NETWORKS AND MATCHINGS

Michael A. Trick

Author addresses:

Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 15213

Contents

CHAPTER I

Shortest Paths

1. Introduction

Given a graph where each arc has a length and two special nodes s and t, the *shortest path problem* is to find a path from s to t that has minimum total length. This problem is useful by itself in such areas as telecommunications, routing, and robot motion planning. More importantly, though, it is a fundamental building block for more complicated network algorithms.

We will look at three algorithms for this problem. The three have a number of differing characteristics that make each appropriate for a certain type of application.

2. Label Fixing Methods

The first algorithm is due to Dijkstra, in 1959. Dijkstra's algorithm finds the shortest path from a given *source* node, s, to all other nodes in the network. These paths combine to form a *shortest path tree* rooted at the source. To get from the source to any other node in the network, it is only necessary to use edges in this shortest path tree.

This algorithm is iterative, where during the kth step, the kth closest node to the source is added to the shortest path tree. To help us keep track of which node is added, we will assign a tentative distance label $D(i)$ to each node i in the network. When we add a node to the shortest path tree, we will make this distance label permanent (for it will give the value of the shortest path from s to i), and signify that by adding the node to a set T (the nodes of the tree).

Initially, we will set $D(s) = 0$ and $D(i) = \infty$ (computationally, you use some large value). During each iterative step, we will take the node with minimum $D(i)$ that is not in T and add it to T. We will then update the tentative distances to all nodes j adjacent to i. If we know that the distance from s to i is $D(i)$, then the distance from s to j is certainly no more than $D(i) + c_{ij}$. If j has a tentative

distance higher than that, then we can update the tentative distance. So for every j adjacent to i, we set

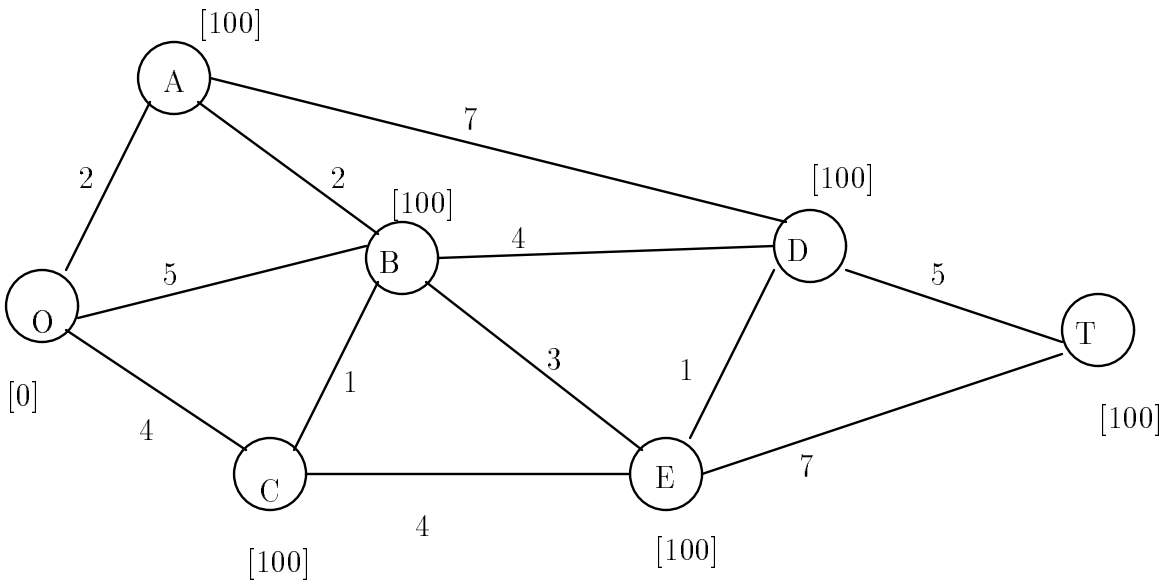
$$D(j) = \min\{d(j), d(i) + c_{ij}\}$$

This completes the iterative step. More formally, the algorithm is as follows:

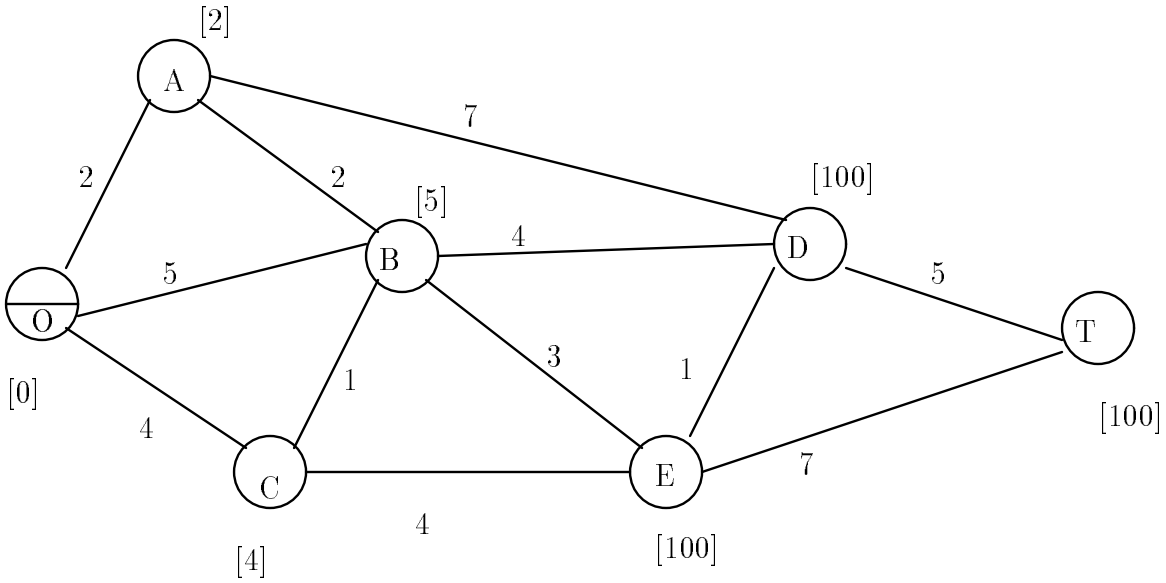
ALGORITHM 1. Dijkstra's Shortest Path

```
ShortestPath(G,s)
  for all i ∈ V
    D[i] = ∞
    parent[i] = NULL
  D[s] = 0
  T = ∅
  repeat
    let i be node with minimum D[i] such that i ∉ T
    if D[i] = ∞
      network is not connected
      T gives the reachable nodes
      return
    for all j adjacent to i
      if D[j] > D[i] + c_ij
        D[j] = D[i] + c_ij
        parent[j] = i
    add i to T
  until T = V.
  return
```

Let's work through an example.

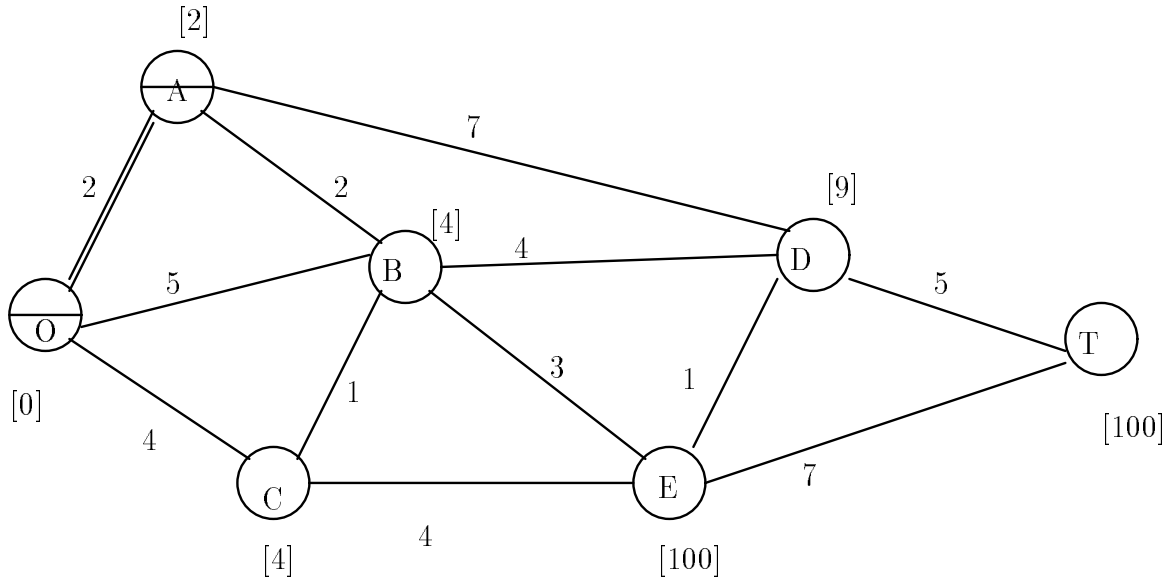


Initial

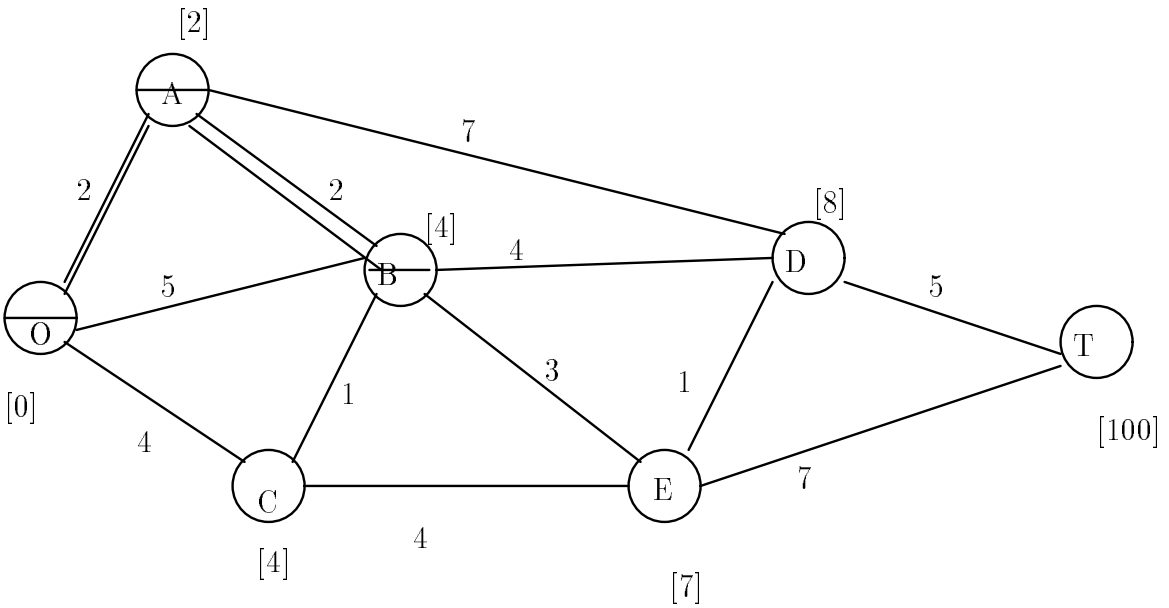


After adding O.

FIGURE I.1. Shortest path example

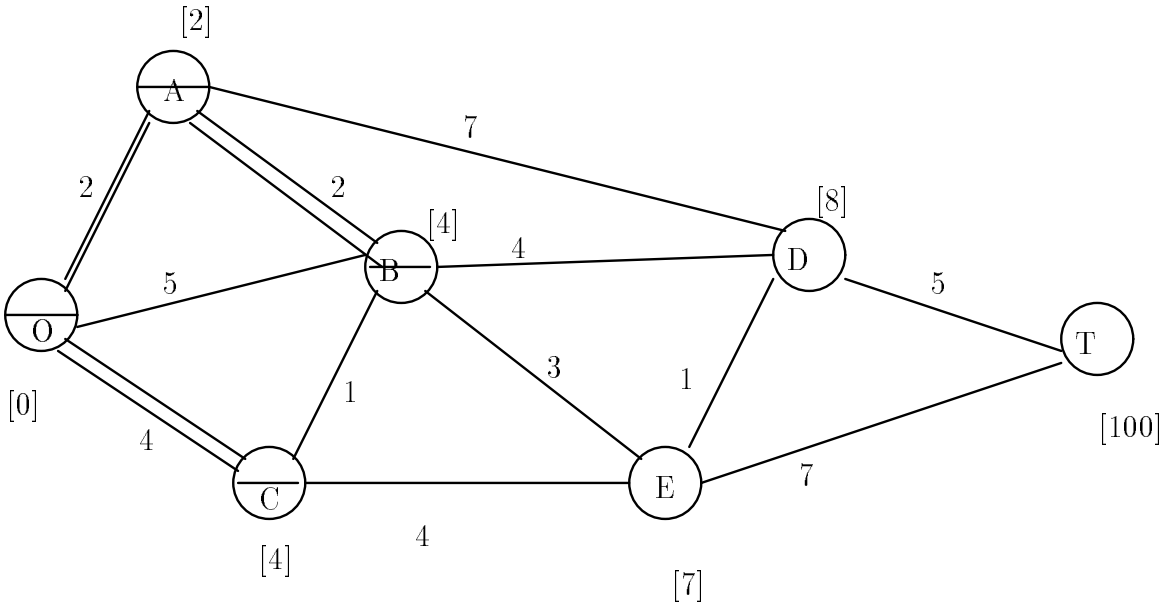


After adding A.

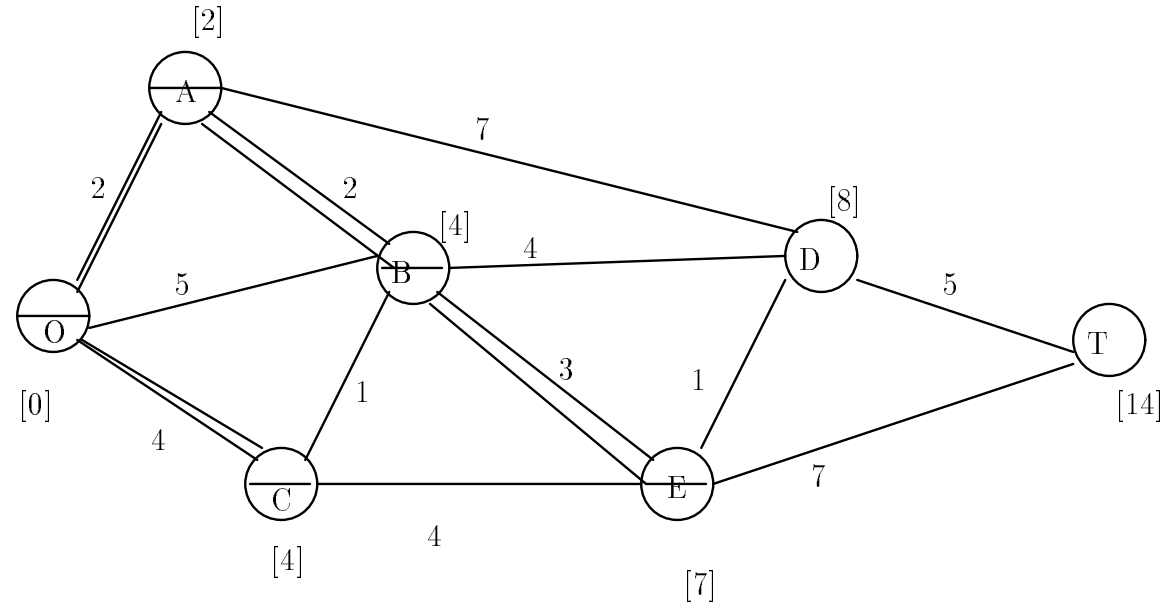


After adding B.

FIGURE I.2. Shortest path example (cont.)

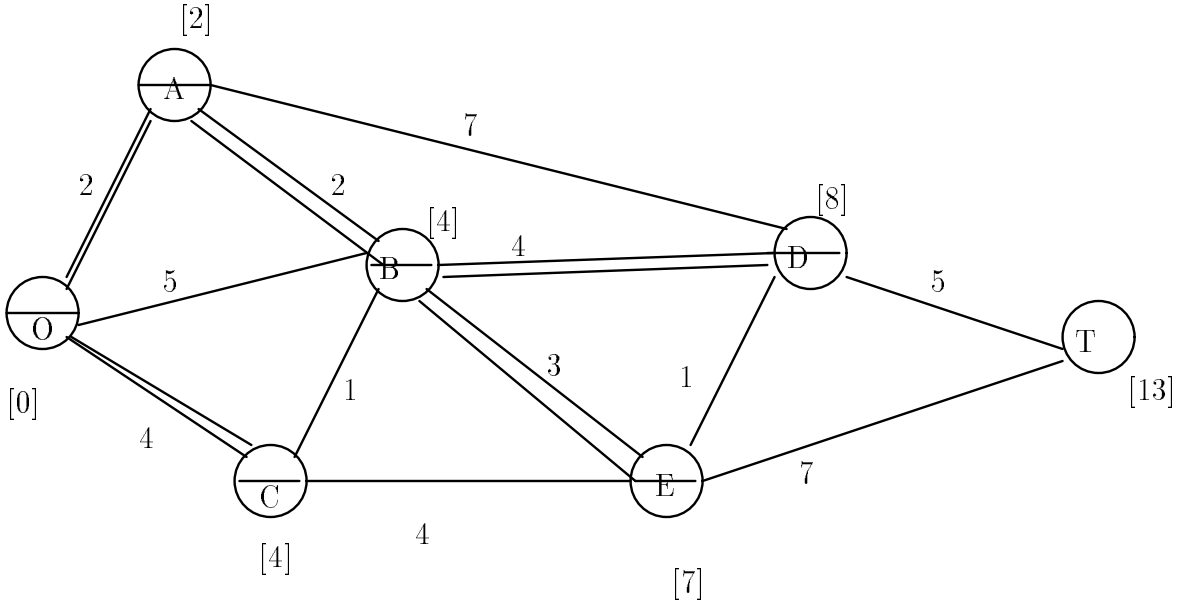


After adding C

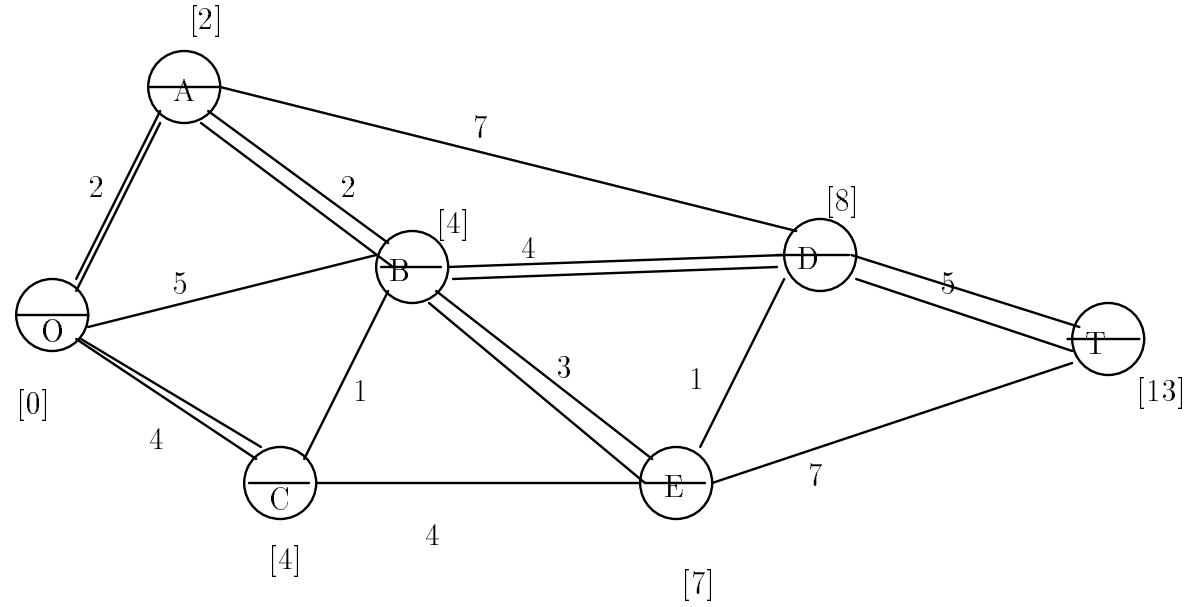


After adding E

FIGURE I.3. Shortest path example (cont.)



