E.G. Coffman et al., Eds., *Handbooks in OR & MS, Vol. 3*

Chapter 8

Design (with Analysis) of Efficient Algorithms

Dan Gusfield

Department of Computer Science, University of California, Davis, CA 95616, U.S.A.

1. Introduction

This chapter is an introduction to the design (with analysis) of efficient computer algorithms. The main theme of this chapter will be to illustrate the interelationship between mathematical insight, data structures, and the design (with analysis) of 'provably' efficient algorithms. The process of designing an efficient algorithm is interwoven with its analysis, the analysis of the data structures to be used, and often with the discovery of mathematical structure underlying the problem that the algorithm is to solve.

The approach taken in this chapter is an experiment. We have chosen not to survey the entire (huge) field of algorithmic design, for to do so would not permit rigorous treatment of any topic. We have chosen instead to discuss a large range of current and historical issues by focusing on a single set of related problems. We will look in detail at the task of calculating a *maximum flow* and a *minimum cut* in a network, along with several associated problems. The intention is not to provide a comprehensive or completely up-to-date discussion of network flow algorithms, but rather to use this focus as a means to discuss in some detail many major ideas in the design or analysis of efficient algorithms, and the computational models that these algorithms are designed for.

We will start with the most basic network flow algorithm for a sequential machine and show how it has been successively improved and changed by additional insights into the network flow problem itself, by new algorithmic ideas, by new data structures, by new methods of analysis, by changes in the accepted notion of 'efficiency', and by changes in the assumed computational model. In this way we will see how related questions are answered under most of the computational settings of current interest. We will discuss sequential algorithms, parallel algorithms, randomized algorithms, parametric algorithms, distributed algorithms, amortized time analysis, approximation algorithms, all-for-one results, results based on preprocessing, and strong versus weak polynomial time algorithms.

2. Maximum network flow on a sequential machine

In this first section we examine the problem of efficiently computing a maximum network flow and minimum-cut, where the computation is to be done on a sequential machine (RAM model) and the measure of goodness of an algorithm is its worst-case running time. We will follow in spirit an abridged history of the improvements, allowing us to illustrate the interplay between the unfolding mathematics of the flow problem, the data-structures proposed, and the resulting algorithmic ideas. However, the history will be apocryphal in places, and it will not detail sparse versions of the results, nor the most recent advances in this field. For a comprehensive discussion of these more recent improvements, see Goldberg, Tardos & Tarjan [1990] and Ahuja, Magnanti & Orlin [1989].

Definitions. Let $G = (N, E)$ be a *directed* graph on the set of nodes N and set of edges E, and let $c(i, j)$ be a positive real number on directed edge (i, j), called the *capacity* of (i, j). In our notation, the edge is directed from the first node in the pair to the second node. We designate two particular nodes s and t as the *source* and the *sink*, respectively. A *flow* f is an assignment of real numbers to edges such that the following conditions are satisfied:

(1) For every edge (i, j), $0 \leq f(i, j) \leq c(i, j)$. This is called the *capacity constraint*.

(2) For every node i other than s or t,

$$\sum_{j:\,(j,i)\in E} f(j, i) = \sum_{j:\,(i,j)\in E} f(i, j)\,.$$

In other words, the flow into i equals the flow out of i. This constraint is called the *conservation constraint*.

It is easy to prove that in any flow f, $\Sigma_i f(s, i) - \Sigma_i f(i, s) = \Sigma_i f(i, t) - \Sigma_i f(t, i)$, and we call this the *value* of flow f, and denote it by $f_{s,t}$, or by f when the source and sink are clear by context. Intuitively, the flow value is the net amount of flow that is sent out of s and also the net amount of flow that is received at t. We will henceforth assume that there are no edges directed into s or out of t, since such edges are useless in computing a maximum s, t flow. Consequently the flow value $f_{s,t} = \Sigma_i f(s, i) = \Sigma_i f(i, t)$.

An s, t *cut* of G is a partition (X, Y) of the nodes of G where $s \in X$ and $t \in Y$. The *capacity* of the cut (X, Y), denoted $C(X, Y)$, is the sum of the capacities of the edges directed from X to Y, i.e., $C(X, Y) = \Sigma_{i\in X, j\in Y} c(i, j)$. Given a flow f, and an s, t cut (X, Y), the net flow across the cut, denoted $f_{(X,Y)}$, is

$$\sum_{i\in X, j\in Y} f(i, j) - \sum_{i\in Y, j\in X} f(i, j)\,.$$

The connection between s, t flows and s, t cuts is very fundamental. The first

fact, which follows immediately from the definitions, is that for any s, t cut (X, Y) and any s, t flow f, $f_{(X,Y)} \leq C(X, Y)$. That is, the net flow across any cut cannot exceed the capacity of the cut. The next fact, which is intuitive, is that for any s, t flow f and any s, t cut (X, Y), $f_{s,t} \leq f_{(X,Y)}$. That is, the flow from s to t cannot exceed the net flow across any s, t cut. This fact is physically intuitive, and we omit a formal proof of it, although in general in this field it is not a good idea to rely exclusively on physical intuition.

Combining the above two facts we have the following lemma and the theorem that follows immediately from it.

Lemma 2.1. *For any flow f and any s, t cut (X, Y), $f_{s,t} \leq C(X, Y)$.*

Theorem 2.1. *If there is an s, t cut (X, Y) and an s, t flow f such that $f_{(X,Y)} = C(X, Y)$, then f is a maximum s, t flow and (X, Y) is a minimum capacity s, t cut.*

In the next section we will show that the converse of this theorem also must hold.

2.1. First methods

A history of maximum flow algorithms will normally begin with the Ford–Fulkerson algorithm, although there were related mathematical theorems and even algorithmic methods that precede that method.

The Ford–Fulkerson method starts with an assignment of zero flow f to each edge, i.e., $f(i, j) = 0$ for each edge (i, j). At a general iteration of the algorithm there is a flow f which is not necessarily maximum. From that flow f, the algorithm constructs a graph G^f, called the *residual* graph of f, according to the following rules:

(1) If $f(i, j) > 0$, then create the edge (j, i) in G^f and assign it a capacity of $f(i, j)$. Edges of this type are called *backward* edges.

(2) If $f(i, j) < c(i, j)$, then create the edge (i, j) in G^f and assign it a capacity of $c(i, j) - f(i, j)$. Edges of this type are called *forward* edges.

Note that every edge in the residual graph has a strictly positive capacity. The residual graph is used to determine whether f is a maximum flow, and if not, to indicate how to augment the flow. These two tasks are accomplished in the Ford–Fulkerson algorithm by details suggested in the following theorem.

Theorem 2.2. *The flow f is a maximum s, t flow if and only if there is no directed path from s to t in G^f.*

Proof. Let S be the set of nodes which are reachable from s via some directed path in G^f, and let $T = V - S$.

Suppose first that $t \notin S$, so that (S, T) is an s, t cut. Consider the capacity of the cut. For every edge (i, j) in G where $i \in S$ and $j \in T$, it must be that

$f(i, j) = c(i, j)$, for otherwise edge (i, j) would be a forward edge in G^f and so j would be reachable from s and hence j would be in S. Similarly, for every edge (i, j) where $i \in T$ and $j \in S$, it must be that $f(i, j) = 0$, for otherwise edge (j, i) would be a backward edge in G^f and i would be in S. Hence

$$f_{(S,T)} = \sum_{i \in S, j \in T} f(i, j) - \sum_{i \in T, j \in S} f(i, j) = \sum_{i \in S, j \in T} c(i, j) = c(S, T) .$$

Then, by Lemma 2.1, f is a maximum s, t flow and (S, T) is a minimum s, t cut.

For the converse, suppose that $t \in S$ and consider a directed simple path (one with no cycles) P in G^f from s to t that starts at s and ends at t. Let δ be the minimum capacity in G^f of the edges on P. We will show that the total flow from s to t can be increased to $f'_{s,t} = f_{s,t} + \delta$. To accomplish this, for every edge (i, j) on P which is a forward edge in G^f, set $f'(i, j)$ to $f(i, j) + \delta$; for every edge (i, j) on P which is backward edge in G^f, set $f'(j, i)$ to $f(j, i) - \delta$; for every other edge on G, set $f'(i, j)$ to $f(i, j)$.

We will show that f' is an $s - t$ flow of value $f_{s,t} + \delta$. First, note that s is incident with exactly one edge (directed out of s) on P, t is incident with exactly one edge (directed into t) on P (and both must be forward edges), hence $\Sigma_i f'(s, i) = \delta + \Sigma_i f(s, i)$ and $\Sigma_i f'(i, t) = \delta + \Sigma_i f(i, t)$.

Next, note that for every other node i on P is incident with exactly one edge on P directed in, and exactly one of P directed out of i. We will show that f' satisfies the conservation constraint at each such node i. If both the edges of P incident with i are forward edges, exactly one is into i and one is out of i, so

$$\sum_{j:(j,i) \in E} f'(j, i) = \delta + \sum_{j:(j,i) \in E} f(j, i)$$

and

$$\sum_{j:(i,j) \in E} f'(i, j) = \delta + \sum_{j:(i,j) \in E} f(i, j) .$$

Since f is a flow, it follows that

$$\sum_{j:(j,i) \in E} f'(j, i) = \sum_{j:(i,j) \in E} f'(i, j) .$$

A similar argument holds if both are backward edges. Now if the edge of P into i is a forward edge, and the edge of P put of i a backward edge, then

$$\sum_{j:(j,i) \in E} f'(j, i) = \delta - \delta + \sum_{j:(j,i) \in E} f(j, i)$$

and

$$\sum_{j:(i,j) \in E} f'(i, j) = \sum_{j:(i,j) \in E} f(i, j)$$

hence

$$\sum_{j:\,(j,i)\in E} f'(j,i) = \sum_{j:\,(i,j)\in E} f'(i,j)\,.$$

A similar argument holds if the in-edge of P at i is a backward edge and the out-edge of P at i is a forward edge. So the assignment f' satisfies the conservation requirement.

The only thing left to check is that $0 \leq f'(i,j) \leq c(i,j)$ for each edge (i,j). Recall that δ is the minimum of capacities in G^f of the edges on P. So, if (i,j) is a forward edge, then

$$0 \leq f(i,j) + \delta = f'(i,j) \leq f(i,j) + c(i,j) - f(i,j) \leq c(i,j)\,.$$

If (i,j) is a backward edge, then

$$c(j,i) \geq f(j,i) \geq f'(j,i) = f(j,i) - \delta \geq f(j,i) - f(j,i) = 0\,.$$

Hence we have shown that f' is an s,t flow, and since $\Sigma_i f'(i,t) = \delta + \Sigma_i f(i,t)$, and $\delta > 0$, it follows that $f'_{s,t} > f_{s,t}$, and f is not a maximum flow. □

The Ford–Fulkerson method

All the essential elements of the Ford–Fulkerson method, and most of the proof of correctness, have been outlined in the proof of Theorem 2.2. In detail, the algorithm is the following:

(1) Set $f(i,j) = 0$ for every edge (i,j).
(2) Construct the residual graph G^f from f.
(3) Search for a directed path P from s to t in G^f. If there is none, then stop, f is a maximum flow.
Else, find the minimum capacity δ in G^f of any of the edges on P.
(4) If (i,j) is a forward edge on P, then set $f(i,j)$ to $f(i,j) + \delta$. If (i,j) is a backward edge on P, then set $f(j,i)$ to $f(j,i) - \delta$.
(5) Return to Step 2.

2.2. Termination

Since $\delta > 0$, every iteration of the Ford–Fulkerson algorithm increases the amount of flow $f_{s,t}$ sent from s to t. Further, since edge capacities are only changed by addition and subtraction operations, if all the edge capacities in G are integral, then δ is always integral and hence at least one. Since the maximum flow is bounded by the capacity of any s,t cut (Lemma 2.1), we have the following theorem.

Theorem 2.3. *If all the edge capacities are finite integers, then the Ford–Fulkerson algorithm terminates in a finite number of steps.*

By the same reasoning, it is also easy to see that the theorem holds when all the capacities are rational. However, the theorem does not hold for irrational capacities. We will see later how to fix this.

Now, in the proof of Theorem 2.2, *if* the algorithm terminates, the final flow f saturates the s, t cut S (defined in the proof of Theorem 2.2), so that f is a maximum flow of value $f_{s,t}$ and $(S, V - S)$ is a minimum s, t cut of capacity $f_{s,t}$. So for the case when the capacities of G are rational, we have proved the converse to Theorem 2.1, the famous *max-flow min-cut theorem*:

Theorem 2.4. *The maximum s, t flow value is equal to the minimum capacity of any s, t cut.*

Theorem 2.4 is an example of a *duality theorem* or *min = max* theorem. Such duality theorems appear extensively in combinatorial optimization, and often are the key to finding efficient methods, and to their correctness proofs.

There is a problem in extending the theorem to the case of irrational capacities: we do not yet have a proof that the Ford–Fulkerson algorithm terminates when capacities are irrational. In fact, it is known that the algorithm, as given, might not terminate in this case. This issue will be resolved in Section 3.3 and then the max-flow min-cut theorem for any capacities will have been proven.

Efficiency

Eventual termination is not the only issue. We want the algorithm to terminate as quickly as possible. Unfortunately, even in the case that all the capacities are integral, and hence the algorithm terminates, the Ford–Fulkerson algorithm can require as many as $f_{s,t}$ iterations of Step 3 [Ford & Fulkerson, 1962]. Hence the only provable time bound for the algorithm (with integer capacities) is $O(ef_{s,t})$, where e is the number of edges. Each residual graph and augmenting path can certainly be found in $O(e)$ time.

The time bound of $O(ef_{s,t})$ is not considered a polynomial time bound in either a strong or a weak sense. To be a (weakly) polynomial time bound, it must grow no faster than some polynomial function of the total number of bits used to represent the input. But a family of examples can be constructed where the capacities of the edges can be represented in $O(\log f_{s,t})$ bits, and where the algorithm uses $\Omega(f_{s,t})$ iterations. Hence, the number of iterations is exponentially larger than the number of bits used in the input.

A stronger notion of a polynomial bound would require that the bound grow no faster than some polynomial function of the number of items in the input, i.e., $e + n$. The bound $O(ef_{s,t})$ certainly does not fit that criterion since $f_{s,t}$ is not even a function (let alone a polynomial) of n and e. We will discuss the

distinction between strong and weak polynomial bounds more deeply in Section 14.

The first strongly polynomial bound for network flow was shown by Dinits [1970] and independently by Edmonds & Karp [1972], who both proposed a modification of the Ford–Fulkerson algorithm to be discussed below. They proved that the modified algorithm terminates correctly within $O(n^5)$ time, where $n = |N|$. This time bound is correct even if the edge capacities are irrational. We will examine the Edmonds–Karp method as we derive the even faster Dinits method.

3. Ford–Fulkerson leads 'naturally' to Dinits

In this section we develop the Dinits algorithms for network flow. We also show a continuity of ideas that leads 'naturally' from the Ford-Fulkerson and Edmonds–Karp methods to the Dinits method. The word 'naturally' is in quotes because the continuity was seen only in retrospect, and because the Dinits algorithm actually predates that of Edmonds–Karp. The Dinits method was developed in the Soviet Union in 1970, but became known in the West only in the later part of that decade. During that time a different algorithm containing some of the ideas of the Dinits method was independently developed in the West by Jack Edmonds and Richard Karp, but the ideas were not as fully developed as in the Dinits method. In fact, it was not even recognized, when the Dinits method first became known in the West, that the Dinits methods could be viewed as a natural refinement of the Ford–Fulkerson method – it looked very different at first. We now see it as essentially a more efficient implementation of the Ford–Fulkerson method.

The Ford–Fulkerson (FF) method is a fairly natural algorithm not far removed from the definitions of flow. The Edmonds–Karp (EK) and Dinits algorithms to be discussed can be derived from the FF algorithm by exploiting deeper observations about the behavior of the FF algorithm. As a result, these algorithms are less natural and farther removed from the basic definitions of flow.

3.1. Path choice for the Ford–Fulkerson method

The Ford-Fulkerson algorithm builds a succession of residual graphs, finds an augmentation path in each, and uses the path to augment the flow. However, there can be more than one s, t path in a residual graph, and the method does not specify which path to use. It is easy to construct networks [Ford & Fulkerson, 1962] where the Ford–Fulkerson method could use just a few augmenting paths, but if it chose paths unwisely, it would use a huge number of paths. Hence the question of which paths to select is important. One reasonable suggestion is to pick the path that increases the flow by the largest amount. This idea was explored in the early 1970s [Edmonds & Karp,

1972], but a different idea was found to be better. The idea is to choose the augmentation path with the *fewest* edges. This is the first idea of the Dinits method, and is also the key idea in the Edmonds–Karp method [Edmonds & Karp, 1972]. Later, this same idea was applied to the bipartite matching problem, yielding the fastest known method for that problem [Hopcroft & Karp, 1973].

We hereafter refer to the Ford–Fulkerson algorithm where each augmentation path is the shortest s, t path in the residual graph, as the EK algorithm.

We now explore the idea of choosing the augmentation path with fewest edges (the EK algorithm), and show how the Dinits algorithm evolves naturally from it, although we note again that this exposition is a corruption of the true history of network flow algorithms.

Definition. For i from 1 to r, let G^i be the ith residual graph constructed by the EK algorithm, and let P_i be the s, t path found by the algorithm in G^i. Then for node v, let $D^i(v)$ be the smallest number of edges of any v, t path in G^i, and let $d_i = D^i(t)$. Any v, t path with $D^i(v)$ edges will be called a *shortest* v, t path.

The following two facts are easy to verify, and are left to the reader (these facts are true for the Ford–Fulkerson algorithm as well as for the EK algorithm).

Fact 1. *The capacity in G^{i+1} of an edge (x, y) is less than its capacity in G^i if and only if edge (x, y) is on P_i.*

Fact 2. *The capacity in G^{i+1} of an edge (x, y) is greater than its capacity in G^i if and only if the edge (y, x) is in G^i and is on P_i. As a special case of this, any edge (x, y) is in $G^{i+1} - G^i$ only if the edge (y, x) is in G^i and is on P_i.*

As a consequence of these facts, the capacities of all edges not in P_i are the same in G^i and G^{i+1}.

Lemma 3.1. *For i from 1 to r, and for any node v, $D^i(v) \leq D^{i+1}(v)$, and so $1 \leq d_1 \leq d_2 \leq \cdots \leq d_r \leq n$.*

Proof. Let P_i be a shortest s, t path in G^i. The EK algorithm augments flow on P_i, and then creates G^{i+1} from G^i by changing some edge capacities, by deleting any edges whose capacities fall to zero, and by possibly adding some new edges not in G^i.

To see how $D^{i+1}(v)$ compares to $D^i(v)$, we create G^{i+1} from G^i in two steps: first, delete all the edges in $G^i - G^{i+1}$; second, add in all the edges in $G^{i+1} - G^i$. After the first step, $D^{i+1}(v) \geq D^i(v)$, since deletion of edges from G^i certainly does not decrease any D value. We will add the new edges in one at a time and see that after each addition the D values remain the same or increase, but never decrease.

Let (x, y) be a new edge added. By Fact 2, edge (x, y) in G^{i+1} is the reversal

of an edge (y, x) on P_i in G^i. But, P_i is a shortest s, t path in G^i, and so $D^i(y) = D^i(x) + 1$. (If $D^i(y) \leq D^i(x)$, there is an s, t path of fewer edges than P_i, by following P_i to y and then going from y to t with $D^i(y)$ edges.) Clearly, the addition of (x, y) does not decrease the distance from y. Further, since $D^i(y) > D^i(x)$, any path from any node z to t using edge (x, y) will have distance greater than $(D^i(z, x) + D^i(x)) \geq D^i(z)$, where $D^i(z, x)$ is the shortest distance from z to x in the ith augmentation graph. Therefore, the addition of (x, y) cannot decrease the distance to t from any node, and so $D^i(v) \leq D^{i+1}(v)$, for all nodes v, and in particular $d_i \leq d_{i+1}$. Further, $d_r \leq n$, since no simple path can be longer than n, the number of nodes in the graph. □

Given Lemma 3.1, we can partition the execution of the EK algorithm into *phases*, where in each phase, all the augmentation paths used by the algorithm have the same number of edges, and all the augmentations of that length are in that single phase. More formally:

Definition. A *phase* of the EK algorithm is a maximal portion of the execution of the algorithm where all the augmentation paths have equal length. If G^i is the first residual graph and G^k is the last residual graph in a phase, then $d_{i-1} < d_i = d_k < d_{k+1}$.

It follows immediately from Lemma 3.1 that in the EK algorithm there are at most n phases.

The idea of the Dinits algorithm is to efficiently find the augmentation paths inside a single phase. We will argue that inside a phase we can streamline the way each successive augmentation graph is constructed from its predecessor. In particular, we will see that inside a phase we can completely ignore any edges whose residual capacities are increased by the EK algorithm, including all the new residual edges that the EK algorithm adds. This streamlining may at first seem only a cosmetic improvement, but, in fact, it holds the key to a significant speed up.

Lemma 3.2. *For any node v, let $P(v)$ be any shortest v, t path in G^{i+1}. If $P(v)$ contains at least one edge of $G^{i+1} - G^i$, then $P(v)$ has at least $D^i(v) + 1$ edges.*

Proof. Let (x, y) be the closest edge to t on $P(v)$ such that $(x, y) \in G^{i+1} - G^i$. By Fact 2, (y, x) must have been on P_i. P_i is a shortest s, t path in G^i, so $D^i(y) = D^i(x) + 1$, and it follows that $D^{i+1}(x) = 1 + D^{i+1}(y) \geq 1 + D^i(y)$ since all the edges from y to t on $P(v)$ are in G^i. But $1 + D^i(y) = D^i(x) + 2$, so $D^{i+1}(x) > D^i(x)$. Now let (x', y') be the next closest edge to t on $P(v)$ such that (x', y') is in $G^{i+1} - G^i$. Again (y', x') must have been on P_i, and $D^{i+1}(x') = 1 + D^{i+1}(y') > 1 + D^i(y')$. The inequality follows from the fact that all the edges on $P(v)$ from y' to x are also in G^i, and that $D^{i+1}(x) > D^i(x)$. Iterating this argument along edges on P_i that are in $G^{i+1} - G^i$, we obtain the lemma. □

The following corollary is immediate.

Corollary 3.1. *If for some $i < j$, $P(v)$ contains at least one edge of $G^j - G^i$, then $P(v)$ has at least $D^i(v) + 1$ edges.*

A short digression

Digressing briefly from the exposition of the Dinits algorithm, we can now bound the running time for the EK algorithm.

