CS787: Advanced Algorithms

Scribe: Daniel Wong and Priyananda Shenoy | **Lecturer:** Shuchi Chawla
Topic: Applications of Network Flow | **Date:** 9/14/2007

5.1 Introduction and Recap

In the last lecture, we analyzed the problem of finding the maximum flow in a graph, and how it can be efficiently solved using the Ford-Fulkerson algorithm. We also came across the Min Cut-Max Flow Theorem which relates the size of the maximum flow to the size of the minimal cut in the graph.

In this lecture, we will be seeing how various different problems can be solved by reducing the problem to an instance of the network flow problem. The first application we will be seeing is the Bipartite Matching problem.

5.2 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 $e \epsilon E$ has one end in X and the other end in Y. A matching M is a subset of edges such that each node in V appears in at most one edge in M. A Maximum Matching is a matching of the largest size.

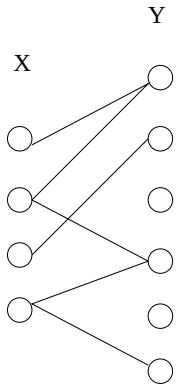

Fig 1 : Bipartite Graph

A clarification about the terminology:

Definition 5.2.1 (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 5.2.2 (Maximum Matching) *A maximum matching is one which is the largest possible; it is globally optimal.*

It can be shown that for any maximal matching $M_i, |M_i| \geq \frac{1}{2}|M|$ where M is the maximum matching. Given this, it is straightforward to obtain 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, 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 weight.

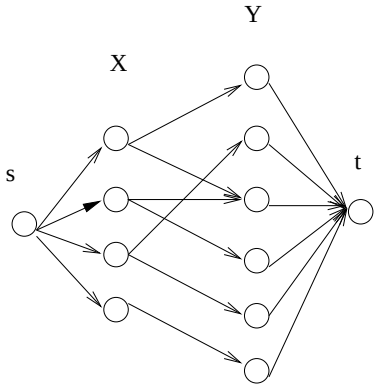

Fig 2: The Bipartite Graph converted into a flow graph

Let f be the maximum 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 the maximum matching for the graph G.

The running time for the Ford-Fulkerson algorithm is $O(mF)$ 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 running time is $O(mn)$ where n is the number of nodes.

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

5.2.1 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 5.2.3 *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 $|\Gamma(S)| \geq |S|$*

We will not discuss the proof of this theorem in the class.

5.3 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: There are k vacation periods each spanning multiple contiguous days. Let D_j be the set of days included in the j^{th} vacation period. There are n doctors in the hospital, each with a set of vacation days when he or she is available. We need to maximize the assignment of doctors to days 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 with unit capacity from start node s to u_i and from v_j to sink t. To include the constraints, the way the graph is constructed in the following way:

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

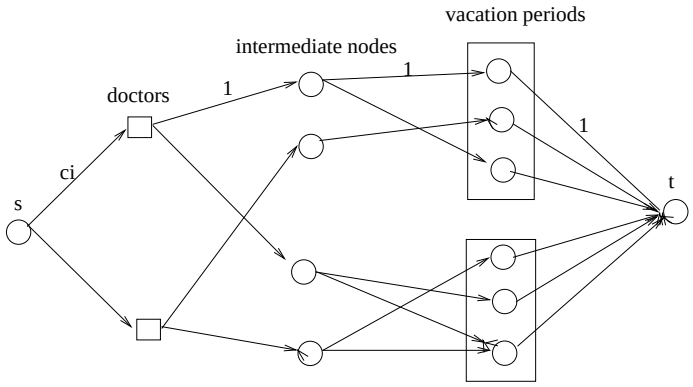

Fig 3: Scheduling Problem with constraints

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.

5.4 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 min-cut rather than max-flow problem. The first example we will see is the Image segmentation problem.

5.4.1 Image Segmentation

Consider the following simplification of the 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, there are costs for segmenting pixels such that pixels from differant regions are adjacent. The goal then is to find an assignment of all pixels such that the costs are minimized.

Let f_i be the cost of assigning pixel i to the foreground.
Let b_i be the cost of assigning pixel i to the background.

Let s_{ij} be the cost of separating neighboring pixels i and j into different regions.

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

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

$$cost(S) = \sum_{i \in S} b_i + \sum_{i \notin S} f_i + \sum_{\substack{i \in S, j \notin S \\ i \text{ and } j \text{ are neighbours}}} s_{ij} \tag{5.4.1}$$

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, s and t, respectively which have edges to every pixel. The edges from s to each pixel have capacity b_i. The edges from each pixel to t have capacity f_i.

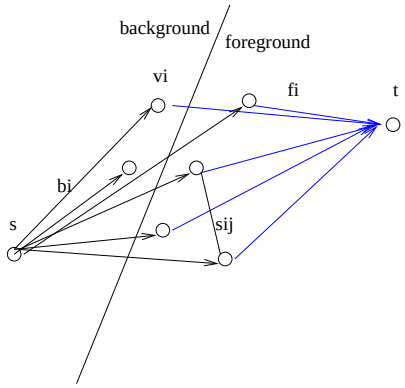

Fig 3: Image segmentation flow graph

Any cut in the graph will then naturally separate the pixels into two groups. With the arrangement the capacity of the cut is the cost associated with that partition. The min-cut max-flow algorithm will find such a partition of minimum cost.

5.4.2 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.

Problem: Find an assigment such that for all pixels, $p \in S$, the set of foreground pixels, or $\bar{S}$, the set of background pixels,such that the global value is maximized. Where value is:

$$val(S) = \sum_{i \in S} f_i + \sum_{i \notin S} b_i - \sum_{\substack{i \in S, j \notin S \\ i \text{ and } j \text{ are neighbours}}} s_{ij} \tag{5.4.2}$$

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":

$$cost(S) = \sum_{i} (b_i + f_i) - val(S) \tag{5.4.3}$$

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

$$cost(S) = \sum_{i \in S} b_i + \sum_{i \notin S} f_i + \sum_{\substack{i \in S, j \notin S \\ i \text{ and } j \text{ are neighbours}}} s_{ij} \tag{5.4.4}$$

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 s to v_i is set as f_i and weight of v_i to t is set as b_i. Solving this using min-cut will give us the partition which maximizes the benefit.

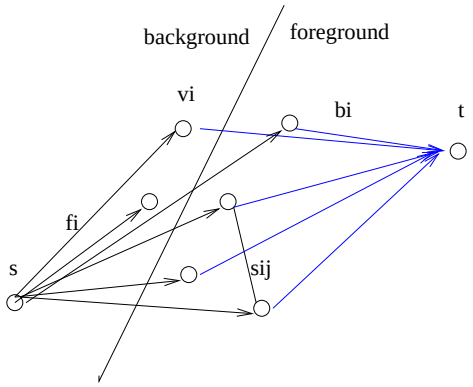

Fig 4: Flow graph for maximizing benefits

5.5 Max-Flow with Costs

Let us consider an extension of the perfect matching problem where different edges have different costs. The naturally leads to the min-cost max-flow problem. In this case, not only do we have to find a max flow through the graph, but also have to find one with the least cost.

Likewise, in the min-cost perfect matching problem, the cost of a perfect matching M is $\sum_{e \epsilon M} c(e)f(e)$ where $f(e)$ is the cost of including the edge e in the matching.

It can be shown that by directing the flow through the minimum cost augementing path first,we will find the maximum flow with the least cost.

References

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