After adding D.



Final Solution

FIGURE I.4. Shortest path example (cont.)

Why does Dijkstra's algorithm work?

Problem 1. Show that the node added to T in the kth iteration is the kth closest node to s.

This is a simple proof by induction.

Problem 2. Show that the distances assigned to each node are correct, and hence that the algorithm is correct.

Again, this is a straightforward proof by induction.

Problem 3. What is the time complexity of this algorithm?

Every arc is examined just twice while going through the adjacency lists. Finding the node with minimum $D[i]$ takes $O(|V|)$, and must be done $O(|V|)$ times. Therefore, the total complexity is $O(|E| + |V|^2)$ (all other operations take less time). Actually, it is possible to store the node distances in a heap. This reduces the time to $O(|E| + |V| \log |V|)$.

Problem 4. What happens in this algorithm if some of the costs are negative?

It fails miserably (the induction in problem 1 no longer holds).

3. Label Correcting Methods

The next algorithm uses the same essential equation as that in Dijkstra's algorithm, but uses it in a much less organized way. Although the resulting algorithm is less efficient than Dijkstra's algorithm, the flexibility provided by the lack of organization make it ideal for distributed computation. This algorithm also makes a less stringent assumption about the input graph.

Suppose we have a set of tentative distance labels $D(i)$ and go through the nodes in order. If we ever find an edge (i, j) such that $D(j) > D(i) + c_{ij}$, we can reduce $D(j)$ to be $D(i) + c_{ij}$. If we go through all the nodes and edges without finding such a case, then we can terminate. More formally, we get the algorithm BELLMAN–FORD.

Again, we must ask ourselves two questions: why does this algorithm work and what is the complexity of the algorithm?

Problem 1. Show that at the end of the kth iteration of the while loop, the distance label associated with i gives the minimum length path using at most k edges.

Problem 2. How many iterations through the while loop are required? When can this algorithm fail?

ALGORITHM 2. BELLMAN–FORD(G,s)

```
for all i ∈ V
   D[i] = ∞
   parent[i] = NULL
D[s] = 0
done = FALSE
while not done
   done = TRUE
   for each i ∈ V
      for each edge (i, j)
         if D[j] > D[i] + c_ij
            done = FALSE
            D[j] = D[i] + c_ij
            parent[j] = i
return
```

There should be no need for more than $|V|-1$ iterations, since no patch can be longer than that. If, however, there is a negative cost cycle in the graph, the algorithm can "loop" though that cycle, so the algorithm will still be decreasing labels after $|V|$ iterations. It is then that the algorithm can fail.

The time complexity follows directly from this argument. The total number of iterations is $O(|V|)$, and every edge is used twice in each iteration. The total time is $O(|V||E|)$, which is not as good as Dijkstra's algorithm. The advantages of the Bellman–Ford algorithm are:

(1) can handle negative costs (though not negative cycles),
(2) can recognize negative cycles (if the distances change during iterations $|V|$),
(3) easier to work into a distributed environment,

4. All Pairs Shortest Paths

Our final static algorithm does more than the previous two algorithms. Both Dijkstra's and the Bellman–Ford algorithm find the shortest distance from a given node to every other node. The Floyd–Warshall algorithm finds the shortest distance between every two nodes. Again, this algorithm begins with estimates of the distances between the nodes, in this case arranged in a matrix D_{ij}. Initially, the estimates are the length of the edge between i and j (if no edge exists, then the estimate is set to ∞ or some large number). Then, at iteration k, the matrix is updated to be the length of the shortest path using just nodes $0, 1, \ldots, k$ as

intermediate nodes. If we let D_{ij}^k be the distance matrix for the kth iteration, then initially

$$D_{ij}^{(-1)} = c_{ij}$$

The iterative step replaces this with

$$D_{ij}^{(k+1)} = \min\{D_{ij}^{(k)}, D_{i(k+1)}^{(k)} + D_{(k+1)j}^{(k)}\}$$

This is done for $k = 0, 1, \ldots, |V| - 1$ to give $D_{ij}^{(|V|)}$, the final result. Now the minimum distance from i to j is the ijth entry of this matrix.

You should interpret the right hand side as taking the minimum of going using only nodes $0, 1 \ldots, k$ and going from i to $k+1$ and from $k+1$ to j. The correctness of this algorithm follows from this interpretation.

To determine the complexity, note that a matrix with $O(|V|^2)$ entries is updated $O(|V|)$ times. This gives a total complexity of $O(|V|^3)$. This complexity is worse than the other two algorithms if just one shortest path is required. It is better, however, if all pairs are required.

Problem 1. How can this algorithm be implemented to find the shortest paths?

Use a predecessor matrix, and update that matrix whenever a distance matrix entry is updated.

5. Problems

HOMEWORK PROBLEM 1. *Show that for any graph and source node s, it is possible to choose shortest paths from s to every other node such that the arcs in these paths forms a tree (i.e. has no cycles ignoring directions).*

HOMEWORK PROBLEM 2. *One obvious approach to handling negative costs in Dijkstra's algorithm is to add some large constant to the cost on each arc. Show that this method does not work.*

HOMEWORK PROBLEM 3. *An acyclic graph is a graph with no directed cycles. Give an algorithm for finding the longest path in acyclic graphs. What is the complexity of your algorithm?*

HOMEWORK PROBLEM 4. *A Euclidean graph is an undirected graph with distances on the edges. The nodes can be embedded in the euclidean plane such that the distance on every edge equals the euclidean distance between the end points of the edge. (Note that the graph need not be complete.) Modify Dijkstra's algorithm for this problem with the difference that the unlabeled node chosen at each iteration is the one with minimum $d(i)$ + Euclidean–distance(i, t). We are only interested in the distance from s to t, so we can stop as soon as t is chosen.*

1) Give the algorithm you create and prove its correctness (you can use any results on the unmodified Dijkstra's algorithm).

2) Give an example where this algorithm is a large (more than constant time) improvement on the unmodified algorithm.

3) Give an example where the modified algorithm is no different than the unmodified one. Can it be worse?

CHAPTER II

Maximum Flow

1. Introduction

In this section we examine maximum flow algorithms from their roots in the early 1960's to current algorithms. These algorithms use a variety of techniques that will be useful in studying more complicated problems. These include such fundamental concepts as *duality*, *augmenting paths*, *scaling*, and *amortized analysis*.

2. The Maximum Flow Problem

A *network* consists of a directed graph G with node set V and arc set A. We will assume the arcs are directed, so an arc (i, j) is said to be from i to j. Associated with each arc (i, j) is a capacity $c(i, j)$. We assume all capacities are non–negative.

Consider a network with two distinguished nodes: s (the source) and t (the sink). A *feasible flow* in the network assigns a value $x(i, j)$ to each arc (i, j) such that

- $0 \leq x(i, j) \leq c(i, j)$, and
- for every node except s and t, the amount of flow entering the node equals the flow leaving the node.

The *maximum flow problem* is to find a feasible flow that maximizes the sum of the flows out of s.

Problem 1. Formulate the maximum flow problem as a linear program.

Many possibilities. Let the network be (V, A) and create variables $\{x_a : a \in A\}$. Let v denote the maximum flow. The constraints are:

$$\sum_j \{x_a : a = (j,i)\} - \sum_j \{x_a : a = (i,j)\} = 0 \text{ for all } i, i \neq s, i \neq t$$

$$\sum_j \{x_a : a = (j,s)\} - \sum_j \{x_a : a = (s,j)\} + v = 0$$

$$\sum_j \{x_a : a = (t,j)\} - \sum_j \{x_a : s = (t,j)\} - v = 0$$

$$0 \leq x_a \leq c_a \text{ for all } a \in A.$$

The objective function is to maximize v.

Problem 2. What is the dual of this linear program?

Using the above formulation, create a dual variable u_v for each constraint. The constraints are:

$$u_j - u_i + w_a \geq 0 \text{ for all } a = (i,j) \in A$$

$$u_s - u_t \geq 1$$

$$w_a \geq 0 \text{ for all } a \in A$$

The objective is to minimize $\sum_a c_a w_a$.

Problem 3. What are the complementary slackness conditions?

The straightforward interpretation is that

$$x_a > 0 \Rightarrow u_j - u_i + w_a = 0$$

$$w_a > 0 \Rightarrow x_a = c_a$$

A more complicated answer examines the possibilities for u_i and u_j:
$u_i < u_j$. x_{ij} must be equal to 0.
$u_i = u_j$. There may or may not be positive flow on (i,j).
$u_i > u_j$. $x_{ij} = c_{ij}$.

We shall assume for notational convenience that for every arc (i,j) there is another arc (j,i) (possibly with capacity 0). We will also assume there is at most one arc from i to j for every i and j.

Let us start with an easier problem: the non–zero flow problem. Given a network with capacities, find a feasible flow that sends at least one unit of flow from s to t.

Problem 4. Give an algorithm for this problem.

Many algorithms work. The idea is to find a path from s to t such that every arc has capacity greater than 0. Then a flow can be sent from s to t equal to the minimum capacity of the arcs on the path found. Some possibilites: depth–first search (good for following counterexamples), breadth first search (leads to well–known Edmonds–Karp), maximum augmentation (leads to Edmonds–Karp fat–path algorithm — good homework problem, see [57]*).*

We will call a path found by this algorithm an *augmenting path.*

Suppose we have a flow f through the network and that the flow on the arc from 4 to 5 is 10. Suppose the $c(4,5) = 15$ and $f(5,4) = c(5,4) = 0$. Then, to change the flow, we could send either up to 5 more units of flow from 4 to 5 or remove up to 10 units, which is equivalent to sending 10 units from 5 to 4. In other words, we could replace the current arcs 4,5 and 5,4 with new arcs having capacity 5 and 10 respectively. If we do this for every arc in the network, we create the *auxilliary network.* Suppose we find a flow f^* for this network. The flow $f + f^*$ is then feasible for the original problem.

This gives us our first algorithm for the maximum flow problem:

MAX-FLOW(N,f) N: capacitated network, f the maximum flow
$f = 0$
REPEAT
  Create auxilliary network N' with respect to f
  NONZERO-FLOW(N', f')
  $f = f + f'$
WHILE ($f' \neq 0$)
FINISHED

Problem 5. Can this algorithm go on forever?

No, assuming all capacities are finite. The algorithm augments by at least one unit of flow each iteration.

A more difficult problem is to show that this algorithm terminates with a maximum flow to the original problem. Let S be a subset of V with $s \in S$ and $t \notin S$. Let $T = V - S$ (so $t \in T$). Finally, let $C(S,T)$ be the arcs (i,j) with $i \in S$ and $j \in T$. $C(S,T)$ is called a *cut.* The *capacity of a cut* is the sum of the capacities of the arcs in the cut. Clearly, if the capacity of some cut is k then there is no flow from s to t with value more than k.

Problem 6. Prove the above statement using duality.

Let $u_i = 1, i \in S$; $u_i = 0, i \in T$; $w_{ij} = 1, i \in S, j \in T$; and $w_{ij} = 0$ otherwise. Examining the constraints in Problem 2 shows that this is a dual feasible solution with objective $\sum\{c_{ij}w_{ij} : (i,j) \in C(S,T)\}$. By weak duality, this implies that the maximum flow from s to t can be no more than that value.

Problem 7. Suppose we find a cut of capacity k and a flow of value k. What could we conclude?

We have an optimal flow.

Now, for some cut $C(S,T)$ define the flow across a cut to be the sum of the flows on the arcs in $C(S,T)$ minus the flow on arcs from T to S.

Problem 8. Show that the flow across any cut equals the value of the flow (that is, the amount of flow leaving s).

One method is by induction on the set of nodes in S. By definition this is true for $S = \{s\}$. Suppose it is true for all subsets of S containing s. To show for S, take any node $i \in S, i \neq s$. By conservation of flow at i, it is easy to show that the flow across the cut for S equals the flow across the cut for $S - i$. The result follows by induction.

Problem 9. How can the NONZERO-FLOW algorithm fail?

The only way it can fail is to have no path from s to t in the auxilliary graph.

Problem 10. Show that our MAX-FLOW algorithm terminates in a maximum flow.

Let S be the nodes reachable from s in the auxilliary graph and $T = V - S$. Then every edge in $C(S,T)$ has capacity 0. So every edge $(i,j), i \in S, j \in T$ has flow at its capacity and every edge $(i,j), i \in T, j \in S$ has zero flow. Therefore the flow across the cut equals the capacity of the cut. By Problem 7, this gives an optimal flow.

Problem 11. How long does MAX-FLOW take (how many calls to NONZERO-FLOW)? How can you improve this algorithm?

Let v' the the value of the maximum flow. This algorithm can take up to v' calls to NONZERO–FLOW. This bound is tight as can be seen:

3. Shortest Augmenting Paths

The main idea of the algorithm of Edmonds and Karp is to augment along **shortest** augmenting paths. We now will work out why this algorithm works quickly.

With respect to the current flow f, let $\sigma(i)$ be equal to the length of the shortest augmenting path from s to i, and $\tau(i)$ be the length of the shortest augmenting path from i to t.

Problem 1. Show that if augmentations are done along shortest augmenting paths, then $\sigma(i)$ and $\tau(i)$ are nondecreasing for all i.

We will show for σ, the case for τ is similar. Let σ and σ' be the values before and after the augmenation respectively, and assume $\sigma(i) > \sigma'(i)$ and i has the minimum value for σ' among all such. Let (j,i) be the last arc on the augmenting path that gives $\sigma'(i)$. By such a choice, $\sigma'(j) + 1 = \sigma'(i)$ and $\sigma(j) + 1 \leq \sigma'(i)$.
Claim that (i,j) was on the augmenting path. If not then $\sigma(i) \leq \sigma(j) + 1$ (since (j,i) capacity greater than 0). Combined with the above, this gives $\sigma(i) \leq \sigma(j) + 1 \leq \sigma'(i)$ which is a contradiction.
So (i,j) was on the augmenting path so $\sigma(j) = \sigma(i) + 1$. Substituting above, we get $\sigma(i) + 2 \leq \sigma'(i)$, contradiction.

Define *phase k* to be the augmentations during which $\sigma(t) = k$.

Problem 2. Show that during phase k at most one of (i,j) and (j,i) will be used in an augmenting path.

It is straightforward to see that if both (i,j) and (j,i) appear in a phase, either a σ or a τ had to decrease.

An arc is *critical* for an augmentation if it is on the augmenting path and has flow equal to either 0 or its capacity after the augmentation.

Problem 3. Show that no critical arc will appear in a later augmenting path during the same phase.

Once an arc reaches capacity (as critical arcs do) it can not be used again until an augmentation uses its reversal. By Problem 2, this cannot occur until the next phase.

Problem 4. How many augmentations can there be in a phase? How many phases? How much time per augmentation?

Let $|V| = n$ and $|A| = m$. Each augmentation makes at least one arc critical, so there can be no more than A augmentations in a phase. There are at most V phases, and breadth first search requires $O(A)$ time, so the overall complexity is $O(A^2V)$.

4. Layered Networks

At the beginning of phase k, we can create a *layered network* by placing in layer l all nodes i with $\sigma(i) = l$. An arc (i,j) is placed in the network if and only if $\sigma(j) = \sigma(i) + 1$.

Problem 1. Show that finding a maximum flow in this layered network is equivalent to doing all the augmentations in phase k.

Since clearly all the augmenting paths in the layered network are of length k, and all augmenting paths in the layered network correspond to augmenting paths in the original graph, we need only show that there is no augmenting path in the original graph of length k that does not correspond to an augmenting path in the layered graph. Suppose there is such a path P with edge (i,j) not in the layered graph. Then at the beginning of the phase $\sigma(j) \neq \sigma(i)+1$. But now, with σ', $\sigma'(j) = \sigma'(i)+1$. Also, $\sigma(i)+\tau(i) = k$ and $\sigma'(i)+\tau'(j) = k$. If neither σ or τ decreased then $\sigma(i) = \sigma'(i)$. But this also holds true for $\sigma(j)$, which is a contradiction. Hence no P exists.

The next three algorithms we will discuss (Dinits, MPM, and Tarjan) are all alternative ways of finding a maximum flow in a layered network. Dinits' is the most straightforward.

Problem 2. Apply depth first search to the layered network. What happens after k arcs are examined (where k is the phase)?

After k arcs, we have either reached t (in which case we augment), or we are forced to back up. In the former case, one arc becomes critical, and we can remove it from the network. In the latter case, if we back up from node i we know that no augmenting path can go through i, so we can remove it from the graph.

Problem 3. What is the time complexity of Dinits' algorithm?

In iteration k, there can be at most $V+A = O(A)$ examinations of at most k arcs. The time for phase k is therefore $O(kA)$. There are V phases, so the overall complexity is $O(V^2A)$.

Note that this complexity is better than that for Edmonds and Karp, though Dinits was two years earlier. Edmonds and Karp first presented their work in 1969.

5. Multiple Augmentations

Read [42].

The next two algorithms optimize flow in a layered network. The important properties of this network are that it is acyclic and every node has a layer and every arc is from one layer to the next.

For a node i in the layered network with nodes V and arcs A, each with a capacity $c(i,j)$ and a current flow $f(i,j)$, define its *throughput* $\rho(i)$ to be the minimum of the following two numbers:

$$\sum\{c(j,i) - f(j,i) : (j,i) \in A\}$$

$$\sum\{c(i,j) - f(i,j) : (i,j) \in A\}$$

For s ignore the first number, for t ignore the second.

Let the node with minimum throughput be the *reference* node.

Problem 1. Let r be the reference node in a layered network. Show that $\rho(r)$ units of flow can be sent from s to r and the same from r to t.

To move the flow from r to t, simply assign flows to the arcs out of r in any order, filling up one arc before assigning flow to the next. By the definition of throughput, all flow will be assigned before the arc capacities are exhausted. Now repeat this with the nodes in the next layer. Because the amount of flow at each layer equals the minimum throughput, at no time will a node receive more flow than it is able to pass on to the next layer, so all the flow will eventually end up at t. The argument for s to r is the same, except the arcs are treated as though reversed.

Problem 2. Show that after moving the flow in such a manner, node r can be deleted from the layered network.

After the augmentation, either all the arcs entering r or all the arcs leaving r are saturated (at their capacities). Since this is a layered network, acyclicity implies that they will stay saturated. Therefore, no augmenting path can go through r, so r can be eliminated.

During such an augmentation, some arcs will be assigned flow equal to their capacity. Such an arc is *saturated* and the corresponding movement of flow is called a *saturating push* of flow. An arc assigned flow not equal to its capacity is *unsaturated* and the movement of flow is an *unsaturating push.*

Problem 3. How many unsaturating pushes are there in your solution to Problem 1?

Because we fill up one arc completely before beginning to fill up the next, there is at most one unsaturating push at each node each iteration.

Now consider the following algorithm for a layered network: determine the node throughputs, find the node with minimum throughput, augment flow through that node, delete the node from the graph, update the throughputs, repeat until the graph has no nodes (or all throughputs are 0).

Problem 4. Why is this algorithm correct? What is the complexity of this algorithm?

By problem 1, we can correctly push flow through the network. By problem 2, we can correctly delete the node from the network after the flow update. Repeating the process, after at most $|V|$ iterations, there are no nodes in the network.
To get the complexity, note that there are at most $|V|^2$ nonsaturating pushes and $|A|$ saturating pushes (because once an arc gets saturated, it remains that way. Therefore, the time is $O(|V|^2 + |E|)$ which is $O(|V|^2)$. This gives an $O(|V|^3)$ algorithm for the maximum flow problem (why?).

6. The Wave Algorithm

Read [58].

All the previous algorithms had the following property: at every step of the algorithm, the flow into a vertex equals the flow out of the vertex. Karzanov (1974) introduced the concept of *preflows* that relaxes this property. Define the *excess* at a node i relative to a set of arc values f to be:

$$\sum\{f(j,i) : (j,i) \in A\} - \sum\{f(i,j) : (i,j) \in A\}$$

A *flow* (as we have defined it) has zero excess at every node except s and t. A *preflow* has non–negative excess at every node except s and t. A node with zero excess is said to be *balanced*; a node with positive excess is *unbalanced*. Finally, we say a node is *blocked* with respect to a preflow if there is no augmenting path from it to t; otherwise it is *unblocked*.