Corollary 3.2. *The Edmonds–Karp algorithm runs in $O(n^5)$ time.*

Proof. In any augmentation path P_i, at least one edge (x, y) becomes saturated and hence does not appear in any successive residual graphs until (y, x) is used on an augmentation path P_j for some $j > i$. But, by Corollary 3.1, $D^j(y) < D^i(y)$. Hence the reappearance of edge (x, y) in G^{j+1} is associated with an increase in $D(y)$. As $D(y)$ is bounded by n, edge (x, y) can be saturated at most $n + 1$ times, and this holds for each edge into y. Further, there are at most n edges into y, so the total number of times that the edges into y can be saturated by an augmentation in algorithm EK is $O(n^2)$. Therefore, the total number of augmentations of the EK algorithm is $O(n^3)$, and since each augmentation takes $O(e)$ time, the total time for the algorithm is $O(n^3 e) = O(n^5)$. Note that this bound can also be shown to be $O(ne^2)$. □

Back to the Dinits method

Dinits improved upon the EK algorithm, obtaining a running time of $O(n^4)$, by more fully exploiting Lemma 3.2. In particular, by setting v to s in Lemma 3.2 we get:

Corollary 3.3. *Any s, t augmentation path in G^{i+1} which contains an edge in $G^{i+1} - G^i$ has at least $d_i + 1$ edges.*

The importance of this corollary is that if a directed edge (x, y) in G^i is not on any shortest s, t path at the start of a particular phase in the EK algorithm, it will not be on any shortest s, t path during *any* part of the phase. So suppose that a new phase of the EK algorithm has just begun, i.e., the previous augmentation path had been some length d, but there are no length-d augmentation paths in the current residual graph. Let G^i be the residual graph at the start of the phase. Then Corollary 3.3 says that we can execute the entire phase using residual graph G^i, without ever adding new edges to G^i. In fact, Corollary 3.3 says that we might as well remove from G^i any edge that is not on some shortest s, t path in G^i – we can execute the entire phase on this reduced graph without affecting the correctness of the EK algorithm! We now make this idea precise.

Definition. The *layered* graph LG^i for G^i is the graph obtained from G^i by removing all edges which are not on some shortest s, t path in G^i.

Note that in a layered graph, all s, t paths have the same length, so every s, t path is a shortest s, t path. It is easy to find the layered graph LG^i for G^i in $O(e)$ time by *breadth-first search* (BFS). Breadth-first search will give each node v a number $l(v)$ which is the minimum number of edges from s to v along any s, v path in G^i. Then LG^i consists of all edges (u, v) in G^i such that $l(u) = l(v) - 1$, and $l(v) < l(t)$. Once the node labels have been assigned, the proper edges can be selected by scanning through them in $O(e)$ time. Below we give the BFS algorithm for assigning node labels.

Breadth-first search

Let Q be a *queue*, i.e., a list in which new elements are added at the end, and elements are removed from the top. A queue is also known as a *first-in first-out* list.

(1) Set $l(s) = 0$; mark s and add it to Q.
(2) While Q is not empty, execute Steps 3 and 4.
(3) Remove the top node w from Q.
(4) For each unmarked node u connected from w by a directed edge in G^i, mark u, add it to Q, and set $l(u) = l(w) + 1$.

It is easy to prove by induction, that the assigned node labels are correct.

Using the definition of a layered graph, we can summarize the observations so far: if G^i is the augmentation graph at the start of a phase, then the entire phase of the EK algorithm can be executed on the layered graph LG^i in place of G^i. LG^i can be found by breadth-first search in $O(e)$ time.

The algorithmic importance of this may not be at first apparent. The EK algorithm spends $O(e)$ time to build each augmentation graph, and $O(e)$ time to find a shortest s, t path in it. The above observations show that we only need to build a new (layered) augmentation graph at the start of each phase, and this reduces the time involved in building augmentation graphs. But does it lead to an overall speedup in the EK algorithm? If the costs of finding augmentation paths are not reduced, then the answer is 'no' since each path continues to cost $O(e)$. The importance of layered graphs is that they do in fact allow augmentation paths to be found faster, as follows.

Since any s, t path in LG^i is a shortest s, t path, and it can have at most $n - 1$ edges, a shortest s, t augmentation path in LG^i can be found myopically in $O(n)$ time: just follow any sequence of edges from s to t in LG^i. So to implement a phase we do the following:

The Dinits algorithm for a single phase
Repeat

Myopically follow any path P from s, keeping track of the minimum residual edge capacity δ along that path. If the path reaches t, then execute Step A. Else execute Step B.

Until all edges have been removed from LG^i.

Step A (when t is reached): augment the flow f in G along the edges of P by

δ units, and reduce the capacity of these edges in LG^i by δ units. The capacity of at least one edge in LG^i on P will become zero. Remove any such edges from LG^i.

Step B (when the path in LG^i from s reaches a node v which has no edges out of it): remove all edges into v from LG^i.

Lemma 3.3. *The Dinits algorithm for a single phase correctly implements a phase of the EK algorithm, and it runs in time* $O(ne)$.

Proof. First note that if edge (x, y) is in LG^i, then edge (y, x) cannot be in LG^i. This means that during a phase of the EK algorithm executed on LG^i, the capacities of the edges of LG^i never increase. Therefore, once the capacity of an edge becomes zero, the edge can be removed. The correctness of the algorithm then follows from Lemma 3.3.

For the time bound, note that each myopic search for a path takes $O(n)$ time, since the length of any path is at most n. Each such myopic search ends with the removal of at least one edge of LG^i, and hence there are at most e such searches. □

The Dinits *network* flow algorithm is the EK algorithm where each phase is implemented as above. Since in each phase the length of the s, t path increases, there can be at most n phases. Hence, we have:

Theorem 3.1. *The Dinits network flow algorithm runs in time* $O(n^2e) = O(n^4)$.

To review, the speed-up of the Dinits algorithm over the EK algorithm comes first from understanding how successive augmentation paths are related, leading to the notion of a phase, and second from the ability to implement an entire phase on a single layered graph, leading to an $O(n)$ method to find each augmenting path. The EK algorithm needs $\Theta(e)$ time to find each augmentation path because it searches in an arbitrary residual graph, while the Dinits algorithm restricts its search to a layered graph. This illustrates the main theme of this chapter, how mathematical insight and algorithm analysis (in this case of an existing algorithm) leads to the design of a more efficient algorithm.

3.2. An $O(n^3)$ *network flow algorithm*

We now show how to reduce the time for computing a maximum flow from $O(n^2e)$ to $O(n^3)$ by reducing the time for a single phase computation from $O(ne)$ to $O(n^2)$. The idea will be to look further at the set of augmentation paths that *could have* been found in a phase of the EK or the Dinits algorithm. That is, instead of trying to find the paths one at a time during a phase, as the EK and Dinits methods both do, we try to find a subset of edges, and flows on those edges, which could have been obtained from the superposition of augmentation paths found in a phase. It turns out that we can find such a set faster than by actually finding individual augmentation paths, and since this set

of edges could have come from the Dinits or EK algorithm, we can correctly use them in the network flow algorithm.

Before we delve into the details, we have to slightly switch our view of what is being computed on a residual graph.

Definition. For a layered graph LG^i used during phase i of the EK or Dinits algorithms, define $g(u, v)$ to be $c(u, v) - \bar{c}(u, v)$, where $c(u, v)$ is the capacity in LG^i of edge (u, v) at the start of phase i, and $\bar{c}(u, v)$ is its capacity at the end of the phase.

In the Dinits method the flow f is modified in Step A immediately after each augmentation path is found. These modifications are related to, but distinct from, the ongoing modifications of LG^i. Suppose we only make the modifications on LG^i, and delay changing the flow f until the end of the phase. What we would know at the end of the phase is the flow f which is correct for the start of the phase, and the function g. From that we could easily obtain the flow f', the correct flow for the end of the phase, by *superimposing* f and g:

If (u, v) is a forward edge in LG^i, then set $f'(u, v)$ to

$f(u, v) + g(u, v)$.

If (u, v) is a backward edge in LG^i, then set $f'(u, v)$ to

$f(u, v) - g(u, v)$.

So instead of finding augmenting paths one at a time, if we had some other method for determining the function g we could simulate a phase of the Dinits algorithm (which itself simulates a phase of the EK algorithm). So is there any easy way to find the function g? To answer that, we look a little more closely at what g is.

It is easy to verify that the function g is an s, t *flow* in LG^i. However, it is not necessarily a maximum flow, since for any edge (u, v) in LG^i, $g(u, v)$ starts at zero and only increase during a phase. That is, if we consider a phase of the Dinits or EK method to be computing an s, t flow g in LG^i, then that computation never decreases the flow in any edge of LG^i, and by simple example, such an algorithm cannot be guaranteed to find a maximum flow. Instead, g is something called a *blocking flow*.

Definition. A blocking flow in a graph is a flow in which at least one edge on every s, t path is saturated. It is easy to construct examples illustrating that a blocking flow is not necessarily a maximum flow.

Clearly, g is a blocking flow in LG^i, for if it was not blocking, then additional s, t paths in LG^i could be found, and the phase would not be complete. We would like to say that conversely any blocking flow in LG^i can be used to simulate a phase of the Dinits method. Unfortunately this is not quite correct

unless we modify Step A of the Dinits method as follows: after finding a path P with minimum capacity δ', set δ to be any positive value less or equal to δ', and use δ as before. It is easy to see that even with this change the maximum flow is still correctly computed, although one might suspect that if we were to modify the Dinits method in this way, the algorithm would become very inefficient. We will see that this need not be the case.

Lemma 3.4. *Let f and LG^i be as above. If g is any blocking flow in LG^i, then the superimposition of f and g gives a flow f' which could have been obtained by the execution of a phase of the (modified) Dinits algorithm on LG^i.*

Proof. Given g we will find s, t paths which could have been found by the modified Dinits method. First remove any edge (u, v) where $g(u, v) = 0$. There are s, t paths in LG^i, and g is a blocking flow, so there must be at least one edge (s, u) such that $g(s, u) > 0$. Then, since g is a flow, there must be an edge (u, v) where $g(u, v) > 0$. LG^i is acyclic and all paths end at t, so repeating this reasoning, we find an s, t path P in LG^i among edges with positive flow g. Let ε be the smallest g value among the edges on P. Now decrease $g(u, v)$ by ε for every edge on P. What remains is again an s, t flow, so we can again find an s, t path. Repeating this operation, we can decompose g into a set of s, t paths, and for each such path $\varepsilon > 0$. Clearly, these paths could have been found (in any order) by an execution of the modified Dinits method, provided that it chose δ to be ε. Finally, since g is a blocking flow, such an execution of the Dinits algorithm would terminate after finding these flows. □

Hence any phase of the Dinits algorithm can be simulated if we have a blocking flow for the layered graph of that phase. Then, one way to speed up the Dinits algorithm is to find a blocking flow in a layered graph in time faster than $O(ne)$. The first solution [in time $O(n^2 + e)$] to this was proposed by Karzanoff [1974]. A simpler method was later discussed by Malhotra, Kumar & Maheshwari [1978], and many additional methods have since been found. In this section, we follow the method called the *wave* method due to Tarjan [1983], which is itself a simplification of the Karzanoff method.

3.3. The wave algorithm

All of the above methods for finding a blocking flow in a layered graph are *preflow* methods. A preflow is a relaxation of a flow; it is an assignment of non-negative real numbers to edges of the graph satisfying the capacity constraints, but the original conservation constraint is replaced by the following relaxed constraint:

For every node i other than s or t,

$$\sum_{j:\,(j,i)\in E} f(j, i) \geq \sum_{j:\,(i,j)\in E} f(i, j)\,.$$

In other words, the flow into i is greater or equal to the flow out of i.

We define $e(v)$ (*excess* at v) to be the total flow into v minus the total flow out of v. In a preflow f, node v ($v \neq s, t$) is called *unbalanced* if the excess is positive, and is called *balanced* if the excess is zero. The residual graph G^f for a preflow f is defined exactly as a flow f. That is, if $f(u, v) > 0$, the directed edge (v, u) is in the residual graph with capacity $f(u, v)$; if $f(u, v) < c(u, v)$, the directed edge (u, v) is in the residual graph with capacity $c(u, v) - f(u, v)$.

The wave method uses preflows until it ends, at which time each node is balanced, and hence the ending preflow is a flow; we will see that it is in fact a blocking flow. During the algorithm, each node will be called either *blocked* or *unblocked*. Initially s is blocked, but all other nodes are unblocked; when a node becomes blocked, it never again becomes unblocked.

The algorithm tries to balance an unbalanced *unblocked* node v by *increasing* the flow out of v, and to balance an unbalanced *blocked* node v by *decreasing* the flow into v. The algorithm operates by repeatedly alternating a wave of increase steps (where flow is pushed towards t) and a wave of decrease steps (where flow is pushed back towards s). We will describe the increase and decrease steps in detail below, but before we do, we give the high-level description of the algorithm.

The wave algorithm for a blocking flow g in a layered graph

(0) Set $g(u, v) = 0$ for every edge (u, v) in the layered graph LG^i.

(1) Saturate all edges out of s.

(2) Find a topological ordering of the nodes of the layered graph. That is, find an ordering of the nodes such that for any directed edge (u, v), u appears before v in the ordering.

(3) Repeat Steps 4 through 7 until stopped.

(4) Examine each node v in the established topological order, and execute Step 4a.

(4a) If v is unblocked and unbalanced, then attempt to balance it by executing the *Increase Step* for v. If v cannot be so balanced, then declare it *blocked*.

(5) If there is any unbalanced blocked node, then go to Step 6. If there is not such a node, then stop the algorithm.

(6) Examine each node v in reverse topological order and execute Step 6a.

(6a) If v is blocked and unbalanced, then balance it by executing the *Decrease Step* for v. Note that v will always become balanced in this step.

(7) If there is an unblocked, unbalanced node, then go to Step 4, else stop the algorithm.

We can now describe in detail the increase and decrease steps executed while examining a node v.

Increase Step. Repeat the following operation for each neighboring node w of v until either v is balanced, or no further neighboring nodes of v exist.

If w is an unblocked node, and $g(v, w) < c(v, w)$, then increment $g(v, w)$ by $\min[e(v), c(v, w) - g(v, w)]$.

Decrease Step. Repeat the following operation for each neighboring node u until v is balanced:
If (u, v) is an edge with flow $g(u, v) > 0$, then decrease $g(u, v)$ by $\min[g(u, v), e(v)]$.

We prove the correctness and the time bound for the wave method with the following lemmas. Note first that once a node becomes blocked in the algorithm, it never becomes unblocked.

Lemma 3.5. *If node v is blocked, then there is a saturated edge on every path from v to t.*

Proof. The proof is by induction on the order that the nodes become blocked. Node s is initially blocked, and all edges out of s are saturated. So the lemma holds at this point. Suppose it holds after the kth node becomes blocked. Now before the $k + 1$ node becomes blocked some flow could be decreased from a node v to a blocked node w, but this happens only in Step 6a, and only if v is blocked. Hence by the induction assumption, all paths from w contain a saturated edge. The $k + 1$ node v becomes blocked only in Step 4a, and only after all the edges out of v are saturated, hence the inductive step holds. □

Lemma 3.6. *If the method halts, all nodes are balanced and the preflow is a blocking flow.*

Proof. Note that after Step 4, there are no unbalanced unblocked nodes, so if the algorithm terminates in Step 5, then all nodes are balanced. Similarly, after Step 6 there are no unbalanced blocked nodes, so if the algorithm halts in Step 7, all nodes are balanced. Hence if the algorithm terminates, then the preflow is a flow. To see that the flow is blocking, note that for every edge (s, v) out of s, either (s, v) is saturated or v is blocked, since v had to be blocked before flow on (s, v) could be decreased. But if v is blocked, then by Lemma 3.5 all v to t paths are blocked. □

Theorem 3.2. *The wave algorithm computes a blocking flow in a layered graph in $O(n^2)$ time.*

Proof. Nodes only become blocked in Step 4a, and once blocked remain blocked forever. Further, when a blocked node becomes unbalanced in Step 4a, it is immediately balanced in the next execution of Step 6a, and no new unbalanced nodes are created in Step 6. So the algorithm terminates unless at least one new node becomes blocked in each execution of Step 4; hence, there are at most $n - 1$ executions of Step 4 and at most $n - 2$ of Step 6. In each such

step, we attempt to balance $n-2$ nodes, so there are $O(n^2)$ times when the algorithm attempts to balance nodes. Each such attempt (at a node v, say) either succeeds or results in saturating all edges out of v.

To bound the running time of the algorithm it is not enough to bound [by $O(n^2)$] the number of attempts to balance the nodes, since in each attempt to balance a node v, several edges out of v are examined. We now examine how many edges are examined by the algorithm. The flow on an edge (v, w) is increased only if w is unblocked, and is decreased only if w is blocked, hence the flow on (v, w) increases for some time, then decreases, but never again increases. During the increase part, each flow increase either balances v or saturates (v, w). Any edge can be saturated only once, so over the entire algorithm there can be at most $O(n^2 + e)$ edges examined during the Increase Step. Similarly, during the decrease part, each flow decrease on (v, w) either reduces its flow to zero (which can happen only once) or ends an attempt to balance w. Hence there can be at most $O(n^2 + e) = O(n^2)$ edges examined during the Decrease Step. The number of edges examined dominates the running time of the method, so the theorem follows. □

3.4. Section summary

To summarize this section, the maximum s, t flow and minimum cut can be found in $O(n^3)$ time by the Dinits network flow method if the wave algorithm is used to execute each phase. The Dinits method executes at most n phases, where in a phase a blocking flow in a layered graph must be found. The wave algorithm finds a blocking flow in a layered graph in $O(n^2)$ time, so a total time bound of $O(n^3)$ is achieved.

4. The breakdown of phases: Goldberg's preflow-push algorithm

The idea of a *phase* was central to the speedup of the Dinits and wave methods over the Ford–Fulkerson and EK methods. The basic observation was that when shortest augmenting paths are used, the computation naturally partitions into at most n phases, and in each phase a blocking flow in a layered graph is computed; the speedups then resulted from implementing a phase more efficiently. The overall time bound was just the *product* of the bound on the number of phases and an *independent* upper bound on the worst-case running time of phase. But maybe there is some important *interaction* between phases. Perhaps the number of phases affects the total time taken by the phases, or perhaps when one phase takes a long time, the next phases take less than the worst-case upper bound on an arbitrary phase. As long as the worst-case time bounds are obtained by multiplying a bound on the number of phases by an independent bound on the worst-case running time of a phase, no analysis of such interaction is possible.

Goldberg [1987] introduced a network flow algorithm that had the same

dense worst-case running time, $O(n^3)$, as the best previous algorithms, but whose analysis did not divide the algorithm into phases. Hence the time bound for the algorithm is not just a bound on the number of phases times a bound on the time per phase. This sort of analysis, where the bound comes from analyzing an *entire* sequence of operations is often called an *amortized* analysis. Hence the analysis of the Goldberg algorithm differs from that of the Dinits algorithm in that the former analysis is more amortized than the latter. We will see other amortized analyses below when we consider parametric flow, and in a later section when we discuss the problem of computing the connectivity of a graph.

Goldberg's method was modified by Goldberg & Tarjan [1988] and is now generally referred to as the Goldberg–Tarjan (GT) method. The amortized analysis of the GT method allows additional advances, one of which, parametric flow, will be discussed in detail in the next section. Another advance, by Goldberg & Tarjan [1988], was the improvement of the running time to $O(ne \log(n^2/e))$, which is $O(n^3)$ for dense graphs, and $O(ne \log n)$ for sparse graphs. This second advance relies heavily on the use of a data structure called a *dynamic tree* and will not be discussed in this article.

4.1. *The generic algorithm*

The Goldberg method is a preflow method, maintaining a preflow until the end of the algorithm when it becomes a (maximum) flow. During the algorithm each vertex v has an associated label $d(v)$ which is always between 0 and $2n-1$. These d labels are called *valid* for the current preflow f if $d(s) = n$, $d(t) = 0$, and $d(u) \leq d(v) + 1$ for any edge (u, v) in the current residual graph for f. Throughout the algorithm the d labels never decrease and are always kept valid (a fact that we will prove later). A directed edge (u, v) is called *admissible* if and only if it is in the current residual graph and $d(u) = d(v) + 1$.

The basic step of the Goldberg algorithm is called a *node examination* of an active node. In a node examination of active node u, all the excess at u is pushed along *admissible* edges out of u to neighbors of u in the current residual graph. If at any point in the examination, the active node u has no more admissible edges out of it, then $d(u)$ is changed to $\min[d(v) + 1: (u, v)$ is an edge in the current residual graph]. It is easy to prove, by induction on the number of pushes say, that any active node has a residual edge out of it. So the relabeling is always possible and creates an admissible edge from u to v allowing additional flow from u to be pushed to v. Hence, the node examination of u ends only when all excess at u has been pushed out of u, at which point u is no longer active. After a push, of amount δ say, has been made along edge (u, v), we change the preflow f along the edge as follows: if (u, v) is a forward edge (i.e., an original edge in G), then $f(u, v)$ gets increased by δ. If (u, v) is a backward edge, then $f(v, u)$ gets decreased by δ. These changes of f are considered part of the node examination of u.

As an implementation detail, any push along an edge is required to push as

much as possible, i.e., an amount equal to the minimum of the node's excess and the capacity of the edge. A push is called *saturating* if the push fills the residual edge (either forward or backward) to capacity, and *non-saturating* otherwise.

We can now describe the generic Goldberg algorithm.

The generic Goldberg algorithm

Push flow out of s to neighbors of s so that all edges out of s are saturated. {This makes all neighbors of s active.}

Set $d(s) = n$, $d(t) = 0$, and $d(u)$ equal to the number of edges on the shortest path from u to t in the graph.

While there are any active nodes other than s or t

begin

Pick an active node u other than s or t and perform a node examination of u.

end.

Note that the notion of the residual graph is important in the algorithm even though it is not explicitly in the algorithm description. It is important because the definition of an admissible edge ultimately depends on the current residual graph. Hence the residual graph must be (explicitly or implicitly) updated as the computation proceeds (after each node examination).

It is easy to see that the algorithm always maintains a preflow. Clearly then, if the algorithm terminates, the preflow is a flow, because when a node has no excess the flow into it equals the flow out of it. We will show that the algorithm does terminate after some additional implementation detail is presented. But with the help of the following lemma, we can already prove that at termination the flow is a maximum flow.

Lemma 4.1. *Throughout the algorithm, the d node labels are valid.*

Proof. It is immediate that the initial d values are valid. Now consider the first point in time where an invalid labeling occurs, and suppose it is $d(u) > d(v) + 1$ for a residual edge (u, v). What could have happened between the time all labels were valid, and the point of first invalidity? There are three things that could have happened: either (u, v) was already a residual edge and $d(u)$ changed or $d(v)$ changed, or (u, v) was not a residual edge but residual edge (u, v) got created the point of invalidity. Whenever the value $d(u)$ is changed it is set to $\min[d(v) + 1: (u, v)$ is an edge in the current residual graph], hence this does not create an invalid node label. If $d(v)$ is changed it must increase (since node labels never decrease); but just before that point $d(v) \leq d(v) + 1$, since (u, v) was a residual edge and the node labels are valid. When a new edge (u, v) is added to the residual graph it is because of a push from v to u in the residual graph. But this means that the edge (v, u) was admissible, so $d(v) = d(u) + 1$. Certainly then $d(u) \leq d(v) + 1$, the requirement for validity on

the new edge (u, v) entering the residual graph. Edges leaving the residual graph have no affect on validity. □

Theorem 4.1. *Assuming the algorithm terminates, the preflow at termination is a maximum flow.*

Proof. We have already noted that at termination the preflow is a flow, hence the residual graph is then a residual graph for an *s, t flow*. The flow will be maximal if and only if there is no s to t path in that residual graph. Since the node labels are always valid and $d(s) = n$ and $d(t) = 0$ throughout the algorithm, any path from s to t in the residual graph would have to have n edges, hence $n + 1$ nodes, which is not possible. Hence there is no s to t path in the residual graph at termination (or ever), and so the preflow is a maximum flow. □

4.2. Additional implementation details

Before we can completely prove termination and worst-case time bounds, there are two important implementation details to discuss: how admissible edges are searched for during a node examination, and which active node to examine if there are choices. We first address the admissible edge question.

