In the last lecture, we looked at the problem of finding the maximum flow in a graph, and how it can be efficiently solved using the Ford-Fulkerson algorithm. We also proved the Min Cut-Max Flow Theorem which states that the size of the maximum flow is exactly equal to the size of the minimum cut in the graph.

In this lecture, we will see how various different problems can be solved by reducing the problem to an instance of the network flow problem.

5.1 Bipartite Matching

A Bipartite Graph $G = (V, E)$ is a graph in which the vertex set V can be divided into two disjoint subsets X and Y such that every edge $e \in E$ has one end point in X and the other end point in Y. A matching M is a subset of edges such that each node in V appears in at most one edge in M.

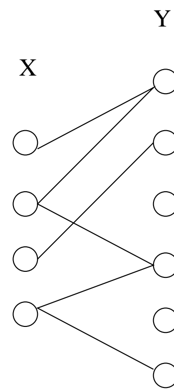


Figure 5.1.1: A bipartite graph

We are interested in matchings of large size. Formally, maximal and maximum matchings are defined as follows.

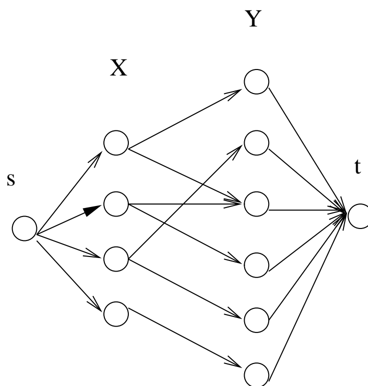
Definition 5.1.1 (Maximal Matching) *A maximal matching is a matching to which no more edges can be added without increasing the degree of one of the nodes to two; it is a local maximum.*

Definition 5.1.2 (Maximum Matching) *A maximum matching is a matching with the largest possible number of edges; it is globally optimal.*

Our goal is to find the maximum matching in a graph. Note that a maximal matching can be found very easily — just keep adding edges to the matching until no more can be added. Moreover, it can be shown that for any maximal matching M, we have that $|M| \geq \frac{1}{2}|M^*|$ where M^* is the maximum matching. Therefore we can easily construct a "2-approximation" to the maximum matching.

The problem of finding the maximum matching can be reduced to maximum flow in the following manner. Let $G(V, E)$ be the bipartite graph where V is divided into X and Y. We will construct a

directed graph $G'(V', E')$, in which V' which contains all the nodes of V along with a source node s and a sink node t. For every edge in E, we add a directed edge in E' from X to Y. Finally we add a directed edge from s to all nodes in X and from all nodes of Y to t. Each edge is given unit capacity.



Let f be an integral flow of G' of value k. Then we can make the following observations:

1. There is no node in X which has more than one outgoing edge where there is a flow.
2. There is no node in Y which has more than one incoming edge where there is a flow.
3. The number of edges between X and Y which carry flow is k.

By these observations, it is straightforward to conclude that the set of edges carrying flow in f forms a matching of size k for the graph G. Likewise, given a matching of size k in G, we can construct a flow of size k in G'. Therefore, solving for maximum flow in G' gives us a maximum matching in G. Note that we used the fact that when edge capacities are integral, Ford-Fulkerson produces an integral flow.

Let us now analyze the running time of this algorithm. Constructing the graph G' takes $\mathcal{O}(n+m)$ time where $n = |V|$ and $m = |E|$. The running time for the Ford-Fulkerson algorithm is $\mathcal{O}(m'F)$ where m' is the number of edges in E' and $F = \sum_{e \in \delta(s)}(c_e)$. In case of bipartite matching problem, $F \leq |V|$ since there can be only $|V|$ possible edges coming out from source node. So the total running time is $\mathcal{O}(m'n) = \mathcal{O}((m+n)n)$.

An interesting thing to note is that at any iteration of the algorithm, any s-t path in the residual graph will have alternating matched and unmatched edges. Such paths are called **alternating paths**. This property can be used to find maximum matchings even in general graphs.