Karzanov's algorithm (as modified by Tarjan) begins by making s blocked and then attempts to make all the nodes balanced. Consider the excess at an unbalanced vertex. If the vertex is not blocked, then the flow still has a possibility to get to t, so it should be sent forward (to the next layer). If the vertex is blocked, it has no such chance by going forwards, so it should be sent back towards s by sending it backwards (to the previous layer).

To send excess forward from i we repeat the following step until its excess is 0 (i is balanced) or there is no unsaturated edge (i,j) with j unblocked:

Forward Flow. Let (i,j) be an unsaturated edge with j unblocked. Increase $f(i,j)$ by $\min\{c(i,j) - f(i,j), \text{excess of } i\}$.

To send excess back from i we repeat the following step until its excess is zero:

Backward Flow. Let (j,i) be an edge of positive flow. Decrease $f(j,i)$ by $\min\{f(j,i), \text{excess of } i\}$.

Problem 1. Why must repeated application of *Backward Flow* terminate with excess of zero?

We can set the incoming flow to zero, which clearly has to make the excess nonpositive.

The maximum flow in a layered network algorithm is now the following:

Step 0. (Initialize) For every edge (s, j) set $f(s, j) = c(s, j)$. Set all other flows to zero. s is now blocked.

Step 1. (Send flow forward) Let i be an unblocked, unbalanced node of minimum layer number (s has layer 0, those next to s layer 1 and so on) other than t. If no such node exists, go to Step 2. Attempt to balance the node by sending flow forward. If i is still unbalanced, mark i blocked. Go to Step 1.

Step 2. (Send flow backwards) Let i be a blocked, unbalanced node of maximum layer other than s or t. If no such node exists, Go to Step 3. Attempt to balance the flow by sending flow backwards. Go to Step 2.

Step 3. (Terminate?) If there are no unbalanced nodes, halt. Otherwise go to Step 1.

Problem 2. Which nodes are unbalanced at the end of when Step 2 is called from Step 1? How many iterations of Steps 1 and 2 (that is, how many Step 3) are required?

Only nodes that were marked blocked in the previous applications of Step 1 are now unbalanced. Therefore, there are at most $|V|$ iterations of Step 1 and Step 2 pairs (since each one makes at least one vertex blocked).

Problem 3. How many times will an attempt be made to balance a node (either by sending flow forwards or backwards)?

Once we balance a blocked node, we no longer need to examine that node. We balance an unbalanced node at most once each iteration, therefore the number of balancing tries for a node is $O(|V|)$, for a total of $O(|V|^2)$ balancings.

Problem 4. Someone claims that he has a problem where first the flow on an edge increases, then it decreases, then it increases again. Do you believe him?

No, this cannot happen. An arc (i, j) has flow increased only if j is unblocked and decreased only if j is blocked. Since a blocked nodes remains that way, the flow first increases then decreases on an edges (hence the "wave").

Problem 5. How many increasing and decreasing steps are there?

Each Forward Flow either saturates the edge or terminates an attempt to balance a node. Similarly each Backward Flow step either puts the flow to zero or terminates a balance attempt. Therefore there are at most $O(|V|^2 + 2|A|)$ applications of Forward and Backward Flow.

Problem 6. What is the complexity of this algorithm?

Each call to Forward or Backward Flow takes constant time so the overall complexity is $O(|V|^2)$, giving another $O(|V|^3)$ algorithm.

Problem 7. Compare the algorithms we have covered so far (Ford–Fulkerson, Edmonds–Karp, Dinits, MPM, Tarjan). What are the advantages and disadvantages of each? Which do you think you would work best in practice? How could you test your guesses?

There are many ways to answer this, examining such things as worst–case time complexity, data structures required, advantages and disadvantages for sparse and dense graphs, etc. The only way to tell for sure is to do some sort of computational experiment, but even then the problems solved probably have little to do with real–world problems.

7. Preflow–Push

Read [28].

The past three algorithms find a maximum flow in a layered network. Goldberg and Tarjan ask the question "Is the layered network necessary?" Their insight was to avoid the layered network by working with approximate distance labels. These labels are combined with the preflows of Karzanov to form a very simple and appealing algorithm.

A distance labeling d is a nonnegative integer valued function on the vertices such that $d(t) = 0$, $d(i) > 0$ for all $i \in V$, and $d(j) \geq d(i) - 1$ for every edge (i, j) in the auxilliary network with positive capacity.

Problem 1. Show that τ gives a valid distance labelling. Give another labelling that is valid for any flow.

Clearly, $\tau(t) = 0$ and $\tau(i) > 0$, for $i \neq t$. Consider an edge (i, j) with positive capacity in the auxilliary network. Since i can reach t through j, the shortest path from i is no more than 1 more than the shortest path from j, so $d(i) \leq d(j) + 1$ as needed. Another valid labelling is to set $d(t) = 0$ and $d(i) = 1$ for $i \neq t$.

The algorithm of Goldberg and Tarjan has two phases: in the first phase flow is sent from s as far forward as possible (ideally as far as t). In the second, excess flow is sent back to s. Ahuja and Orlin (1987) modified the algorithm slightly, giving a one–phase approach. It is their method we adopt here.

An initial preflow can be created by saturating each arc out of s.

Problem 2. Given that initial flow, show that setting $d(t) = 0$, $d(s) = V$ and $d(i) = 1, i \neq s, t$ is a valid distance label.

Again clearly, $d(t) = 0$ and $d(i) > 0, i \neq t$. Consider an edge (i, j) with positive capacity in the auxilliary network. Since all edges out of s were saturated, $i \neq s$, so $d(i) \leq 1$, which means the restriction on $d(j)$ is $d(j) \geq 0$ which is clearly so.

A node i is *active* if it has positive excess and $i \neq s, t$. The simplest version of the algorithm is to repeat the following in any order until there are no active vertices:

Push. Select any active vertex i. Select any edge (i, j) with positive capacity $c(i, j)$ in the auxilliary network and $d(i) = d(j) + 1$ Send min($c(i, j)$,excess(i)) units of flow from i to j.

Relabel. For some active vertex i, replace $d(i)$ by $\min\{d(j) + 1(i, j)$ is an edge with positive capacity in the auxilliary graph$\}$.

Problem 3. Show that d is always a valid labelling, and that for any i, $d(i)$ is nondecreasing.

Since the initial solution is valid, consider the push or relabel step which made the labelling invalid. A push on (i, j) may add the edge (j, i) to the auxilliary network, so we need $d(j) \leq d(i) + 1$. But since we pushed on (i, j) we know that $d(j) = d(i) + 1$. A push on (i, j) may also delete (i, j) from the auxilliary network, which does not affect the validity conditions. Finally, the relabel step explicitly ensures the validity conditions.
If $d(i)$ were to decrease during a relabelling, there would be an edge (i, j) with positive capacity in the auxilliary network such that before the relabelling $d(i) < d(j) + 1$, contrary to the validity conditions.

Problem 4. Show that every if i has positive excess then there is a directed path from i to s in the auxilliary graph. How big can $d(i)$ get?

Consider any s–t cut. It is simple to prove by induction that the flow across the cut equals the flow into t plus the sum of the excesses of nodes on the t side of the cut. For any node i with positive excess, claim there is an s–i path consisting of arcs with positive flow. If not, then there is an s–i cut where all the forward edges have zero flow (by the max–flow/min–cut theorem). But this contradict the first statement (since the sum of the excess is strictly positive). Therefore, there is an s–i path of flows, so its reversal has positive capacity in the auxilliary graph.
Since there is a path of length at most $V - 1$ from i to s and an arc is relabelled only if it has positive excess, $d(i) \leq d(s) + V - 1 = 2V - 1$.

Problem 5. How many saturating pushes are there?

If an arc (i, j) becomes saturated, then it remains so until $d(j)$ increases by at least 2. Thus, an arc can be saturated at most V times, for a total of at most VA saturating pushes.

Problem 6. How many nonsaturating pushes. Hint: Consider $\sum\{d(i)i$ is active$\}$. How does it increase? How does it decrease? Is it ever negative?

Consider $\Phi = \sum\{d(i) i \text{ is active}\}$. Each nonsaturating push causes Φ to decrease by at least one. A saturating push increases Φ by at most $2V$. A relabelling increases it by the increase in the label so there is an $O(V^2)$ increase due to relabellings. Since Φ is never negative, the maximum number of times Φ can be decreased is equal to the initial value plus the total increases. This is $O(V^2A)$ by problem 5.

Problem 7. Show that this algorithm terminates in a maximum flow, provided each relabel step changes a label.

If each relabelling changes a label, the algorithm must terminate in a flow (for there can be no more active vertices). Since d is valid and $d(s) = V$, there can be no augmenting path from s to t, which implies optimality.

Problem 8. What is the complexity of this algorithm, provided each relabel step changes a label?

If each relabel changes a label, the dominating term is the number of non–saturating pushes, which is $O(V^2A)$.

To show a polynomial bound, then, we need only ensure that every relabel step changes a label. One way is to place an order on the edges incident to each node. Each node has a current arc, which is the current candidate for pushing flow out on. We can replace the *Push* and *Relabel* routines with one *Push/Relabel* routine:

Push/Relabel Select any active vertex i and let (i, j) be its current arc. If $d(j) = d(i) - 1$ and (i, j) has positive capacity then *push* flow from i to j. Otherwise replace (i, j) by the next arc incident to i. If (i, j) is the last arc, *relabel* i and make the first arc out of i the current arc for i.

Problem 9. Show that each relabel step changes a distance label.

Since a relabel is called only when there are no arcs (i, j) with $d(i) = d(j) + 1$, the label must strictly increase.

The resulting algorithm is $O(V^2A)$. To improve this to $O(V^3)$ we must examine nodes in a certain order. This order is in queue order. The following step is repeated until the queue Q is empty:

Discharge. Select the vertex i from the front of Q and remove it. Apply *push/relabel* to i at least until its excess is 0 or $d(i)$ changes. If a push from i to j causes j to get positive excess, add j to the rear of Q. If i is still active after the *push/relabel* steps, add i to the end of the queue.

To analyze this algorithm (whose analysis is much like magic), define a *pass* over the queue as follows: the first pass is the discharging step applied to the nodes incident to s (which have positive excess due to our initial flow); pass p consists of those nodes added to the queue during pass $p - 1$.

Problem 10. How many passes are there? Hint, consider $\max\{d(v)v \text{ is active}\}$. Suppose it stays the same or increases during a pass. What must have happened? How many times can that happen? Suppose it decreases. How many times can that happen?

Let $\Phi = max\{d(i)i \text{ is active}\}$. Consider what happens to Φ during a pass. If Φ remains the same, then some distance label must increase. If Φ increases, some distance label increases the same amount. Therefore, the total number of passes for which Φ increases or remains the same is $O(V^2)$. Also, since $\Phi > 0$, the number of passes where it decreases is $O(V^2)$, giving $O(V^2)$ passes.

Problem 11. How many non–saturating pushes are there? What is the time complexity of this algorithm?

There is at most one nonsaturating push per node per pass. Therefore, there are $O(V^3)$ nonsaturating pushes, leading to an $O(V^3)$ algorithm.

The overall complexity of this algorithm is $O(V^3)$. To get the complexity down further, complicated data structures must be used, the so–called dynamic trees, which turn up everywhere in network algorithms (well, everywhere Tarjan is writing anyway).

8. Pushing Large Excesses

Read [2].

In their new paper, Ahuja and Orlin modify the order in which nodes are examined in the Goldberg–Tarjan algorithm. The resulting algorithm is still very simple, but its time bound is better than any others, as long as the capacities are not too big (even if the capacities are very large, you can argue that this is the fastest algorithm).

The essential idea is to push flow from nodes with large excess to nodes with small excess, never creating too much excess at a node. Let U be the maximum capacity of any edge. The algorithm performs $\lceil \log U \rceil + 1$ scaling iterations. For each iteration, there is a limit on the maximum excess permitted, Δ. We will assume Δ is a power of two. In an iteration, every non–saturating push sends at least $\Delta/2$ units of flow and no excess is created of size more than Δ. To ensure this we will always send flow from a node with excess at least $\Delta/2$ to a node with excess less than $\Delta/2$.

Using our notation for Goldberg–Tarjan, given we are in a scaling iteration with Δ, we choose the node with minimum distance such that the node has excess at least $\Delta/2$. We then apply *Push/Relabel* except that we never create excess more than Δ (if we would, we stop sending flow when the excess reaches Δ). If no node

exists with excess at least $\Delta/2$, we replace Δ with $\Delta/2$ and begin a new scaling iteration.

Problem 1. Show that our choice of node means that we always send flow from a node with excess at least $\Delta/2$ to one with excess less than $\Delta/2$.

If we push on (i,j) then we have $e_i \geq \Delta/2$. Since we chose i so that it had minimum distance among all such nodes, $e_j < \Delta/2$.

Problem 2. Show that each nonsaturating push sends at least $\Delta/2$ units of flow.

There are three limitations we can reach: the arc capacity, the excess of the node we push from, and the limitation that the excess of the node we push to is no more than Δ. The latter two limitations are at least $\Delta/2$. A nonsaturating push doesn't reach arc capacity, so we push at least $\Delta/2$ units.

Problem 3. Show that there are $O(V^2)$ nonsaturating pushes in any scaling iteration. (Hint: Examine $\sum e_i d_i$ where e_i is the excess of node i). How many nonsaturating pushes are there in total? How many saturating pushes?

Consider $\Phi = \sum e_i d_i$. Initially this is bounded by $2V^2\Delta$, for each $e_i \leq \Delta$ and each $d_i \leq 2V$. When the algorithm examines node i one of two things happens:
(1) The algorithm increases its distance label. The number of times this can be done for each node is no more than $2V$ and each increases Φ by Δ, so the increase in Φ due to relabellings is bounded by $2V^2\Delta$.
(2) The algorithm pushes along an arc. Each nonsaturating push decreases Φ by at least $\Delta/2$. Each saturating push decreases Φ by some amount. Therefore, the number of nonsaturating pushes is no more than the initial value of Φ plus the increase in Φ divided by the minimum amount the push decreases Φ. This turns out to be $8V^2$.
Since there are $\log U$ scaling iterations, there are $O(V^2 \log U)$ nonsaturating pushes. There are $O(VA)$ saturating pushes for the reasons given in Goldberg–Tarjan.

In order to show that the overall algorithm takes time $O(V^2 \log U)$, it is necessary to give data structures that permit the finding of the minimum distance node with large enough capacity, and other aspects of this algorithm. This makes the algorithm a bit nasty looking, but doesn't change the essential aspects. Read Ahuja and Orlin for further details.

9. Conclusions and Further Research

In this tutorial, we have examined seven algorithms for the maximum flow problem: those by Ford and Fulkerson, Edmonds and Karp, Dinits, MPM, Karzanov (via Tarjan), Goldberg and Tarjan, and Ahuja and Orlin. The complexity of the

versions we examined ranges from $O(v^*)$ (the value of the maximum flow) for Ford–Fulkerson down to $O(V^3)$ for MPM, Karzanov, and Goldberg–Tarjan and $O(V^2 \log U)$ for Ahuja–Orlin. In this tutorial we have omitted one very important topic: sophisticated data structures, such as dynamic trees (see [28]) as well as a number of other algorithms (see [2] for references). Even at this stage, however, there are a number of intriguing research questions:

Research Problem 1. Is there an $O(VA)$ algorithm for the maximum flow problem or can you create a nontrivial lower bound? This bound seems to be the natural bound, but no one has found such an algorithm.

Research Problem 2. Which of these algorithms works best in practice? Do some algorithms work better with some network structures than others? Is there a hybrid algorithm that somehow identifies and takes advantages of these differences? Are there variations on these algorithms that work especially well in practice? For example, in Ahuja–Orlin, pushing is done from the minimum level, whereas in practical terms, it seems better to push from the maximum level. Can you still retain a polynomial bound? Does it work better in practice?

Research Problem 3. What other maximum flow algorithms can you find? For instance, can you combine the MPM idea of throughput with the Goldberg–Tarjan pushing of flows ideas? Is there a variation on the Edmonds–Karp "fat–path" routine (see [19, 57]) whose complexity doesn't depend on the capacities (this would be particularly attractive in practice, for it seems reasonable to augment by as much as possible)?

Research Problem 4. There are a number of problems that use a maximum flow routine as a subroutine, for instance the problem of finding a minimum cut tree ([33, 10]). Do any of these routines, or variations on them, work particularly well for one of these problems?

Research Problem 5. There are a number of generalizations of maximum flow, including polymatroidal flow [40], submodular flow [23], and generalized flow (flow with gains and losses), and there has been some success in generalizing the maximum flow algorithms ([56],[27]). Can some others, or variations thereof, be similarly generalized?

Research Problem 6. There are alternative measures of an algorithm than sequential worst case time. Some of these include parallel complexity (discussed in [2, 28]) and average case behaviour. Is there a reasonable definition of average case, and do the algorithms discussed differ in their average case behaviour? Is there a completely different algorithm that is particularly suitable in parallel? (The latter has been done for matching, using randomization and a translation to a matrix question (see section 4.5.)

Research Problem 7. Another possibility is to create a randomized algorithm with good worst case behaviour. Randomization prevents systematic errors causing poor time complexity. For instance, it is easy to show in the Goldberg–Tarjan algorithm that simply taking the node with minimum level is not a good algorithm. What about randomly picking a node with positive excess? What is the worst expected performance of this algorithm? Note that this is different from average case behaviour, for the probability model is in the algorithm, not the input.

Research Problem 8. The parametric maximum flow problem has the arc capacities a function of a single parameter. The objective is to do such things as find the optimal value for the parameter, determine the breakpoints in the solution value and so on. The excellent paper by Gallo, Grigoriadis, and Tarjan [26] gives a number of references and shows how the Goldberg–Tarjan algorithm can be used to solve these problems in time proportional to the time to solve a single maximum flow problem (previous algorithms solved a series of maximum flow problems). One intriguing question they raise is whether such a modification can be done for **any** maximum flow algorithm. Or, more realistically, can the Ahuja–Orlin algorithm (or some modification) be so modified? See also [50].

CHAPTER III

Minimum Cost Flow

1. Introduction

The minimum cost flow model is one of the most frequently used in Operations Research. This is due to three reasons: it is easy to understand, it is widely applicable, and solutions to reasonable problems can be found very quickly. These notes concentrate on the third aspect: solvability. For more information on the applicability of network models see [1].

In this section, we will concentrate on various general solution techniques, with an emphasis on papers and results from the past two or three years. We begin with two cycle canceling algorithms. These algorithms have the advantage of being conceptually very simple and perhaps practical. Following sections study scaling algorithms.

2. Canceling the Best Cycle

Let $G = (V, A)$ be a directed network with a cost on each arc a of c_a and a capacity u_a. Associated with each node $i \in V$ is an integer $b(i)$ representing its supply or demand ($b(i) < 0$ means i has a demand for flow; $b(i) > 0$ is a supply). The minimum cost flow problem is to move the flow from the supply vertices to the demand vertices along the arcs so as to minimize the total cost. This can be written as a linear program as follows:

$$\min cx$$

$$\sum_{\{(i,j)\in A\}} x_{(i,j)} - \sum_{\{(j,i)\in A\}} x_{(j,i)} = b_i$$

$$0 \le x_a \le u_a$$

Let $n = V$ and $m = A$.

There are a number of optimality conditions for the minimum cost flow problem. We will begin with one based on negative cycles. First a few definitions. An arc (i,j) with flow $x(i,j)$ has *residual capacity* $u(i,j)-x(i,j)$ and cost $c(i,j)$. It also gives rise to an arc (j,i) with residual capacity $x(i,j)$ with cost $-c(i,j)$. An arc is a *residual arc* if it has positive residual capacity. A *residual cycle* is a directed cycle of residual arcs and has cost equal to the sum of the costs on the cycle.

The following theorem is the main theorem of this section:

Theorem. *A feasible solution to a minimum cost flow problem is optimal if and only if there are no negative cost residual cycles.*

Problem 1. Prove the above theorem. (Hint: Argue that any solution with net flow of zero at each node can be decomposed into at most A cycles. The theorem then follows easily.)