How to search for an admissible edge

For each node v, the algorithm keeps a list $I(v)$ in arbitrary but fixed order, containing every node w such that either edge (v, w) or (w, v) is in G. Hence $I(v)$ represents all the edges (v, w) which could possibly be admissible. At any point during the algorithm there is a pointer $p(v)$ into $I(v)$. At the start of the algorithm each $p(v)$ points to the top of $I(v)$. When node v is examined the algorithm finds admissible edges out of v by searching through $I(v)$ in order, starting at $p(v)$, advancing $p(v)$ each time a new node of $I(v)$ is considered. Further, the algorithm will only consider updating $d(v)$ when it has passed the bottom of $I(v)$. The algorithm remains correct, because in the generic algorithm $d(v)$ is changed when there are no admissible edges out of v, and although that might happen before $p(v)$ is at the bottom of $I(v)$, it certainly cannot hurt to explicitly check all the remaining potential residual edges. So the algorithm will only consider changing $d(v)$ when $p(v)$ passes the bottom of $I(v)$. This detail by itself does not imply that $d(v)$ will definitely change at that point, however, we will prove that implication. That is, we will show that if the bottom node of $I(v)$ is passed, then there are no admissible edges out of v. At that point then, $d(v)$ is changed to $\min[d(w) + 1: (v, w)$ is an edge in the current residual graph], $p(v)$ is set to the top of $I(v)$, and the examination of v continues.

Since the change of $d(v)$ creates a new admissible edge out of v, this cycling scan through $I(v)$ always results in all excess being pushed out of v during an examination of v. Note that at most one non-saturating push (the last one, if any) from v occurs during a single examination of v.

Lemma 4.2. *When the bottom of $I(v)$ is passed (but before $d(v)$ is changed), there are no admissible edges out of node v in the current residual graph.*

Proof. For any node w on $I(v)$, $p(v)$ passes w during a node examination of v only when edge (v, w) is not admissible. Since that time, the algorithm might have examined nodes other than v, and so we have to see what effect this might have on the admissibility of (v, w).

At the moment that $p(v)$ passes w edge (v, w) is not admissible, so either edge (v, w) was not in the residual graph, or $d(v)$ was strictly less than $d(w) + 1$. In the latter case $d(v)$ is still strictly less than $d(w) + 1$ when $p(v)$ passes the bottom of $I(v)$, because $d(v)$ has not changed and $d(w)$, if changed, has only increased. In the former case, it may be that edge (v, w) is in the current residual graph, although it was not in the residual graph when $p(v)$ passed w. This can only happen if there was a push from w to v in the meantime; at that time $d(w)$ equaled $d(v) + 1$. But this again implies that $d(v) < d(w) + 1$ when $p(v)$ passes the bottom, so (v, w) is still inadmissible. Since w was arbitrary, we have proved that there are no admissible edges out of v when $p(v)$ passes the bottom of $I(v)$. □

Note we have actually shown something a little stronger which will be needed later. We have shown that if w is above $p(v)$ in $I(v)$, then edge (v, w) is inadmissible.

We can now complete a little of the timing analysis, and hence also a little of the termination argument.

Lemma 4.3. *For any node v, $d(v)$ is always less than $2n$.*

Proof. First, only node labels of active nodes are changed, so once a node becomes permanently inactive its node label is fixed. So we only need to show what happens to active nodes.

Next, we claim that for any active node v there is a directed path in the current residual graph from v to s. For suppose not, and let W be the set of nodes which are reachable from v along any directed path from v in the current residual graph. Let $\bar{W} = V - W$. By assumption $s \in \bar{W}$. By the maximality of W, all $f(x, y) = 0$ for all $x \in W$, $y \in \bar{W}$, otherwise there would be a residual edge from W to $\bar{W}$. But if there is no flow (or preflow) into W from $\bar{W}$, there certainly cannot be any excess at any node in W. So there is a directed path from v to s in the current residual graph. Let w be the node adjacent to s on this path. Because $d(s) = n$, and node labels are always valid, it follows that $d(w) \leq n + 1$. Repeating this argument along the path to v, and using the fact that there are only n nodes, we see that $d(v) < 2n$. □

Lemma 4.4. *The generic algorithm does at most $O(n^2)$ node relabel operations, and at most $O(ne)$ saturating pushes.*

Proof. Since $d(v) < 2n$ for each node v, and each relabel of v increases $d(v)$,

the total number of relabels is bounded by $2n^2$. Now $d(v)$ increases each time the bottom of $I(v)$ is passed, since there are no admissible edges out of v at that point, so the bottom of v can be passed at most $2n$ times. Each saturating push along an edge out of v advances $p(v)$ by one position, so the total number of saturating pushes out of v is $2n$ times $|I(v)|$ (which is the number of neighbors of v in G). Summing this over all nodes bounds the total number of saturating pushes by $2ne$. □

To complete the time analysis of the algorithm we essentially need only consider the number of non-saturating edge pushes. This is most easily done by adding computational implementation detail given below.

4.3. How to chose among active nodes

There are two well-studied specialized versions of the generic algorithm. In the FIFO version, nodes are placed on the end of a queue as they become active, and are picked for examination off the top of the queue. In the max-d version, the active node picked for examination is always the one with the largest d label. Both methods lead to an $O(n^3)$ time algorithm, but the max-d method is easier to analyze (following an argument given by Cheriyan & Maheshwari [1989]), and has additional applications we will discuss later.

Note that Lemmas 4.3 and 4.4 remain valid for the max-d version of the algorithm since they did not rely in any way on how active nodes were chosen. The next lemma does rely on the max-d version of the algorithm, and nearly completes the remaining time analysis.

Lemma 4.5. *The* max-d *algorithm performs only* $O(n^3)$ *non-saturating pushes.*

Proof. In the max-d algorithm, at most n consecutive node examinations can occur without at least one node label increasing. To see this, note that excess is always pushed from the highest labeled active node to a lower labeled node (since it is pushed along admissible edges). So, since each node pushes out all its excess when examined, if no node labels change during n node examinations, then all excess in the network will either be pushed forward to t or backward to s. At that point the algorithm terminates with a maximum flow, since there will be no active nodes. Each node examination can do at most one non-saturating push, so there can be at most n non-saturating pushes between node label increases. Each node label is bounded by $2n$, and node labels never decrease, so there can be at most $O(n^3)$ non-saturating pushes. □

Theorem 4.2. *The* max-d *version of the Goldberg algorithm finds a maximum flow in* $O(n^3)$ *worst case time.*

Proof. The time analysis is divided between the time for all the non-saturating pushes, and the time for all-other-work. Lemma 4.5 showed that the total

number of non-saturating pushes is bounded by $O(n^3)$, and each can clearly be done in $O(1)$ time.

Ignoring for now the time to find the max-d active node, the time for 'all-other-work' is just $O(ne)$ as follows. Each operation other than a non-saturating push or a relabeling causes $p(v)$ to advance for some v, and $d(v)$ advances every time $p(v)$ passes the bottom of $I(v)$. Since $d(v) < 2n$, and $|I(v)|$ is the sum of the in and out degrees of v in G, the total advancement of all n p-pointers is bounded by $2n \Sigma_v |I(v)| = O(ne)$. Relabeling also only costs $O(ne)$ since a node gets relabeled at most $2n$ times, and the time for each relabel is bounded by the sum of its in and out degrees. So the time for all-other-work is $O(ne)$.

Now we discuss how to find an active node of maximum d label. To do this efficiently, the algorithm keeps a set A of $2n - 1$ linked lists of active nodes, each indexed by a number from 1 to $2n - 1$. List j keeps all the active nodes whose d label is j. A is used to locate an active node of maximum d label. If there is more than one, then the node picked for examination is the first one on the list. Each push from a node v must be to a node with d label equal to $d(v) - 1$, so finding the next active node of maximum d value takes constant time. Further, updating A after a node relabeling involves only constant work, so A can be maintained and used in $O(n^2)$ time plus $O(1)$ time per push. □

5. Parametric flow: The value of amortizing across phases

The worst-case (dense) running time of the Goldberg algorithm presented above is no better than that of earlier algorithms. However, the analysis of the Goldberg algorithm is not divided into phases and this amortization across phases can be very useful in analyzing more complex applications of network flow algorithms. In this section we give one example in detail.

5.1. The problem

One of the most useful applications of network flow is in the solution of combinatorial problems by a *sequence* of maximum flow or minimum cut calculations. For examples of such problems, see Picard & Queyranne [1982], Gusfield & Martel [1989], Gusfield & Tardos [1991], Gusfield [1991], Cunningham [1985], and Gallo, Grigoriadis & Tarjan [1989]. In many problems the networks in the sequence are similar and differ only by a systematic change in some of the edge capacities. In particular, the edge capacities are often functions of a single parameter λ, and the particular combinatorial problem is solved by finding the value of λ whose corresponding maximum flow or minimum cut meets some side constraint(s). Hence problems of this type are solved by searching over the possible values of λ (in some efficient manner), solving a maximum flow problem for each fixed value of λ generated.

Of course, for each fixed value of λ in the generated sequence we could solve

the corresponding maximum flow problem from scratch, but the similarity of the problems can often be exploited to solve the entire sequence faster. For a large and important class of such problems, we will see that a sequence of $O(n^2)$ maximum flow problems can be solved with the same worst case time bound as just a single maximum flow problem. The use of the Goldberg (and Goldberg–Tarjan) algorithms in parametric analysis was initiated by Gallo, Grigoriadis & Tarjan [1989] who showed that a sequence of $O(n)$ maximum flow problems can be solved in the worst-case time bound for only a single flow. The result given here is an improvement on that result and is taken from Gusfield [1990b], and Gusfield & Tardos [1991].

Definition. In a *monotone parametric flow network* G, the capacities of the edges out of s are *non-decreasing* functions of the real parameter λ, and the capacities of the edges into t are *non-increasing* functions of λ. All other edge capacities are fixed, as in a normal flow network. For a given value λ^*, we define $G(\lambda^*)$ as the ordinary flow network that results from plugging in λ^* into the capacity functions of the edges out of s and into t. Given a sequence of values $\lambda_1 < \lambda_2 < \cdots < \lambda_k$ (in sorted order), the *parametric flow problem* is to compute the maximum flow and minimum cut in $G(\lambda_1), \ldots, G(\lambda_k)$. We will let f_j denote a maximum flow in $G(\lambda_j)$.

5.2. *The central idea*

In $G(\lambda_{j+1})$ the edge capacities out of s increase and those into t decrease compared to $G(\lambda_j)$, so f_j is a legal preflow in $G(\lambda_{j+1})$, and we can start the computation of f_{j+1} with the initial preflow f_j rather than starting from the zero flow. This 'common-sense' idea has been around (in use with other flow algorithms) for a long time. What is new is that by using this idea together with the max-d version of the Goldberg algorithm, it can be proved that the total work involved in the k network flow computations is at most $O(n^3 + kn)$.

The fundamental result of Gallo, Grigoriadis & Tarjan [1989] is the following:

Theorem 5.1. *In a monotone parametric flow network G, if the values $\lambda_1 < \lambda_2 < \cdots < \lambda_k$ (or $\lambda_1 > \lambda_2 > \cdots > \lambda_k$) are specified in this order, then a maximum flow and a minimum cut in each of the networks $G(\lambda_1), \ldots, G(\lambda_k)$ can be computed on line in $O(n^3 + kn^2)$ total time.*

Thus for $k = O(n)$, all the flows can be done in the same worst-case time as the fastest known algorithm for a single flow. This result has many applications and leads to the fastest solutions of many combinatorial problems. It is important to note that in all of these applications it is the minimum cut that is needed; the maximum flow is computed in order to find the minimum cut. Martel [1989] showed that the $O(n^3 + kn^2)$ bound can also be obtained by

using either the Karzanoff or the wave versions of the Dinits maximum flow algorithm in place of the GT algorithm.

The GGT algorithm [Gallo, Grigoriadis & Tarjan, 1989] (establishing Theorem 5.1) is based on the idea that the maximum flow in $G(\lambda_i)$ can be used to obtain a good initial preflow in $G(\lambda_{i+1})$ and that, if care is taken, the last d labels in $G(\lambda_i)$ remain valid for this initial preflow. In detail, for each i, the initial preflow in $G(\lambda_{i+1})$ is obtained from the maximum flow in $G(\lambda_i)$ by increasing the flow in every edge (s, v) to $c_v(\lambda_{i+1})$ if $d(v) < n - 1$, by reducing the flow in every edge (v, t) to $c_v(\lambda_{i+1})$, and by leaving all other edge flows as they are in the maximum flow in $G(\lambda_i)$. Each $d(v)$ is unchanged from its last value in $G(\lambda_i)$.

It is easy to verify, by the fact that $\lambda_i < \lambda_{i+1}$ and the monotonicity of the capacity functions, that this initial flow assignment is a preflow in $G(\lambda_{i+1})$; it is also easy to verify that the d labels are valid for this preflow. After the initial preflow is set, the maximum flow in $G(\lambda_{i+1})$ is found by resuming the GT flow algorithm and running it to completion.

5.3. *Parametric flow with the* max-d *version*

Theorem 5.2. *In a monotone parametric flow network G, if the values $\lambda_1 < \lambda_2 < \cdots < \lambda_k$ (or $\lambda_1 > \lambda_2 > \cdots > \lambda_k$) are specified in this order, then a maximum flow and a minimum cut in each of the networks $G(\lambda_1), \ldots, G(\lambda_k)$ can be computed on line in $O(n^3 + kn)$ total time.*

Proof. As mentioned above, we use the max-d version of the Goldberg algorithm. However, we note two additional implementation details that are needed for this result. First, when beginning the flow computation in $G(\lambda_{i+1})$, the initial position of each $p(v)$ is its ending position in the flow computation in $G(\lambda_i)$. Second, s must be at the bottom of any $I(v)$ list that it is in, and similarly for t. The first modification is needed for the time analysis below, and the second modification is needed for the correctness of the method. The reason is the following. For the correctness of the Goldberg algorithm, whenever $d(v)$ changes there must be no admissible edges out of v. From the comment after Lemma 4.2 we know that for a single flow computation if w is a node above $p(v)$ in $I(v)$, then (v, w) is not admissible; so when $p(v)$ passes the bottom of $I(v)$ there are no admissible edges out of v. To ensure that after a capacity change, edge (v, w) is still inadmissible if w is above $p(v)$, we always put s and t at the bottom of any list they are in, since an inadmissible edge not incident with s or t is clearly still inadmissible after a capacity change.

For the time analysis, we note how the $O(n^3)$ bound for a single flow computation is affected when k flows are computed by the method described above. Again, the analysis is divided into time for non-saturating pushes, and all-other-work. In the above parametric method, the d labels never decrease, and each is bounded by $2n$ no matter how large k is. Further, $p(v)$ is not moved when the edge capacities change. So the analysis for all-other-work

inside the k flow computations is unchanged, and the time is again $O(ne)$. The time for making the capacity changes is certainly bounded by $O(kn)$ as is the time to set up A after each capacity change. Note that A is empty at the end of each flow computation. So all-other-work inside and between flow computations is bounded by $O(ne + kn)$. Note that we have amortized over the entire sequence of k flows. Had we just taken the $O(n^3)$ bound on the time for any single flow and multiplied by k, our desired bound would be impossible.

To analyze the number of non-saturating pushes, note that *inside* any of the k flows there cannot be more than n non-saturating pushes before a d label increases, for precisely the same reason as in a single flow. However, there may be $n-1$ non-saturating pushes, then a capacity change, and then another $n-1$ non-saturating pushes, all without a label change. So each time the capacities change the bound on the total number of allowed non-saturating pushes increases by n. Hence the total number of non-saturating pushes over k flows is bounded by $O(n^3 + kn)$. Therefore, the time to compute the k flows is $O(n^3 + kn)$. The analysis here is again amortized over all the k flows.

We now discuss how to find the k minimum cuts. Define S_i to be the set of nodes reachable from s in the residual graph obtained from the maximum flow in $G(\lambda_i)$; let $T_i = N \backslash S_i$. It is known [Ford & Fulkerson, 1962] that S_i, T_i is the unique 'leftmost' minimum s, t cut. That is, if S', T' is another minimum s, t cut in $G(\lambda_i)$, then $S_i \subseteq S'$. We will find S_i, T_i in each $G(\lambda_i)$, but we cannot search naively from s since that would take $\Omega(km)$ total time. We note the following two facts. First, $S_i \subseteq S_{i+1}$ for every i [Stone, 1978; Gallo, Grigoriadis & Tarjan, 1989]; second, if w is in $S_{i+1} \backslash S_i$, then w must be reachable in the $G(\lambda_{i+1})$ residual graph from a node $v \in S_{i+1} \backslash S_i$ such that (s, v) is an edge in G and $c_v(\lambda_{i+1}) > c_v(\lambda_i)$ [Gallo, Grigoriadis & Tarjan, 1989]. To search for S_{i+1} we start at from such nodes v, and we delete any edge encountered that is into S_i. Hence the search for S_{i+1} examines some edges not previously examined in any search for S_j, $j < i$, plus at most n edges previously examined. So, the time to find all the k cuts is $O(m + kn)$. □

5.4. *Parametric flow for parameters given out of order*

In Theorem 5.2 the values of λ changed monotonically and there are many applications where this is the case. However, it is even more useful to be able to handle the case when the λ values change in no ordered manner. It was shown by Gusfield & Martel [1989] that a sequence of flows determined by k values of λ given in any order can be computed in $O(n^3 + kn^2)$ time; later this was improved to $O(n^3 + kn)$ by Gusfield [1990b] and Gusfield & Tardos [1991]. We will not give either result here, but examine an important special case where the values of λ are given out of order.

The special case of binary search

We consider the special case where the λ values are generated by some binary search over the space of possible λ values (this is the case in several of the applications). What is special about such a sequence of λ values is that at

any instant there is an interval $[\lambda_l, \lambda_r]$ such that the next λ value is guaranteed to fall into this interval, and such that either λ_l or λ_r will next be set to λ. Hence λ_l monotonically increases, λ_r monotonically decreases, and the intervals are nested over time. The nested interval property will make the algorithm and the analysis particularly simple: we will be able to charge all the flow computations to exactly two monotone sequences of continued flows.

The binary search process iterates the following until the desired value of λ is found or $\lambda_l = \lambda_r$. Assume initially that λ_l and λ_r are known, that the maximum flows $f(\lambda_l)$ and $f(\lambda_r)$ in $G(\lambda_l)$ and $G^R(\lambda_r)$, respectively, are also known.

(1) Given a new value λ^* between λ_l and λ_r, dovetail the following two computations: compute the maximum flow in $G(\lambda^*)$ by continuing the flow computation from $f(\lambda_l)$ (this corresponds to increasing λ to λ^*); compute the maximum flow in $G^R(\lambda^*)$ by continuing the computation from $f(\lambda_r)$ (i.e., decreasing λ to λ^*). Stop as soon as either of these flow computations finishes, and call the resulting flow $f(\lambda^*)$.

(2) Use $f(\lambda^*)$ to determine (in the binary search process) which of λ_l or λ_r should be changed to λ^*, and make the change.

If it is λ_l that changed to λ^*, and the dovetailed flow computation from $f(\lambda_l)$ finished, then set $f(\lambda_l)$ to $f(\lambda^*)$; if it is λ_l that changed but the dovetailed computation from $f(\lambda_l)$ did not finish, then finish it, and set $f(\lambda_l)$ to the resulting flow.

If it is λ_r that changed and the dovetailed flow computation from $f(\lambda_r)$ finished, then set $f(\lambda_r)$ to $f(\lambda^*)$; if it is λ_r that changed but the dovetailed computation from $f(\lambda_r)$ did not finish, then finish it and set $f(\lambda_r)$ to the resulting flow.

Time analysis

Consider an iteration when λ_l is changed to λ^* (in Step 2). The work in that iteration is either involved in finding $f(\lambda^*)$ by continuing the flow from $f(\lambda_l)$ (the value of λ_l is before Step 2), or in finding (or attempting to find) $f(\lambda^*)$ by continuing the flow from $f(\lambda_r)$ (λ_r before Step 2). Because of the dovetailing, the amount of the latter work is dominated by the amount of the former work and so the total work in the iteration is at most twice the amount used to continue the flow from $f(\lambda_l)$. By symmetry, the same conclusion holds when λ_r changes to λ^*.

We will identify two sequences of λ values, one increasing and one decreasing, such that the associated flows for each sequence can be charged as in Theorem 5.2, and such that all other work is dominated by the work done in these two sequences. The first sequence (of increasing λ values) will be called S_l, and the other sequence (of decreasing λ values) will be called S_r. The first values of S_l and S_r are the first λ_l and λ_r values used in the binary search.

In general, whenever λ_l is changed to λ^* (in Step 2), add λ^* to the end of S_l, and whenever λ_r is changed to λ^*, add λ^* to the end of S_r. By construction, if λ_i and λ_j are two consecutive values in S_l, then $f(\lambda_j)$ is computed in the algorithm by continuing the flow from $f(\lambda_i)$. Hence all of the flows associated

with S_l form one (increasing) sequence, and Theorem 5.2 applies. Similarly, all the flows associated with S_r form one (decreasing) sequence and Theorem 5.2 again applies. By the comment in the first paragraph, all other work is dominated by the work involved in computing the flows in these two sequences. Hence we have the following.

Theorem 5.3. *In a binary search that probes k values of* λ, *the total time for all the flows is* $O(n^3 + kn)$. *So if the binary search is over D possible values for* λ, *then the time for the binary search is* $O(n^3 + n \log D)$.

5.5. An application: Network reliability testing

In a communication network $G = (V, E)$, each node v can test $k(v)$ incident lines per day, and each line e must be tested $t(e)$ times. The problem is to find a schedule to minimize the number of days to finish the tests. This problem can be solved as a parametric network flow problem in the following bipartite network GB: there are n nodes (one for each node of V) on one side and m nodes (one for each edge of E) on the other side; there is an edge (v, e) from node v to node e in GB if and only if node $v \in V$ is an endpoint of edge $e \in E$ of graph G. Each edge (e, t) in GB has capacity $t(e)$, and each edge (s, v) capacity $k(v)$. If we multiply the capacity of each (s, v) edge by a parameter λ, then the problem is to find the minimum integer value of λ such that there is a flow saturating the edges into t.

A direct method to solve this problem is to search for the proper λ by binary search. Let $T = \Sigma_e\, t(e)$; then the binary search would solve $O(\log T)$ network flow problems. These flows are in the form assumed for Theorem 5.3, hence the optimal schedule can be found in $O(n^3)$ time for $T = O(2^{n^2})$.

6. Computing edge connectivity: The amortization theme writ small

The edge connectivity of a connected undirected graph is the minimum number of edges needed to be removed in order to disconnect the graph, i.e., after the edges are removed, at least two nodes of the graph have no path between them. The set of disconnecting edges of minimum size is called a *connectivity cut*. The computation to find edge connectivity, and a connectivity cut, is based on network flow but we will see that by exploiting properties of the problem, and by *amortizing* the analysis, we can obtain a much faster algorithm than at first may be suspected.

