it is not as fast, but theoretically it provably runs in polynomial time.

Exercise 16.2.1 *Express the network flow instance in Figure* **??** *as a linear program.*

16.3 Exercises

Exercise 16.3.1 *(See solution in Section* **??***) Let $G = (L \cup R, E)$ be a bipartite graph with nodes L on the left and R on the right. A matching is a subset of the edges so that each node appears at most once. For any $A \subseteq L$, let $N(A)$ be the neighborhood set of A, namely $N(A) = \{v \in R \mid \exists u \in A \text{ such that } (u, v) \in E\}$. Prove Hall's Theorem which states that there exists a matching in which every node in L is matched if and only iff $\forall A \subseteq L,\ |A| \leq |N(A)|$.*

1. *For each of the following two bipartite graphs, either give a short witness to the fact that it has a perfect matching or to the fact that it does not. Use Hall's Theorem in your explanation as to why a graph does not have a matching. No need to mention flows or cuts.*



2. *$\Rightarrow$: Suppose there exists a matching in which every node in L is matched. For $u \in L$, let $M(u) \in R$ specify one such matching. Prove that $\forall A \subseteq L,\ |A| \leq |N(A)|$.*

3. *Look at both the slides and section 19.5 of the notes. It describes a network with nodes $\{s\} \cup L \cup R \cup \{t\}$ with a directed edge from s to each node in L, the edges E from L to R in the bipartite graph directed from L to R, and a directed edge from each node in R to t. The notes gives each edge capacity 1. However, The edges $\langle u, v \rangle$ across the bipartite graph could just as well be given capacity ∞.*

   Consider some cut (U, V) in this network. Note U contains s, some nodes of L, and some nodes of R, while V contains the remaining nodes of L, the remaining nodes of R, and t. Assume that $\forall A \subseteq L,\ |A| \leq |N(A)|$. Prove that the capacity of this cut, i.e. $cap(U, V) = \sum_{u \in U} \sum_{v \in V} c_{\langle u,v \rangle}$, is at least $|L|$.

4. *$\Leftarrow$: Assume that $\forall A \subseteq L,\ |A| \leq |N(A)|$ is true. Prove that there exists a matching in which every node in L is matched. Hint: Use everything you know about Network Flows.*

5. *Suppose that there is some integer $k \geq 1$ such that every node in L has degree at least k and every node in R has degree at most k. Prove that there exists a matching in which every node in L is matched.*

Chapter 21

Reductions and NP-Completeness

21.1 An Algorithm for Bipartite Matching using the Network Flow Algorithm

Up to now we have been justifying our belief that certain computational problems are difficult by reducing them to other problems believed to be difficult. Here, we will give an example of the reverse, by proving that the problem *Bipartite Matching* can be solved easily by reducing it to the Network Flows problem, which we already know is easy because we gave an polynomial time algorithm for it in Section 16.

Bipartite Matching: Bipartite matching is a classic optimization problem. As always, we define the problem by given a set of instances, a set of solutions for each instance, and a cost for each solution.

Instances: An input instance to the problem is a bipartite graph. A bipartite graph is a graph whose nodes are partitioned into two sets U and V and all edges in the graph go between U and V. See the first figure in Figure **??**.

Solutions for Instance: Given an instance, a solution is a matching. A matching is a subset M of the edges so that no node appears more than once in M. See the last figure in Figure **??**.

Cost of a Solution: The cost (or success) of a matching is the number of pairs matched. It is said to be a perfect matching if every node is matched.

Goal: Given a bipartite graph, the goal of the problem is to find a matching that matches as many pairs as possible.

Network Flows: Network Flow is another example of an optimization problem that involves searching for a best solution from some large set of solutions.

Instances: An instance $\langle G, s, t\rangle$ consists of a directed graph G and specific nodes s and t. Each edge $\langle u, v\rangle$ is associated with a positive capacity $c_{\langle u,v\rangle}$.

Solutions for Instance: A solution for the instance is a *flow* F which specifies a flow $F_{\langle u,v\rangle} \leq c_{\langle u,v\rangle}$ through each edges of the network with no leaking or additional flow at any node.

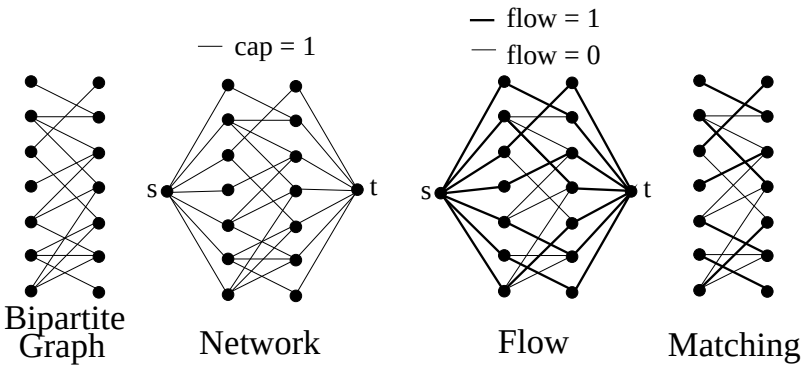


Figure 21.1: The first figure is the bipartite graph given as an instance to Bipartite matching. The next is the network that it is translated into. The next is a flow through this network. The last is the matching obtained from the flow.

Measure Of Success: The cost (or success) of a flow is the the amount of flow out of node s.

Goal: Given an instance $\langle G, s, t\rangle$, the goal is to find an optimal solution, that is, a maximum flow.