(if) If an optimal solution had a negative residual cycle then, after augmenting around the cycle, a better feasible solution could be found, a contradiction.
(only if) Let f be any feasible solution and let f^* be the optimal solution. Assume f is not optimal. Examine the flow $x = f^* - f$, where negative flow on an arc is taken to be positive flow on the arc in the opposite direction. Since both f and f^* are feasible, x has net flow 0 at each node. Now examine any arc e with x_e nonzero. We can create a path of nonzero flows beginning at the head of e. Since the net flow at each node is 0, if we enter a node we have not visited before we can leave it. This continues until we enter a node we have visited, at which point we have found a cycle. We can remove an amount of flow equal to the minimum flow on the cycle from each arc on the cycle, deleting one arc from x. We can do this at most m times, decomposing x into m cycles. Each cycle is a residual cycle with respect to f (for it is a cycle in $f^* - f$). Also, f^* equals f augmented by these cycles. Since the cost for f^* equals the cost for f plus the cost on the cycles and has smaller cost, at least one of the cycles must have negative cost.

Define the *improvement* of a negative cycle to be the product of the cost of the cycle and the flow of the cycle.

Problem 2. Assume you have a feasible solution and a method for finding the negative cycle with maximum improvement. Give a polynomial time algorithm for the minimum cost flow problem.

The algorithm is simply find the maximum improvement cycle, augment around it, repeat until no negative cycle remains. By the proof of problem 1, the maximum improvement cycle must improve by at least $1/m$ of the current distance from optimality. The initial solution is at most mUC away from optimal (U is the maximum capacity, C the difference between the maximum and minimum costs). Some calculations show that $O(m \log(mUC))$ iterations are required before the distance is less than 1.

Problem 3. Show that finding the negative cycle with maximum improvement in a graph is an NP–complete problem.

Reduction from hamiltonian cycle. From a hamiltonian cycle instance, create a graph with costs of -1 on all edges corresponding to edges in the instance. A maximum improvement cycle is a hamiltonian cycle, if one exists.

The next section presents an algorithm by Barahona and Tardos that modifies a much earlier paper by Weintraub to give a polynomial algorithm. The main idea is to find a set of cycles at least as good as the minimum cost negative cycle.

3. Canceling Many Cycles

Read [8].

We saw in the previous section that if we could find the maximum improvement cycle in time K then we could solve the minimum cost flow problem in time $O(Km\log(mUC))$. Unfortunately, K is probably exponential. Barahona and Tardos [8], modifying very early work by Weitraub [62], give a method for finding a set of cycles at least as good as the maximum improvement cycle.

For a graph $G = (V, A)$ with a cost function c on the edges, create an auxilliary bipartite graph $B = (V' \cup V'', A')$ as follows:

For each node $v \in V$ create two copies $v' \in V'$ and $v'' \in V''$. Create an arc $a' = (i', j''$ if either $i = j$ or $(i, j) \in A$. Define the cost of the arc $(i', j''$ to be 0 if $i = j$ and $c(i, j)$ otherwise.

A *perfect matching* in B is a set of arcs so that each node is incident to exactly one arc. Solving the matching problem in bipartite graphs (also known as the assignment problem) is distinctly easier than the minimum cost flow problem ([14, 57]).

Problem 1. Show that each perfect matching in B corresponds to a set of node–disjoint cycles in G.

Consider a matching M. Suppose $(i', j'') \in M$ with $i \neq j$. Then $(j', k'') \in M$ for $k \neq j$ and so on until $(l', i'' \in M$. The nodes $i, j, k, \ldots, l$ then form a cycle in the original graph.
If $(i', i'') \in M$ then i is not part of a cycle.

Finding disjoint cycles in the residual graph, even those with minimum cost, is not enough to find the a maximum improvement set. For that, we must solve a sequence of assignment problems. Let $\lambda_1 > \lambda_2 > \lambda_3 > \ldots > \lambda_k > 0$ denote the different values of the residual capacities in G. Let $G(\lambda) = (V, E(\lambda))$ where $E(\lambda)$ are all arcs with residual capacity at least λ.

Problem 2. Consider solving an assignment problem using the auxilliary bipartite graph of $G(\lambda)$. What is a lower bound on the improvement when augmenting around the corresponding cycles in G?

If c is the sum of the costs in the cycles, then, since every cycle can be augmented by at least λ, the improvement is at least $c\lambda$.

Problem 3. Suppose the assignent problem corresponding to $G(\lambda_i)$ for $i = 1, \ldots, k$ is solved and the best lower bound used. Show that the improvement is at least that of the maximum improvement (single) cycle in the original graph.

Let λ_i be the residual capacity of the maximum improvement cycle. Then $G(\lambda_i)$ contains every edge of the cycle. Therefore the cycle is feasible for the matching problem, so the minimum cost solution will be at least as good as that for the single cycle.

The result of this problem gives an algorithm for the minimum cost flow problem: create the residual graph, solve a series of assignment problems, augment along the best set of cycles found, repeat. We previously showed that no more than $O(m \log(mUC))$ cancellations are required. In the above definition of λ_k, clearly $k \leq m$, so each cancellation requires solving at most m asssignment problems. The overall time bound is $O(m^2 A \log(mUC))$, where A is the time to solve an assignment problem. This can be improved as follows:

Note that the assignment problems are very similar to each other: one extra edge is added to get from one to the next. It is possible to show that given a solution to one, the solution to the next can be solved with one shortest path calculation, where the corresponding graph has negative weights.

The final time bound is $O(m^2(m + n \log n) \log(mUC))$.

4. Canceling a Good Cycle

Read [30].

An alternative to canceling a set of cycles at least as good as the maximum improvement cycle is to cancel a single cycle that is "good enough." Goldberg and Tarjan [30] show that the cycle that has minimum mean cost is such a cycle. The mean cost of a cycle is defined to be the cost of the cycle divided by the number of edges in the cycle.

In order to show that this algorithm works we need to understand some concepts from duality theory. A price function p assigns a value to each node of G. The reduced cost for an arc (i, j) with respect to p is $c_p(i, j) = c(i, j) + p(i) - p(j)$.

Problem 1. Show that the reduced cost of a cycle is the same for every value of p.

Consider an adjacent pair of edges on the cycle (i,j) and (j,k). The cost from this pair is $c(i,j)+p(i)-p(j)+c(j,k)+p(j)-p(k) = c(i,j)+c(j,k)+p(i)-p(k)$. Continuing around the cycle cancels off all the p values.

An alternative optimality criterion is as follows:

Theorem. *A feasible solution f is optimal if and only if there is a price function p so that the reduced cost for every residual edge is nonnegative.*

Tardos [55] and Bertsekas [9] independently defined the notion of ϵ–optimality. A feasible circulation is *ϵ–optimal* if there is a price function p so that the reduced cost for every residual edge is $\geq -\epsilon$.

Problem 2. Show that if all the arc costs are integer and $\epsilon < 1/n$ then ϵ–optimality implies optimality. (Hint: What is the cost of the minimum cost cycle?)

If each arc has cost $> 1/n$ then the minimum cost cycle has cost > -1. Since all the arc costs are integer, the minimum cost cycle must have cost ≥ 0 so the solution is optimal.

For a feasible flow f, let $\epsilon(f)$ be the minimum epsilon such that there exists a p so that f is ϵ–optimal with respect to p. We can relate this value to the minimum mean cost residual cycle. Let $\mu(f)$ be the mean cost of a minimum mean residual cycle.

Problem 3. Show that for any circulation f, $\epsilon(f) = \max\{0, -\mu(f)\}$.

Consider any cycle Γ with l edges. If we add $\epsilon(f)$ to each edge, Γ must end up with nonnegative cost, so $c(\Gamma)+\epsilon l \geq 0$. This means $c(\Gamma)/l \geq -\epsilon(f)$ for all Γ so $\epsilon(f) \geq -\mu(f)$.
Now, suppose $\mu(f) \geq 0$. Then $\epsilon(f) = 0$ (the solution is optimal). Otherwise, add $-\mu(f)$ to each arc cost. Since there are no longer any negative cycles, there exists a cost function that gives a postive reduced cost to each residual edge. Now subtract off the $-\mu(f)$. The reduced costs are all at least $\mu(f)$, so $\epsilon \leq -\mu(f)$.

Problem 4. Show that canceling a minimum mean cycle cannot increase $\epsilon(f)$.

In order for $\mu(f)$ to equal $-\epsilon(f)$ there must be a price function that realizes $\epsilon(f)$ for which all arcs on the minimum mean cycle have reduced cost $-\epsilon(f)$. Augmenting around this cycle only creates arcs with reduced cost $\epsilon(f)$ so ϵ has not increased.

For the proof, we need to show how quickly $\epsilon(f)$ decreases. Let f be a feasible solution, let $\epsilon = \epsilon(f)$, and let p be a price function that gives ϵ. Consider a sequence of m minimum–mean cycle cancellations.

Problem 5. Suppose the reduced cost for every arc in each cycle cancelled is negative. Show that the final solution is optimal.

By the argument in problem 3, if an augmentation is done using only negative reduced cost arcs only positive reduced cost arcs are created. After at most m augmentations there are no negative reduced cost arcs, so the solution is optimal.

Problem 6. Suppose an arc is used with non–negative reduced cost. Show that just before the augmentation $\epsilon(f) = (1 - 1/n)\epsilon$.

Let the length of the cycle be l. If one arc is non–negative, then the cost of the cycle is $\geq -\epsilon(l-1)/l \geq -\epsilon(n-1)/n$. But that implies that the $\epsilon(f) \leq (1-1/n)\epsilon$.

Some logarithmic calculations show that for integer costs $O(nm\log(nC))$ iterations are required for optimality. This can be changed to $O(nm^2 \log n)$ for a strongly polynomial bound.

Karp [37] gives an algorithm for finding the minimum mean cycle that requires $O(nm)$ time. This gives a total time of

$$O(n^2m^2 \min\{log(nC), m\log n\})$$

on networks with integer arc costs.

5. Cost Scaling

Read [29].

The fundamental idea behind scaling is to ignore most of the problem data and solve an approximation to the instance. More data is added and the previous solution is used to find a more accurate solution. We begin with an algorithm by Goldberg and Tarjan ([29]) that uses cost scaling. In this case, most of the cost information is ignored initially and added in steps. Recall the concept of ϵ–optimality from the cycle cancelling section. We will begin with a flow that is ϵ–optimal, where ϵ is the maximum cost in the network. We will then improve this flow so that it becomes $\epsilon/2$ optimal. After $\log nC$ such improvements, the flow is ϵ–optimal for $\epsilon < 1/n$. Assuming the costs are integer, this implies the flow is optimal, as we previously proved.

A generic description of a cost scaling algorithm is as follows:

Procedure Min-cost(V,A,u,c)
  $\epsilon = \max_{(i,j)\in A} |c(i,j)|$
  f_ϵ = some feasible flow
  $p_\epsilon(v) = 0$, for all v
  while $\epsilon \geq 1/n$ **do**
    ImproveApproximation(ϵ, f, p)
    $\epsilon = \epsilon/2$
    endwhile
  end.

ImproveApproximation is a routine that takes an ϵ–optimal solution and creates an $\epsilon/2$–optimal solution.

Goldberg and Tarjan suggest a number of possibilities for ImproveApproximation varying in the sophistication of the data structures, the ability to be parallelized, and so on. We will examine two: a generic pseudoflow based method, and an improvement thereof. These algorithms are highly reminiscent of Tarjan's wave algorithm for maximum flow ([58]).

We begin with the concept of a *pseudoflow*. A pseudoflow is an assignment of values to the arcs that satisfies capacity and lower bound constraints, but does not necessarily satisfy conservation of flow at the nodes (remember a *preflow* had nonnegative excess at each node; a pseudoflow allows positive or negative). We extend the concept of ϵ–optimality to pseudoflows in the obvious way: a pseudoflow is ϵ–optimal if there is a price function p such that the reduced cost of every residual arc is at least $-\epsilon$.

Problem 1. For a given p, give a method for finding a 0–optimal pseudoflow.

Simply set $f(i,j) = u(i,j)$ if $c_p(i,j) < 0$ and $f(i,j) = 0$ otherwise.

So, given an ϵ–optimal flow, we can create an $\epsilon/2$–pseudoflow. We now give methods for changing an $\epsilon/2$–psueudoflow into an $\epsilon/2$–optimal flow.

Let $e(i)$ be the net flow into node i minus its requirement. We say that i is *active* if $e(i) > 0$.

Problem 2. Show that a pseudoflow with no active node is a flow.

*The sum of the balances always
and once negatively). If no balan
is a flow.*

Recall that $c_p(i,j)$ is the reduced cost of arc (i,j) relative to p. We will say that a residual arc (i,j) is *admissible* relative to p if it has positive residual capacity and $-\epsilon/2 \leq c_p(i,j) < 0$. Let $r(i,j)$ be the residual capacity of (i,j). For an active node i, consider the following routine:

PushRelabel(i)
  if there exists an admissible arc (i,j) **then**
    push $\min(e(i), r(i,j))$ from i to j
  else $p(i) = p(i) + \epsilon/2 + \min\{c_p(i,j) : (i,j) \in A, r(i,j) > 0\}$
end.

Problem 3. Show that *Pushrelabel(i)* preserves $\epsilon/2$–optimality.

Pushing on an arc can only create an arc in the residual graph with positive reduced cost. Now consider relabelling node i. The reduced cost for any arc (i,j) is decreased by at most $\epsilon/2$ plus the previous reduced cost. Therefore the reduced cost of the arc is at least $-\epsilon/2$, so the solution is still $\epsilon/2$–optimal.

This leads to our first algorithm for *ImproveApproximation*:

ImproveApproximation(ϵ, f, p)
  Create initial solution
  while there is an active node **do**
    select an active node i
    PushRelabel(i)
  endwhile
end.

From problems 2 and 3, if we show that *ImproveApproximation* terminates, then it must terminate in an $\epsilon/2$–optimal flow. The proof of termination is very similar to the case for maximum flow.

Problem 4. Show that $p(i)$ only increases and that it increases by at least $\epsilon/2$.

Relabelling is done only when there is no admissible arc, so $\min\{c_p(i,j) : (i,j) \in A, r(i,j) > 0\} \geq 0$. *The amount added to* $p(i)$ *is at least* $\epsilon/2$.

We now wish to bound the number of times p increases for each node. Let f' be the ϵ–optimal flow at the beginning of the phase and f the current pseudoflow. Let i be an active node. Using decomposition techniques similar to those used for maximum flow, we can show that there exists a node j and a path P from i to j such that j has negative excess, P is an augmenting path with respect to f, and the reversal of P is augmenting with respect to f'. Formalizing this is somewhat tedious:

Problem 5. Show the above statement.

Let $G_+ = (V, E_+)$ *where* $E_+ = \{(i,j) : (f'(i,j) > f(i,j))$ *or* $(f'(j,i) < f(j,i))\}$, *and* $G_- = (V, E_-)$ *is the corresponding graph with the roles of* f *and* f' *reversed. Any edge in* E_+ *is a residual edges with respect to* f *and any in* E_- *is residual with respect to* f'. *Furthermore, if* $(i,j) \in E_+$ *then* $(j,i) \in E_-$. *So, if we find a suitable* P *in* G_+ *then we are done.*
Fixing i, *suppose we can reach no node* j *with negative excess in* G_+. *Let* S *be the set reachable from* i *and* $S' = V - S$. *The flow across any cut with respect to* f' *is zero. Also, every edge from* $k \in S$ *to* $j \in S'$ *must have* $f(k,j) \geq f'(k,j)$ *and every edge from* j *to* k *has* $f(j,k) \leq f'(j,k)$. *So the flow across the cut in* f *is non-negative. But it is easy to see that the flow across a cut equals* $-b(S) < 0$, *a contradiction.*

This leads to a bound on the number of times $p(i)$ is increased.

Problem 6. Suppose j has negative excess. What is the relation between $p(j)$ now and at the beginning of the phase?

If j has negative excess, then at no point would we have chosen j for PushRelabel so its p value is unchanged.

Problem 7. Get a bound on $p(i)$ based on P. Do the same for $p'(i)$ based on the reversal of P. Combine them and bound the number of increases to $p(i)$.

For i with positive excess, find P and j as in problem 5. Since P is augmenting with respect to f and f is $\epsilon/2$–optimal, $p(i) \leq p(j) + P\epsilon/2 + \sum_{(i,j)\in P} c(i,j)$. Similarly, at the beginning of the phase, f' is ϵ–optimal, so $p'(j) \leq p(i) + P\epsilon + \sum_{(j,i)\in P'} c(j,i)$. Using $p(j) = p'(j)$ and $c(i,j) = -c(i,j)$ gives $p(i) - p'(i) \leq (3n/2)\epsilon$.

We say that a push is *saturating* if the residual capacity of the arc after the push is 0.

Problem 8. How many saturating pushes are there in a phase?

A saturating push removes the arc from the residual graph. It cannot appear again until (j,i) is used. But the dual for j must have increased in the interim. This gives an $O(n)$ bound per arc, or $O(nm)$ in total.

We need one more result to bound the number of unsaturating pushes.

Problem 9. Show that the graph of admissible arcs is acyclic.

Initially the admissible graph is empty, so is acyclic. A push operation creates no new admissible edges. After a relabel, no admissible edge enters i (the dual of i increases by at least $\epsilon/2$ so all arcs entering must get non–negative reduced cost). If the graph was acyclic before the relabel, then it must remain so.

Let $h(i)$ be the number of nodes reachable from i in the graph of admissible arcs. We will now use the potential function $\Phi = \sum\{h(i) : i \text{ is active}\}$.

Problem 10. Show that each relabel or saturating push increases Φ by at most n and each nonsaturating push decreases Φ by at least 1. What is the number of nonsaturating pushes?

A nonsaturating push on (i,j) makes i inactive while possibly making j active. But $h(j) \leq h(i) - 1$, since i reaches everything j does as well as i itself, so Φ decreases by at least 1.
A relabel on i increases only $h(i)$ (since no arc enters i afterwards), so the increase is at most n. A saturating push on (i,j) can only increase $h(j)$, so again the increase is at most n.
Since Φ is non–negative and has an initial value of at most $O(n^2)$, the number of nonsaturating pushes is $O(n^2m)$.

Problem 11. What is the complexity of this algorithm?

Each call to ImproveApproximation requires $O(n^2m)$ time (the nonsaturating push time dominates). The number of scaling phases is $O(\log nC)$ for a total of $O(n^2m \log nC)$.

We can improve this slightly by examining the nodes in a certain order (just like the maximum flow case). Because the admissible graph is acyclic, we can order the nodes in the network so that if (i, j) is an admissible arc, then $i < j$ (this is called the *topoligical ordering*).

Problem 12. Suppose the nodes are examined in topological order and no relabel is required. Show that the resulting pseudoflow is a flow.

Every push is to a higher node. Therefore, if no relabel is done, there are no further active nodes, so the result is a flow.

Problem 13. How many node examinations are required if the nodes are examined in topological order?

Since there are $O(n^2)$ relabels, there are $O(n^3)$ node examinations. Each node examination has at most one non–saturating push, so $O(n^3)$ nonsaturating pushes are required.

Note that a push does not change the topological order, only a relabel does. It is possible to determine the topological order of a graph in $O(m)$ time. Since there are $O(n^2)$ relabellings this leads to $O(n^2m)$ time. We can improve on this by the following: Starting with a topological order of the nodes, after a relabel of node i, move i to the first position.

Problem 14. Show that this is a valid topological ordering for the new admissible graph.

Suppose we relabel i. We have already shown that no admissible edge enters i after a relabelling. For each (i, j), i comes before j. For each (k, j) with $k \neq i$, neither the admissible graph or the topological order change, so the order is still valid.

Problem 15. What is the complexity of this algorithm?

$O(n^3 \log nC)$.

Goldberg and Tarjan continue by adding dynamic trees, replacing an $O(n^2)$ term with one of $O(m \log n)$, and then discuss other improvements and the parallel case.

Attached to this set of notes is a listing of a straightforward implementation of this algorithm. One useful project is to examine the code to see how simple the algorithm really is. How could you improve the execution time of this code? How fast does it work in practice (say, compared with standard network codes)? Does the computation time vary much depending on the network topology? Cost ranges? Does the time seem to follow our worst case analysis or does it seem to work differently in practice? These are a sample of the questions that might be asked.

6. Shortest Path Augmentations

Read [49]

In the previous algorithm, we had the following problem: Transform a given pseudoflow into an optimal flow. The previous algorithm "almost" did that; it found an ϵ–optimal flow. An alternative, and perhaps more natural, approach is to directly move flow from nodes with positive excess to nodes with negative excess. Of course, we can not do this movement arbitrarily. We must be able to prove that we have an optimal flow at termination. We do this by keeping a dual feasible solution at all times. Then, if we can prove that we must terminate in a flow, then we will terminate in an optimal flow.