The most direct way to compute edge connectivity is to put a unit capacity on each edge, and then consider the $\binom{n}{2}$ minimum cut problems obtained by varying the choice of source and sink pair over all possibilities. The smallest of these pairwise cuts is clearly a connectivity cut. With this approach, the edge

connectivity can be computed in $O(n^2F(n))$ time, where $F(n)$ is the time to find a minimum cut between a specified source–sink pair in a graph with n nodes. In a graph where all the edges have capacity of one, it is known [Even & Tarjan, 1975] that the Dinits algorithm will have at most $O(n^{2/3})$ phases, rather than the general bound of n phases. So, for the connectivity problem $F(n) = O(n^{2/3}e)$, and the edge connectivity and a connectivity cut can be found in $O(n^{2+2/3}e)$ time.

The first improvement over this direct method is to note that only $n-1$ flows need to be computed. To achieve the bound of only $n-1$ flows, simply pick a node v arbitrarily and compute a minimum cut between v and each of the other $n-1$ nodes. The minimum cut over these $n-1$ cuts will be a connectivity cut because a connectivity cut must separate at least one node w from v, and clearly no v, w cut can be smaller than the connectivity cut. Hence the minimum cut separating v from w is also a connectivity cut. So a time bound of $O(n^{1+2/3}e)$ is achieved.

6.1. An $n-1$ flow method allowing amortization

There are actually several ways, different than the approach above, to organize the flow computations so that only $n-1$ total flows are necessary. We will follow one such method that will allow us to amortize the time for these $n-1$ flows, achieving a total running time bounded by $O(ne)$. Not only is this a speed up over the above approach based on the Dinits method, but the algorithm will be considerably simpler than the Dinits method.

Let us assume an arbitrary ordering of the nodes $v_1, v_2, \ldots, v_n$, and define V_i to be the set $\{v_1, v_2, \ldots, v_i\}$. The graph G_i is the graph obtained from the original graph G by contracting the set V_i into a single node called v_0. That is, the nodes V_i are removed and a node v_0 is added, and any edge (v_j, v_k) from a node $v_j \notin V_i$ to a node in $v_k \in V_i$ is replaced by an edge from v_j to v_0. Note that G_i could also be defined as the graph obtained from G_{i-1} by contracting v_i into v_0. Let C_i denote a minimum cut between v_{i+1} and v_0 in G_i.

Lemma 6.1. *The smallest of the cuts C_i $(i = 1, \ldots, n-1)$ is a connectivity cut of G.*

Proof. Let C be a connectivity cut in G, and let $k+1$ be the smallest integer such that v_1 and v_{k+1} are on opposite sides of C. Hence all the nodes of V_k are on one side of C and v_{k+1} is on the other side. Note that for any i, any cut in G separating V_i from v_{i+1} is a v_0, v_{i+1} cut in G_i and conversely. So C and C_k must have the same number of edges, and C_k must be a connectivity cut in G. □

Now we consider the total time to compute the $n-1$ C_i cuts. As should be expected, each C_i will be obtained from a maximum v_{i+1}, v_0 flow in G_i. To compute these flows efficiently, we will use a slight variation of the Ford–

Fulkerson algorithm, differing from the original Ford–Fulkerson algorithm in two implementation details.

First, for every iteration $i = 1, \ldots, n-1$, after the maximum v_{i+1}, v_0 flow and minimum cut C_i are found in G_i, we scan the current adjacency list of v_{i+1}, and for any node $v_j \neq v_0$ adjacent to v_{i+1} we mark v_j and the edge v_j, v_{i+1}. Then we contract v_{i+1} into v_0, updating the appropriate adjacency lists, thus creating graph G_{i+1}. Note that at this point the set of nodes which are adjacent to v_0 in G_{i+1} and the edges incident with v_0 are all marked.

Second, when computing a maximum v_{i+1}, v_0 flow in G_i we start by searching for paths of length one or length two in G_i and flow one unit on each such path found. These paths will be called *short* paths. Note that these paths are in G_i and not in a residual graph. The node and edge markings discussed above make this search particularly easy: simply scan the current adjacency list of v_{i+1} for either v_0 or a marked node (note that v_0 can appear more than once because of previous node contractions). Every occurrence of v_0 indicates an edge between v_{i+1} and v_0, and every marked node $w \neq v_0$ indicates a path of length two from v_{i+1} to v_0. In particular, this path consists of the unique edge (v_{i+1}, w) followed by one of the possibly many (due to previous contractions) edges (w, v_0). Note that the node and edge markings allow each node w to be processed in constant time.

The search for short paths ends when all paths of length one or two are blocked. Because each edge saturated at this point is either connected to v_0 or v_{i+1}, there is no need to ever undo flow on these edges, so we can remove from further consideration any edges that are saturated at this point. This will be important for the time analysis below.

To complete the v_{i+1}, v_0 maximum flow in G_i we follow the original Ford–Fulkerson algorithm, i.e., successively building residual graphs (ignoring the saturated edges on short paths found above), finding augmentation paths, and augmenting the flow by one unit for each such path. All paths found during this part of the algorithm will have length three or more, and are called *long* paths.

6.2. *Analysis*

The modified Ford–Fulkerson method clearly finds a maximum v_{i+1}, v_0 flow, and C_i is easily obtained from it, as shown in Theorem 2.2. Then by Lemma 6.1, the connectivity and connectivity cut are correctly obtained from the smallest of these $n-1$ cuts.

Theorem 6.1. *The total time for computing the* $n-1$ C_i *cuts is* $O(ne)$.

Proof. Note first that the time for computing any particular cut C_i cannot be bounded by $O(e)$, but rather the $O(ne)$ bound will be obtained by amortizing

over all the flows. The time needed to find a minimum cut given a maximum flow is clearly just $O(e)$ per flow, as is the time to mark nodes and edges and do the updates to create the next G_i. Hence the total time for these tasks is $O(ne)$.

Now in any iteration i, the short paths in G_i are found in time $O(n)$, since the size of v_{i+1}'s adjacency list is at most size $n-1$, and, by use of the markings, processing of any node on the list takes constant time. Hence over the entire algorithm the time for these tasks is $O(ne)$, although a closer analysis gives an $O(e)$ time bound.

So the key to the time analysis is to show that the searches for long paths (done by using the original Ford–Fulkerson method) take only $O(ne)$ total time over the $n-1$ flows. Each search for a long augmentation path takes $O(e)$ time, and we will show that at most $n-1$ such long searches are done over the entire algorithm.

In iteration i, suppose a long augmentation path begins with the edge (v_{i+1}, w). At the end of iteration i node v_{i+1} is contracted into v_0, so node w will be adjacent to v_0 thereafter. Hence in any iteration $j > i$, if v_{j+1} is adjacent to w in G_j, then there is a path of length two from v_{j+1} to v_0 through w. The edge (v_{j+1}, w) is unique and so will become saturated during the search for paths of lengths one or two. The flow in that edge will not be decreased, so in iteration j, w cannot be the second node on a long augmentation path. It follows that over the entire algorithm a node can be the second node on a long augmentation path at most once. Hence the number of long augmentation paths is at most $n-1$, and the theorem is proved. □

The amortization in the above analysis is on the number of long augmentation paths. In any of the n iterations the number of long augmentations might approach $n-1$, but over the entire algorithm the total number of long augmentations is never more than that number.

6.3. *Historical note*

The history of the above $O(ne)$ connectivity algorithm is similar to the history of the first $O(n^4)$ and $O(n^3)$ network flow algorithms, which were discovered in the Soviet Union in the early 1970s, but were unknown to the western researchers for some time after that. The $O(ne)$ connectivity algorithm discussed in this section is due to Podderyugin [1973] [see Adelson-Velski, Dinits & Karzanov, 1975] who developed it in 1971. However, it was only published in Russian, and, as far as we know, the method was never discussed in the western literature until now (this article). In the meantime, Matula [1987] independently developed an $O(ne)$ connectivity algorithm in 1986 which incorporates many of the same ideas as the above method, but which will in general do fewer than $n-1$ flows. Matula further showed that connectivity can be found in $O(\lambda n^2)$ time, where λ is the connectivity of the graph. Note that since $\lambda \leq e/n$, $\lambda n^2 \leq ne$.

7. Matching: Optimal, greedy and optimal-greedy approaches

A *matching* in a graph is a set of edges such that no two edges have a node in common. A *maximum cardinality matching*, or maximum matching for short, in a matching with the largest number of edges over all matchings. A *perfect* matching is one where every node is incident with an edge in the matching. In many problems solved by matching, each edge of the graph has a *weight*, and the problem is to find a matching maximizing the sum of the edges in the matching. We call this the *weighted matching problem.*

7.1. Cardinality matching

The cardinality matching problem in a bipartite graph $G = (N_1, N_2, E)$ can be solved by general network flow algorithms as follows. Let N_1 and N_2 denote the nodes in the two sides of bipartite graph G. We introduce two new nodes, s and t, connecting every node in N_1 to s and every node in N_2 to t. The resulting graph consists of all the original edges of G plus these new edges. Let H denote the new graph, and give every edge a capacity of one.

Theorem 7.1. *An integral maximum s, t flow in H defines a maximum matching.*

Proof. Note that because all edge capacities are one, and the maximum flow is integral, the flow can be decomposed into node-disjoint s, t paths consisting of three edges. The set of middle edges on these paths clearly defines a matching. Conversely if the matching is not maximum, then there is some edge (v, w) with $v \in N_1$ and $w \in N_2$ which can be added to the matching. But, since all edge capacities are one and the flow is integral, the path s, v, w, t with flow one could then also be added to the flow, contradicting the assumption that it is a maximum flow. Hence the matching defined by the flow is maximum. □

Since network flow on n nodes runs in $O(n^3)$ time, maximum cardinality bipartite matching can be computed in $O(n^3)$ time. However, we will show that the Dinits algorithm only uses $O(\sqrt{n})$ phases in H, rather than the general bound of n phases. Each phase can still be implemented in $O(n^2)$ time, hence bipartite matching runs in $O(n^{2.5})$ worst-case time using Dinits algorithm. We begin with the following lemma.

Lemma 7.1. *Let H be as above, and let F be the maximum flow value in H, and let F' be the maximum flow value after phase $i-1$ of Dinits algorithm. Then $i \leq n/(F - F')$.*

Proof. The total flow F in H can be obtained by superimposing the flow of value F' in H obtained at the end of phase $i-1$ with the maximum s, t flow

obtained in the residual graph G^i defined from F'. Further, we can think of the maximum flow problem in G^i as a new flow problem on a graph with zero flow, and we know that it will have a maximum flow of value $F - F'$. In H every node either has one edge into it from s or one edge out of it to t, and all edges in G have capacity one. Further, the Dinits algorithm always maintains an integral flow, so every edge either has flow one or zero. Now consider a node $v \in N_1$. If there is any flow from s to v at this point, then in G^i there will be no (s, v) edge (since it is saturated) but there will be exactly one edge into v from a node $w \in V_2$ (w is the node that v sent its unit flow to). If there is no flow into v at this point, then there is exactly one edge, namely (s, v) into v. Similarly, for any node w on the t side of H there is exactly one edge out of w. It follows that the maximum integral s, t flow in G^i is partitioned into s, t paths that share no nodes except for s and t. Since there are only n nodes in G^i, the shortest of these paths must be less than or equal to $n/(F - F')$. Hence the length of the layered graph LG^i obtained from G^i is at most $n/(F - F')$. Since the length of the layered graphs grow by at least one in each phase, $i \leq n/(F - F')$. □

Theorem 7.2. *In graph H defined above the Dinits algorithm can only use* $2\sqrt{n}$ *phases.*

Proof. Assume the total flow value F is greater than $\sqrt{n}$, since otherwise the theorem is immediate. Now consider the phase i in which the flow reaches or exceeds $F - \sqrt{n}$. At the start of phase i the flow F' is less than $F - \sqrt{n}$, so $F - F' > \sqrt{n}$. By Lemma 7.1, $i \leq n/(F - F') < \sqrt{n}$. Further, at the end of phase i, the flow is at least $F - \sqrt{n}$ so there can be at most $\sqrt{n}$ additional phases, and the theorem follows. □

This theorem is of importance in its own right, but it also illustrates a common theme in the design of efficient algorithms: seeking out and taking advantage of special structure. There is no fancy name or philosophy suggesting how to identify which structures will be the most useful, but digging for structure and exploiting what is found is certainly one of the most productive techniques in finding efficient algorithms. We will see another example of this later in this section.

We should also mention that a maximum cardinality matching in a general graph can be found in $O(n^{2.5})$ time by a much more complex algorithm.

7.2. Weighted matching and the greedy paradigm

We now look at a simple and very fast heuristic, *the greedy method*, for finding a maximum weight matching in any graph. The heuristic does not always produce an optimal matching, but it is guaranteed to find one with weight at least *one half* that of the optimal.

Greedy matching

Input: A weighted graph $G = (N, E)$; $w(e)$ represents the weight of edge e.

Output: A matching M, with total weight at least half that of the maximum weight matching.

(1) Set M to the empty set, and set A to E.
(2) While A is not empty do
    Begin
    (2.1) Pick the edge e in A with the largest weight $w(e)$ of all edges in A.
    (2.2) Put edge e in set M; delete e and all edges incident with it from A.
    End
(3) Output edge set M.

Theorem 7.3. *Let M' be the optimal matching in G, and let $c(M')$ and $c(M)$ denote the weights of the optimal and the greedy matchings, respectively. Then $c(M)/c(M') \geq \frac{1}{2}$.*

Proof. Initially, every edge of M' is in A. Each time an edge e is chosen for M, the only edges of M' that get deleted from A are those edges incident with an endpoint of e. M' is a matching, and so at most two edges, e_1 and e_2, of M' are deleted from A at each step. Since e is chosen over both e_1 and e_2, $w(e) \geq w(e_1)$ and $w(e) \geq w(e_2)$, so $w(e) \geq \frac{1}{2}(w(e_1) + w(e_2))$. Eventually, A is empty; at that point, every edge e' in M' is associated with an edge e in M, such that $w(e) \geq w(e')$, and at most two edges in M' are associated with the same edge in M. Hence $c(M) \geq \frac{1}{2}c(M')$. □

Corollary 7.1. *M' has cardinality at least one half that of the maximum cardinality matching in G.*

In fact, if edges are picked arbitrarily in Step 2.1, the resulting matching has cardinality at least one half the maximum. This is easy to prove, and is left as an exercise.

It should be clear why this approach to matching is called a 'greedy algorithm'. Another suggestive term for it is a 'myopic algorithm'. Greedy algorithms are generally fast, but in most cases give results that deviate from the optimal solutions by large or even unbounded amounts. A very elegant theory based on matroids has been developed which explains and predicts when a certain type of greedy approach will be guaranteed to yield an optimal solution [see Lawler, 1976]. However, not all greedy methods can be explained by matroid theory. The example below is such a case.

7.3. A problem where the greedy algorithm finds the optimal matching

Weighted matching is a very important problem because many, varied, combinatorial problems can be cast and solved in term of weighted matching.

Sometimes the problem being solved via matching can be shown to have special structure, and this structure can be exploited to speed up finding the optimal weighted matching. Below we can discuss a case where the edge weights have special structure allowing the optimal weighted matching to be found in $O(e)$ time via a greedy approach.

The box inequality

Let u, u', v, v' be four nodes in a complete bipartite graph $G = (N_1, N_2, E)$ (a complete bipartite graph is one where there is an edge between every node in N_1 and every node in N_2) such that u and u' are in N_1 and v and v' are in N_2. Suppose w.l.o.g. that $w(u, v) \geq \max[w(u, v'), w(u', v), w(u', v')]$. If $w(u, v) + w(u', v') \geq w(u, v') + w(u', v)$, then these four nodes satisfy the *box inequality*. A complete bipartite graph is said to satisfy the box inequalities if the box inequality is satisfied for any two arbitrary nodes from N_1 together with any two arbitrary nodes from N_2.

Below we will discuss a problem solved by weighted bipartite matching, where the graph satisfies the box inequality. But for now we show the following theorem.

Theorem 7.4. *If the complete bipartite graph satisfies the box inequalities, and all weights are non-negative, then the greedy matching is a maximum weight matching.*

Proof. Let M be the matching found by the greedy algorithm. If there is more than one maximum weight matching, then let M' be a maximum weight matching which contains the largest number of edges also in M. Suppose M' has weight strictly greater than M. Since G is a complete bipartite graph and all weights are non-negative, both M and M' will be perfect matchings.

Let $M \oplus M'$ be the symmetric difference of the sets of edges in M and in M'. That is, $M \oplus M'$ is the set of edges which are in exactly one of the two matchings M or M'. $M \oplus M'$ must be non-empty for otherwise $M = M'$. Since both matchings are perfect, any edge in $M \oplus M'$ is part of an even length *alternating cycle* where every other edge is from M (M'). Let C be such an alternating cycle, and let $e = (u, v)$ be the first edge in C considered by the greedy algorithm. We claim edge e must be in M. If not, then at the time e was considered either edge (u, v') and (u', v) was in M, for some nodes u', v'. W.l.o.g. say that (u, v') was in M. Then v' is not in C, since e is the first edge on C considered by the algorithm. But this is not possible, because u is in C so the other endpoint (v') of the edge in M touching u must, by definition, also be in C. Hence $e = (u, v)$ is in M.

Let (u, v') and (u', v) be the edges of M' in C that touch u and v. We know these edges exist since C must have at least four edges. Since (u, v) is the first edge in C considered by the greedy algorithm, $w(u, v) \geq \max[w(u, v'), w(u', v)]$. We claim also that $w(u, v) \geq w(u', v')$. This is clearly true if $(u', v') \in M$, since it then would be in C. So suppose (u', v') is not in M. But both u' and v' are in C, so neither were matched at the time (u, v) was

examined, so if $w(u, v) < w(u', v')$, then (u', v') would have been taken into M, a contradiction. Hence $w(u, v) \geq \max[w(u, v'), w(u', v), w(u', v')]$, and by the box inequality, $w(u, v) + w(u', v') \geq w(u, v') + w(u', v)$.

Now consider the matching formed from M' that omits edges (u', v) and (u, v') and includes edges (u, v) and (u', v'). It is immediate that this is still a perfect matching, that it contains one more edge of M than did M', and (by the box inequality) that it has weight greater or equal to that of M'. But that contradicts the choice of M', proving the theorem. □

A matching problem in molecular biology where the box inequalities hold

A central task in molecular biology is determining the nucleotide sequence of strings of DNA. For our purposes, DNA is just a string composed from an alphabet of four symbols (nucleotides): A, T, C, or G. Efficient methods exist for determining the sequence of short strings of DNA, but in order to sequence long strings (which are of more interest) the long strings of DNA must first be cut into smaller strings. Known ways of cutting up longer strings of DNA result in the pieces becoming randomly permuted. Hence, after obtaining and sequencing each of the smaller strings one has the problem of determining the correct order of the small strings. The most common approach to solving this problem is to first make many copies of the DNA, and then cut up each copy with a different cutter so that the pieces obtained from one cutter overlap pieces obtained from another. Then, each piece is sequenced, and by studying how the sequences of the smaller pieces overlap, one tries to reassemble the original long sequence.

Given the entire set of strings, the problem of assembling the original DNA string has been *modeled* as the *shortest superstring problem*: find the shortest string which contains each of the smaller strings in the set as a contiguous substring. In the case when the original DNA string is linear, the problem of finding the shortest superstring is known to be NP-complete but there are approximation methods which are guaranteed to find a superstring at most three times the length of the shortest string [Blum, Jiang, Li, Tromp & Yannakakis, 1991].

The methods discussed by Blum, Jiang, Li, Tromp & Yannakakis [1991] are heavily based on weighted matching, where weight $w(u, v)$ is the length of the longest suffix of string u that is identical to a prefix of string v. In the initial part of the method, the weight $w(u, v)$ is computed for each ordered pair (this can be done in linear time [Gusfield, Landau & Schieber, 1992]), and a bipartite graph $G = (N_1, N_2, E)$ is created with one node on each side of the graph for each of the small DNA strings. The weight of edge (u, v), $u \in N_1$, $v \in N_2$ is set to $w(u, v)$, and a maximum weight matching is found. This matching is then used to construct a superstring which is at most four times the length of the optimal superstring. This error bound is then reduced to three, by another use of weighted matching using suffix–prefix lengths derived from strings obtained from the factor-four solution.

The point of interest here is that for weights equaling the maximum

suffix–prefix overlap lengths, as above, the weights satisfy the box inequality, and hence the two matchings needed in the above approximation method can be found by the greedy algorithm.

Theorem 7.5. *Let graph G with weights w be obtained from a set of strings as defined above. Then G satisfies the box inequalities. That is, if* $w(u, v) \geq \max[w(u, v'), w(u', v), w(u', v')]$, *then* $w(u, v) + w(u', v') \geq w(u, v') + w(u', v)$, *for any nodes u, u' in N_1 and v, v' in N_2.*

Proof. Assume w.l.o.g. that $w(u', v) \geq w(u', v')$. We divide the proof into two cases. Either $w(u', v') = 0$ or $w(u', v') > 0$. The first case is shown in Figure 1. Since it is a suffix of u' that matches a prefix of v, the left end of u' cannot be to the right of the left end of v. Also, since $w(u, v) \geq w(u', v)$, the right end of u' can also not be to the right end of u. So, we define $x = w(u', v) \leq w(u, v)$. With the lengths x, y, z as shown in the figure,

$$w(u, v) + w(u', v') = x + y + z \geq z + x = w(u, v') + w(u', v) .$$

The second case when $w(u', v') > 0$ is shown in Figure 2. With x, y, z as shown in the figure

$$\begin{aligned} &w(u, v) + w(u', v') \\ &\quad = (x + y + z) + y \geq (y + z) + (x + y) = w(u, v') + w(u', v) . \end{aligned}$$ □

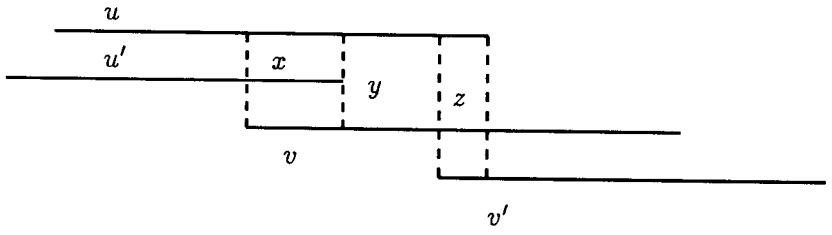

Fig. 1. First case.

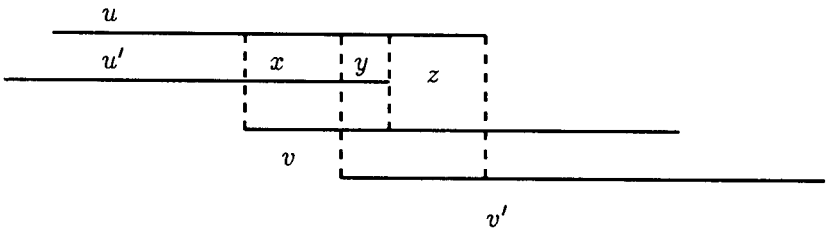

Fig. 2. Second case.

8. Parallel network flow in $O(n^2 \log n)$ time

In this section we discuss how to compute network flow on a parallel computer. But first, we have to define what we mean by a parallel computer. There are many ways that parallel computers have been defined and modeled, some more realistic than others. The most critical differences have to do with whether more than one processor can read or write the same memory cell concurrently, and how the processors communicate. One model that is generally agreed as meaningful is the concurrent read exclusive write parallel random access machine (CREW PRAM).