5.1.1 Perfect Matching

A perfect matching is a matching in which each node has exactly one edge incident on it. One possible way of finding out if a given bipartite graph has a perfect matching is to use the above algorithm to find the maximum matching and checking if the size of the matching equals the number of nodes in each partition. There is another way of determining this, using Hall's Theorem.

Theorem 5.1.3 *A Bipartite graph G(V,E) has a Perfect Matching iff for every subset $S \subseteq X or S \subseteq Y$, the size of the neighbors of S is at least as large as S, i.e* $|\Gamma(S)| \geq |S|$

This theorem can be proven using induction. We will not discuss the proof in detail here.

5.2 Scheduling Problems

We will now see how the Max Flow algorithm can be used to solve certain kinds of scheduling problems. The first example we will take will be of scheduling jobs on a machine.

Let $J = \{J_1, J_2, ..., J_n\}$ be the set of jobs, and $T = \{T_1, T_2, .., T_k\}$ be slots available on a machine where these jobs can be performed. Each job J has a set of valid slots $S_j \subseteq T$ when it can be scheduled. The constraint is that no two jobs can be scheduled at the same time. The problem is to find the largest set of jobs which can be scheduled.

This problem can be solved by reducing it to a bipartite matching problem. For every job, create a node in X, and for every timeslot create a node in Y. For every timeslot T in S_j, create an edge between J and T. The maximum matching of this bipartite graph is the maximum set of jobs that can be scheduled.

We can also solve scheduling problems with more constraints by having intermediate nodes in the graph. Let us consider a more complex problem: A hospital has n doctors, each with a set of vacation days when he or she is available. There are k vacation periods, each spanning multiple contiguous days. Let D_j be the set of days included in the jth vacation period. We need to maximize the assignment of doctors to days (one doctor per day) under the following constraints:

1. Each doctor has a capacity c_i which is the maximum total number of days he or she can be scheduled.

2. For every vacation period, any given doctor is scheduled at most once.

We will solve this problem by network flow. As was done earlier, for every doctor i we create a node u_i and for every vacation day j we create a node v_j. We add a directed edge from start node s to u_i and from v_j to sink t. Other edges and capacities are added as follows to satisfy the above constraints:

- The doctor capacities are modeled as capacities of the edge from the source to the vertex corresponding to the doctor.

- The capacities of edges from nodes v_j to the sink t are all set to 1, to represent choosing one doctor for each vacation day.

- To prevent the doctor to be scheduled more than once in a vacation period, we introduce intermediate nodes. For any doctor i and a vacation period j, we create an intermediate node w_{ij}. We create an edge with unit capacity from u_i to w_{ij}. For every day in the vacation period that the doctor is available, we create an edge from w_{ij} to the node corresponding to that day with unit capacity.

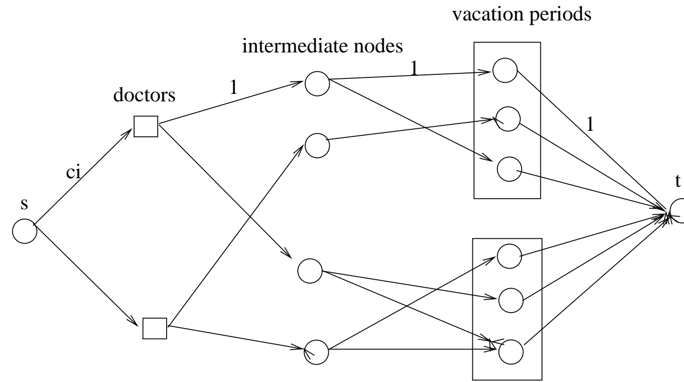


Figure 5.2.2: Scheduling problem with constraints

Let us see if the an integral flow through the graph produces a valid schedule. Since the edge connecting s to the node corresponding to the doctor has the capacity equal to the total availability of the doctor, the flow through the doctor node cannot exceed it. So the first constraint is satisfied. From any doctor to any vacation period the flow is atmost one, since the intermediate node has only one incoming edge of unit capacity. This makes sure that the second criteria is met. So the flow produced satisfies all the constraints.