Problem 1. Show that if we have a feasible flow f and a price function p such that $c_p(i,j) \geq 0$ for every (i,j) with positive residual capacity, then f is optimal.

This follows directly from the complementary slackness conditions.

Problem 2. Give a method for finding a pseudoflow f and price function p such that every residual arc has nonnegative reduced cost.

For every arc with negative reduced cost with respect to p, set the flow on the arc equal to the upper bound.

Given a pseudoflow and associated price function, we will try to update the flow and price so as to keep dual feasibility but reduce the sum of the positive excesses. Choose a node i with positive excess. We can easily calculate the shortest path (with costs as distances) from i to every other node using arcs in the residual network.

Problem 3. Show that the shortest path is the same whether original costs or reduced costs are used.

The cost on a path P from i to j is $\sum_{(l,k)\in P} c'(l,k) = \sum_{(l,k)\in P}(c(l,k) - p(l) + p(k) = p(j) - p(i) + \sum_{(l,k)\in P} c(l,k)$. Therefore, the cost function simply adds a constant to the length of every path, so the shortest path remains the same.

Now let j be some node reachable from i, and P the shortest path that connects them.

Problem 4. What is the maximum flow can be sent from i to j along P?

The minimum of the excess at i, the negative of the excess at j, and the residual capacity of an arc on P.

But we must update the price function so the reduced cost for any residual arc are nonnegative. Let $d(k)$ be the shortest distance from i to k in the residual graph (with respect to the reduced costs). Consider the cost function $p' = p - d$.

Problem 5. Show that the reduced cost with respect to p' for any arc in the residual graph before augmenting is nonnegative.

Since d gives the shortest paths in the residual graph, $d(k) \leq d(l) + c'(l,k)$ for (l,k) in the residual graph. Substituting in gives new residual costs of $c''(l,k) = c'(l,k) - d(l) + d(j) \geq 0$.

Problem 6. What arcs are added to the residual graph after augmenting? What are their reduced costs?

The only arcs added to the residual graph are the reversals of those on the augmenting path. But arcs on the augmenting path have new residual cost 0, so the reversals do also.

Therefore, we have reduced the positive excess while keeping a dual feasible price function.

Problem 7. How many shortest path calculations are required? What algorithm can be used for this shortest path calculation? What is the complexity of this algorithm?

If U is the total positive excess created during initialization, then U shortest path calculations are required. Since we work with reduced costs that are always nonnegative, we can use fast, Dijkstra–like algorithms for the shortest path calculation. The best bound is then is $O(U(m + n \log n))$.

Problem 8. Consider the assignment problem. What is the complexity of this algorithm for that problem?

Here $U = O(n)$ so we get a bound of $O(mn + n^2 \log n)$.

7. Capacity Scaling

The correctness of the previous algorithm is independent of the choice of i and j. Of course, this means we can play a number of games in order to get a polynomial bound. One method is to scale the arc capacities in such a way as to replace the U term with a $\log(U)$ term. There are a number of ways of presenting this. We will change our basic model to that of uncapacitated networks with supplies and demands. We also add infinite capacity arcs between every i and j with sufficiently high cost to ensure that they will not be used in any optimal solution. (This is

mainly for notational convenience). We will also assume all costs are positive. The supply at i will be denoted $b(i)$ (negative $b(i)$ represents demand).

Problem 1. Give a method for translating a circulation problem with upper bounds to an uncapacitated network with supplies and demands.

Replace each capacitated edge (i,j) with two edges, (i,k) and (j,k). Let the cost of (i,k) be c_{ij} and the cost of (j,k) be 0. Add u_{ij} to the supply on j, and keep the supply on i the same. Let the supply on k be $-u_{ij}$ (a demand).

Our goal is to push only from nodes with high excess to those with high negative excess. We begin with a pseudoflow where there is a Δ such that either all the positive excesses are less than 2Δ or all the negative ones have absolute value less than 2Δ, or possibly both. We then do augmentations to make this condition true for Δ. For this, we will want to ensure that every augmentation pushes at least Δ.

For scaling phase Δ, let $S(\Delta) = \{i : e(i) \geq \Delta\}$ and $T(\Delta) = \{i : e(i) \leq \Delta\}$. Thus, at the beginning of scaling phase Δ, either $S(2\Delta)$ or $T(2\Delta)$ is empty.

Problem 2. How many scaling phases are there?

Once $\Delta < 1$ we can terminate, so there are $O(\log U)$ scaling phases.

The capacity scaling algorithm is exactly the shortest augmentation algorithm with the following differences:

(1) Augmentations are done from $i \in S(\Delta)$ to $j \in T(\Delta)$.
(2) Exactly Δ units of flow are pushed, independent of the excesses or the arc capacities.

Problem 3. Show that during scaling phase Δ the capacity of each arc is an integer multiple of Δ. Show that the push above neither violates arc capacities nor changes a node from positive to negative excess or from negative to positive excess.

Each arc capacity begins as an integer multiple of Δ. Since every augmentation pushes exactly Δ, this property is retained. Finally, since we push from nodes with excess greater than Δ to those with excess less than $-\Delta$, no excess changes sign.

Problem 4. How many augmentations are there in a scaling phase?

Consider the case when $S(2\Delta)$ is empty at the beginning of the phase. Each augmentation then reduces $S(\Delta)$ by one, so at most n augmentations are required. The case for $T(2\Delta)$ empty is similar.

Problem 5. What is the overall complexity of this algorithm?

$O(n \log U)$ times the time for a shortest path calculation with nonnegative distances. But this is in terms of the transformed problem. In terms of the capacitated circulation problem, the bound is $O(m \log U)$.

Thus far, we are following a version of Edmonds and Karp's algorithm ([19]). But is is possible to improve on this bound by removing the effect of U completely. The idea is to identify arcs that must have nonzero flow on them in any optimal solution.

Problem 6. What is the most a flow on an arc can change during a scaling phase? How about for the rest of the algorithm?

Since each augmentation changes the flow by Δ, and there are at most n augmentations, the change is $n\Delta$. The total over all remaining augmentations is then $2n\Delta$.

So any arc with flow more than $2n\Delta$ must have positive flow for any optimal solution. Such an arc must have reduced cost zero for the remainder of the algorithm. A natural operation is to *shrink* the arc, combining its endnodes. We replace i and j with a node k, and replace any arc incident to i or j with one incident to k. The supply at k is the sum of the supplies of j and i. We now have a problem with one less node.

The following problem shows that after a limited number of scaling phases we can shrink an arc.

Problem 7. Suppose at the end of phase Δ, $\Delta < |b(i)|/(4n^2)$ for some node i. Then there is an arc incident to i that can be shrunk.

First, note that the excess at i is less than $2n\Delta$ since the total excess in the network is less than that amount. Suppose $b(i)$ is positive. The least amount leaving i is $b(i) - e(i)$. This can go out on no more than $n-1$ arcs, so one arc must have flow at least $2n\Delta$.

Problem 8. Show that the first arc is contracted after $O(\log n)$ scaling phases.

Initially, $b(i) = \Delta$ for some i. After $O(\log n)$ phases, Δ is small enough to satisfy the conditions of the previous problem.

If this could be repeated, then it would lead to $O(n \log n)$ applications of the shortest path algorithm. Unfortunately it cannot, for the nodes created by contraction may have very small $b(k)$ but large excesses. Many phases are required to push Δ small enough. To fix this, you must identify when this anomalous result occurs and create *special* augmentations to fix it. See the Orlin paper for details.

One practical question.

Problem 9. Is arc contracting likely to be useful in practice? Should you add it to a capacity scaling code?

8. Generalized Networks

Read [27]

Up until now, we have implicitly made an important assumption on how flows move: whatever enters an arc leaves an arc. In many important applications, this assumption does not hold. Rather, a certain proportion of the flow that enters will leave. This proportion may be less than 1 (flow is lost) or greater than 1 (flow is gained). Because of leakage, not all the flow entering a water pipe or electical wire may leave the other end. The ability to modify the flow along an arc is also useful for modeling purposes, converting one unit into another. This truly is a useful generalization.

Unfortunately, it does not seem simple to optimize such networks, despite a large amount of work done. For instance, the only work done on solving minimum cost generalized flow has been simplex based methods. Even there, such fundamental questions as a combinatorial anti–stalling rule are not known. No fully combinatorial method is even conjectured (if anyone is looking for a dissertation topic, I can't make it any more obvious).

Recently, matters have improved for the maximum generalized flow problem. In this very impressive paper, the authors have given polynomial algorithms for finding the maximum generalized flow. The algorithms are fairly complicated, but should be reminiscent of the minimum cost flow algorithms we have discussed. This is no coincidence. In 1977, Truemper ([61]) noted the relationship between maximum generalized flow and minimum cost (pure) flow.

We begin with a few definitions. Every arc in a generalized flow problem has a multiplier a_{ij} associated with it. If $f(i,j)$ units of flow leave i then $a_{ij}f_{ij}$ units enter j. Based on this, the conservation of flow constraint for i is

$$\sum_j a_{ji}f_{ji} - \sum_j f_{ij} = 0.$$

An arc with multiplier greater than 1 is a *gain* arc; one with multiplier less than 1 is a *loss* arc. The *gain* of a directed cycle of arcs is the product of the arc multipliers around the cycle.

A generalized flow satisfies conservation of flow at all nodes except a given node s.

Problem 1. In pure maximum flow, we have both a source and a sink. Why is that not required here?

Cycles with gain less than 1 act as sinks (they can absorb flow); cycles with gain greater than 1 act as sources (they create flow). Only cycles with gain equal to 1 do not create or destroy flow.

Although we call s the source, in keeping with the paper, we will attempt to maximize the flow in to s.

We define the residual graph as we did for pure network flows, except the multiplier on a reverse arc is the inverse of the multiplier on the forward arc.

Problem 2. What is the upper bound on the reverse arc of an arc with multiplier 2 and flow 5?

10.

One fundamental result, due to Onaga ([47]) relates a generalization of augmenting paths to optimal flows. A *generalized augmenting path (GAP)* is a flow generating cycle together with a directed path from a node on the cycle to s (the path may be empty).

Problem 3. Prove that if the residual graph contains a GAP then the flow is not optimal.

If the graph contains a GAP, then augmenting around it creates flow at any chosen node. This flow can then be sent to s along the path.

Proving the reverse implication, though involved, is just like the case for pure network flows. We prove a decomposition theorem, assume that we have a nonoptimal flow, take the difference between the optimal flow and the nonoptimal flow, and show that this implies a GAP. Here is the decomposition theorem. Given a generalized flow, we can write it as the sum of

(1) A flow generating cycle and a path to s,
(2) A cycle with unit gain, and
(3) A pair of cycles, one with gain ¿ 1, one with gain ¡ 1 and a path from the first to the second.

each of which has flow values that satisfy conservation of flow (except at node s).

Problem 4. Prove this decomposition.

Let f be the flow, and G′ the subgraph of G with arcs with positive value. By conservation of flow G′ must be empty (in which case we are done) or it must contain a cycle. If the cycle has gain 1, it is of type 2, so we can cancel the minimum flow around the cycle. Otherwise, assume G′ has a flow creating cycle. After subtracting flow from the cycle, keeping conservation of flow) until one arc gets value 0, the tail of that node must have flow leave on some noncycle arc. We follow this flow (from conservation of flow) until it either reaches s, or it is destroyed by a cycle. This results in type 1 or type 3 component respectively. The proof now follows by induction.

(Hmmm - the proof in the paper is more general and nicer).

Problem 5. Show that if a flow is not optimal, then there is a GAP.

Consider an non-optimal flow f and the optimal flow f^*. $f - f$ is a flow so it can be decomposed as above. Since more enters s under f^* than f, there must be at least one component of type 1. Since everything in $f^* - f$ is in the residual graph of f, the theorem follows.*

So our goal is simply to find and augment around GAPs. This turns out to be simpler if we know where the cycle is. We can transform the problem so that every flow–generating cycle goes through the source. The idea is simple: saturate all gain arcs and add some arcs to imitate their effect. The transformation begins by setting the flow on every gain arc to its upper bound. In general, this is not a feasible flow: there is either a positive or negative excess on each node. If the excess is negative (more leaves the node than enters it) we add an edge from s to i with a very high multiplier and capacity equal to minus the excess divided by the multiplier. If the excess is positive, then we add an edge from i to s with high multiplier and upper bound equal to the excess. We then take the residual graph together with these added edges as our transformed network. A maximum flow in this network can be transformed into a maximum flow in the original network just by adding in the flows we initially set. (wrestle, wrestle). The transformed problem is called the *restricted problem.*

Problem 6. Take a small example, do the transformation, and convince yourself of its validity.

A lot of handwaving.

Problem 7. What can be said about all the flow generating cycles?

At least initially, they all go through s, because all gain arcs are incident to s.

One simple approach is to find a gain cycle and cancel it (the advantage of working with the restricted problem is that we no longer need to find a path back. We will have to prove that we do not destroy the structure of the restricted problem (say by creating gain cycles not through s), but we save that for the moment.

Just as we could apply a price function in minimum cost flow, we can relabel the nodes of the graph changing the arc multipliers and capacities. Consider rescaling the units exiting an arc by some positive multiplier (say changing the flow leaving the network from pennies to dollars).

Problem 8. What changes in the network are required to keep the network "the same"?

If the multiplier is μ_i, we must replace u_{ij} with u_{ij}/μ_i and the multiplier a_{ij} with $a_{ij} * \mu_i/\mu_j$.

We call this updated network the *relabeled network*. It is straightforward to prove that a network and its relabeled version have many similar properties: one can convert a flow from one to the other and the residual networks are the same.

We can define a labeling that helps us find gain cycles. We can find the highest gain path from each i to the source. Suppose we take μ_i as the inverse of this value for each i (with $\mu_s = 1$).

Problem 9. Show that the relabelled multiplier for every arc not leaving s is ≤ 1. Show that for every node i with a path to s that there is a path of unit arcs in the relabeled graph.

The key is to recognize that finding gain paths is exactly like finding shortest paths where we use the logarithm of the multipliers as distances. This then follows from the principle of optimality.

Problem 10. What does the most efficient (i.e. highest gain) cycle look like.

It is a (i, s) path of unit arcs together with an arc from s to i with maximum multiplier.

The only thing left to prove is that augmenting around the maximum gain cycle doesn't add any gain cycles not through s and, furthermore, does not increase the value of the maximum gain cycle. The proof of this is analagous to similar facts about cancelling the minimum cost cycle on a graph, so is omited (i.e. I am wimping out).

This algorithm is not very good. It behaves like augmenting along a minimum cost path for minimum cost flow: the objective may change by only a minimal amount. For this case, the minimal amount may be very small indeed. It is possible to improve this somewhat, by augmenting along many cycles of the same value. Suppose the maximum gain is α and that there are a number of edges in the relabelled graph with value α.

Problem 11. How can you find the maximum flow using only cycles with gain α?

This is simply a (normal) maximum flow problem in the graph defined on the unit edges and those with gain α.

At the end of this, there are no cycles of gain α so the bound is based on the number of different values of cycle gain. For example, if all multipliers are powers of two, then this algorithm runs in polynomial time.

The paper continues with algorithms that give a polynomial bound. The first is like the above, but it uses a minimum cost flow routine to work with a larger graph than just those with unit flow or gain α. The second looks for GAP that are large enough to significantly improve the objective. There is clearly room for improved algorithms.

One interesting exercise is to see if the reliance on restricted graphs is necessary or merely convenient. I lose a lot of feeling for the problem as soon as we move to the restricted problem and it would be nice to work with the original problem directly.

Overall, though this paper is a start, the application of network ideas to generalized networks is almost wide open.

9. THE NETWORK SIMPLEX METHOD

The network simplex method is an alternative for solving network flow problems. In this section, we examine the network simplex method and its specializations to such problems as the shortest path problem and the maximum flow problem. We also examine extensions to generalized networks and some possible extensions to matchings.

9.1. Fundamental Algorithm. Recall the minimum cost flow problem: Given a graph with vertices V, each with a supply b_i, and arcs A, each with a cost c_{ij}, the problem is to

$$\text{Minimize } \sum c_{ij}x_{ij}$$

$$\sum_{\{j:(i,j)\in A\}} x_{ij} + \sum_{\{j:(j,i)\in A\}} x_{ji} = b_i \quad \text{for all } i \tag{1}$$

$$x_{ij} \geq 0 \tag{2}$$

This problem is simply a linear program so solutions can be found by using the simplex method. The simplex method can be steamlined considerably by exploiting the network structure. This both clarifies the algorithm and greatly enhances its practical effectiveness.

This is a (very) brief review of the simplex method for linear programming. Consider the problem Minimize $\{cx : Ax = b, x \geq 0\}$. A fundamental concept is that of a *basis*. A basis is a maximum size set of columns (variables) that is *linearly independent* (that is, no column can be expressed by a linear combination of the others). Given any basis B, there is an associated primal solution $x = B^{-1}b$ and an associated dual solution $y = c_B B^{-1}$. A basis with corresponding $x \geq 0$ is a *feasible basis*, and x is a *basic feasible solution*. Every variable j has a *reduced cost* measuring the improvement in objective if j were to enter the basis. The reduced cost for j is $c_j - ya_j$ where a_j is the column associated with j in A.

The simplex method begins with a basic feasible solution with the associated dual values. The reduced cost for each variable is calculated. If no variable has negative reduced cost, the current solution is optimal. Otherwise, some variable with negative reduced cost is chosen to enter the basis (the *entering variable*). Some variable must be found to leave the basis. This is done by calculating the change in value for each basic variable as the value of the entering variable is increased. The first variable to reach 0 is chosen as the *exiting variable*. If more than one variable reaches zero simultaneously then any such variable can be the exiting variable. The solution and duals are recalculated for the current basis. This completes one pivot. Pivots continue until no variable has negative reduced cost.

To specialize this method for network problems we take advantage of the special structure of the basis. First we need to determine the size of the basis. An obvious limit is n.

Problem 1. Is it possible to find n linearly independent columns for a network problem?

No. The sum of the n rows is 0, so there must be less than n linearly independent columns.

We will assume that G is connected. We know from the above that there can be no more than $n-1$ variables in a basis. To show that exactly $n-1$ suffices, examine the relationship among arcs in the basis.

Problem 2. Can some of the arcs in a basis form a cycle?

No. Multiply some of the arcs by -1 so all arcs around the cycle go in the same direction. Now take any arc (i,j) on the cycle. The sum of all the other arcs is exactly -1 times (i,j).

A set of $n-1$ arc without cycles is a spanning tree of the underlying undirected graph if we ignore directions. To show that these are linearly independent, we examine the corresponding matrix. Examine the matrix of a spanning tree. One row is redundant (from problem 1) so we will delete row 1. Now rename the rows 2, 3, ... , n and the columns $a_2, a_3, \ldots, a_n$ so that one end of a_i is i and the other is j for some $j < i$.

Problem 3. Show that such a reordering is possible. What does the resulting matrix look like? What does that imply about the arcs?

Beginning at node 1, examine the nodes in breadth–first search. Each node, when labelled, is adjacent to exactly one labelled node. The edges in this order give the required ordering.
The resulting matrix is upper triangular, hence of full rank, which is $n-1$, since we deleted row 1.

Problem 4. Given a basis and a supply vector b, show how to find the corresponding basic solution.

Using the reverse order as in Problem 3, simply assign flows to the arcs as required by feasibility.

It is equally straightforward to find a corresponding set of dual values. We are looking for $y_1, y_2, \ldots, y_n$. Since the basis has rank $n-1$, one of these values is arbitrary, so we will set $y_1 = 0$. Given that, and the requirement that the reduced cost for basic arcs is zero, the remainder of the duals can be found.

Problem 5. Give a method for finding the dual solution.

Using the node order of Problem 3, assign dual values to keep the reduced cost of basic arcs 0. Since every node, when examined, will be adjacent to an examined node, the dual value will be determined.

Given the primal and dual solution, we now try to find an arc with negative reduced cost.

Problem 6. What is the reduced cost of arc (i, j)?

$c_{ij} + y_i - y_j$.

The calculation of the exiting arc is streamlined considerably by the simple way in which flows must change if the flow on the entering arc is increased. Let (i, j) be the entering arc. The basis tree, T, together with (i, j) has exactly one cycle. Some of the arcs on the cycle (the *forward* arcs) are in the same direction as (i, k); some (the *reverse* arcs) are opposite in direction.