In the CREW PRAM model there are k processors that are identical except that they each have a unique identifying number. These processors can execute their programs in parallel, and each processor is a general purpose sequential computer, in particular a RAM (random access machine). The processors are assumed to work in lock-step synchrony following a central clock that divides the work of the processors into discrete time units. The processors may have separate memories, but they share a common memory that any of them can read from or write into. Each processor can read or write a single memory cell in a single time unit. We assume that the hardware allows any number of processors to read the same memory cell concurrently (concurrent read), but if more than one processor tries to write to the same cell in the same time step, then the result is unassured (exclusive write). Hence any program for this parallel system should avoid concurrent writes. The processors communicate with each other via the shared memory. The parallel time for a parallel algorithm refers to the number of primitive time units (each of which is enough for a single step of the algorithm on any processor) that have passed from the initiation of the algorithm to its termination, no matter how many processors are active at any moment.

Below we will describe an implementation of the max-d GT algorithm on a CREW PRAM using $k = O(e)$ processors, and a shared memory of size $O(e)$ cells. In particular, there will be a constant number of memory cells and a constant number of independent processors for each node and each edge of G. The algorithm will run in $O(n^2 \log n)$ parallel time. This speed up over $O(n^3)$ is not very dramatic, however, it is the best that is known (for dense graphs) and has been obtained by a variety of different methods, suggesting that improving the bound may be a difficult problem. This bound was first obtained by Shiloah and Vishkin [1982] using only $O(n)$ processors. We will follow ideas that are closer to those given by Goldberg & Tarjan [1988], but again using $O(e)$ processors rather than $O(n)$ processors.

The parallel GT algorithm will be broken up into stages, and each stage will be broken up into four substages. In the first substage of a stage, the label of any active node v is set to $\min[d(w) + 1: (v, w)$ is an edge in the current residual graph]. In the next substage, all the active nodes of max-d label are identified. In the third substage each such node v will, in parallel with the others, push excess out along admissible incident edges until either v has no

more excess or until v's label must be incremented. In the final substage, each node 'receives' its new flow (if any) and recomputes its excess. After this, the stage ends. The algorithm ends in the stage when there are no active nodes. We will provide implementation details and justify the time bound below, but first we establish the following.

Theorem 8.1. *The parallel GT algorithm introduced above correctly computes a maximum flow.*

Proof. In the sequential algorithm, it was easy to establish that if the algorithm terminated, then it terminated with a maximum flow. The key to this was that the node labels were always valid. It is easy to verify that in the parallel GT algorithm, the node labels are valid after the first substage in each stage. Hence it is again easy to see that, if the algorithm terminates, it terminates with a maximum flow. We will establish termination below by bounding the number of stages. □

Lemma 8.1. *In the parallel* max-d *algorithm, there can be at most* $O(n^2)$ *stages.*

Proof. Between two consecutive stages either the max-d value will increase, remain constant, or decrease. In the first two cases the node label of at least one node will increase, and hence there can be at most $2n^2$ stages of this type (the second case is a little more subtle than the first, but is not hard). For the third case, note that since the max-d label of any active node is between $2n$ and 0 (this follows from validity), at any point in time the number of stages that end with max-d decreasing can be at most $2n$ larger than the number of stages that end with max-d increasing. Since there can be only $O(n^2)$ increasing cases, there can be at most $O(n^2)$ stages of the third type as well. □

Below we will show how to implement a stage in $O(\log n)$ parallel time using $O(e)$ processors, yielding an $O(n^2 \log n)$ parallel time algorithm.

8.1. Parallel implementation

We will discuss each substage and show how it can be implemented in $O(\log n)$ parallel time. The central idea in each substage is the same, the use of a binary tree with at most n leaves. The easiest conceptual way to think of the tree is that each vertex in the tree contains a dedicated processor. Since the tree is binary, the number of processors in any tree is $O(n)$, and the height of the tree is $O(\log n)$.

How to update node labels in parallel

A node v can compute its updated $d(v)$ label during the first substage as follows: it signals the processors associated with the possible residual edges out of v to determine (in parallel) which of them are residual edges. Each edge

processor can do that because it knows the current flow on the edge. A processor representing residual edge (v, w) then reads $d(w)$ and inserts it into a leaf in a binary tree for v, in a predefined leaf specified for (v, w). A processor representing a possible residual edge out of v which is not in the current residual graph enters a value large than $2n$ in its predefined leaf. This tree has d_v leaves, where $d_v = |I(v)|$, the number of possible residual edges out of v. Then in $O(\log d_v)$ parallel time, the minimum of the values at the leaves is computed by the obvious tournament computation. That is, once the value is known for each of the two children of a vertex in the tree, the value at that vertex is set to the minimum of the value of its children. The value at the root of the tree is the overall minimum. If the root value is greater than $2n$, then there are no residual edges out of v, and $d(v)$ is unchanged. Otherwise, the processor at the root changes $d(v)$ to one plus the value at the root.

For any node v of G, the total number of processors needed to implement the tournament is $O(d_v)$ since the binary tree contains only that many vertices. So the number of processors involved in this substage is $\Sigma_v d_v = O(e)$.

How to find the active nodes of max-*d label*

The processor for any node v knows the current excess at v and the updated $d(v)$. Each processor associated with an active node inserts the current d value in a predefined leaf of a binary tree. Then by the tournament method again, and in $O(\log n)$ time using $O(n)$ processors, the maximum of these values is found and placed in a memory cell. All the active node processors then read this cell in unit time to determine if their node is of max-d.

How to push in parallel

For each node v in G, we will again use a binary tree with d_v leaves, where each leaf in the tree is associated with a possible residual edge out of v. Again, each vertex in the tree has an assigned dedicated processor. Hence over all the trees there are $O(e)$ processors assigned. Essentially, the leaves of the tree correspond to the list $I(v)$ described in Section 4.2.

We will determine in $O(\log d_v)$ parallel time which edges v should use to push out its excess, and how much to push on each edge. At the start of this substage, the processor for the leaf corresponding to edge (v, w) determines whether (v, w) is admissible (by reading $d(v)$, $d(w)$ and knowing whether (v, w) is a residual edge), and what the capacity $c(v, w)$ in the current residual graph is. If the edge is admissible, then its current capacity is written at the leaf. If the edge (v, w) is not admissible, then a zero will be written instead of its capacity.

Processing the vertices of the tree bottom up from the leaves, we collect at each vertex the sum of the numbers written at the leaves in its subtree. This process is completed in $O(\log d_v)$ parallel time, since the entry at a vertex is the just the sum of the two entries of its two children, and the depth of the tree is $O(\log d_v)$. At this point, all the processors associated with leaves of v's tree

can read, in unit time, the sum at the root, to determine if all their associated edges will become saturated or not.

If the number at the root is smaller or equal to $e(v)$, then every admissible edge out of v will get saturated. This message is sent back down the tree to the leaf processors which make the flow changes on their associated edges.

If the sum at the root of v's tree is greater than $e(v)$, then only some of the admissible edges out of v will be used. However, at most one edge will get new flow without getting saturated. The allocation to edges is done in $O(\log d(v))$ time by working down from the root. Essentially, we will write at each vertex x of the tree the amount of flow to be pushed from node v along edges associated with leaves in the subtree of x. Hence the number written at a leaf for edge (v, w) is the amount to be pushed along edge (v, w) in the residual graph. In detail, we do the following: let $n(x)$ denote the number presently written at tree vertex x. First change $n(r)$, the number written at the root, to $e(v)$. For any tree vertex x, let x' be the right child of r and x'' be its left child. Set $n(x')$ to the minimum of $n(x')$ and $n(r)$; then set $n(x'')$ to the maximum of $n(r) - n(x')$ and zero. Once the numbers at the leaves have been written, the processors associated with those edges can make the flow changes on these edges in parallel.

How to receive flow

In the previous substage flow was pushed along certain edges. The amount pushed out of a node v is known during the substage and so the excess at v is updated then. However, the total flow entering a node w must also be determined. This is again easily computed in $O(\log n)$ parallel time using a binary tree for each node, where each leaf is associated with a possible residual edge into w. We omit the details.

8.2. Final result

Since there are $O(n^2)$ stages in the parallel method and each can be implemented to run in $O(\log n)$ parallel time with $O(e)$ processors, we get the following theorem.

Theorem 8.2. *A maximum flow can be computed in* $O(n^2 \log n)$ *parallel time using* $O(e)$ *processors.*

8.3. Work versus time

In the above implementation we used $O(e)$ processors to compute maximum flow in $O(n^2 \log n)$ parallel time. If we were to convert this parallel algorithm to a sequential one we would immediately obtain a sequential time bound of $O(en^2 \log n)$, which is much larger than the known $O(n^3)$ bounds. This leaves the intuition that a 'better' parallel algorithm may be possible, one which either uses fewer processors or has a smaller worst-case bound.

To examine this issue we define the *work* of a parallel algorithm to be the product of the parallel time bound and the maximum number of processors needed by the algorithm. There are network flow algorithms which run in $O(n^2 \log n)$ time using only $O(n)$ processors, hence using only $O(n^3 \log n)$ work [Shiloah & Vishkin, 1982], and similar results have been obtained for sparse graphs [Goldberg & Tarjan, 1988]. The notion of work is important not only as a way of distinguishing between equally fast parallel algorithms, but because the number of assumed processors may not be available. In that case the degradation of the running time is related to the assumed number of processors; the smaller the number of assumed processors, the smaller the degradation.

In addition to reducing the work needed by the parallel algorithm, most parallel algorithms for network flow have been implemented on the EREW PRAM model. In that model only one processor can read (exclusive read) any memory cell in one unit. Since this is a more restrictive assumption than for a CREW PRAM, results on EREW PRAMs are considered more realistic, or at least more likely to be implementable on real parallel machines.

9. Distributed algorithms

In this section we briefly discuss another form of parallel computation, namely *distributed computation*. There are numerous particular models of distributed computation but they all try to capture the notion of several autonomous, often asynchronous, processors each with their own memory solving some problem by their joint actions and restricted or local communication. This is in contrast to the single processor model, and even in contrast to models of synchronous parallel computation, where there is an assumed common clock, shared memory, and possibly a very rich or highly structured communication network. In particular, in the distributed model, computation on any processor is considered to be much faster than interprocessor communication, and so it is the later time that is to be minimized.

The efficiency of a distributed algorithm is usually measured in terms of the number of messages needed, and the complexity of the computations that each processor does. Alternatively, one can measure the worst-case, or average-case, parallel time for the system to solve its problem. To make this measure meaningful, we usually assume some maximum fixed time that a processor will wait after it has all the needed inputs, before executing a single step of the algorithm. However, it is possible to sometimes relax even this assumption, and there also are general techniques for converting asynchronous distributed algorithms to synchronous ones with only a small increase in the number of messages used.

In this article we can only give some flavor of the nature of distributed computation, and will examine only a very simple problem, the *shortest path communication* problem. Although simple, this problem actually arises in some distributed versions of the Goldberg–Tarjan network flow algorithm.

9.1. Shortest path communication

Consider the situation where each node v on a directed graph must repeatedly send messages to a fixed node t, and so wants to send the message along a path with the fewest number of edges (a shortest path). The nodes are assumed only to know who their neighbors are in the graph. In order to know how to route a message on a shortest v to t path, v only needs to know the first node w after v on such a shortest path, since the edge (v, w) followed by a shortest w to t path is shortest v to t path. When the system first starts up, or after some edges have been removed, the distances to t increase, no node can be sure of the shortest path from it to t. The goal at that point is for the system to begin some asynchronous, distributed computation, so that at the end of the computation, every node v learns which one of its neighbors is the first node on the shortest path from v to t. We ignore for now the question of how the nodes know to start this process. In the KE, Dinits or GT network flow algorithms, the distance to t from a node v in the evolving residual graph does only increase as was shown earlier.

We first assume that a central clock is available, so that the actions of the processors can be divided into distinct iterations. Later we will remove this assumption. Let H be any directed graph with designated nodes s and t. For any node v, let $D(v)$ denote the number of edges on the shortest v to t path in H.

Lemma 9.1. *For each node v, let $\hat{D}(v)$ be a number assigned to node v, such that $\hat{D}(t) = 0$. If for every node v, $\hat{D}(v) = 1 + \min[\hat{D}(w)$: (v, w) is a directed edge in $H]$, then $\hat{D}(v)$ is the distance of the shortest (directed) path from v to t in H.*

Proof. Suppose not, and let v be the closest node to t such that $\hat{D}(v) \neq D(v)$, the true distance from v to t. Let P_v be the shortest path from v to t and let w be the node after v on P_v. By assumption $\hat{D}(w) = D(w)$. But then $\hat{D}(v) \leq \hat{D}(w) + 1 = D(w) + 1 = D(v)$. So, suppose that $\hat{D}(v) < D(v)$; then there is an edge (v, u) to a node u such that $\hat{D}(u) < \hat{D}(w)$ and, by assumption, $\hat{D}(u) = D(u)$. But that would contradict the assumption that P_v is the shortest v to t path. Hence $\hat{D}(v) = D(v)$ for every node v. □

We will use the above lemma to compute the shortest distances from each node to t. Suppose we start off with numbers $\hat{D}(v)$, where for all v $\hat{D}(v) \leq D(v)$, and $\hat{D}(v) \leq 1 + \min[\hat{D}(w)$: (v, w) is a directed edge in $H]$. We call a node v *deficient* if the second inequality above is strict. We can think of $\hat{D}(v)$ as an underestimate of $D(v)$. We will modify the $\hat{D}(v)$ values in the following algorithm, then show that the algorithm terminates, and that at termination, $\hat{D}(v) = D(v)$ for each v.

Distance estimate algorithm
Repeat

Step A. For at least one node v such that the current $\hat{D}(v)$ is less than $1 + \min[\hat{D}(w)$: (v, w) is an edge in $H]$, i.e., v is deficient, set $\hat{D}(v) = 1 + \min[\hat{D}(w)$: (v, w) is an edge in $H]$.

Until $\hat{D}(v) = 1 + \min[\hat{D}(w)$: (v, w) is an edge in $H]$ for every node v.

Note that if more than one deficient node is selected in Step A (any number are possible), then the updates to the $\hat{D}$ values are made in parallel.

Theorem 9.1. *The distance estimate algorithm terminates, and upon termination* $\hat{D}(v) = D(v)$ *for all* v.

Proof. First we show that $\hat{D}(v) \leqslant D(v)$ throughout the algorithm. Suppose not, and let v be the first node set to a value above $D(v)$. As before, let w be the first node after v on P_v. Then at the point that $\hat{D}(v)$ is set above $D(v)$, $\hat{D}(w) \leqslant D(w)$, so $\hat{D}(v) \leqslant 1 + D(w) = D(v)$, which is a contradition. Now the algorithm must terminate since each time a $\hat{D}(v)$ value is changed it is increased by a least one, and $\hat{D}(v)$ is bounded by $D(v)$. At termination, the conditions of Lemma 9.1 are satisfied, hence the theorem is proved. □

We now add implementation detail to the distance update algorithm so that it runs efficiently. We will assume that at the beginning of the algorithm the minimum of $\{\hat{D}(w)$: (v, w) is a directed edge in $H\}$ is known for each v, and hence that all the initially deficient nodes are known. Note that a deficient node remains deficient until its $\hat{D}$ value is increased.

Although we assume that initially deficient nodes are known, we will need an efficient way to identify nodes which become, or become again, deficient during the algorithm. It is not efficient to have each node scan the $\hat{D}$ values of its neighbors at each iteration of Step A. That would take $O(e)$ messages per iteration. Instead, whenever a $\hat{D}(v)$ is changed, the updated $\hat{D}(v)$ is sent to each node u such that (u, v) is a directed edge in H. If a node u receives such a message in one iteration, in the next iteration it compares its current $\hat{D}(u)$ against one plus the values it received in the previous iteration. The smaller of these candidate values is then taken as the current $\hat{D}(u)$. We associate the work to send these messages and then to do the comparisons, to the nodes of the previous iteration whose values have changed. In this way, the total amount of work involved in such checking is just $O(n)$ times the number of nodes whose $\hat{D}$ value changed in the previous iteration. Hence we have the following theorem.

Theorem 9.2. *Let* $\hat{D}(v)$ *be the initial values given to each node* v, *and let* $S = \Sigma_v [D(v) - \hat{D}(v)]$, *where* $D(v)$ *is the correct* v *to* t *distance. Ignoring the work to locate the initially deficient nodes, the distance estimate algorithm uses at most* $O(nS)$ *message passes.*

9.2. The asynchronous case

The distance estimate algorithm is more sequential than is permitted in the distributed model. The problem is that in the above discussion of the distance estimate algorithm there are definite iterations of Step A, and there is a centralized clock that establishes the starts and ends of each iteration. The implementation specified that every node receiving one or more messages from its neighbors in one iteration should use this information to update its $\hat{D}$ value in the next iteration. But in the distributed model, the processors run at differing speeds, and there is no central clock, so the notion of iterations is not applicable. This difficulty is easily handled.

We change the implementation of the algorithm so that whenever a node v decides to update its distance estimate, it simply uses its knowledge of who its neighbors are (this knowledge is assumed to be current), and the $\hat{D}$ values that it has received *since* $\hat{D}(v)$ *was last changed*. The algorithm with this modification is truly distributed, and the estimates will converge to the correct distances. The proof of this is almost identical to that of Theorem 9.1. Hence if $D(v)$ is the correct distance from v to t, and $\hat{D}(v)$ is the initial estimate, then the system will converge to the correct distance values using only $n \Sigma_v [D(v) - \hat{D}(v)] = nS$ messages.

Although the $\hat{D}$ values will converge to the correct distances, how will the nodes in the system know when this has happened? Whenever a node v adjacent to t sees that $\hat{D}(v) = 1$, it knows that $\hat{D}(v) = D(v)$. This will eventually happen for every node v whose shortest path to t is the single edge (v, t), and there certainly must be one such v. When a node realizes it has the correct D value, it sends an appropriate message to its neighbors. In general, any node v which learns that the true distance of all its neighbors have been established, and which sees that $\hat{D}(v) = 1 + \hat{D}(w)$ for its neighbor w with minimum $\hat{D}(v)$, knows that $\hat{D}(v) = D(v)$ and should send a message to its neighbors. In this way, all the nodes in the system eventually realize that the correct distances have been found, and only e additional messages have been passed. So the total number of messages used is $nS + e$.

We leave to the reader the problem of modifying the distance estimate algorithm so that it can work even when path distances decrease, and in the case that edge distances can take values other than one.

10. Many-for-one results

10.1. Introduction

It often happens that a sequence of related instances of a problem must be solved. In some cases, each instance must be solved from scratch, but in many notable cases it is possible to solve all the instances at a cost which is substantially less than the total cost of solving each instance from scratch.

Results of this type are called *many-for-one* results, or *quantity-discount* results. If one can solve all the problem instances for the price of one, then the result is called an *all-for-one* result. We have actually already seen such a result, namely the parametric flow problem. There we saw that $O(n)$ problem instances, in a highly structured sequence of problem instances, could be solved in $O(n^3)$ time, which is the best (dense) bound presently known for even one problem instance. Here we consider another important class of flow related problems for which there is an elegant many-for-one result.

To start, we consider the problem of computing the maximum flow value for each of the $\binom{n}{2}$ possible source–sink pairs in an undirected, capacitated graph G. If each pair were considered separately, then $\binom{n}{2}$ solutions of a network flow problem would be required. However, we will show that all $\binom{n}{2}$ flow values can be determined after computing the flow between only $n-1$ source–sink pairs. Hence the total computation is a factor of n faster than the straightforward approach.

This result was originally obtained by Gomory & Hu [1961]. Their method produces an edge weighted tree with n nodes, such that the value of the maximum flow in G for any pair of nodes, say s and t, is the minimum weight of the edges on the path between s and t in the tree. A tree which represents the flow values in this way is called an *equivalent-flow tree*. Gomory and Hu's method in fact constructs a special equivalent flow tree called a *cut-tree* with an additional desirable feature: for any pair of nodes (s, t) if you remove the minimum weight edge on the path from s to t, then the resulting partition of nodes defines a minimum (s, t) cut in G. Hence a cut-tree not only compactly represents flow *values*, but also compactly represents one easily extracted minimum cut for each pair of nodes. It is not true that every equivalent-flow tree for G is also a cut-tree for G.

The key algorithmic feature of the Gomory–Hu method is the maintenance of 'non-crossing' cuts. In the method, if a minimum cut $(X, \bar{X})$ has been found between a pair of nodes, then every successive cut $(Y, \bar{Y})$ computed by the method (for any other pairs of nodes) must have the property that either all of X, or all of $\bar{X}$ is on one side of the $(Y, \bar{Y})$ cut. That is, the $(Y, \bar{Y})$ cut splits only one side of the $(X, \bar{X})$ cut. The implementation detail to enforce this non-crossing property of the cuts makes the method complicated to program.

Simpler methods which avoid the need to find non-crossing cuts, and which also require only $n-1$ flow computations, were later obtained [Gusfield, 1990c] for finding equivalent-flow trees and cut-trees. These methods work with any minimum cuts, whether they cross or not. Although the algorithms do not need to maintain non-crossing cuts, their *existence* is central to the *proofs* of correctness used by Gusfield [1990c].

Recently, the role of non-crossing cuts has been further reduced with the development, by Cheng & Hu [1989], of a new, equally efficient algorithm, that produces a tree called an *ancestor cut-tree*. This tree has all the advantages of an equivalent-flow tree (and others as we will see), but lacks some of the advantages of a true cut-tree.

10.2. The Cheng–Hu method for ancestor cut-trees

We will represent the minimum cut values of a graph G, containing n nodes, with a binary tree T. Each internal vertex of T will be labeled with a source–sink pair (p, q), and will be associated with a minimum (p, q) cut in G; each leaf of T will be labeled by one node in G, and each node in G will label exactly one leaf of T. Further, for any two nodes i, j in G, if the least common ancestor of i and j in T is labeled by the pair (p, q), then the associated minimum (p, q) cut is a minimum (i, j) cut as well. Hence this tree represents the maximum flow values for every pair of nodes, and allows the retrieval of one minimum cut for any pair of nodes. However, it is not as compact as a cut-tree, for a cut-tree takes only $O(n)$ space, while the cuts associated with an ancestor cut-tree are stored explicitly and hence take $\Omega(n^2)$ space.

As an example, the graph shown in Figure 3a has an ancestor cut-tree shown in Figure 3e.

Note that for clarity, the word 'node' refers to a point in G, while 'vertex' refers to a point in an ancestor cut-tree.

The algorithm builds successive trees $T_0, T_1, \ldots, T_{n-1} = T$, each containing one more leaf than its predecessor. The following fact about any T_i will be proved in Lemma 10.1 later.

Fact. *If v is any internal vertex of T_i labeled with the pair (s, t), and its two children are labeled with the pairs (i, j) and (p, q), then i and j are together on one side of the (s, t) minimum cut associated with v, and p and q are together on the other side of the cut.*

In order to describe the algorithm, we first define T_i' to be the subgraph of T_i consisting of the internal vertices of tree T_i, and describe how to place the leaves of T_i, given tree T_i'. This process is called *sorting* the nodes of G into T_i'.