If k is the size of an integral flow through the graph, then the total number of vacation days which have been assigned is also k, since the edges which connect the nodes corresponding to the vacation days to the sink node have unit capacity each. So the size of the scheduling is equal to the size of the flow. From this we can conclude that the largest valid scheduling is produced by the maximum flow.

5.3 Partitioning Problems

Now we will see how certain kinds of paritioning problems can be solved using the network flow algorithm. The general idea is to reduce it to a min-cut rather than a max-flow instance. The first example we will see is the image segmentation problem.

5.3.1 Image Segmentation

Consider the following simplification of an image segmentation problem. Assume every pixel in an image belongs to either the foreground of an image or the background. Using image analysis techniques based on the values of the pixels its possible to assign probabilites that an individual pixel belongs to the foreground or the background. These probabilites can then be translated to "costs" for assigning a pixel to either foreground or background. In addition, since adjacent pixels are expected to be mapped in the same way, there are costs for assigning a pixel to the foreground and its neighbor to the background. The goal then is to find an assignment of all pixels such that the costs are minimized.

Formally, let f_i be the cost of assigning pixel i to the foreground, b_i be the cost of assigning pixel i to the background, and s_{ij} be the cost of separating neighboring pixels i and j into different regions.

Goal: Find a partition of pixels into S, the set of foreground pixels, and $\overline{S}$, the set of background pixels, such that the global cost of this assignment is minimized.

Let $(S, \overline{S})$ be the partition of pixels into foreground and background respectively. Then the cost of this partition is defined as:

$$cost(S) = \sum_{i \in S} f_i + \sum_{i \notin S} b_i + \sum_{\substack{i \in S, j \notin S \\ i \text{ and } j \text{ are neighbors}}} s_{ij} \tag{5.3.1}$$

We need to find an S which minimizes *cost*. This problem can be reduced to a min-cut max-flow problem. The natural reduction is to a min-cut problem.

Let each pixel be a node, with neighboring pixels connected to each other by undirected edges with capacity s_{ij}. Create additional source and sink nodes, B and F, respectively which have edges to every pixel. The edges from B to each pixel have capacity f_i. The edges from each pixel to F have capacity b_i.

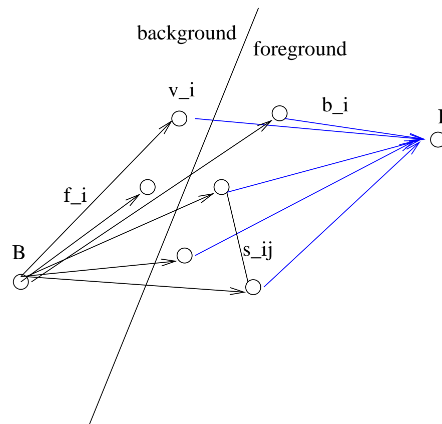


Figure 5.3.3: Image segmentation flow graph

Any cut in the graph will then naturally separate the pixels into two groups — S (the foreground) representing the group of pixels not reachable from B (see the figure above). Now it's easy to see that for any B-F cut, the cost of the cut is exactly equal to the cost associated with the corresponding partition. The min-cut max-flow algorithm will find such a partition of minimum cost.

5.3.2 Image Segmentation cont.

Consider a modification of the original problem. Replace f_i and b_i with values representing the "benefit/value" for assigning the pixel i to either the foreground or background.

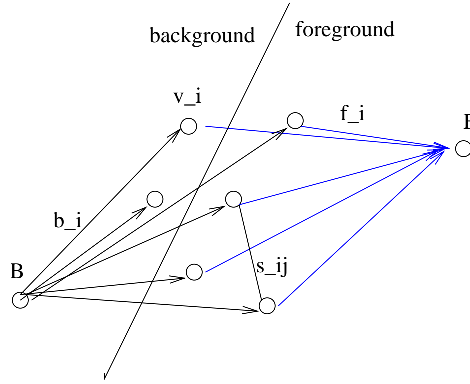


Figure 5.3.4: Flow graph for maximizing benefit

Goal: Find a partition of all pixels into S, the set of foreground pixels, and $\overline{S}$, the set of background pixels, such that the global value is maximized. Here value is defined as

$$val(S) = \sum_{i \in S} f_i + \sum_{i \notin S} b_i - \sum_{\substack{i \in S, j \notin S \\ i \text{ and } j \text{ are neighbors}}} s_{ij} \tag{5.3.2}$$