Problem 7. Show that if the flow on the entering arc is increased to δ then increasing the flow on the forward arcs by δ and decreasing the flow on reverse arcs by δ preserves primal feasibility.

Any node not on the cycle is unaffected by this flow change. Consider node k on the cycle. If it is adjacent to two forward arcs then one must be (i, k) and the other (k, j) for some i and j, so flow is conserved. Similarly for the case of two reverse arcs. One forward and one reverse must either be (i, k) and (j, k) or (k, i) and (k, j). In either case, since the flow on one is increased by δ and the flow on the other decreased by δ, there is no net change.

Problem 8. Which arc(s) are eligible to leave the basis?

Any reverse arc that has minimum flow among all reverse arcs.

Finding an initial feasible basis can be done in many standard ways (adding artificial arcs is one way). This completes the generic network simplex method. The advantages over the general simplex method occur throughout the algorithm: every step is simplified. The disadvantages are the same, however. The simplex

method as presented is not formally an efficient algorithm. In fact, without further modification, the algorithm may *cycle* by repeating a basic feasible solution without proving optimality. In the next sections we will see how to arrange our arbitrary choices (choice of entering and exiting variables) to ensure finiteness, and even provide polynomial bounds for several interesting cases.

9.2. Prohibiting Cycling. Cunningham ([12]) gives a very simple, and very elegant method for avoiding cycling. This method is independent of the rule used to choose the entering variable: it only restricts the choice of exiting variable. We can think of the basis tree T as being rooted at an arbitrary node (say, node 1).

First note that cycling can occur only during a sequence of *degenerate* pivots (pivots which do not change the flow). Therefore, cycling can only occur when there are degenerate arcs (arcs in the basis with zero flow). Cunningham restricts how these arcs can be in the basis: every arc in the basis with zero flow is directed away from the root. Such a basis is called *strongly feasible*. There are two parts to his proof: he first shows that it is possible to pivot from one strongly feasible basis to another; he then shows that strongly feasible bases prohibit cycling.

Beginning with a strongly feasible basis, there is a very easy, appealing rule for pivoting to another for any entering arc (i, j). Examine the two paths from i to the root and from j to the root. There is a unique node k that is on both paths and is maximal (in terms of arcs) distance from the root. This node is called the *join*. Cunningham's rule is simply to begin at the join, traverse the cycle in the same direction as (i, j) and choose the first eligible arc as the exiting arc.

Problem 1. Show that if the pivot is nondegenerate, the resulting basis is strongly feasible. Now do the same for the case of a degenerate pivot.

If the pivot is nondegenerate, the candidates for leaving are exactly those that end up with zero flow. Taking the first from the join in the direction of the entering arc (say, at node k) implies that all other arcs with zero flow will be between k and the root and pointed towards k, hence away from the root.
If the pivot is degenerate, then any zero flow arc between the join and i (where (i, j) is the entering arc) will point away from the root, hence be a forward arc and ineligible to exit the basis. So the exiting arc is between j and the root. By taking the first one, all remaining zero flow arcs will still point away from the root.

To show that the method prohibits cycling we show that during a degenerate pivot the sum of the duals decreases. Repeating a basis would require an increase in this sum, which can only occur during a nondegenerate pivot.

Problem 2. Let (i, j) be the newly entered arc. Examine the subtree "below" (i, j). What happens to the duals of nodes not in that subtree? What about those in the subtree? What happens to the sum of the duals?

Since (i,j) caused a degenerate pivot, it must point away from the root. The duals above (i,j) do not change. Those below are changed as follows: we need $c_{ij}+y_i-y_j = 0$. But before this value was less than 0. Since y_i does not change, y_j must have decreased. Now for any other node below (i,j), the same arcs are in the basis, so, by induction, they must each decrease also.
Therefore, the sum of duals decreases.

This simple method suffices. It also leaves us complete flexibility in choosing the entering arc. In the next section we will see how different choices for the entering arc can avoid exponential sequences of degenerate pivots.

9.3. Prohibiting Stalling. Strong feasibility avoids the possibility of repeating a basis. However, an exponential number of bases may still be required. To date, there is still no rule known that guarantees a polynomial bound on the number of iterations required by the network simplex method. Cunningham ([13]) gives rules that give only a polynomial number of consecutive *degenerate* pivots when combined with his method of strongly feasible trees.

Let $T_0, T_1, \ldots, T_p$ be a sequence of degenerate pivots, each with primal solution x^0. We break this sequence into consecutive *stages*: during each stage each arc has nonnegative reduced cost (i.e. is not eligible to enter the basis) at least once. In a moment we will see that the number of stages is bounded, but first we will examine a rule that guarantees that the length of a stage is not too long.

Problem 1. Consider the entering edge rule that arbitrarily orders the arcs and enters the first arc with negative reduced cost. Then, if arc i is entered, the next iteration starts examining the arcs at $i+1$ and so on. How long can a stage be? (This rule is called *Least Recently Considered (LRC)*).

The maximum number of pivots in a stage is $O(|E|)$. When arc i is examined either it is not eligible to enter the basis, or it enters the basis immediately after which it is ineligible to enter the basis. Therefore, after at most i pivots, i is ineligible to enter the basis.

So we know that a stage is not too long. Now we will show that there are not too many stages. The idea is to show that certain dual values get fixed after each stage of the sequence of pivots. Examine the connected components of the graph induced by edges with strictly positive x^0 values.

Problem 2. Suppose node i is in the same connected component as the root. What happens to the dual of i during the pivots?

Its dual does not change. Since every arc with $x^j{}_0$ remains in the basis, all edges from the root to the node stay in the basis, so its dual remains the same.

Now examine all nodes that use exactly one edge with value 0 on the unique path to the root in T_p. We would like to show that the duals for these nodes are fixed after the first stage.

Problem 3. For some node i, suppose the dual is not fixed after the first stage. What is the relationship between the dual value after the first stage and the final dual value? What does that imply about the reduced cost of the zero edge in T_p used by i in the first stage? What can you conclude?

Since the dual is not equal to the final value, and duals only decrease, it must be larger than the final value. Now, examine the zero flow arc (j,i) from i to the root in T_p fir any pivot in the first stage. Its reduced cost is $\pi(j)-\pi(i)-c(i,j)$ (by strong feasibility, since all zero flow arcs point away from the root). But $\pi(j)$ does not change during the stage (by Problem 2), so this is less than the final reduced cost of the arc (since $\pi(i)$ decreases). But (j,i) is in the final basis tree, so the final reduced cost is 0. Therefore, the reduced cost for (j,i) is less than 0 for the first stage, contradicting the definition of stage. Therefore, after the first stage, such dual values are fixed to their final value.

It is straightforward to modify the argument in Problem 3 to be an induction on the number of zero edges on the path from i to the root in T_p. This means that the number of stages is no more than the number of arcs in a basis with zero flow.

Cunningham examines three other rules, two which reduce the pivots per stage to $O(|V|)$. He also gives an example of an exponential degenerate sequence for Bland's rule, which takes the first edge elgible to leave the basis (always starting at edge 1).

9.4. Other Papers. Recently there have been a number of interesting papers on the network simplex method. These are listed here, with the intention that someday the notes will be continued to include some or all of these papers.

Specialization to Shortest Paths. Goldfarb, Hao, and Kai ([32]).

Specialization to Maximum Flow. Goldfarb and Hao ([31]).

Non–polynomial, But Subexponential Bounds. Tarjan ([59]).

Dantzig's Pivot Rule, and Variations. Orlin ([48]) and Ahuja and Orlin ([3]).

Generalized Networks. Elam, Glover and Klingman ([20]) and Trick ([60]).

CHAPTER IV

Matchings

1. Introduction

The matching problem is a cornerstone problem in combinatorial optimization. Despite the fact that no polynomially sized linear program for this problem is known, polynomial algorithms exist for solving this problem. In these notes, we examine a selection of these algorithms, ranging from augementing path algorithms of the 1950s and 1960s to facet generation and randomized algorithms of the 1980s.

While we concentrate on algorithms, it should be noted that there are many applications for the matching problem. Some include facility design [44], plotter movement [], crew scheduling [6], and machine scheduling [24]. Furthermore, matchings have been used as a subroutine for the traveling salesman problem [10], the chinese postman problem [18], and the set covering problem [46]. The matching model is truely a useful model.

In these notes, we concentrate on the various techniques for solving matching problems. In order to simplify the exposition of these techniques, we concentrate on a simple case: the cardinality matching problem, where our goal is to maximize the number of edges in a matching. In the final sections we will examine the conceptually similar but more difficult weighted matching problem.

The following is an outline of the notes:

(1) Bipartite matching, cardinality case. Augmenting path method, theorems of Berge, Hall, and König.
(2) General matching, augmenting path algorithms. Algorithm of Edmonds, theorems of Tutte and Gallai and Edmonds.
(3) Polyhedral structure, cut generation algorithms. Algorithms of Grötschel and Holland and Trick using results of Padberg and Rao.
(4) Randomized/Parallel algorithms. Determinants, algorithm of Mulmuley, Vazirani, and Vazirani.
(5) Weighted Matching. Algorithm of Edmonds.

2. Bipartite Matchings

Given a graph $G = (V, E)$, $M \subseteq E$ is a *matching* if every node in V is incident to at most one member of M. A matching is *perfect* if every node is incident to exactly one member. The *maximum cardinality matching problem* is to find a matching of largest size. We begin with the case G is bipartite with bipartition (S, T). The number of nodes of G is n, the number of edges m.

Problem 1. Show that in this case, the cardinality matching problem is a special case of maximum flow.

Add two nodes s and t. Add edges of capacity 1 from s to every member of S, and edges of capacity 1 from every member of T to t. Set the capacity of every edge of E to infinity. A maximimum flow in this graph corresponds to a maximum matching.

Due to the simple structure of the graph and capacities, most maximum flow theorems have simpler statements for the matching problem. The fundamental concept is that of augmenting path relative to a current matching M. An *alternating path* is a path in G where the edges are alternately in and not in M. An *augmenting path* is an alternating path where the endpoints are not incident to any member of M. If a node is incident to an element of M, we will say that the node is *matched*. Similary, an edge is *matched* if it is in M. The following theorem is due to Berge.

Theorem. *M is a maximum matching if and only if there is no augmenting path relative to M.*

Problem 2. Prove the above theorem directly (i.e. don't use knowledge about network flows).

One direction is straightforward. If there is an augmenting path, simply reversing the membership in M along that path creates a matching of size one larger.
Suppose M is the current matching and there is a larger matching M'. Take the symmetric difference of M and M'. Since every node has degree at most 2 in this graph, this consists of alternating paths and cycles. Since M' is larger, there exists at least one path that begins and ends with an edge of M'. It is clear that the endpoints of the path are unmatched by M, so this is an augmenting path.

In the bipartite case, this leads to an algorithm: Search for an augmenting path, if found augment, otherwise terminate. The following algorithm searches for an augmenting path by finding an *alternating forest*. A node in the forest if an alternating path is found to it from an unmatched S node. It is marked either "odd" or "even" depending on the parity of the path length. Each component of the forest is rooted an an unmatched S node.

0) (initialize) Mark each unmatched S node "even" and place into a queue Q.

1) (grow forest) Remove the first member of Q, i. For each edge $(i,j) \in E$, one of the following cases is executed:

(j unmarked, matched) Let k be the node j is matched with. Add (i,j) and (j,k) to the forest, mark j "odd" and k "even;" add k to Q.

(j unmarked, unmatched) An augmenting path has been found between j and the root of the component containing i. Augment along it.

(j marked "odd") Do nothing.

This algorithm stops either when the queue is empty or when an augmenting path is found. In the first case, the current matching is optimal; in the second, a new alternating tree can be grown.

Problem 3. Why can j not have the marking "even" in the case analysis?

It follows by induction that "even" nodes are always S nodes and "odd" nodes are T nodes. Furthermore, only S nodes are in Q, so j is a T node.

Problem 4. Prove the correctness of the algorithm. What is the overall complexity of this algorithm?

It follows by induction that if a node is marked, then there is an augmenting path with the correct parity. Therefore, a claimed augmenting path really is an augmenting path. Suppose there is an augmenting path P that is not found by the algorithm where P starts at an S node. Let i by the first node of P that is not marked, and let j be its predecessor. If i is an S node, then (j,i) must be in the matching (by the definition of augmenting path). But j is marked, and must be marked "odd" (since it is a T node). But a node is marked "odd" only if its matched node is marked even. So i is a T node. But then j is an S node and (i,j) is not in the matching, so i would be marked when j reached the front of Q.
Each edge is examined once per augmentation and there are at most $n/2$ augmentations so the complexity is $O(nm)$.

Hopcroft and Karp [35] describe an implementation of this algorithm that requires overall time of just $O(\sqrt{n}m)$. This algorithm (but not the complexity!) is simply Dinics' [15] algorithm applied to this case.

The algorithm also provides an easy way to prove various structure theorems, which give conditions under which matchings can occur. For instance, the following is due to Frobenius (1912). For $X \subseteq S$, let $N(S)$ be the set of nodes connected to a member of X by at least one edge.

Theorem. *A bipartite graph G with partition (S,T) has a perfect matching if and only if $|S| = |T|$ and, for each $X \subseteq S$, $|X| \leq |N(X)|$.*

Problem 5. Prove the above theorem (one direction is straightforward, for the other direction, the violating X can be found from the final alternating tree).

If the graph has a perfect matching, then clearly $|S| = |T|$. Also, if any set X has too small a neighborhood, then no matching is possible.
If G has no perfect matching and $|S| = |T|$, then we must exhibit an X with too small a neighborhood. Let A be the set of vertices marked by the algorithm, and let $X = S \cap A$. A matched $i \in S$ is marked only if its corresponding j is marked. All other nodes incident to i are marked. By the initialization, all the nodes incident to an unmatched S node are marked. Therefore, $T \cap A = N(X)$. But every node in $T \cap A$ is matched, so there is a unique corresponding element of X among the matched nodes. But there are unmatched elements of X, so $|X| > |N(X)|$, as needed.

Another theorem by König (1936) is particularly nice, for it provides a min–max result. A *cover* of the edges of G by nodes is a subset of nodes so that every edge is incident to at least one member of the cover. Let $\tau(G)$ denote the minimal node cover of G.

Theorem. *If G is bipartite then the maximum number of edges in a matching equals $\tau(G)$.*

Problem 6. Prove the above theorem.

Let $\nu(G)$ be the size of a maximum matching. Let M be any matching and C be any cover. Every edge of M must be incident to at least one member of C and every member of C is incident to at most one member of M. Therefore, $|M| \le |C|$, so, in particular, $\nu(G) \le \tau(G)$.
Let M be a maximum matching. If M matches all elements of S then S is a cover of the same size as M. Otherwise, let A be the nodes marked by the algorithm. Let $S' = X \cap S$ and $T' = X \cap T$. Now T' are all matched, since M is maximum. Consider the set $X = (S - S') \cup T'$. For any edge (i, j) either i or j is matched (by maximality). If i is matched, either i is not marked (so $i \in X$) or i and j are marked (so $j \in X$). If j is matched, then either it is marked (and in X) or not, in which case i is not marked (so it is in X). Therefore X is a cover. Furthermore, for an edge $(i, j) \in M$ either i or j, but not both are in the cover, so $|X| \le |M|$, as needed.

According to the folklore, after the work of Berge and Norman and Rabin, the feeling was that there was no more to do on matchings. Edmonds (1965) proved how wrong they were.

Problem 7. Adapt the algorithm for nonbipartite graphs. Where does the algorithm fail?

Where indeed? See next section.

3. General Matchings

To adapt the previous algorithm, we can no longer rely on the fact that S nodes will be marked "even" and T nodes marked "odd." The following algorithm adapts

the bipartite algorithm as much as possible:

0) (initialize) Mark each unmatched node "even" and place into Q.

1) (grow forest) Remove first node of Q, i. For each edge $(i,j) \in E$, execute one of the following cases:

(j unmarked, matched) Let k be the node j is matched with. Add (i,j) and (j,k) to the forest, mark j "odd" and k "even;" add k to Q.

(j "odd") Do nothing.

(j "even," in different component than i) An augmenting path has been found between the roots of the two components. Augment along it.

(j "even," in same component as i) ????

It is in the last case that there is some problem. We have found an odd length augmenting path to a node for which we previously found an even length augmenting path. Two obvious solutions are to ignore the problem (give the node the label it first had) or to allow nodes to be labelled both odd and even, placing only nodes with even labels in the queue.

Problem 1. Show that these solutions will not work, the first because it misses some augmenting paths, the second because it identifies some "paths" that are not paths at all.

Both of the counterexamples in the above problem rely on a cycle with an odd number of nodes where every node on the cycle except one is matched by edges of the cycle. Edmonds ([16]) showed that this construct (which he called a *blossom*) is the only construct that causes difficulty.

Problem 2. Show that there is a blossom when the last case of the algorithm occurs.

Since both i and j are in the same component their paths back to the root must meet at some node k. The paths from k to i and from k to j must have the same parity and be alternating. Together with (i,j), this forms an odd alternating cycle.

Edmonds' idea was to *shrink* the blossom into a single node. To shrink a blossom B, replace the nodes of the blossom by a single node b. If (i,j) is an edge in the original graph, with $i \in B$ in the blossom and $j \notin B$, then create an edge (b,j). The key idea is that a blossom is *hypomatchable*: for every node k in the blossom, there is a matching within the blossom that hits every node except k. Edmonds proved the following theorem:

Theorem. *If B is a blossom of G with respect to M, then there is an augmenting path in G if and only if there is an augmenting path in G with B shrunk.*

Problem 3. Prove the if part of this theorem.

Let P be the path in G with B shrunk. If the path avoids b, then it is a path in G. If it uses B, then either b is an endpoint or an intermediate point of the path. In the first case, let j be the node of B the path uses and let i be the unmatched node of B. There are two paths from j to i in B: one odd length and one even length (which may be empty). The even length one added to the remainder of P is the augmenting path. Similarly, if it goes through B, one direction around the cycle will be the augmenting path.

Now, we complete our algorithm with the following case:

(j "even," in same component as i) Identify the blossom and shrink it to a single node.

We call a shrunk blossom a *pseudonode*.

Problem 4. Show that every blossom is an even node.

The paths found by problem 2 meet in a node which is either the root or has degree at least 3. In either case, the meeting node is an even node, so collapsing the cycle will result in an even node.

To prove the only if part of Edmonds theorem, it is easier to use duality and the results of the algorithm. We define an *odd cover* of a graph to be a collection of odd sets so that for every edge either

i) it is incident to a singleton of the cover, or

ii) both ends are contained in some odd set.

We define the size of an odd set to be 1, if the set is a singleton, and to be r if it has size $2r + 1$ if it is a larger odd set. The size of a cover is the sum of the sizes of its sets.

Problem 5. Show that every odd cover is at least as large as every matching.

Each singleton is incident to at most one matching edge. Each larger set of size r contains at most r matching edges. Therefore, the size of the set is at least the number of edges in the matching.

Problem 6. Suppose our algorithm ends without an augmenting path. Show that the current matching is optimal.

Form an odd cover as follows: take every odd node as a singleton; every outermost blossom as an odd set; and one arbitrary singleton from the unmarked nodes and the remaining unmarked nodes in a single odd set. Every edge of G is either in a blossom, incident to an odd node, or has both its endpoints unmarked. Therefore, this collection of sets is an odd cover. It is straightforward to see that the size of this odd set is exactly the number of matching edges.

Since we have a matching and an odd set with the same size, the matching must be maximum (by problem 5).

Problem 7. Show that every augmenting path found by the algorithm is valid.

An augmenting path is found when i and j are in different components and are both marked even. Therefore, there is an odd alternating path between there corresponding roots (the even paths together with (i,j)). By problem 3, we can expand the blossoms to find an augmenting path in G.

The time complexity of this algorithm depends on the data structures used to handle the blossoms. A straightforward bound is $O(n^4)$. Lawler ([39]) and Gabow ([25]) give implementations that reduce this to $O(n^3)$, and other implementations reduce it further to $O(nm)$. The adaptation of Dinics' algorithm done by Karp in the bipartite case has been generalized by Even and Kariv ([21]) and Micali and Vazirani ([43]) to give a $O(\sqrt{n}m)$ bound.