Starting at the root of T_i', we separate the nodes of G according to the cut specified at the root node. For example, if the cut at the root is an (s, t) cut, then we place on one branch out of the root all the nodes of G on the s side of the cut, and on the other branch out of the root we place all the nodes of G on the t side of the cut. There still is a question of which edge to use for which set. Suppose the two children of the root are labeled with the pairs (i, j) and (p, q). Given the fact stated above, we use the following rule to assign the two parts of the (s, t) cut to the two edges out of the root of T_i': the part of the (s, t) cut containing i and j is placed on the edge from the root to its child labeled with the (i, j) cut, and the part containing p and q is placed on the other edge out of the root.

In general, at any vertex x to T_i', we split the nodes of G that are on the edge leading to x into the two edges out of x, according to how the cut at x separates these nodes. To decide which set goes on which of the two edges out

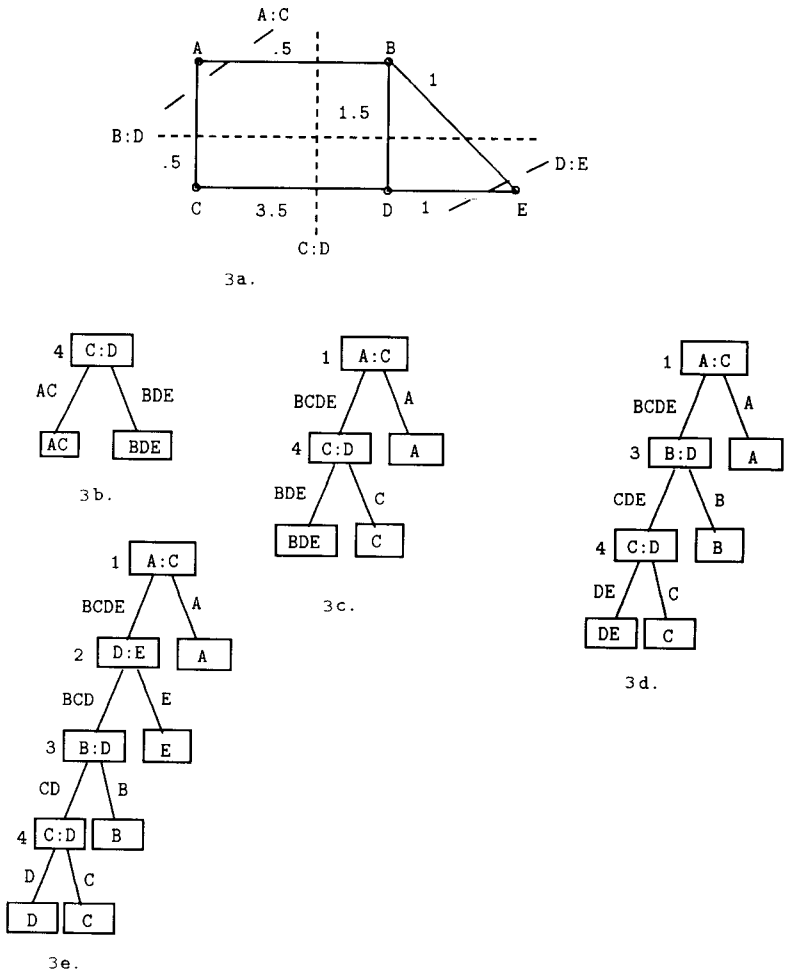

Fig. 3. Graph G and the cuts used by the algorithm.

of x, we follow the same rule stated for the root. If x is a leaf of T_i', then the nodes of G on the edge entering x are split into two children of x according to how the cut labeling x splits these nodes. The children of x are then leaves in T_i.

For example, in Figures 3b through 3e, the nodes written on the edges of the intermediate trees show the sorting process.

An added feature of any tree T_i, which will be maintained inductively, is that after each iteration of the algorithm, the set of nodes contained in any leaf will have exactly one designated node called the *representative* of that set.

The full algorithm is now the following.

The Cheng–Hu algorithm

Set k to 0.

The initial tree T_0 consists of a single leaf containing all the nodes of G. Arbitrarily set one of the nodes to be the representative of this leaf.

Repeat

Pick a leaf x of T_k which contains more than one node of G; suppose i is the representative of x, and let j be any other node of G in x. Declare j to be a representative.

Find a minimum cut $(X, \bar{X})$ between i and j; let its value be $f(i, j)$, and assume that $i \in X$.

Find the closest ancestor vertex y of x in T_k whose cut value is less than or equal to $f(i, j)$; let z be the vertex below y on the path from y to x in T. Create a vertex labeled with (i, j), and place it between y and z in T_k. Remove all the leaves of T_k, creating tree T'_{k+1}; then sort the nodes of G into T'_{k+1}, creating tree T_{k+1}. Set $k := k + 1$.

Until each leaf node of T contains only a single node of G.

Note that z may be a leaf of T_k. Note also that at least one of the children of vertex x is a leaf of T_{k+1}.

Correctness of the algorithm

The key to the correctness of the algorithm is that every intermediate tree T'_{k+1} is sortable. For a tree to be sortable, we need that the (s, t) cut (say) at any internal vertex x partitions the nodes on the edge coming into x such that all nodes in labels of the internal vertices in the left subtree of x are on one side of the (s, t) cut, and all nodes in labels in the right subtree of x are on the other side of the cut. If this condition is satisfied, then the tree is sortable. To prove that the tree is always sortable, we start with the following definition.

Definition. For a vertex x in T_k, let p and q be any two nodes of G which are each used in some label (not necessarily the same label) of a vertex in the subtree of T_k rooted at x. We say that p and q are *connected* in the subtree of x, if there exists a sequence $(p, v_1), (v_1, v_2), \ldots, (v_j, q)$, where the second node in each pair is the first node in the succeeding pair, and each pair is a label of a vertex in the subtree of x.

For example, in Figure 3e let x be the root of the tree, and let p be A and q be B. Then p and q are connected in the subtree rooted at x through the path $(A, C), (C, D), (D, E)$.

Lemma 10.1. *Any intermediate tree T'_{k+1} produced by the algorithm is sortable. In addition, all of the nodes in pairs labeling the vertices in the subtree of x are connected.*

Proof. We prove the lemma inductively. T_0 is clearly sortable, and since it has no internal nodes, it has the claimed connectedness property. Suppose T'_k is sortable and has the connectedness property. Let (i, j) be the source–sink pair used by the algorithm to create T_{k+1}, where i is a representative of a leaf w in T_k. Let x denote the new vertex created, labeled with the pair (i, j).

Suppose first that in T'_{k+1}, x takes the place that w occupied in T_k, then T'_{k+1} is sortable since T'_k was, and the new (i, j) cut simply splits the nodes coming into x into two branches. Further, the label of the parent of x must contain either i or j (by the way that representatives of leaves are created, and the fact that T'_k was sortable). Suppose, w.l.o.g., its label is (i, s). Then, there is a sequence (s, i), (i, j) in T_{k+1}, so j is also connected to s in T_{k+1}. Since the connectedness property holds for T_k, s is connected to all nodes in labels of the subtree of x in T_{k+1}.

Now suppose that x is inserted between two internal vertices y and z, where y is the parent of z. We first show that T'_{k+1} is sortable. All nodes that were on the incoming edge into z in T_k are now on the incoming edge into x in T'_{k+1}. Hence, all vertex labels of the subtree rooted at x are contained on the set of nodes coming into x. In particular, i and j are on that edge.

For any internal vertex, labeled (s, t), in the subtree rooted at x, $f(s, t) > f(i, j)$, and therefore the (i, j) cut cannot separate s from t. Further, by induction, all nodes used in labels in this subtree are connected, so it follows that the (i, j) cut cannot separate any two nodes p and q which are used as labels in this subtree. For suppose that the (i, j) cut did separate p from q. Then the (i, j) cut must also separate two nodes in a label on the chain of labels connecting p and q, a contradiction. Thus, the cut at x partitions the incoming nodes into two sets, such that one set contains all labels in the subtree of x. The part of the cut containing these nodes is passed on to z, while the other part becomes a leaf of T_{k+1} below vertex x. From z downwards, the tree is certainly sortable as before.

To show that the connectedness property holds for T_{k+1}, consider an arbitrary subtree of T_{k+1}. If it does not contain the new vertex x, then the claim clearly holds for it. If it does contain x, then it must contain the immediate ancestor of leaf w in T_k. As before, assume that the label of that ancestor is (i, s). Now (i, s), (i, j) is a sequence in the subtree of T_{k+1} rooted at x. Further, by the induction hypothesis, in T_k, s and t are connected in the subtree of z if t is used in a vertex label in the subtree of z. It follows that i and t and j and t are connected in the subtree of x in T_{k+1}, and the connectedness property holds. □

Lemma 10.2. *If the least common ancestor of nodes i and j has label* (p, q), *then the associated minimum* (p, q) *cut separates i and j, and so* $f(i, j) \leq f(p, q)$.

Proof. Follows immediately from the sorting process. □

Lemma 10.3. *Let $S = (i, v_1), (v_1, v_2), \ldots, (v_k, j)$ be a sequence of node pairs, where the second node of each pair is the first node of the succeeding pair. Then $f(i, j) \geq \min[f(x, y)$: (x, y) is a pair in $S]$.*

Proof. Let $(X, \bar{X})$ be a minimum (i, j) cut. Since $i \in X$ and $j \in \bar{X}$, there must be a pair $(v_h, v_{h+1}) \in S$ such that $v_h \in X$ and $v_{h+1} \in \bar{X}$, and hence $f(v_h, v_{h+1}) \leq f(i, j)$, and the lemma follows. □

Theorem 10.1 *For any nodes i and j in G, the cut (p, q) at the least common ancestor x of i and j in T is a minimum (i, j) cut in G.*

Proof. By Lemma 10.2, $f(i, j) \leq f(p, q)$. Now consider the set of vertex labels in the subtree of T rooted at x. By applying the connectedness property shown in Lemma 10.1, we can connect *all* the vertex labels in the subtree of x into a single sequence $(i, v_1), (v_1, v_2), \ldots, (v_k, j)$ (vertex labels may be repeated). Then by Lemma 10.3, $f(i, j) \geq \min[f(u, v)$: (u, v) is a vertex label in the subtree of $x]$. But by construction of T, x has a smaller associated cut than any vertices in its subtree, and so $f(p, q) \leq f(u, v)$ for any label (u, v) in the subtree of x. Therefore $f(i, j) \geq f(p, q)$, and the theorem follows. □

So the ancestor cut-tree can be built with $n - 1$ flow computations, takes $O(n)$ space, and can be used to retrieve in $O(n)$ time the minimum cut value of any pair of nodes in G. In fact, any value could be retrieved in $O(1)$ time after an initial preprocessing phase taking $O(n)$ time. This is accomplished by using a fast *least common ancestors* algorithm [Harel & Tarjan, 1984, Schieber & Vishkin, 1988] that will be briefly discussed in the next section. If simpler methods are desired, it is not difficult to devise a method to collect all the $\binom{n}{2}$ values from the tree in $O(n^2)$ time. Hence the ancestor cut tree has all the advantages of an equivalent-flow tree. Further, by Lemma 10.2, for any pair of nodes (i, j), the tree can be used to retrieve an actual (i, j) minimum cut. However, $n - 1$ minimum cuts need to be explicitly stored, so it does not have all the advantages of a cut-tree.

We should note that the use of representatives in the algorithm makes the correctness proof easier, but their use is not essential. In fact, the algorithm would be correct if we arbitrarily select any pair of nodes in a leaf. Also, if the values for only a subset of the node pairs are needed, then the algorithm can be terminated early. We leave the details to the reader.

10.3. Additional uses of the ancestor cut-tree

Closer examination of the proof of Theorem 10.1 yields the following important observation made by Cheng & Hu [1989]. Suppose that instead of defining the value of a cut $(X, \bar{X})$ as the sum of the edge capacities crossing the cut, we give the cut an arbitrary value. Then for a pair of nodes (i, j) we define $f(i, j)$ as the minimum value of all the cuts separating i from j. Under this

definition of $f(i, j)$, all of the lemmas and theorems in the preceding section remain valid, for all of them depend only on the fact that $f(i, j)$ is the minimum value of the cuts separating i from j. Hence there exists ancestor cut-trees for any such values applied to cuts in G. Further, if it is possible to efficiently find a cut of value $f(i, j)$ when a pair (i, j) is specified, then an ancestor cut-tree for these cut values can be found efficiently, with only $n-1$ calls to the routine which gives the cut values and the cuts.

There are many applications of this more general cut framework. As one useful application, let G be a *directed* graph with edge capacities, let $C(X, \bar{X})$ be the sum of the capacities of the edges crossing from X to $\bar{X}$, and let $C(\bar{X}, X)$ be the sum of the capacities of edges from $\bar{X}$ to X. Then we define the *value* of the cut (partition) X, $\bar{X}$ to be the minimum of $C(X, \bar{X})$ and $C(\bar{X}, X)$, and we define $f(i, j)$ as before to be the minimum *value* of all the cuts which separate i from j. By the max-flow min-cut theorem $f(i, j) = \min[F(i, j), F(j, i)]$, where $F(i, j)$ is the maximum flow value from i to j, and $F(j, i)$ is the maximum flow value from j to i. Since G is directed, $F(i, j)$ need not be equal to $F(j, i)$. This particular function $f(i, j)$ was studied by Schnorr [1979] who called the function $\beta(i, j)$ and used it in computing the directed connectivity of a graph. He showed that all the $\beta(i, j)$ values could be computed with $O(n \log n)$ minimum cut calculations in a graph of n nodes, although these flows can be implemented to run in $O(n^4)$ amortized time. However, the ancestor cut-tree, when G has been given the above cut values, clearly also represent $\beta(i, j)$. Moreover, for any i and j, $\beta(i, j)$ and the associated cut can be found with only two network flow computations – the maximum flow from i to j, and the maximum flow from j to i. Hence, an ancestor cut-tree for the β function can be constructed in $2n-2$ flow computations on G. This achieves the same overall time bound as the Schnorr method, $O(n^4)$, but it does not need any of the involved implementation details that the Schnorr method uses to achieve that amortized bound. Further, when the graph has special properties allowing a specialized faster than general network flow method to be used, the time to build the ancestor-tree is automatically improved, while the Schnorr method may not be able to exploit the faster flow.

One of the applications of the β function is in the area of data security [Gusfield, 1988], where it is shown how to compute the tightest upper bounds on a secure matrix entry (i, j) by computing $\beta(i, j)$ in a directed graph derived from the matrix. Hence the tightest upper bounds on all the cell values can be computed with only $O(n)$ flow computations, even though there may be $\Omega(n^2)$ upper bounds that need to be determined. The time to determine all these values is then the time for just $n-1$ flow computations, plus a total of $O(n^2)$ time. Full details of this application are given by Gusfield [1990a].

11. The power of preprocessing: The least common ancestor problem

The previous section discussing the Cheng–Hu algorithm stated that the least common ancestor of any two leaves of a tree can be found in *constant* time,

after a *linear* time amount of preprocessing. That is, the tree is first preprocessed in linear time, and thereafter each least common ancestor query takes only constant time, no matter how large the tree is. Without preprocessing, the best worst-case time bound for a single query is linear, so this is a most surprising and useful result. Since it has applications in network algorithms and it illustrates the tremendous speedup that can sometimes be achieved through preprocessing, we will briefly discuss some of the ideas used for this method.

The actual method we will present is not 'correct' because it needs a few (probably realistic) assumptions and needs to allocate (but not use) more than linear space. Still, it will capture the spirit of the correct method and will be practical for reasonable sized problems.

11.1. An 'incorrect' method

Suppose first that the tree T is a rooted binary tree, so that every internal node has exactly two children. Let d be the length of the longest path from the root to a leaf in T and let n be the number of nodes in T. We will first label each node v in T with a description of the unique path from the root to node v as follows. Counting from the leftmost bit of the desired node label, the ith bit corresponds to the ith edge on the path from the root to v; a zero in bit i indicates that the ith edge goes to a left child, and a one in bit i indicates a right child. So for example a path that goes left twice then right and then left again ends at a node which will be given the label 0010. We will now extend these node labels so that the root also has a label and so that all labels consist of $d + 1$ bits. To do this we add a 1-bit to the right end of every label, and then add 0-bits to the right of each label so that each resulting label has exactly $d + 1$ bits. We will use $L(v)$ to denote the resulting label of node v.

It is not difficult to see how to set these labels in $O(n)$ time during a depth first traversal of T, *if* we assume that multiplication by a number as big as 2^d can be done in constant time. During the depth first traversal, we construct the path label of v by shifting the path label of its parent v' one bit to the left (multiplying by two) and adding one if v is a right child of v'. To get the final label for v we multiply its label by two, add one, and then multiply by 2^{d-k} (shift left by $d - k$ bits).

We keep $L(v)$ at node v in T, so that when given v we can retrieve $L(v)$ in constant time. Conversely, we will also need to be able to find any node v from its label $L(v)$ in constant time. One simple, but very space inefficient way to do this is to reserve a space A of 2^{d+1} words addressed by all the possible binary numbers of length $d + 1$. Whenever a label $L(v)$ is computed, we write v in $A(L(v))$. So the total time we set up this space is only $O(n)$. A more practical approach would be to hash the L labels into a space much smaller than A.

Now suppose we want to find the least common ancestor of nodes x and y, and say that it is node z. We first take the *exclusive or* (XOR) of $L(x)$ and $L(y)$ and look for the leftmost 1-bit in the resulting number. The XOR of two bits is 1 if and only if the two bits are different. The XOR of two numbers, each of

which consists of $d+1$ bits, is just the XOR of the $d+1$ pairs of bits taken independently. For example, XOR of 00101 and 10011 is 10110.

Suppose that the leftmost 1-bit in the XOR of $L(x)$ and $L(y)$ is in position k counting from the left. Then the leftmost $k-1$ bits of $L(x)$ and $L(y)$ are the same, and hence the paths to x and y agree for the first $k-1$ edges, and then diverge. It follows that $L(z)$ consists of the leftmost $k-1$ bits of $L(x)$ [or $L(y)$] followed by a 1-bit followed by $d+1-k$ zeroes. We assume that our machine can find the leftmost 1-bit in a number in constant time. Again, this is not an unreasonable assumption on most machines, depending on how big n is. So $L(z)$ can be found in constant time. Given $L(z)$ we find z in entry $A(L(z))$ in constant time.

11.2. How to handle non-binary trees

If the original tree T is not binary we modify any node with more than two children as follows. Suppose node v has children $v_1, v_2, \ldots, v_k$. Then we replace the children of v with two children v_1 and v^* and make nodes $v_2, \ldots, v_k$ children of v^*. We repeat this until each original child v_i of v has only one sibling, and we place a pointer from v^* to v for every new node v^* created in this process. Later, whenever any such a new node v^* is returned by the least common ancestor algorithm, the pointer at v^* leads to the true least common ancestor v in T. Note that the transformed tree has at most $2n$ nodes.

So assuming a shift (multiply) by up to d bits can be done in constant time, that XOR on d bits can be done in constant time, that the leftmost 1-bit can be found in constant time, and that we have space of size 2^{d+1}, the above gives a linear time preprocessing method, and constant time look-up method for the least common ancestor problem. All these assumptions are reasonable for reasonable values of n, except for the space required. However, we can still obtain a practical method (but lacking the theoretical constant time guarantee) if we use hashing for A, since we only hash n numbers.

11.3. A peek at a correct method

The deficiencies of the above method are: we must allow word sizes to be $d+1$ bits which in worst case is $n+1$ bits (although in any actual case of $d=n$, the ancestor problem is trivial); we assume the ability to shift (by at most d bits), to do XOR and to find the leftmost 1-bit in constant time; and we need space 2^{d+1} or must use hashing.

The correct method avoids all of these deficiencies with the same idea. It efficiently maps the nodes of T into a *balanced* binary tree B with n nodes in such a way that if x maps to $B(x)$ and y maps to $B(y)$, then the least common ancestor of $B(x)$ and $B(y)$ in B can be used to quickly find the least common ancestor of x and y in T. Since B is balanced its maximum depth d is $O(\log n)$. That means that only $O(\log n)$ bits need to be used for L labels, and that the other assumptions needed in the first method need not be made. The way that the mapping is done is complex, and is a major achievement.

This peek at the correct method has not done justice to the real method, and the interested reader is referred to the paper by Schieber & Vishkin [1988]. The fact that the problem can be solved in constant time was first shown by Harel & Tarjan [1984].

12. Randomized algorithms for matching problems

We will now consider the use of randomization in algorithms. In particular we will discuss how randomization can be used in a fast parallel algorithm for *constructing* a maximum cardinality matching in a bipartite graph. The major focus is randomization, but the specific topic will also allow an additional look at parallel algorithms.

Recall from Section 7 that a maximum cardinality matching can be computed in $O(n^{2.5})$ time by maximum flow in a graph where all edges have capacity one. This is faster than the $O(n^3)$ bound for flow in general graphs. As another distinction between flow in this special bipartite graph and general maximum flow, we will discuss here very efficient (polylog time) parallel randomized algorithms for bipartite matching, while we saw earlier that the best available (deterministic) parallel algorithm for network flow runs in $O(n^2 \log n)$ time. Even when randomization is allowed, no one knows a fast parallel algorithm for the general network flow problem.

We will discuss in detail matching under the *Monte Carlo* model, but also introduce the other common model, the *Las Vegas model*. The matching result is due to Mulmuley, Vazirani & Vazirani [1987]. In our discussion we will first assume that a perfect matching exists in the bipartite graph, discuss how to find such a perfect matching, and then discuss how to reduce the problem of constructing a maximum cardinality matching to the perfect matching problem. Along the way we will also see how to test if a graph has a perfect matching.

12.1. The Monte Carlo model

By randomization we mean that the algorithm (not the input) has some random component. Typically there is some point where the algorithm randomly generates a number according to some distribution, and then uses that number to direct its computation in some way. It may seem strange at first that certain problems can be efficiently 'solved' by a randomized algorithm, but not efficiently by any known deterministic algorithm. Parallel matching is one such problem.

To define a *Monte Carlo* algorithm we restrict attention to decision problems, i.e., problems which have either a 'yes' answer or a 'no' answer. A $T(n)$ time Monte Carlo algorithm is a randomized algorithm which outputs either 'yes' or 'no' and has the following three properties.

(1) On any input of size n, the algorithm halts in $O(T(n))$ time.

(2) For any input, if the algorithm answers 'yes', then 'yes' is definitely the correct answer.

(3) For any input, if the algorithm answers 'no', then 'no' is the correct answer with probability at least one-half.

Notice that the probability in the last statement is taken over the randomized executions of the algorithm, and *not* over the distribution of the input – the probability of one-half holds for *any* input. This is the novel and surprising aspect of randomizing the algorithm. Notice also that each time the algorithm is run, the probability of an incorrect 'no' answer is independent of all other executions of the algorithm so if we run the algorithm 100 times, say, and get a 'no' answer each time, then the probability that 'no' is incorrect is less than $1/2^{100}$.

A parallel algorithm which always terminates in $O(\log^k n)$ time, for some fixed k, is said to run in *polylog* parallel time. A Monte Carlo algorithm which runs in polylog time on a parallel machine (PRAM model) with at most a polynomial number of processors (as a function of n), is said to be in the complexity class RNC. An algorithm in class RNC is referred to as an RNC algorithm.

The Monte Carlo model has been defined for 'yes/no' problems, but its definition can be easily extended to optimization problems or construction problems. In that case the algorithm produces a proposed solution which is correct with probability at least one-half.