Since the min-cut approach is a minimization problem, we need to convert it into a minimization problem in order to reduce it to min-cost. Let $(S, \bar{S})$ be the partition of pixels into foreground and background respectively.

We can relate *cost* (defined as per 5.4.1) and *val* by assuming cost to mean "lost benefit":

$$cost(S) = \sum_i (b_i + f_i) - val(S) \tag{5.3.3}$$

Since $\sum_i (b_i + f_i)$ is a constant for any given instance, the problem of maximising val reduces to minimizing the cost.

We can reformulate cost as

$$cost(S) = \sum_{i \in S} b_i + \sum_{i \notin S} f_i + \sum_{\substack{i \in S, j \notin S \\ i \text{ and } j \text{ are neighbors}}} s_{ij} \tag{5.3.4}$$

Comparing this to the previous problem, we see that this is the same as interchanging b_i and f_i in the graph we used to solve the first problem; i.e. the weight of the nodes from B to v_i is set as f_i and weight of v_i to F is set as b_i. Solving this using min-cut will give us the partition which maximizes the benefit.

5.4 Finding maximum flows via linear programming

We now discuss a different way of finding maximum flows in polynomial time.

5.4.1 Linear programming

Linear Programming is a kind of optimization problem where one optimizes a linear multivariate function over linear constraints.

Definition 5.4.1 *A linear program is a collection of linear constraints on some real-valued variables with a linear objective function.*

Example: Maximize $5x + y$ subject to the constraints

$$2y - x \leq 3$$
$$x - y \leq 2$$
$$x, y \geq 0$$

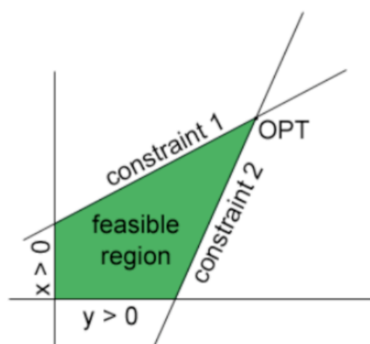


Figure 5.4.5: The feasible region in some linear programming problem.

To get an idea of what this looks like plot the set of pairs (x, y) that satisfy the constraints. See Figure 5.4.5 for a plot of the feasible region.

The *feasible region*, the region of all points that satisfy the constraints, is the shaded region. Any point inside the polytope satisfies the constraints. Our goal is to find a pair that maximizes $5x + y$. The solution is the point $(x, y) = (7, 5)$. Note that the optimum is achieved at a corner point of the feasible region. ■

Definition 5.4.2 (Convex Polytope) *A polytope is an n-dimensional body with "flat" faces. A polytope is called convex if for any two points in the polytope, the line segment joining the two points lies entirely within the polytope. The feasible region of a linear program is a convex polytope.*

Definition 5.4.3 (Extreme point or basic solution) *An extreme point or a basic solution is a corner point of the feasible region. An equivalent definition of an extreme point is any point that cannot be expressed as the convex combination of two or more points in the polytope.*

The optimal solution to a linear program always occurs at an extreme point.