As in the bipartite case, this algorithm gives simple proofs of a number of previously known structure theorems. We have already proved a min–max result regarding odd covers first proved by Edmonds. Another theorem is by Tutte (1947). Let $c(G)$ be the number of components of G with an odd number of nodes.

Theorem. *G has a perfect matching if and only if $c(G-S) \leq |S|$ for all $S \subseteq V$.*

Problem 8. Prove the above theorem.

Since at least one node from every odd component must be matched to some member of S, if G has a perfect matching then $c(G-S) \leq |S|$ for all S.
For the other direction, assume that G does not have a perfect matching. Let S be set set of nodes marked "odd" by the algorithm. Each even node or blossom forms a component of $G-S$. There are more even nodes and blossoms than odd nodes, so $c(G-S) > |S|$.

Finally, an important theorem related to the set of all matchings of a graph is the Gallai–Edmonds decompostion theorem. Let $D(G)$ be the set of all nodes that are not covered by at least one maximum matching of G. Let $A(G)$ be the set of nodes adjacent to at least one element of $D(G)$. Let $C(G)$ be the remaining nodes. One crucial element of the Edmonds–Gallai stucture theorem is:

Theorem. *If M is any maximum matching of G, then*

i) within each component of $D(G)$ it misses exactly one node,

ii) it contains a perfect matching of $C(G)$, and

iii) every element of $A(G)$ is matched with a node in distinct components of $D(G)$.

We can identify these sets directly from the algorithm: $D(G)$ are those nodes marked "even" or those within blossoms, $A(G)$ are those marked "odd," and $C(G)$ are those not marked by the algorithm. To prove this, we would have to show that a node is marked even or in a blossom if and only if it is unmatched by some maximum matching.

Problem 9. Prove the only if part (i.e. if it is marked even or in a blossom, then it is unmatched by some maximum matching).

Every tree can be rerooted at any even node, leaving that node or blossom unmatched. Every node within a blossom can be unmatched by hypomatchability.

For the if part, see the text by Lovász and Plummer [41].

4. Cut Generation

Read [34].

In the previous section, we saw an augmenting path algorithm for the cardinality matching problem. In the next two sections, we will examine radically different algorithms for this problem: a cut generation approach and a randomized algorithm based on taking determinants.

The cut generation approach attempts to treat the matching problem as a linear program. First we need to find a suitable formulation. We begin with the most obvious constraints, the requirement that at most one edge be incident to each node. The problem becomes:

$$\text{Maximize } \sum x_{ij}$$

$$\sum_{\{j:\{i,j\}\in E\}} x_{ij} \leq 1 \quad \text{for all } i \tag{3}$$

$$x_{ij} \geq 0 \tag{4}$$

Problem 1. Prove that this formulation is not sufficient to define the matching problem.

The triangle provides such a counterexample.

Another obvious constraint, motivated by the blossoms of the augmenting path algorithm, is that no odd set can be perfectly matched by itself. In other words:

$$\sum_{\{i,j\}\in E: i\in S, j\in S} x_{ij} \leq (|S|-1)/2 \quad \text{for all } S \subseteq V, S \text{ odd}. \tag{5}$$

Problem 2. Prove that adding these constraints is sufficient to define the matching problem.

The dual of (1), (2), and (3) is simply the odd cover problem. Since the constraints are valid, this proves that they suffice.

Unfortunately, there are an exponential number of constraints of type (3). It is not known if there is a formulation of the matching problem that has only a polynomial number of constraints.

It is possible to use the ellipsoid algorithm to solve the matching problem using the above constraints. The ellipsoid algorithm requires a *separation* algorithm: given a point y, determine if y is in the polytope and, if not, give a violated constraint. It is straightforward to provide separation routines for constraints (1) and (2). It is not obvious how to find a violated constraint of type (3). Padberg and Rao [51] provide an algorithm for this problem. For any y satisfying (1) and (2) they give an algorithm that will decide if y satisfies (3), and, if not, will give an S whose corresponding constraint is violated.

Suppose we have a fractional solution y for $G = (V, E)$. We create an auxilliary graph G' as follows: beginning with G, we place a capacity on each edge equal to y_{ij}. We then add a node (the *slack node*) to the graph and edges between each node and the the new node. These edges represent the slack variables and have capacity equal to the value of the corresponding slack variable.

Problem 3. Show that a violated constraint is exactly an odd set in G' not containing the slack node where the edges leaving the set have total capacity less than 1.

Suppose we have a violating $|S|$. The amount on the edges within $|S|$ is more than $(|S|-1)/2$. But, by the constraints (2), this means the amount leaving S is less than $|S| - 2(|S|-1)/2 \leq 1$.
For the other direction, suppose we find an S not including the slack node with value k leaving it. The amount within it must be no more than $(|S|-k)/2$. Therefore, if $k < 1$ then S corresponds to a violated inequality.

The problem becomes finding an odd set with the minimum capacity leaving it. If the problem was merely to find a set with minimum capacity leaving it, we could solve this by standard maximum flow techniques.

Problem 4. Give an algorithm for finding a set with minimum capacity leaving it (hint: solve $O(n^2)$ maximum flow problems).

Simply find the minimum cut between every two pairs of nodes and choose the minimum cut.

Gomory and Hu [33] give a faster algorithm for problem 4 using only $O(n)$ maximum flow calculations. The idea is to create a *cuttree* for the graph. A cuttree is a tree that represents the minimum cuts for all pairs of nodes in an undirected graph. Each are of the tree has a label; to find the minimum cut value between to nodes, find the minimum label on the unique path between them in the tree. Removing the minimum edge breaks the tree into two pieces which corresponds to the partition of the nodes to give the cut value.

The Gomory and Hu algorithm is as follows:

0) Place all nodes in a single group.
1) Choose two nodes in a group. Contract all other groups; find a maximum flow between the two nodes.
2) Use the cut to divide the group into two smaller groups, adding an edge of the cuttree between them. If there is no group of size at least 2, then stop. Otherwise go to step 1.

Padberg and Rao show that it is possible to find an odd cut by forming the cuttree as above. The minimum odd cut corresponds to the minimum weight edge of the cuttree whose removal leaves a component with an odd number of nodes (not counting the slack node). It is clear that such an edge corresponds to an odd cut. It is not so clear that it gives the minimum odd cut. Let C be an optimal odd cut and let (S, S') be the bipartition of nodes induced by the cut. Let T be the cuttree generated. Examine the edges $T' \subseteq T$ that correspond to odd cuts in T. Clearly, every node in the tree is incident to at least one member of T' and the degree of T' at each node is odd. Now examine the nodes corresponding to S in T'. Since S has an odd number of nodes, there must be an edge in T' from some element of S to an element in S'. This edge is the minimum cut C' between its endpoints, so the size of C' is less than or equal to the size of C. But, by the definition of T', C' is an odd cut, so it is a minimum odd cut.

This gives an algorithm for matching: Use the ellipsoid algorithm and the separation routines given. Grötschel and Holland ([34]) replace the ellipsoid method with the simplex algorithm to give a cut generation technique similar to that used for the traveling salesman problem. The algorithm has an exponential worst case bound but may work well in practice. The algorithm is:

0) Create a linear program with constraints (1) and (2).
1) Solve current linear program. If solution is integer, the solution is optimal. Otherwise proceed to step 2.
2) Generate violated constraint(s) and add to linear program. Go to step 1.

Creating the modified cuttree is a very expensive operation. Grötschel and Holland provide some heuristics for finding violated inequalities. Only if these heuristics fail do they resort to the cuttree.

Problem 5. Give a very quick algorithm to determine if there is an odd set with capacity 0 out of it.

Delete all edges with capacity 0 in G'. An odd size connected component corresponds to an odd set with 0 leaving it.

Trick ([60]) suggests replacing the simplex code with a generalized network with

side constraints code. This code works very well provided relatively few constraints are generated.

Everything in this section applies to the weighted matching problem. The computational results of Grötschel and Holland and Trick suggest that cut generation techniques may be comparable in speed to augmenting path techniques. There are many more advantages. The generalization to b–matching is straightforward, as opposed to the more complicated combinatorial algorithm. The linear relaxation of matchings with side constraints can be solved routinely, without resorting to lagrangian relaxation or other techniques. The main disadvantages are stability and pathological slowness. An edge in a combinatorial approach is either in or not in the matching. A value of 0.99 in a cut generation approach is more problematical: is it really 1, and the L.P. had round off errors, or is it really 0.99? Finally it must not be forgotten that these methods might generate a very large number of constraints. While the randomly generated problems tested on do generate few constraints, real problems may require more and may be harder to solve.

5. Determinants and a Randomized Algorithm

Read [45].

The next method for solving matchings reveals a somewhat surprising relationship between matching and determinants. These algorithms have a very different feel from the previous algorithms. We begin with the bipartite case.

Consider the matrix A, where $a_{ij} = 1$ if $(i,j) \in E$ and 0 otherwise. Let $\det(A)$ denote the determinant of A.

Problem 1. Suppose G has no perfect matching. What is $\det(A)$? What if G has a perfect matching? Suppose for some G, $\det(A) = 5$. What can you conclude? What if G has a unique perfect matching?

Let σ denote a permutation of $1, \ldots, n$. The definition of determinant is $det(A) = \sum_\sigma sign(\sigma)\Pi_i a_{i,\sigma(i)}$. Each term corresponds to a possible perfect matching. The term will be nonzero if and only if each entry is nonzero. This occurs exactly when the corresponding edges are in the graph. Therefore, if G has no perfect matching, the determinant is 0. If it does have a perfect matching, it may be zero (due to cancellation of terms) or it may not. If it is nonzero, then it definately has a perfect matching. Finally, if G has a unique perfect matching, then the determinant will be nonzero.

So, taking determinants does not provide a characterization of graphs with matchings. We can give a characterization by replacing the entries in A with indeterminates. Let $a_{ij} = x_{ij}$ if $(i,j) \in E$ and 0 otherwise, where x_{ij} are indeterminates.

Problem 2. Now what is the relationship between $\det(A)$ and whether or not G has a perfect matching? What is a drawback of this approach?

In this case, since different terms involve different variables, no cancellation can occur. Therefore, the determinant is nonzero if and only if G has a perfect matching. Unfortunately, the determinant may have an exponential number of terms.

Instead of using indeterminates (which require an exponential amount of time) or 1s (which tend to cancel too much), we will use entries for A drawn at random from the range $\{1, \ldots, N\}$. It is possible for the terms to cancel, but it is highly unlikely. Schwartz ([54]) shows that the probability is at most $(2/N)^{|E|}$.

Problem 3. Given a 1000 edge graph, suppose 100 sets of numbers in the range $\{1, \ldots, 2000\}$ are generated and the corresponding determinant is 0 in all cases. What is the probability that G has a perfect matching?

For each set, if G had a perfect matching, the probability the determinant is zero is at most $.001^{1000} = 10^{-3000}$. Therefore, the overall probability is $10^{-300000}$ (which is pretty darn small).

To extend this to the nonbipartite case, we need a somewhat more complicated matrix. For $(i,j) \in E$, we will define $a_{ij} = x_{ij}$ if $i < j$, and $a_{ij} = -x_{ij}$ if $i > j$. If $(i,j) \notin E$ then $a_{ij} = 0$. The following theorem is more difficult than the bipartite case (the problem is that since each variable appears twice, there are opportunities for cancelation):

Problem 4. Show that $\det(A) = 0$ if and only if G has no perfect matching.

For each permutation σ of $1, \ldots, n$, define $v(\sigma) = \Pi_i a_{i\sigma(i)}$. $Det(A) = \sum_\sigma sign(\sigma)v(\sigma)$. Thus $v(\sigma) \neq 0$ if and only if $(i, \sigma(i)) \in E$ for all i.
Now, create an auxilliary graph on V. For a permutation σ, add the edges $(i, \sigma(i))$ for all i. Clearly each node has degree 2 (possibly due to two edges (i,j)). Matchings correspond exactly to those σ that have no cycles with more than two edges. Suppose there is an odd cycle for σ. There is another σ' that is identical to sigma except it traverses the edges in the reverse direction. It is easy to show that $v(\sigma) = -v(\sigma')$ so these cancel in determining the determinant.
Therefore, the only terms that contribute towards the determinant have no odd cycles. But any graph with even cycles of length greater than 2 are simply unions of matchings. Therefore, if the determinant is nonzero, then there is a perfect matching. To show the converse, simply note that a perfect matching uniquely gives the σ, so it cannot be cancelled out, so a graph with a perfect matching has nonzero determinant.

Again, if we replace the indeterminates with numbers in the range $\{1, \ldots, N\}$ then the probability of a nonzero (indeterminate) determinant equaling 0 is at most $(2/N)^{|E|}$. This gives a probabilistic algorithm that will determine if a graph has a perfect matching with arbitrarily high probability. It is only slightly more difficult to find such a matching (again with very high probability).

Problem 5. Give an algorithm to find a perfect matching in a graph if one exists.

Confirm that G has a perfect matching. For each edge (i,j) determine if $G-(i,j)$ has a perfect matching. If so, then delete (i,j) and continue. Otherwise, place (i,j) in the matching and continue. After m attempts, the edge set is exhausted.

One very interesting reason for examining this algorithm is its ability to be computed in parallel. Imagine a computer with K processors, solving a problem with an $O(F)$ algorithm. Ideally, this computer would solve these problems in time $O(F/K)$. This is *optimal speedup*. Most algorithms do not have known optimal speedups for any K. Consider the following variant. Suppose we allow a computer to have any number of processors polynomial in the size of the problem. How fast can the computer solve the problem? For an NP–Complete problem it is clear that the computer cannot solve it in polynomial time (by any known algorithm). For a problem in P, it clearly requires at most polynomial time. The interesting cases are those that require less time: either logarithmic time or constant time. Very few of the latter class are known, so researchers have concentrated on algorithms that require time that is polynomial in the logarithm of the size of the problem. The class of problems solvable in polylog time is called *Nick's Class* or *NC* for short. The class of problems that is probabilistically likely to give a correct answer is Random NC (*RNC*).

Some examples of problems in NC are finding maximal independent sets, finding a depth first search path, two processor scheduling, and, most important to us, finding the determinant of a matrix and inverting a matrix ([52]). Together with the above argument, this shows that determining whether or not a graph has a perfect matching is in RNC. Unlike the case for sequential algorithms, however, the search and decision problems for parallel algorithms do not seem to be directly equivalent.

Problem 6. Can the algorithm given in problem 5 be run in parallel?

I don't think so.

To *find* a perfect matching in parallel requires a much more complicated algorithm. The algorithm presented here is due to Mulmuley, Vazirani, and Vazirani ([45]). Another algorithm is found in [38].

First note that if G has at most one perfect matching then there are no possibilities for cancellation. This algorithm extends this to a weighted case and shows that if a graph has a unique minimum weighted perfect matching then the determinant of a related matrix is nonzero. Suppose we have a graph with weights w_{ij} on each edge. Create the matrix A as above replacing the indeterminates x_{ij} with the value $2^{w_{ij}}$.

Problem 7. Suppose G has a unique minimum weight perfect matching with

weight w. Show that $\det(A) \neq 0$ and that the highest power of 2 which divides $\det(A)$ is 2^{2w}. (This modifies the proof of problem 4.)

Using the same auxilliary graph as in problem 4, it is easy to see that any σ with an odd cycle is still cancelled out. Now examine the permutation associated with any minimum matching M with weight w. Its value is $\pm 2^{2w}$. We need to show that any other permutation has a higher value. Certainly this is true for any non–minimum matching. Examine a permutation associated with even cycles. This can be seen to be the union of two matchings, at least one with weight more than w, so that term has value more than 2^{2w}.

This gives a quick method for determining the weight of the optimal matching: simply find the largest w so that 2^{2w} divides the determinant.

The same argument suffices to give a characterization of which edges are in the unique minimum perfect matching. Let A_{ij} be the minor of A with the ith row and jth column removed.

Problem 8. Show that (i, j) is in the unique minimum matching if and only if $(det(A_{ij})2^{w_{ij}})/2^{2w}$ is odd.

Notice that

$$det(A_{ij})2^{w_{ij}} = \sum_{\sigma:\sigma(i)=j} sign(\sigma)v(\sigma)$$

.

Examining the auxilliary graph, if there is an odd cycle, there is one that avoids (i, j), so the cancellation argument in problem 7 still holds.
If (i, j) is in the minimum weight matching, then there is a term with value $\pm 2^{2w}$ and all other terms have higher values. On the other hand, if it is not in the minimum weight matching all terms have higher values. Therefore, the statement follows.

The final step is to assign weights to an unweighted graph so that it contains a unique minimum weight matching. Suppose we assign weights uniformly and independently from $\{1, \ldots, 2|E|\}$.

Problem 9. Show that the probability there is a unique minimum matching is $\geq 1/2$.

This is a nice probabilistic proof. Fix the weights of all elements except some (i,j). For that (i,j), there is a value α such that if $w_{ij} > \alpha$ then (i,j) is in no minimum weight matching. Clearly, if $w_{ij} < \alpha$ then it is in every minimum weight matching. It is only if $w_{ij} = \alpha$ that there is a minimum weight matching that contains it and another that does not. We call such an element ambiguous. Note that this argument is independent of w_{ij}, so the probability that (i,j) is ambiguous is exactly the probability that $w_{ij} = \alpha$ which is $\leq 1/2n$. Therefore, the probability there is an ambiguous edge is $\leq n/2n = 1/2$. This is equal to the probability that there is not a unique minimum matching.

The problem of determining all the numbers required can be done in parallel by the algorithm of Pan [52]. This gives an algorithm that requires time $O(\log^2 m)$ with $O(n^{3.5}m)$ processors. This shows that finding a perfect matching is in RNC.

Some corollaries of this theorem are that the following are in RNC:

- constructing a maximum cardinality matching,
- constructing a matching in a graph with weighted vertices that covers a set of maximum weight (with weights in binary),
- finding a maximum flow in a directed graph with edge weights given in unary.

In contrast, the maximum flow problem with binary weights is complete for P (with a suitable definition of completeness). Showing it in RNC would imply that every problem in P is in RNC, a fairly unlikely possibility.

This algorithm is a *Monte Carlo* algorithm: it always returns an answer, but the answer may be wrong. A *Las Vegas* algorithm recognizes when it has the correct answer, so it either returns a correct answer, or it returns "failure" (it should be clear that any Las Vegas algorithm can be turned into a Monte Carlo algorithm). Karloff ([36]) gives a method for transforming any Monte Carlo algorithm for matching into a Las Vegas algorithm. Not surprisingly, it uses duality.

Recall from the Gallai–Edmonds theorem that the size of a maximum matching in a graph is related to the set of all nodes missed by at least one maximum matching ($D(G)$) and its neighbor set ($A(G)$). The size of a maximum matching is $(|V|+|A|-c(D))/2$ where $c(D)$ is the number of odd components in the subgraph induced by D. Therefore, if we identify $D(G)$ we can find the size of a maximum matching. We also know that the size of a perfect matching is always less than or equal to $(|V| + |S| - c(V - S))/2$ for any S implies that if we misidentify D then we get an upper bound on the size of a maximum matching.

Problem 10. Give a parallel algorithm to find D supposing you have an algorithm that finds a maximum matching with probability at least $1 - 1/2^n$. What is the probability of it finding an incorrect answer?

For each node v, determine if the graph $G - v$ has the same size matching as G. If so, place v in D, otherwise not. The probability of our oracle giving at least one wrong answer is at most $(n+1)2^{-n}$, so the probability of getting the correct D is $1-(n+1)2^{-n}$.

This gives a Las Vegas algorithm: Run the above algorithm and the matching algorithm given in parallel. If they agree on the answer, return it. Otherwise return "failure."

Finally, consider the exact matching problem: each edge of the graph is colored either red or blue and an integer k is given. The problem is to find a perfect matching with exactly k red edges. No deterministic polynomial algorithm is known for this problem, but it is easy to modify the above algorithm to solve it in random logarithmic time. According to all the assumptions about class inclusions, this should imply that there is a polynomial algorithm for this problem.

6. Weighted Matching

We have seen three different methods for the cardinality matching problem: augmenting paths, cut generation, and determinants. In this section we discuss generalizing these methods for the case where each edge has a weight and the objective is to maximize the total weight of the matching. We begin with an augmenting path algorithm due to Edmonds ([16, 17]).