12.2. Self-reduction in sequential and parallel environments

A natural approach to designing a parallel algorithm is to divide the problem into *independent* pieces so that the solution to the original problem can be constructed quickly from information obtained about each piece. Independence of the pieces means that work on the pieces can be done in parallel. Of course, the key problem is to find such a nice division.

One idea for dividing up the matching problem was suggested by the early result that the following *decision question* can be solved, as we will see below, by an RNC algorithm.

Decision problem. For any fixed edge e, is e in some perfect matching in G?

Now in the sequential environment, the ability to answer this decision question can easily be used to *construct* a perfect matching as follows. Order the edges arbitrarily as $e_1, e_2, \ldots, e_m$. Test if e_1 is in *some* perfect matching of G, and if so modify G by deleting the endpoints of e_1 and all incident edges from G. Next test whether e_2 (if it has not been removed) is in some perfect matching in the new G. In general, for every edge e_k there is a current G, and if e_k is in a perfect matching in the current G, then G gets modified as above. When all edges have been removed, the set of edges which were found to be in some perfect matching of their associated G, form a perfect matching in the original G. This process of constructing a solution by repeatedly solving a decision question is called *self-reduction*.

Difficulties in the parallel environment

Although we will see that the above *decision* question for matching can be solved fast in parallel, the self-reduction used seems to be a very sequential process, so it is not clear how to use it to *construct* a perfect matching fast in parallel. A natural response to this difficulty is to ask all the decision questions in parallel, i.e., for each edge e determine whether there is a perfect matching in the *original* G containing e. These questions are independent and so can be solved in parallel. But now the set of edges which are in some perfect matching may not themselves form a matching. The problem is that the perfect matching may not be unique, and two adjacent edges in G may be in separate perfect matchings. However, the above method would work *if* there were only one perfect matching in G.

Given the problem of non-unique matchings, the next immediate idea is to find a way to distinguish the perfect matchings so that one of them is unique. An easy way of doing this is to introduce edge weights that make the *minimum weight* perfect matching unique. In particular, give edge e_k weight 2^k. Then the sum of the weights in any edge set is different from the sum for any other edge set. Hence there can be no ties for the minimum weight perfect matching. With this idea the appropriate decision question is the following.

Weighted decision question. Given a fixed edge e, is e in some minimum weight perfect matching?

We can ask this decision question about each edge e in parallel, because the questions are independent of each other. Further, since the minimum weight perfect matching is unique, the set of edges which are in some minimum weight perfect matching form the unique perfect matching. So *if* we could solve the above weighted form of the decision question fast in parallel we would have a good parallel method to construct a perfect matching.

The problem with the above idea is that we do not know how to solve the stated decision question fast in parallel *when the weights are so large*. However, the decision question can be solved fast in parallel (as we will see) when the weights are 'small' (polynomial in m). But then the minimum weight perfect matching is not always unique, and when it is not unique the 'yes' answers to all the questions asked in parallel do not specify a matching. So we seem to have taken one step forward and one step back.

Here is where the power of randomization comes in. We will show that the minimum weight perfect matching is unique *with high probability* if the weights are chosen *uniformly* over the interval 1 through $2m$. This is the basis for the following randomized matching method.

Randomized matching algorithm

(1) For each edge e choose a weight $w(e)$ uniformly from the interval 1 through $2m$.

(2) For each edge e test whether e is in some minimum weight perfect matching for the above weights. If yes, place e in set S.

We will see that S is a perfect matching (always assuming that one exists in G) with probability at least one-half. We will also see how to solve Step 2 fast in parallel.

If we want to exclude the possibility that S is not a perfect matching, then we should add an additional Step 3: check whether S is a perfect matching, and if not, return to Step 1. Since the weight assignments in each execution of Step 1 are independent and the probability that an execution gives a perfect matching is at least one-half, the *expected* number of iterations until S is a perfect matching is at most two. With Step 3, the algorithm becomes a very fast *expected* time parallel method.

We now begin the detailed investigation of these claims.

Lemma 12.1. *If each edge weight $w(e)$ is selected uniformly from the integers in the range 1 to $2m$, then the minimum weight perfect matching is unique with probability greater than one-half.*

Proof. There are $(2m)^m$ equally likely assignments of integers to the edges of G. We will estimate how many of them have more than one minimum weight perfect matching. Fix a particular edge e and then fix an assignment Q of weights for all edges *other than* e. Since there are $2m$ choices for $w(e)$, there are $2m$ assignments of weights to all the edges which agree with Q. Given Q, let $M(e)$ be the minimum weight of any perfect matching which excludes e, and let $M'(e)$ be the total weight of the edges other than e in any minimum weight perfect matching which contains e. $M(e)$ is defined to be some large finite number if there is no perfect matching excluding e, and similarly for $M'(e)$. If both perfect matchings exist, then the second one will have the same weight as the first only when $w(e)$ is exactly $M'(e) - M(e)$. This happens in at most one of the $2m$ assignments which agree with Q. So there is a minimum weight perfect matching containing e and a minimum weight perfect matching not containing e, in at most one of these $2m$ assignments. Letting Q vary over all possible assignments, we conclude there is both a minimum weight perfect matching containing e and one not containing e in at most $1/2m$ of the $(2m)^m$ weight assignments.

Now for any fixed assignment of edge weights, the minimum weight perfect matching is unique unless there is at least one 'witness' edge e where the minimum weight perfect matchings with and without e have equal weight. As shown above, any fixed edge e is the 'witness' of non-uniqueness in at most $1/2m$ of the weight assignments. Since there are only m edges, at most one-half of all the assignments have such a witness, and the lemma follows. □

Corollary 12.1. *For any graph G containing a perfect matching, the random matching algorithm constructs a perfect matching with probability at least one-half.*

Proof. When the minimum weight perfect matching is unique, the 'yes' answers to the weighted decision questions, asked in parallel, specify a perfect matching. With the edge weights selected as above, the minimum weight perfect matching is unique with probability at least one-half. □

Note that the lemma and corollary hold for non-bipartite graphs as well as bipartite graphs. However, the details of how to solve the decision question are more involved for the non-bipartite case, and for convenience we will only discuss the bipartite case.

12.3. How to solve the decision problem

We now begin to show how to solve the weighted decision question. Given a set of edge weights in the range 1 to $2m$, and assuming that G has a unique minimum weight perfect matching M, is e in M?

For any edge $e = (i, j)$, let $G(e) = (X, Y, E)$ be the graph obtained from G by deleting i and j and all incident edges. We assume that the nodes of X are renumbered consecutively from 1 to $|X|$, and similarly for the nodes of Y. Hence the correspondence of numbers to nodes in each $G(e)$ may be different from each other and from G. We let M be the unique minimum weight perfect matching in G, and let w denote its weight. We also assume for now that we know w (but of course not M); we will see how to compute w below.

The solution to the decision problem is based on the relationship between matchings in a bipartite graph and the determinant of a certain matrix obtained from it. The method will be applied both to G and to $G(e)$ for each e, but we will focus on $G(e)$; the case of G is simpler. We start with an idea that does not quite work and then add in the needed modification.

The simple matrix A

Let the two sides of a bipartite graph $G(e)$ be denoted by X and Y, where $|X| = |Y|$. Let A be the adjacency matrix of $G(e)$, i.e., $a_{i,j} = 1$ if $i \in X$, $j \in Y$ and $(i, j) \in G(e)$; $a_{i,j} = 0$ otherwise. Now the determinant of A is defined as

$$\sum_{\sigma} s(\sigma) \prod a_{i,\sigma(i)},$$

where σ is a permutation of the integers from 1 to $|X|$, and $s(\sigma)$ is a function which evaluates to either $+1$ or -1. Details of this function are not needed in this discussion.

It is useful to consider a particular permutation σ as describing a 'potential' perfect matching containing the potential edges $(i, \sigma(i))$ for $i \in X$, $\sigma(i) \in Y$. Since σ is a permutation, each node $i \in X$ and $\sigma(i) \in Y$ is incident with exactly one of the 'potential' edges described by σ. If $(i, \sigma(i))$ is an edge in $G(e)$ for each $i \in X$, then the potential matching is a perfect matching in $G(e)$, and $\prod a_{i,\sigma(i)} = 1$. If $(i, \sigma(i))$ is not an edge in $G(e)$ for some i, then $\prod a_{i,\sigma(i)} = 0$ and σ does not describe a perfect matching in $G(e)$.

Hence if the det A is not zero, then $G(e)$ must contain a perfect matching, although the converse does not hold. Each perfect matching contributes either a $+1$ or -1 to the determinant, and by cancellation the determinant could be zero even when $G(e)$ has a perfect matching. So with the current A, we cannot determine from det A whether $G(e)$ has a perfect matching. We will modify A so that this, and more, will be possible.

A modified matrix A

Matrix A is now defined as follows: set $a_{i,j} = 2^{w(i,j)}$ if (i, j) is an edge in $G(e)$, and $a_{i,j} = 0$ otherwise. Then we have the following.

Theorem 12.1. *If* det $A = 0$, *then e is not in M. If* det $A \neq 0$, *then* $e \in M$ *if and only if* $2^{w-w(e)}$ *is the largest power of two which evenly divides* det A.

Proof. If $e \in M$ then let σ be the permutation of the integers in X which describes the perfect matching $M - e$ in $G(e)$. Then in A, $\Pi a_{i,\sigma(i)} = 2^{w-w(e)}$.

Since M is the unique minimum weight perfect matching in G, the minimum weight perfect matching in $G(e)$ (if one exists) has weight at least $w - w(e)$. Further, it has exactly that weight only if $e \in M$. Therefore, any permutation $\sigma' \neq \sigma$ contributes either a zero term to det A [when σ' does not describe a perfect matching in $G(e)$], or contributes a term of $\pm 2^k$ with $k > w - w(e)$. Hence no subset of these other terms can sum to $\pm 2^{w-w(e)}$. It follows that if $e \in M$, then det $A \neq 0$, and so det $A = 0$ implies that $e \notin M$. The first statement of the theorem is proved.

For the second part, recall that all terms in det A are powers of two with absolute value larger or equal to $2^{w-w(e)}$. So $2^{w-w(e)}$ divides det A when it is not zero. Now $2^{w-w(e)+1}$ does not divid det A (assumed to be non-zero) if and only if $\pm 2^{w-w(e)}$ is a term in det A (this follows simply from the fact that det A is the sum of numbers which are powers of two). But the only permutation that can contribute $\pm 2^{w-w(e)}$ is σ, so $M - e$ must be a perfect matching in $G(e)$, so $w^{w-w(e)+1}$ does not divide det A if and only if $e \in M$. □

At this point we can also describe a randomized method for the decision question: is e in some perfect matching in G, assuming that G has a perfect matching? This is equivalent to asking if there is a perfect matching in $G(e)$, which is answered by the following immediate corollary of Theorem 12.1.

Corollary 12.2. *Assume that G has a perfect matching. If* det $A \neq 0$ *then there is a perfect matching in* $G(e)$. *If* det $A = 0$ *then the probability that* [det $A = 0$ *and* $G(e)$ *has a perfect matching*] *is at most one-half.*

How to compute w

Replace $G(e)$ with G in the above discussion and let A be defined for G. Since M is the unique minimum weight perfect matching in G, det A will not be zero, and $\pm 2^w$ will be the smallest term. Therefore, 2^w will be the largest power of two that divides det A. To obtain w, simply test in parallel powers of

two to find the largest which divides det A. The number of such independent tests is bounded by log(det A), which is polynomial in m, so only a polynomial number of processors are needed.

Summary of the method

The one detail that we cannot explain here is that the determinant of a symmetric n by n matrix can be computed fast in parallel, in fact by an NC^2 algorithm. An NC^2 algorithm is a deterministic parallel algorithm which uses a polynomial number of processors and runs in $O(\log^2 n)$ parallel time. With that assumption, it should be clear that the following algorithm can be implemented to run in parallel in $O(\log^2 n)$ time with just a polynomial (in m) number of processors.

(1) Select edge weights uniformly from 1 to $2m$.

(2) Form A from G and compute w, the weight of the (assumed unique) minimum weight perfect matching in G.

(3) For each edge e in G form A [conceptually from $G(e)$] and compute det A. If det $A = 0$ then place e into set S. If det $A \neq 0$ then find the largest power of two, z, which divides det A. If $z = 2^{w-w(e)}$ then place e in S.

We have shown that S is the perfect matching M under the assumption that G has a unique minimum weight perfect matching, and that this happens with probability at least one-half for the edge weights chosen. So we have an NC^2 method which is guaranteed to find a perfect matching with probability at least one-half, assuming there is a perfect matching in G. Note that this gives an RNC method to determine if a graph has a perfect matching (when we cannot assume that it does): run the above algorithm and examine S; if S is a perfect matching, then 'yes' is definitely the correct answer; if S is not a perfect matching, then 'no' is the correct answer with probability at least one-half.

The reader should consider at this point why we cannot make the above method deterministic? We mentioned earlier that if we set $w(e_k) = 2^k$, then the minimum weight perfect matching is definitely unique, so the only source of randomness in the above method would be eliminated. So why did we not use these weights? The problem is that $a_{i,j}$ would then be 2^{2^k} for $(i, j) = e_k$. These numbers would then not be representable in polynomial space as a function of n, and operations on them would require more than polynomial time in n. So such large numbers simply violate the basic model of what kinds of numbers are permitted. Because of the requirement to use numbers in the correct range, we needed to introduce randomization into the method.

12.4. The cardinality matching problem

We started with the problem of constructing in parallel a maximum cardinality matching, but have focussed above on the perfect matching problem. How do we use the latter to solve the former?

Suppose that the maximum cardinality matching in G is $n - k$, where each

side of G has n nodes. Let $G(k)$ be obtained from G by adding k nodes to each side of G and connecting each of them to all the n nodes on the other side. Then there is a perfect matching in $G(k)$. So the maximum cardinality matching in G can be obtained by finding the smallest k such that $G(k)$ has a perfect matching. We saw above how to test if $G(k)$ has a perfect matching so that the probability of an incorrect 'no' answer is at most one-half. Each iteration of this test is independent of the others, so if it gives a 'no' answer q times, the probability of error [$G(k)$ has a perfect matching but the algorithm did not find one] is less than $1/2^q$. So if we test each k in parallel say 100 times, and k' is the smallest value such that the test for $G(k')$ found a perfect matching, the probability that the maximum matching is greater than $n - k'$ is less than $1/2^{100}$. So we can use this method to quickly find a matching which, with very high probability, is a maximum cardinality matching.

If we run the Monte Carlo perfect matching algorithm just on a single $G(k)$, then we have a fast parallel Monte Carlo method for the decision question: is there a matching of size $n - k$ or more? When the method says 'yes' it is certainly correct, and when it says 'no' it is correct with probability at least one-half.

12.5. A Las Vegas extension

The method above has the property that when it says 'yes' there is a matching of size $n - k$ or more, it is certainly correct, but when it says 'no', it could be wrong with some small probability. If we could construct a symmetric algorithm that was surely right when it said 'no' and wrong with some small probability when it said 'yes', then we could dovetail the two computations to get a method that was already right. This approach leads to the idea of a Las Vegas algorithm.

A Las Vegas algorithm also has a random component, but compared to a Monte Carlo algorithm it is a more reliable algorithm, although not as certain to be fast. In particular, a randomized 'yes/no' algorithm is called a $T(n)$-time Las Vegas algorithm if it satisfies the following two properties:

(1) *If* the algorithm halts, then it definitely outputs the correct answer.

(2) For *any* input of size n, the *expected* running time of the algorithm is bounded by $O(T(n))$, where the expectation is taken over the random choices of the algorithm.

More generally, an optimization algorithm, such as one which finds a maximum cardinality matching, is called a $T(n)$-time Las Vegas algorithm if it is randomized and has the properties: (a) if it halts then it has the optimal solution; (b) the expected running time, averaged over the random choices of the algorithm, is bounded by $O(T(n))$.

In the case of bipartite graphs, we can very simply turn the Monte Carlo algorithm we have for finding the maximum cardinality matching into a Las Vegas algorithm for maximum matching.

Las Vegas bipartite matching

(0) Set $k' = 0$.

(1) Repeat the Monte Carlo decision algorithm once, in parallel, for each $G(k)$, $k \leq n - k'$, in parallel.

(2) In parallel, examine each output set of edges S, checking for a matching. If no matching is found, repeat Step 1. Else let M be the largest matching found.

(3) If the size of M is less than n, then test whether a larger matching is possible as follows:

(3a) Considering the matching problem as a flow problem as detailed at the start of Section 7, build the residual graph for the flow corresponding to matching M.

(3b) Search for an s to t directed path in the residual graph. In other words, test whether t can be reached from s via a directed path.

(3c) If there is no path, then M must be the maximum cardinality, so report with certainty that M is optimal and stop.

(3d) If there is a path, then a larger matching is possible. Set $k' = |M| + 1$ and return to Step 1.

The expected number of iterations of Step 1 before the maximum cardinality matching is found is two. When M is the maximum cardinality matching, Step 2 will determine that for sure and stop the algorithm. So the method is Las Vegas. Steps 2 and 3 can be implemented to run in polylog time with a polynomial number of processors (we leave the details as an exercise). Hence the method is a polylog parallel Las Vegas algorithm using only a polynomial number of processors to find a maximum cardinality matching in a bipartite graph. Notice that in this Las Vegas algorithm the only randomness comes in the phase where matchings are constructed. The test of optimality is deterministic.

For non-bipartite graphs Karlof [1986] extended the Monte Carlo matching algorithm to a polylog parallel Las Vegas algorithm using only a polynomial number of processors. He first developed a complementary RNC Monte Carlo algorithm to solve the decision question: does G have a perfect matching. Karloff's algorithm is complementary to the one we discussed in that when it says 'no' it is certainly correct, and when it says 'yes' it is correct with probability at least one-half. So if you alternate running the two Monte Carlo algorithms until either the first one says 'yes' or the second one says 'no', the resulting answer will certainly be correct. If the correct answer is 'yes', then the expected time before the first algorithm halts is within two iterations; similarly if the correct answer is 'no', then the expected time for the second algorithm to halt is within two iterations. The dovetailed algorithm therefore has expected running time of no more than two iterations. Hence it runs in expected polylog time with a polynomial number of processors, and if it halts it always gives the correct answer. The dovetailed algorithm is therefore a Las Vegas algorithm for testing for a perfect matching. The extension to the maximum cardinality case is left for the reader.

13. A matching problem from biology illustrating dynamic programming

Having discussed several variants of the matching problem on different machine models, we turn now to a very special variant that arises in biology. This problem will allow us to introduce and discuss an important algorithmic technique, namely *recursive programming* and a speedup of it known as *dynamic programming*.

The following is a simple version of a problem that arises in predicting the secondary folding structure of transfer RNA molecules. Let L be a string of n binary characters, i.e., each character is either 0 or 1. We define a *matching* as set of pairs of characters in L, each pair containing *exactly* one one and one zero, such that no character appears in more than one pair. We say that characters i and j of L are *matchable* if and only if exactly one of them is a one and the other is a zero.

We consider the string L to be arrayed around a circle and define a *nested matching* as a matching where each matched pair is connected by a line inside the circle such that no two lines cross each other. That is, for any positions $i < j$ in L, if the character in position i is matched to the character in position j, then no character below i or above j can be matched to a character between i and j. The problem is to find a nested matching of largest cardinality. It is the nesting property of the matching that makes this problem interesting; without that constraint, the problem is easily cast and solved as a maximum flow problem. We will solve the problem by *recursive programming* and show how that leads to a *dynamic programming* solution.

13.1. A recursive solution

Define $C(i, j)$ as the value of the optimal nested matching on the substring defined by characters in positions i through j of L. Clearly then, we seek the value $C(1, n)$. For the base case, $C(i, i+1) = 1$ if the two characters are different and $C(i, i+1) = 0$ if they are the same.

We approach the problem of computing $C(1, n)$ by thinking recursively, starting with the question: what are the possible matches for character 1? Either character 1 is not involved in a match, or it matches with some character $k \leq n$, where characters 1 and k are matchable. In the first case $C(1, n) = C(2, n)$. In the second case $C(1, n) = 1 + C(2, k-1) + C(k+1, n)$, where we define $C(p, q) = 0$ if $p \geq q$.

So $C(1, n) = \max[C(2, n), \max_{k \leq n}[1 + C(2, k-1) + C(k+1, n)]$: i and k are matchable].

Of course, this leaves the question of what $C(2, n)$ is and what $C(k+1, n)$ is for each $k \leq n$. Still, with only this small effort and little notation, we could actually program a computer to compute $C(1, n)$, provided that we use a programming language that allows recursion, i.e., that allows a function with parameters to be defined in terms of itself. For example, we can write the following recursive 'program' to compute $C(i, j)$ for $i < j$.

```
If i = j Then C(i, j):= 0 Else
  Begin
  If i = j - 1 Then
    Begin
    If characters i and j are different
    Then C(i, j):= 1
    Else C(i, j):= 0
    End
  Else
    Begin
    K:= C(i + 1, j)
    For k between i + 1 and j - 1
      Begin
      If characters i and k are matchable
      Then K:= max[K, C(i + 1, k - 1) + C(k + 1, j) + 1]
      End
    End
  C(i, j) = K
  End
```

13.2. Defects of the recursive solution

Then we can start the recursive program by calling $C(1, n)$, sit back and wait for the output. The computer will make all the subsequent required recursive calls, stacking all pending calls, until the base cases are reached. Then it climbs back up the tree of pending calls filling in the computed values, etc., until $C(1, n)$ is learned. This recursive program is easy to think up and program, and it leaves all the drudge work to the computer (a good use for a computer), but it is not satisfactory for large n because of the time it takes to execute. The call to $C(1, n)$ makes about $2n$ calls to compute other values of C, and in general $C(i, j)$ makes about $2(j - 1)$ calls to other C values. So the number of calls in total when computing $C(1, n)$ in this way is $\Omega(n!)$, a very unsatisfactory growth rate.

The reason for this large growth is that any particular $C(i, j)$ can be called a very large number of times in the above recursive program. And yet, there are only $O(n^2)$ distinct choices for the pair i and j. This suggests that if we can avoid duplicate calls to the function with the same choice of parameters, then we can speed up the solution. Indeed, it is possible to avoid all duplicate calls in a *top–down* manner, i.e., using the above top–down recursive method that starts by calling $C(1, n)$. This leads to a solution that runs in $O(n^3)$ time for the RNA folding problem. Further, it is possible to write a general recursion implementing system which will automatically avoid duplicate calls, and hence find by itself an efficient implementation of the otherwise inefficient recursive program. However, this approach is not usually taken. Instead, a different, slightly more efficient but less automatic method is often used to overcome the problem of duplicate calls. That method is called *dynamic programming*.

13.3. Dynamic programming

In the context of the RNA folding problem, dynamic programming is simply the above recursive solution to computing $C(1, n)$, but with the modification that the $C(i, j)$ values are computed *bottom–up* rather than *top–down*. That is, we start by computing $C(i, i+1)$ for each i, and continue computing $C(i, j)$ in order of increasing difference $j - i$. In this way, once $C(i, j)$ has been computed and stored for each pair i, j where $j - i < t$, then we have all the needed values to compute $C(i, j)$ for $j - i = t$, using only $O(j - 1)$ table look-ups per pair. Hence $C(i, j)$ can be computed for all pairs i, j in $O(n^3)$ total time.