The following are standard ways of writing a linear program:

$$\begin{aligned} \text{maximize } \; c^{\mathrm{T}}x \; \text{ subject to} \\ Ax \leq b \\ x \geq 0 \end{aligned}$$

or,

$$\begin{aligned} \text{minimize } \; c^{\mathrm{T}}x \; \text{ subject to} \\ Ax \geq b \\ x \geq 0 \end{aligned}$$

Here A is an m by n real matrix, $A \in \Re^{m \times n}$, b is an $m \times 1$ real vector, c is an $n \times 1$ real vector, and x is an $n \times 1$ vector of variables. It is easy to see how to convert any given linear program into standard form. For example,

$$\begin{aligned} \text{maximize } \; 3x + 5y - z \; \text{ subject to} \\ x \leq 2 \\ z - 5y \geq 4 \\ 2y + z = 1 \\ y, z \geq 0 \end{aligned}$$

can be written as

$$\begin{aligned} \text{maximize } \; 3(x_1 - x_2) + 5y - z \; \text{ subject to} \\ x_1 - x_2 \leq 2 \\ 5y - z \leq -4 \\ 2y + z \leq 1 \\ -2y - z \leq -1 \\ x_1, x_2, y, z \geq 0 \end{aligned}$$

5.4.2 Solving a Linear Program

Given a linear program, three things can happen:

1. The feasible region is empty
2. The feasible region and the optimal value are unbounded
3. The feasible region is a bounded convex polytope and an optimal value exists.

Generally the goal is to determine whether (1) happens, and if (3) happens, what the optimal value is. These problems are reducible to each other.

Linear programs can be solved in time polynomial in n and m, where n is the number of variables and m is the number of constraints.

There are three ways of solving a linear program:

1. **The simplex method.** This approach starts on an extreme point of the polytope and follows the edges of the polytope from one extreme point to another doing hill-climbing of some sort, until it reaches a local optimum. Note that a local optimum is always a global optimum due to convexity. The worst-case complexity of this algorithm is exponential (the polytope may contain an exponential number of extreme points). However its "average case complexity", defined appropriately, is polynomial-time. In practice, the simplex method is the most frequently used approach to solve linear programs.

2. **The ellipsoid method.** This was the first polynomial time algorithm developed for linear programming. The idea is to start with some ellipsoid containing the feasible region. The center of the ellipsoid is then checked for feasibility. If it is found to be feasible, the algorithm terminates. Otherwise, there is some constraint that is violated. This constraint divides the ellipsoid into two equal halves, one of which entirely contains the feasible region. The algorithm then encloses this half into a smaller ellipsoid and continues. The main insight behind the algorithm is that the size of the ellipsoid is cut by a constant fraction at every iteration. Unfortunately this algorithm is quite slow in practice.

3. **Interior point methods.** These methods start with a feasible point and move towards the optimum within the interior of the polytope, as opposed to the exterior in the simplex method. This approach leads to polynomial time algorithms that are also reasonably efficient in practice.

5.4.3 Max flow via linear programming

Linear Programming is interesting to us because 1. it is solvable in polynomial time, 2. a closely related problem (Integer Programming) is NP-hard, and 3. LP-duality, which we will study a little later, gives us insights connecting different optimization problems. We will see later that we can take any NP-complete problem, reduce it to an instance of Integer Programming, and then try to solve the new problem using the same techniques that solve Linear Programming problems.

For now, we show that max flow can be reduced to linear programming. Consider an instance $G = (V, E)$ of the max flow problem with edge capacities c_e for $e \in E$. Let f_e be the variable representing the amount of flow on edge e. Then the following LP encodes the max flow problem.

$$\text{maximize} \sum_{v:(s,v)\in E} f_{s,v} \ \text{subject to}$$

$$\sum_{u:(u,v)\in E} f_{(u,v)} = \sum_{u:(v,u)\in E} f_{(v,u)} \qquad \forall v \in V, v \neq s, t \qquad \textbf{(Conservation)}$$

$$f_e \leq c_e \qquad \forall e \in E \qquad \textbf{(Capacity constraints)}$$

$$f_e \geq 0 \qquad \forall e \in E$$

Note that the Ford-Fulkerson algorithm is greatly preferable to this approach in terms of running time as well as because the linear program doesn't necessarily return an integral solution, which may sometimes be preferable. Nevertheless this reduction lets us determine quickly what variants of max-flow are likely to be solvable in polynomial time.

References

[1] Eva Tardos, Jon Kleinberg Algorithm Design. *Pearson Education*, 2006.