Bipartite Matching $\leq_{poly}$ Network Flows: We go through the same steps as before.

3) Direction of Reduction and Code: We will now design an algorithm for Bipartite Matching given an algorithm for Network Flows.

4) Look For Similarities: A matching decides which edges to keep and a flow decides which edges to put flow though. This similarity suggests keeping the edges that have flow through them.

5) InstanceMap, Translating the Bipartite Graphs into a Network: Our algorithm for Bipartite Matching takes as input a bipartite graph $G_{bipartite}$. The first step is to translate this into a network $G_{network} = InstanceMap(G_{bipartite})$. See the first two figures in Figure **??**. The network will have the nodes U and V from the bipartite graph and for each edge $\langle u, v\rangle$ in the bipartite graph, the network has a directed edge $\langle u, v\rangle$. In addition, the network will have a source node s with a directed edge from s to each node $u \in U$. It will also have a sink node t with a directed edge from each node $v \in V$ to t. Every edge out of s and every into t will have capacity one. The edges $\langle u, v\rangle$ across the bipartite graph could be given capacity one as well, but they could just as well be given capacity ∞.

6) SolutionMap, Translating a Flow into an Matching: When the Network Flows algorithm finds a flow S_{flow} through the network, our algorithm must translate this flow into a matching $S_{matching} = SolutionMap(S_{flow})$. See the last two figures in Figure **??**.

SolutionMap: The translation puts the edge $\langle u, v\rangle$ in the matching if there is a flow of one through the corresponding edge in the network and not if there is no flow in the edge.

Warning: Be careful to map *every* possible flow to a matching. The above mapping is ill defined when there is a flow of $\frac{1}{2}$ through an edge. This needs to be fixed and could be quite problematic.

Integer Flow: Luckily, Exercise **??** proves that if all the capacities in the given network are integers, then the algorithm always returns a solution in which the flow through each edge is an integer. Given that our capacities are all one, each edge will either have a flow of zero or of one. Hence, in our translation, it is well-defined whether to include the edge $\langle u, v \rangle$ in the matching or not.

7) Valid to Valid: Here we must prove that if the flow S_{flow} is valid than the matching $S_{matching}$ is also valid.

Each u Matched At Most Once: Consider a node $u \in U$. The flow into u can be as most one because there is only one edge into it and it has capacity one. For the flow to be valid, the flow out of this node must equal that in. Hence, it too can be at most one. Because each edge out of u either has flow zero or one, if follows that at most one edge out of u has flow. We can conclude that u is matched to at most one node $v \in V$.

Each v Matched At Most Once: See Exercise **??**

Cost to Cost: To be sure that the matching we obtain contains the maximum number of edges, it is important that the cost of the matching $S_{matching} = SolutionMap(S_{flow})$ equals the cost of the flow. The cost of the flow is the amount of flow out of node s, which equals the flow across the cut $\langle U, V \rangle$, which equals the number of edges $\langle u, v \rangle$ with flow of one, which equals the number of edges in the matching, which equals the cost of the matching.

8) ReverseSolutionMap: The reverse mapping from each matching $S_{matching}$ to a valid flow $S_{flow} = ReverseSolutionMap(S_{matching})$ is straight forward. If edge $\langle u, v \rangle$ is in the matching, then put a flow of one from the source s, along the edge $\langle s, u \rangle$ to node u, across the corresponding edge $\langle u, v \rangle$, and then on through the edge $\langle v, t \rangle$ to t.

9) Reverse Valid to Valid: We must also prove that if the matching $S_{matching}$ is valid then the flow $S_{flow} = ReverseSolutionMap(S_{matching})$ is also valid.

Flow in Equals Flow Out: Because the flow is the sum of paths, we can be assured that the flow in equals the flow out of every node except for the source and the sink. Because the matching is valid, each u and each v is matched either zero or once. Hence the flows through the edges $\langle s, u \rangle$, $\langle u, v \rangle$, and $\langle v, t \rangle$ will be at most their capacity one.

Cost to Cost: Again, we need to prove that the cost of the flow $S_{flow} = ReverseSolutionMap(S_{matching})$ is the same as the cost of the matching. See Exercise **??**.

10 & 11: These steps are always the same. $InstanceMap(G_{bipartite})$ maps bipartite graph instances to network flow instances G_{flow} with the same cost. Hence, because algorithm Alg_{flow} correctly solves network flows quickly, our designed algorithm correctly solves bipartite matching quickly.

In conclusion, bipartite matching can be solved in the same time that network flows is solved.

Exercise 21.1.1 *Give a proof for the case where each v is matched at most once.*

Exercise 21.1.2 *Give a proof that the cost of the flow $S_{flow} = ReverseSolutionMap(S_{matching})$ is the same as the cost of the matching*

Exercise 21.1.3 *Section* **??** *constructs three dynamic programming algorithms using reductions. For each of these, carry out the formal steps required for a reduction.*

Exercise 21.1.4 *There is a collection of software packages $S_1, \ldots, S_n$ which you are considering acquiring. For each i, you will gain an overall benefit of b_i if you acquire package S_i. Possibly b_i is negative, for example, if the cost of S_i is greater than the money that will be saved by having it. Some of these packages rely on each other; if S_i relies on S_j, then you will incur an additional cost of $C_{i,j} \geq 0$ if you acquire S_i but not S_j. Unfortunately, S_1 is not available. Provide a polytime algorithm to decide which of $S_2, \ldots, S_n$ you should acquire. Hint: Use max flow / min cut.*