We have a linear programming formulation for the cardinality case (see section 4). We have not yet proved that this formulation is sufficient for the general case. Again, we will prove the sufficiency by giving an algorithm that gives both primal and dual solutions. The primal problem is to

$$\text{Maximize } \sum c_{ij}x_{ij}$$

$$\sum_{\{j:\{i,j\}\in E\}} x_{ij} \le 1 \quad \text{for all } i \tag{6}$$

$$\sum_{\{i,j\}\in E: i\in S, j\in S} x_{ij} \le (|S|-1)/2 \quad \text{for all } S \subseteq V, S \text{ odd}. \tag{7}$$

$$x_{ij} \ge 0 \tag{8}$$

With the dual variables associated with 6 be u and those associated with 7 be w, the dual is

$$\text{Minimize } \sum_i u_i + \sum_S r_S w_S$$

$$u_i + u_j + \sum_{\{S:i,j\in S\}} w_S \leq c_{ij} \tag{9}$$

$$u, w \geq 0 \tag{10}$$

where r_S is $(|S|-1)/2$.

There are two ways to show that a primal and a dual solution are optimal: either show their costs are the same, or show that they satisfy complementary slackness. We adopt the second method here. The complementary slackness conditions are:

$$x_{ij} > 0 \Rightarrow u_i + u_j + \sum_{\{S:i,j\in S\}} w_S = c_{ij} \tag{11}$$

$$u_i > 0 \Rightarrow \sum_{\{j:\{i,j\}\in E\}} x_{ij} = 1 \tag{12}$$

$$w_S > 0 \Rightarrow \sum_{\{i,j\}\in E:i\in S,j\in S} x_{ij} = r_S \tag{13}$$

In this algorithm, we always have a primal feasible solution and a dual feasible solution. These solutions will satisfy 11 and 13. As soon as they also satisfy 12 then we terminate.

Problem 1. Give primal and dual feasible solutions that satisfy everything except 12.

The primal solution is $x_{ij} = 0$ for all i, j. Set $w_S = 0$ for all S and $u_i = 1/2c^*$ where c^* is the maximum cost in the graph.

To ensure that 11 is satisfied, we will only do augmentations in the *equality–constrained subgraph*: those arcs that satisfy 9 with equality. All edges outside of this subgraph will be set to zero. To ensure 13 we will only assign postive w values to odd sets that are shrunk in the current graph.

Given a current pair of solutions, there are two ways to satisfy 12 at a node: either the node becomes adjacent to a matching edge (by augmenting from it) or its dual u is decreased to zero. We create the alternating forest from the unmatched nodes using only edges in the equality–constrained subgraph (perhaps shrinking blossoms along the way). Either an augmenting path is found or we will be able to

decrease the duals on the unmatched nodes keeping primal and dual feasibility as well as 11 and 13. Let's begin by seeing how this dual change can be done. Suppose we decrease the duals of all the unmatched nodes by δ. Let the alternating forest be F.

Problem 2. What to we have to do to the duals of nodes next to an unmatched node in F to satisfy 9? What about those next to *them*? In general, what do we have to do to the duals of nodes based on their label odd or even? What about unlabeled nodes?

We must increase the duals next to the unmatched nodes, then decrease the next layer of nodes and so on. Every even node (or node in an even pseudonode) has its nodes decreased by δ and every odd node (or node in an odd pseudonode) is increased by δ. Unlabelled nodes are not changed.

It turns out that even nodes, or nodes contained in pseudonodes marked even have their duals decreased by δ. Odd nodes, or those in odd pseudonodes, are increased by δ. This has a bad effect on edges completely within a shrunk blossom. Fortunately, we can modify the w values to offset this.

Problem 3. How can the w values be changed to keep 9 for edges within a blossom?

The dual for odd blossoms can be decreased by 2δ and that for even blossoms increased by 2δ.

We still have to find δ. As we change the duals we must be certain that we do not violate dual feasibility.

Problem 4. There are four possibilities for violating dual feasibility. What are they?

We decrease duals in two cases: the u for even nodes (by δ) and the w for odd blossoms (by 2δ). Neither of these values can go negative. An edge between two even nodes in different blossoms has its reduced cost decreased by 2δ and this cannot go below c_{ij}. Finally, an edge between an even node and an unlabeled node has its reduced cost decreased by δ and it also cannot go below c_{ij}.

This gives an algorithm for doing the dual change. Simply calculate the largest δ that keeps dual feasibility. After the dual change we expand all blossoms with $w_S = 0$. Those with $w_S > 0$ are not expanded. This implies that shrunk blossoms may end up later as odd nodes.

As new edges are added to the graph it may be that an edge is added between two even nodes. If the endnodes are in different trees then an augmentation can be made; otherwise a blossom can be shrunk. After this, the forest can be retained and the search for an augmenting path resumed.

The algorithm is as follows:

0) Find a primal and dual feasible solution that satisfy 11 and 13.

1) Create an alternating forest in the equality–constrained subgraph. If an augmenting path is found, go to step 2. Otherwise go to step 3.

2) Augment along the path. Expand all blossoms with $w_S = 0$. If $u_i = 0$ for all unmatched nodes then stop. Otherwise go to step 1.

3) Change the dual solution using the calculations above. If $u_i = 0$ for all unmatched nodes then stop. Otherwise add edges to the equality constrained subgraph. If an edge is added between two even nodes in different components then go to step 2. If an edge is added between two even nodes in the same component then shrink the blossom. Go to step 1, continuing from the current alternating forest.

Problem 5. Show that 11 and 13 are satisfied after an augmentation.

Since we only augment on the equality–constrained subgraph, 11 *must be satisfied. For* 13 *not to be satisfied, we must augment through a set* S *with* $w_S > 0$. *But in that case,* S *is a pseudonode, so after augmenting* r_S *edges of it are used.*

Problem 6. Show that the dual change is nondegenerate ($\delta > 0$).

We will show that each of the four bounds must not be zero.
We know that at least one unmatched node has dual ¿ 0 (since there is such an unmatched node). But it must always have been unmatched. Furthermore, since we began with each node having the same dual, an unmatched node must have the lowest dual value of all nodes (decreases happen only to even nodes and an unmatched node is always even). Therefore, all even nodes have dual ¿ 0.
We decrease blossom duals only if the blossom is odd. But an odd blossom cannot be formed this iteration, so it must have been formed at a previous augmentation. Then its w value must have been ¿ 0 (or else we would have expanded it.
For an edge with two even endnodes, if its 9 *value was at equality then it would be in the equality–constrained subgraph so would give rise to either a blossom or an augmenting path. Therefore such an edge between two blossoms is not in the equality–constrained subgraph, so* $\delta > 0$.
Similary an edge between an even node an an unlabeled node cannot be in the equality–constrained subgraph, so $\delta > 0$.

Problem 7. Show that 11 and 13 are satisfied after a dual change.

This is a tedious case analysis, but follows directly from our argument for how to change the duals.

The only other point to prove is that the algorithm eventually terminates. It is clear that no more than $n/2$ augmentations can be done. We can bound the number of dual changes between augmentations.

Problem 8. Show that the number of dual changes between augmentations is $O(n)$. (Hint: consider the ways δ is bounded and show that each way cannot occur too often).

Consider the four bounding cases.
If we ever are bound by the dual of an even node, all unmatched nodes must get dual 0, so we terminate.
If we are bound by an odd pseudonode, then we expand that pseudonode. We do not create new odd pseudonodes until we augment and there are $O(n)$ odd pseudonodes, so this case can occur only $O(n)$ times.
If we are bound by an edge with two even endnodes, we either augment or shrink a blossom. Since we do not expand even pseudonodes until after augmentation, this case can occur only $O(n)$ times.
If we are bound by an edge from an even node to an unlabeled node we add that node to the alternating tree. After $O(n)$ such additions, there are no more nodes to add.
Overall, there are $O(n)$ dual changes.

Problem 9. What is the complexity of this algorithm?

Between dual changes, $O(m)$ work must be done, so the total time is $O(n^2m)$.

This algorithm shows a number of very important points: the constraints 6, 7, and 8 are sufficient to define the matching polytope. Since these are the same constraints as we used in section 4 for cut generation this implies that all the results there hold just as well for weighted matching.

There are a number of programming tricks known to make this algorithm more efficient. Lawler ([39]) gives an $O(n^3)$ implementation. The fastest known implementation is that of Ball and Derigs ([7]), who have a very nice examination of various strategies for implementing matching algorithms.

No determinant technique for weighted matching is known. A parallel implementation is very unlikely (unless the weights are given in unary), although no conjectured class inclusions preclude it.

7. Generalizations of Matchings

There are a number of generalizations of matchings. In this section we will examine the *capacitated b–matching problem*, a very powerful generalization.

We begin with the *uncapacitated b–matching problem*. In the previous sections, we examined the problem of finding a set of edges so that no node is adjacent to more than one edge. Suppose there is an integer b_i associated with each node i. A natural generalization is to assign integer values to the edges so that the total on the edges incident to node i is no more than b_i for all nodes. This is the uncapacitated b–matching problem. The problem examined in the previous sections has $b_i = 1$ for all i so is termed the *1–matching problem*.

Problem 1. Show that the uncapacitated b–matching problem can be reduced to the 1–matching problem. Is this a polynomial reduction?

Replace each node i with b_i copies. For each arc (i,j) create an arc from each copy of i to each copy of j.

It is also possible to insist that some or all of the nodes have values totaling exactly b_i next to it (the relationship is exactly the same as that between perfect and nonperfect matchings).

Although the reduction is not polynomial, it is possible to examine how a 1–matching algorithm works on this graph and modify it to solve b–matchings. See, for example, Pulleyblank ([53]) and Anstee ([5]).

Another variant is to place upper bounds on the values that can be assigned to each edge. If every upper bound is 1, then the problem is called the *b–factor*, or *f–factor problem*; arbitrary capacities result in the *capacitated b–matching problem.* Again, this is just 1–matching in disguise (provided a psuedopolynomial reduction is allowed).

Problem 2. Show that the capacitated b–matching problem can be reduced to the 1–matching problem.

Replace each edge (i,j) with capacity $u(i,j)$ with a path of three edges (i,k), (k,k'), (k',j). Let the $b(k) = b(k') = c(i,j)$ and insist that k and k' have values exactly $c(i,j)$ incident to it. The edges (i,k) and (k',j) must be given the same value, and that value must be less than or equal to $c(i,j)$. This gives the value of (i,j) in the original graph.

Again, the 1–matching algorithms can be streamlined on these special graphs to lead to polynomial algorithms.

Bibliography

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, "Network flows," Sloan W.P. No. 2059–88, Sloan School of Management, M.I.T., Cambridge, MA (1988).

[2] R.K. Ahuja and J.B. Orlin, "A fast and simple algorithm for the maximum flow problem," Sloan W.P. No. 1905–87, Sloan School of Management, M.I.T., Cambridge, MA (1987).

[3] R.K. Ahuja and J.B. Orlin, "Improved primal simplex algorithms for shortest path, assignment and minimum cost flow problems," Sloan W.P. No, 2090–88, Sloan School of Management, M.I.T., Cambridge, MA (1988).

[4] R.K. Ahuja, J.B. Orlin, and R.E. Tarjan, "Improved time bounds for the maximum flow problem," Sloan W.P. No. 1966–87, Sloan School of Management, M.I.T., Cambridge, MA (1988).

[5] R.P. Anstee, "A polynomial algorithm for b–matchings: an alternative approach," University of Waterloo Research Report CORR 83–22 (1983).

[6] M.O. Ball, L. Bodin and R. Dial, "A matching based heuristic for scheduling mass transit crews and vehicles," *Transportation Science*, **17**, 4–31 (1983).

[7] M.O. Ball and U. Derigs, "An analysis of alternative strategies for implementing matching algorithms," *Networks*, **13**, 517–550 (1983).

[8] F. Barahona and Éva Tardos, "Note on Weintraub's minimimum cost flow algorithm," manuscript (1988).

[9] D.P. Bertsekas, "Distributed asynchronous relaxation methods for linear network flow problems," Lab. for Decision Systems LIDS–P–1986, M.I.T., Cambridge, MA (1985).

[10] N. Chistofides, *Graph Theory — An Algorithmic Approach*, Academic Press, London (1975).

[11] D. Conradt and U. Pape, "Maximales Matching in Graphen," in H. Späth, *Ausgewählte Operations Research in FORTRAN*, Oldenbourg, München (1980).

[12] W.H. Cunningham, "A network simplex method," *Mathematical programming*, **11**: 105–116 (1976).

[13] W.H. Cunningham, "Theoretical properties of the network simplex method," *Mathematics of Operations Research*, **4**:196–208 (1979).

[14] U. Derigs, *Programming in networks and graphs*, Lecture Notes in Economics and Mathematical Systems 300, Springer–Verlag, Berlin (1988).

[15] E.A. Dinits, "Algorithm for solution of a problem of maximal flow in a network with power estimation," *Soviet Math. Dokl.*, **11**, 1277–1280 (1970).

[16] J. Edmonds, "Paths, trees, and flowers," *Canadian Journal of Mathematics*, **17**, 449–467 (1965).

[17] J. Edmonds, "Maximum matching and a polyhedron with 0–1 vertices," *Journal of Research of the National Bureau of Standards*, **69B**, 125–130 (1965).

[18] J. Edmonds and E.L. Johnson, "Matching, euler tours, and the chinese postman," *Mathematical Programming*, **5**, 88–124 (1973).

[19] J. Edmonds and R.M. Karp, "Theoretical improvements in algorithmic efficiency for netowrk flow problems," *Journal of the Association for Computing Machinery*, 19, 248–264 (1972).

[20] J. Elam, F. Glover, and D. Klingman, "A strongly convergent primal simplex algorithm for generalized networks," *Mathematics of Operations Research*, **4**:39–59 (1979).

[21] S. Even and O. Kariv, "An $O(n^{2.5})$ algorithm for maximum matching in general graphs," *Proc. 16th Annual Symposium of the Foundations of Computer Science*, 100–112 (1975).

[22] L.R. Ford and D.R. Fulkerson, *Flows in Networks*, Princeton Unversity Press, Princeton, N.J. (1962).

[23] A. Frank, "Finding feasible vectors of Edmonds–Giles polyhedra," *Journal of Combinatorial Theory, Series B*, 36, 221–239 (1984).

[24] M. Fujii, T. Kasami, and K. Ninomiya, "Optimal sequencing of two equivalent processors," *SIAM Journal of Applied Mathematics*, **17**, 784–789 (1969).

[25] H. Gabow, "An efficient implementation of Edmonds' algorithm for maximum matchings on graphs," *Journal of the ACM*, **23**, 221–234 (1975).

[26] G. Gallo, M.D. Grigoriadis, and R.E. Tarjan, "A fast parametric maximum flow algorithm," LCSR–TR–95, Laboratory for Computer Science Research, Rutgers University, Rutgers, NJ (1987).

[27] A.V. Goldberg, S.A. Plotkin, É. Tardos, "Combinatorial algorithms for the generalized circulation problem," MIT/LCS/TM–358, Laboratory for Computer Science, MIT, Cambridge, MA (1988).

[28] A.V. Goldberg and R.E. Tarjan, "A new approach to the maximum flow problem," *Proceedings of the Eighteenth Annual ACM Symposium on the Theory of Computing* (1986).

[29] A.V. Goldberg and R.E. Tarjan, "Solving minimum–cost flow problems by successive approximation," *Proc. 19th ACM Symposium on the Theory of Computing*, 7–18 (1987).

[30] A.V. Goldberg and R.E. Tarjan, "Finding minimum–cost circulations by canceling negative cycles," *Proc. 20th ACM Symposium on the Theory of Computing*, 388–397 (1988).

[31] D. Goldfarb and J. Hao, "A primal simplex algorithm that solves the maximum flow problem in at most nm pivots and $O(n^2m)$ time," Department of Industrial Engineering and Operations Research, Columbia University, New York (1988).

[32] D. Goldfarb, J. Hao, and S. Kai, "Anti–stalling rules for the network simplex algorithm," Department of Industrial Engineering and Operations Research, Columbia University, New York (1987).

[33] R.E. Gomory and T.C. Hu, "Multi–terminal network flows," *SIAM Journal of Applied Math.*, **9**, 551–556 (1961).

[34] M. Grötschel and O. Holland, "Solving matching problems with linear programming," *Mathematical Programming*, **33**, 243–259 (1985).

[35] J.E. Hopcroft and R.M. Karp, "An $n^{5/2}$ algorithm for maximum matching in bipartite graphs," *SIAM Journal of Computing*, **2**, 225–231 (1973).

[36] H. Karloff, "A randomized parallel algorithm for the odd set cover problem," *Combinatorica*, **6**, 387–391 (1986).

[37] R.M. Karp, "A characterization of the minimum cycle mean in a digraph," *Discrete Mathematics*, 23, 309–311 (1978).

[38] R.M. Karp, E. Upfal, and A. Wigderson, "Finding a maximum matching is in random NC," *Seventeenth Annual Symposium on the Theory of Computing*, 22–32 (1985).

[39] E.L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt Reinhart and Winston, New York (1976).

[40] E.L. Lawler and C.U. Martel, "Computing maximal polymatroidal flows," *Mathematics of Operations Research*, 7, 334–347 (1982).

[41] L. Lovász and M.D. Plummer, *Matching Theory*, North Holland, New York (1986).

[42] V.M. Malhotra, M. Pramodh Kumar, and S.N. Maheshwari, "An $O(V^3)$ algorithm for finding maximum flows in networks," *Information Processing Letters* 7, 277–278 (1978).

[43] S. Micali and V.V. Vazirani, "An $O(\sqrt{V}E)$ algorithm for finding maximum matching in general graphs," *Proc. 21st Annual Symposium on the Foundations of Computer Science*, 17–27 (1980).

[44] B. Montreuil, H.D. Ratliff, and M. Goetschalckx, "Matching based interactive facility layout," *IEE Transactions* (1988).

[45] K. Mulmuley, U.V. Vazirani, and V.V. Vazirani, "Matching is as easy as matrix inversion," *Combinatorica*, **7**, 105–113 (1985).

[46] G.L. Nemhauser and G. Weber, "Optimal set partitioning, matchings, and lagrangian relaxation," *Naval Research Logistics Quarterly*, **26**, 553–563 (1979).

[47] K. Onaga, "Dynamic programming of Oprimum flows in lossy communications nets," *IEEE Transactions on Circuit Theory*, **13:**282–287 (1966).

[48] J.B. Orlin, "On the simplex algorithm for networks and generalized networks," *Mathematical Programming Studies*, **24**:166–178 (1985).

[49] J.B. Orlin, "A faster strongly polynomial minimum cost flow algorithm," *Proc. 20th ACM Symposium on the Theory of Computing*, 377–387 (1988).

[50] J.B. Orlin and R.K. Ahuja, "New distance–directed algorithms for maximum flow and parametric maximum flow problems," Sloan W.P. No. 1908–87, Sloan School of Management, Cambridge, MA (1987).

[51] M.W. Padberg and M.R. Rao, "Odd minimum cut–sets and b–matchings," *Mathematics of Operations Research*, **7**, 67–80 (1982).

[52] V. Pan, "Fast and efficient algorithms for the exact inversion of integer matrices," *Fifth Annual Foundations of Software Technology and Theoretical Computer Science Conference*, (1985).

[53] W.R. Pulleyblank, *Faces of the Matching Polyhedron*, University of Waterloo, Ph.D. Thesis (1973).

[54] J.T. Schwartz, "Fast probabilistic algorithms for verification of polynomial identitities," *Journal of the ACM*, **27**, 701–717 (1980).

[55] É. Tardos, "A strongly polynomial minimum cost circulation algorithm," *Combinatorica*, 5, 247–255 (1985).

[56] É. Tardos, C.A. Tovey, and M.A. Trick, "Layered augmenting path algorithms," *Mathematics of Operations Research*, 11, 362–370 (1986).

[57] R.E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA (1983)

[58] R.E. Tarjan "A simple version of Karzanov's blocking flow algorithm," *Operations Research Letters,* 2, 265–268 (1984).

[59] R.E. Tarjan, "Efficiency of the primal network simplex algorithm for the minimum–cost circulation problem," manuscript (1988).

[60] M.A. Trick, *Networks with Additional Structured Constraints*, Georgia Institute of Technology, Atlanta, GA (1987).

[61] K. Truemper, "On max flows with gains and pure min–cost flows," *SIAM Journal of Applied Mathematics*, **32:** 450–456 (1977).

[62] A. Weintraub, "A primal algorithm to solve network flow problems with convex costs," *Management Science*, 21, 87–97 (1974).