The approach to RNA folding is typical of problems solved by dynamic programming. The problems often have a fairly direct recursive solution, but direct recursive implementation of the solution solves the problem top–down and hence duplicates calls to subproblems. However, if the structure of the subproblems is regular enough so that the subproblems can be nicely ordered, then the subproblems can be solved and their values tabulated in a bottom–up manner. This bottom–up tabling of solutions to subproblems is called dynamic programming. In this view, dynamic programming is a way to efficiently *implement* recursive programming, and it works when the structure of the subproblems is sufficiently well behaved. The reader should be aware however that this view of dynamic programming as an implementation technique, reducing its importance as a conceptual technique, is somewhat non-standard.

14. Min-cost flow: Strong versus weak polynomial time

In this section we will briefly discuss a generalization of the network flow problem called the *min-cost flow* problem. As in the case of the original network flow problem, we will first develop a finite time algorithm which is not necessarily polynomially bounded. Then we will discuss some recent developments showing how this algorithm can be converted into a (strongly) polynomial time solution. That solution will be related to the first, merely finite, solution in a way that is very analogous to the relationship of the Edmonds–Karp network flow algorithm to the Ford–Fulkerson algorithm. Finally, we will discuss an earlier, yet still valuable, approach to the problem which first lead to a (weakly) polynomial time solution. This method, also developed by Edmonds & Karp [1972], is called *scaling*.

In an instance of the min-cost flow problem each directed edge (i, j) has an associated cost $w(i, j)$ as well as the normal capacity $c(i, j)$. The *cost* of a flow f from source s to sink t is

$$\sum_{(i,j)} f(i, j) \times w(i, j) .$$

The min-cost flow problem is to find an s, t flow of maximum value which has

minimum cost among all maximum value s, t flows. The problem was shown in 1972 to be solvable in polynomial time using scaling by Edmonds & Karp [1972], but the result is not totally satisfying to some because it is not strongly polynomial.

An algorithm is considered *strongly polynomial* if the *number* of primitive operations of the algorithm is bounded by a polynomial function of the *number of input elements* alone, always with the implicit assumption that the time of each operation is bounded by a polynomial in the size of the input. That is, if some of the input elements consist of numbers, then the polynomial bound on the number of operations is independent of how large the numbers are. In the case of the maximum flow problem, the EK, Dinits, GT and wave algorithms are all strongly polynomial – in each case the worst-case number of primitive operations (additions, subtractions, comparisons, data movements) is a function of the number of nodes and edges of the graph, and not of the size of the edge capacities. Of course, the time to carry out a given operation will be affected by the size of the numbers, but not the number of such operations.

For contrast, recall that an algorithm is considered polynomial if the worst case number of operations is bounded by a polynomial in the *total size* of the input, i.e., the number of bits needed to represent the input. In this model, the number of arithmetic operations need not be related to the number of input elements, just to their total size.

In the first polynomial method for min-cost flow, the scaling method [Edmonds & Karp, 1972], the number of primitive operations is $O(n^4 \log U)$, where U is the largest edge capacity in the graph. This bound is a polynomial function of the size of the input, because the input takes at least $n + m + \log U$ bits.

Now there is a school of thinking which might be called the 'bit-is-a-bit-is-a-bit' school, which considers the polynomial bound of $O(n^4 \log U)$ to be every bit as good as a bound which does not contain U, but many people find themselves in the opposite camp. For them, a strongly polynomial time bound is superior, and so the question of whether the min-cost flow problem could be solved in strongly polynomial time was an attractive open problem for some time. Indeed, the same question is still open for linear programming – polynomial time algorithms are known, but not strongly polynomial ones. The min-cost flow question was solved by Tardos [1985]. We will not describe her solution, but will briefly describe a different strongly polynomial solution. But first we develop a finite method for the problem.

14.1. *Finite time solution for min-cost circulation*

In discussing min-cost flow algorithms it is convenient to first generalize the problem to the min-cost *circulation* problem. A *circulation* is a generalization of a flow where all the conditions for flow must apply, but additionally the inflow must equal the outflow at *all* nodes, i.e., including nodes s and t. The min-cost circulation problem is to find a circulation f of minimum total cost $\sum_{(i,j)} f(i, j)w(i, j)$.

Now the min-cost circulation problem is a true generalization of the min-cost flow problem since not all min-cost circulations will be maximum s, t flows, but a min-cost maximum s, t flow can be achieved as a min-cost circulation as follows: add an edge from t to s with infinite capacity and cost $M < 0$, where $|M|$ is larger than $\Sigma_{(i,j)\neq(t,s)} w(i, j)$. A min-cost circulation in this network will consist of a min-cost (maximum) s, t flow of some value, say v, and a flow along edge t, s of v units. So the min-cost circulation problem generalizes the min-cost flow problem.

An early finite method [Röck, 1980] for solving the min-cost circulation problem is the following.

Cycle augmentation algorithm

(1) Find a circulation f (possibly all zeros) in G. Let G^f be the residual graph.

(2) For every forward edge (i, j) in G^f associate the weight $w(i, j)$, and for every backward edge (i, j) in G^f associate the weight $-w(j, i)$.

(3) Search for a negative weight cycle C in G^f. If there are none, then stop; f is a min-cost circulation.

(4) If C exists, then let u be the minimum residual capacity of any edge in C. For every forward edge (i, j) in C set $f(i, j)$ to $f(i, j) + u$. For every backward edge (i, j) in C set $f(j, i)$ to $f(j, i) - u$. The new f is still a circulation, but it has less cost than the previous f.

(5) Go to Step 2.

It is not difficult to prove that this algorithm is correct, and assuming that all edge costs are rational, that it terminates in finite time. Searching for a negative weight cycle is also not a difficult task; it can be done by a modification of the dynamic programming based shortest path algorithms that allow negative as well as positive distances.

Analogies to Ford–Fulkerson

To see how the above algorithm is analogous to the Ford–Fulkerson maximum flow method, consider how to use it just to find a maximum flow. One simple way is to set the cost of edge (t, s) to -1 and the cost of all other edges to zero. Then if we start with the zero circulation, each negative weight cycle in the residual graph is actually an s, t path in the residual graph, followed by the t, s edge. The Ford–Fulkerson maximum flow algorithm allows *any* (s, t) residual path to be used, and the above algorithm allows *any* negative weight cycle to be used. Further, as in the Ford–Fulkerson method, the number of iterations of the algorithm, although finite, may be exponential in terms of n and m alone.

The final, but very recent, analogy is that the number of iterations in the above circulation method can be made polynomial (in fact strongly polynomial) by a rule for choosing negative weight cycles which generalizes the shortest augmenting path rule of the Edmonds–Karp method. The rule is to

use the negative weight cycle of smallest *mean* weight. That is, if cycle C has total weight $w(C) < 0$ and has k edges, its mean weight is $w(C)/k$. There are strongly polynomial methods for finding the smallest mean weight cycle [Karp, 1978], so this gives a strongly polynomial method for min-cost flow. This approach was discovered by Goldberg & Tarjan [1989]. It is not the fastest presently known solution to min-cost flow, but it is perhaps the most satisfying in the way it ties together the history of the maximum flow and the minimum cost flow problems.

The minimum cycle-mean method is the proper generalization of the EK maximum flow method which augments along shortest s, t paths in the residual graph. Put a cost of zero on all original edges and a cost of -1 on the (t, s) edge; then the minimum cost circulation gives a maximum s, t flow, and the minimum cycle-mean rule specializes to the shortest s, t path rule of the EK algorithm.

14.2. An earlier polynomial method based on scaling

Despite the analogies developed above between the history of the maximum flow and the min-cost flow problems, there is one major difference between the two stories. The first polynomial time methods for maximum flow were also strongly polynomial, while the first polynomial time method for min-cost flow [Edmonds & Karp, 1972] was not strongly polynomial. A strongly polynomial time method did not appear for more than ten years after that [Tardos, 1985]. The first polynomial min-cost flow method introduced a technique called *scaling* which has continued to be an important technique in its own right. For that reason we will now examine a polynomial (but not strongly polynomial) time scaling method for the min-cost flow problem. In particular, we will discuss the capacity scaling method of Röck [1980] modified by Andrew Goldberg.

The idea of the method is to solve the min-cost flow problem on successively closer approximations of the original capacities. Let U be the largest edge capacity and let $J = \lfloor \log_2 U \rfloor + 1$. J is the number of bits used to represent U in binary. The method consists of J iterations. The purpose of iteration k is to compute the min-cost flow where the approximate edge capacity of any edge is the number created by the leftmost k bits of its original capacity. So for example, at the end of the first iteration any original edge capacity equal to or greater than 2^J will be approximated by 1, and all other approximate edge capacities will be zero.

At the end of iteration k, the circulation will be a min-cost circulation for capacities given by the leftmost k bits of the true capacities. At the start of iteration $k + 1$ the current flow assignment and edge capacity of every edge is doubled. Note that the circulation is still min-cost for these new capacities. At this point the edge capacity of any edge is at most one less than the number given by the leftmost $k + 1$ bits of its true capacity. An edge will be called *deficient* if its capacity is less than its $k + 1$ bit capacity. One by one we will

increase the edge capacity of each deficient edge by one unit, and update the circulation to be a min-cost circulation. Iteration $k+1$ ends when there are no remaining deficient edges.

What remains is to explain how to efficiently update the min-cost circulation. Suppose the capacity of edge $e=(i, j)$ is increased by one. If edge (i, j) is already in the residual graph or is new but no negative cycles get created, then the current circulation is optimal. If edge (i, j) is new and its addition to the residual graph creates a negative cycle, then we find the *most negative* cycle C and augment around it by exactly one unit, the residual capacity of edge (i, j). We will show that the resulting circulation is min-cost for the current capacities. Note first that C must contain (i, j) and therefore C is obtained from the shortest path from j to i [excluding edge (j, i)] in the current residual graph. To find that path we delete edges (i, j) and (j, i) from the residual graph; the resulting graph has no negative cycles (although it has negative edges), so dynamic programming methods such as Floyd's method can find the most negative path P from j to i in $O(n^3)$ time.

Lemma 14.1. *After augmenting around the most negative cycle C by one unit, the resulting residual graph contains no negative cycles.*

Proof. Before increasing the capacity of (i, j) the circulation is min-cost, so any negative cycle created (when the capacity of (i, j) is increased) must contain edge (i, j). Suppose that the augmentation around C creates another negative cycle C' in the resulting residual graph. C' must contain at least one edge which was not in the residual graph before C was augmented. Let X be the set of such new edges in C' and note that each of these must be in C but in the opposite direction then they are in C'.

Suppose edge (p, q) is the first edge on the j to i path P such that its reverse [edge (q, p)] is in X. Then consider the j to i path P' formed by taking P until p followed by path C' until q and then P until j. Now both C and C' have negative total weight, and edge (p, q) has a weight which is the negative of the weight of (q, p), so the cycle P' plus edge (i, j) is more negative than C. Note that the cycle P' plus edge (i, j) has fewer edges in X than does C'. Iterating this argument until there are no edges from X on the cycle, we obtain a cycle that only contains edges of the residual graph before C was augmented, and which is more negative than C – a contradiction. □

There are m edges, so every iteration takes $O(mn^3)$ operations, and since there are only $\log U$ iterations, the method uses $O(mn^3 \log U)$ operations, and with a closer analysis the bound can be reduced to $O(n^4 \log U)$. A polynomial but not strongly polynomial time bound.

14.3. Edge cost scaling

The idea of the method is to solve the min-cost flow problem on successively closer approximations of the original costs. Let Cmax be the largest edge cost

and let $J = \lfloor \log_2 \text{Cmax} \rfloor$. J is the number of bits used to represent Cmax in binary. The method consists of J iterations. The purpose of iteration k is to compute the min-cost flow where the approximate edge cost of any edge is the number formed from the leftmost k bits of its original cost. So for example, at the end of the first iteration any original edge cost equal to or greater than 2^J will be approximated by 1, and all other approximate edge costs will be zero. In iteration $k+1$ the approximate edge costs are updated by doubling the approximate edge costs of iteration k and adding one to the cost of any edge whose original cost has a one in bit $k+1$. Then the min-cost flow from iteration k is used as a starting point to efficiently find a min-cost flow for the $k+1$ bit costs. It is not obvious and we will not go into details, but the $(k+1)$st min-cost flow can be found from the previous one using at most n network flow calculations (these do not involve costs) each on a graph that is efficiently derived from the original graph and the most recent flow. So every iteration takes $O(n^4)$ operations, and since there are only log(Cmax) iterations, the method uses $O(n^4 \log \text{Cmax})$ operations. Another polynomial but not strongly polynomial bound.

15. Weighted node cover: Approximation algorithms based on network flow

In this section we consider a common approach that is taken to problems which are known to be NP-hard. We will illustrate the approach with the node cover problem, and a polynomial-time approximation algorithm for it based on network flow.

15.1. The node cover problem

Let G be an undirected graph with each node i given weight $w(i) > 0$. A set of nodes S is a *node cover* of G if every edge of G is incident to at least one node of S. The *weight* of a node cover S is the summation of the weights, denoted $w(S)$, of the nodes in S; the weighted node cover problem is to select a node cover with minimum weight.

The node cover problem (even when all weights are one) is known to be NP-hard, and hence we do not expect to find a polynomial-time (in terms of worst-case) algorithm that is always correct. Therefore, we relax somewhat the insistence that the method be both correct and efficient for all problem instances. There are many types of relaxations that have been developed for NP-hard problems. The most common is the constant-factor, polynomial-time approximation algorithm.

For a graph G with node weights, let $S^*(G)$ denote the minimum weight node cover. Let A be a polynomial time algorithm that always finds a node cover, but one that is not necessarily minimum; let $S(G)$ denote the node cover of G that A finds. Then A is called a *constant-error polynomial-time approximation algorithm* (or approximation algorithm for short) if for any graph G, $S(G)/S^*(G) \leq c$ for some fixed constant c.

For the node cover problem we will give an approximation algorithm, based on network flow, with $c = 2$. First, we observe that the node cover problem has a nice solution when the graph G is bipartite.

For a bipartite graph $G = (N, N', E)$, connect all nodes in N to a new node s, and connect all nodes in N' to a new node t. For every node $i \in N$, set the capacity of edge (s, i) to $w(i)$, and for $i \in N'$, set the capacity of edge (i, t) to $w(i)$. Set the capacity of all original edges in E to be infinity. Call the new graph $\hat{G}$.

Theorem 15.1. *A minimum s, t cut in $\hat{G}$ defines a minimum node cover of G.*

Proof. Let C be a minimum s, t cut in $\hat{G}$. To get a node cover $S^*(G)$ of G we use the rule that if $i \in N$ and edge (s, i) is cut by C or if $i \in N'$ and edge (i, t) is cut by C, then i is in $S^*(G)$. Set $S^*(G)$ is clearly a node cover of G, for if there is an edge (u, v) with neither u or v in $S^*(G)$, then the path s, u, v, t is not cut by C in $\hat{G}$. To see that $S^*(G)$ is a minimum node cover, note that any node cover S' of G defines an s, t cut C' of $\hat{G}$ of equal cost: if $i \in N \cap S'$, then $(s, i) \in C'$, and if $i \in N' \cap S'$, then $(i, t) \in C'$. Since C is a minimum s, t cut, its cost is less than or equal to the cost of any minimum node cover of G, so the node cover $S^*(G)$ is a minimum node cover of $\hat{G}$. □

15.2. The approximation algorithm for general graphs

Given G, create bipartite graph $B = (N, N', E)$ as follows: for each node i in G, create two nodes i and i', placing i on the N side, and i' on the N' side of B; give both of these nodes the weight $w(i)$ of the original node i in G. If (i, j) is an edge in G, create an edge in B from i to j' and one from j to i'. Now find a minium cost node cover $S^*(B)$ of graph B. From $S^*(B)$, create a node cover $S(G)$ in G as follows: for any node i in G, if either i or i' is in $S^*(B)$, then put i in $S(G)$.

It is easy to find examples where $S(G)$ is not a minimum node cover of G, and where $S(G)/S^*(G) = 2$ for any G and any choice of node weights. However, no worse error ever happens.

Theorem 15.2. $S(G)/S^*(G) \leq 2$ *for any G and any choice of node weights for G.*

Proof. If node i is in $S^*(G)$, then put both nodes i and i' of B into a set Q. It is easy to see that Q is a node cover of B, and that its costs is twice that of $S^*(G)$. Hence the minimum node cover of B has cost at most twice that of $S^*(G)$. Now let $S^*(B)$ denote the minimum node cover of B, and for every node i in G, if i or i' of B is in $S^*(B)$, then put node i of G in a set S. It is easy to verify that S is a node cover of G, and its cost is at most the cost of $S^*(B)$. Hence S has cost at most $2S^*(G)$. □

Now the time for this approximation algorithm is $O(n^3)$, the time to find the minimum cut in B. In addition to the above method, there are approximation methods for the node cover problem which also achieve a factor-two approximation and which are not based on network flow. Some of these run in $O(n)$ time [Bar-Yehuda & Even, 1981; Gusfield & Pitt, 1986].

15.3. A small (worst-case) improvement in the approximation

We now examine a modification of the heuristic that improves slightly the worst-case approximation bound, but might be much more effective in practice. First observe that we do not need to double the occurrence of each edge of G. That is, for each edge (i, j) of G, put into B either the edge (i, j') or the edge (j, i'), but not both. It should be clear that with this sparser graph B, a node cover of B still defines a node cover $S(G)$ of G such that $S(G)/S^*(G) \leq 2$. This seems intuitively better than the original construction because the number of edges has been reduced by half, so intuitively the node cover of B should be smaller. However, every node of G still 'appears' twice in B. This doubling can be reduced with the following rule.

First, pick any edge (i, j) in G and place i (but not i') in B. Then for every neighbor k of i in G (including j), place k' (but not k) in B, and put the edge (i, k') into B. Similarly, for every neighbor k (including i) of j in G, put k (but not k') in B, and put the edge (k, j') into B. Then neither i' nor j appears in B, and each neighbor of i or j in G appears only once in B. After putting in these nodes and edges, add all nodes which are not neighbors of i or j, and for each edge (u, v) of G which does not yet appear in B, place either edge (u, v') or (u', v) into B.

Theorem 15.3. *Letting $S(G)$ be the node cover of G defined by the node cover $S^*(B)$, and assuming all nodes have positive weight, $S(G) < 2S^*(G)$.*

Proof. First, any node cover of B defines a node cover of G of no greater cost, since every edge in G is in B and its end points correspond to its correct endpoints in G. Hence the cost of $S(G)$ is no more than $S^*(B)$. Now suppose that edge (i, j) in G is the one picked by the modified method, so that neither i' nor j are in B. Consider any optimal node cover $S^*(G)$ of G. It certainly must include either i or j, since (i, j) is in G. Now create a node cover S' of B from $S^*(G)$ by taking into S' any node k if k is in B and $S^*(G)$, and any node k' if k' is in B and $S^*(G)$. Hence some nodes in $S^*(G)$ may be taken twice into S', but neither of i or j can be taken twice. Since one of i or j must be in $S^*(G)$, at least one of its nodes is taken only once into S', and hence S' cannot be as much as double the cost of $S^*(G)$. Therefore $S(G) \leq S^*(B) \leq S' < 2S^*(G)$. In fact, $S(G) \leq 2S^*(G) - \min[w(i), w(j)]$. □

The best edge to pick (to get the best guaranteed approximation result) is the edge (i, j) in G where $\min[w(i), w(j)]$ is maximized. The heuristic can be

improved by iterating, as long as every edge of G ends up in B at least once. It is an interesting open question how nodes and edges should be picked in order to optimize the effectiveness of this heuristic.

16. Summary and thesis

In this chapter we have discussed a large range of current and historical issues in algorithm design and analysis by focusing on network flow and related problems. In this way we have looked at many computational models, algorithm design and analysis techniques, and general design paradigms. Perhaps more important, we have seen many different types of questions that have been addressed by research in algorithm design and analysis. The field is not just concerned with worst- (or even average-) case running times on algorithms that are always correct, deterministic, run on sequential machines, and assume that all data appear at the start of the algorithm and disappear at its end. It is difficult to set out a taxonomy of all the types of questions, results, computational models and techniques that were discussed, but it is instructive to try, and also to try to set out some conclusions.

16.1. Models

We summarize the broad computational models that were discussed in this chapter. We first used the most familiar model, the random access machine (RAM) model, when we examined the Ford–Fulkerson algorithm, the Edmonds–Karp algorithm, the Dinits and wave algorithm and the Goldberg–Tarjan algorithm in Sections 2, 3, 3.3, 4. Parallel random access machines (PRAMs) were introduced when discussing parallel implementation of the Goldberg–Tarjan algorithm in Section 8 and again in Section 12 where we discussed randomized matching on a parallel machine. In Section 8 we distinguished between concurrent read concurrent write (CRCW) programs and exclusive read exclusive write (EREW) programs. We also introduced there the notion of parallel work. The notion of a distributed computation, in both the synchronous and asynchronous case, was introduced in Section 9 on shortest path communication problems. Randomized algorithms and the class RNC were introduced in Section 12, where randomized methods for finding maximum cardinality matchings were discussed. In that section we looked both at Monte Carlo and Las Vegas randomized models. In Section 14 on minimum cost flow, we introduced strong versus weak polynomial time, addressing the issue of how input should be represented. Since running times of algorithms are expressed as a function of the input size, the question of input size is central to what running times are established.

16.2. Questions

The standard question asked is about the worst-case running time of a sequential algorithm that must always be correct, and where no two instances of the problem can be assumed to be related. This was the assumption during most of the chapter, e.g., in the discussions on maximum flow algorithms on sequential machines. However, in Sections 5, 10 and 11 we discussed parameterized algorithms, many-for-one results, and a method based on preprocessing, where a sequence of problems must be solved, and where the times obtained were significantly better than by solving each instance from scratch. In Section 12 on randomized algorithms we dropped the insistence that the algorithm and its analysis be deterministic, and that it always be right. Further, we distinguished between Monte Carlo randomization where the algorithm must be fast but it can be wrong in certain ways with certain probabilities, and Las Vegas randomization where the algorithm must always be right, but it must only be fast in expectation. In Section 15 we considered approximation algorithms, where the method must be fast in worst-case, but can make errors as long as the maximum deviation from the optimal is bounded by a constant ratio. We distinguished between strong and weak polynomial time and algorithms, where the central issue is whether the algorithm runs in polynomial time as a function of the number of objects in the input, or just in the total size of the input.

16.3. Design and analysis paradigms and techniques

It is sometimes difficult to separate design techniques from analysis techniques since the analysis often guides the design, and the design often suggests the analysis. Still we can identify some broad paradigms and specific techniques.

The most classic design paradigm is to break a problem into smaller pieces or units that one can better understand and effectively solve than the whole original problem. These units must have the property that a solution to the original problem can be obtained either by iterating through a series of solutions to these small problems, or by taking the solutions to a set of smaller problems and using these solutions to construct a solution to the original problem. The former case is illustrated many times in this chapter. The latter case is usually called 'divide and conquer', and almost none of the algorithms in this chapter fall into that description. We will return to this point below.

The paradigm of solving a problem by solving a series of smaller more manageable units was first seen in the Ford–Fulkerson algorithm which iteratively solves the smaller problem: can a given flow be augmented, and if so, how? The problem of computing a flow was understood by Ford–Fulkerson in terms of the repetitive solving of that more limited augmentation problem. The Edmonds–Karp algorithm maintains that essential structure. Dinits expan-

ded the basic unit to a phase, in which a maximal flow is computed on a layered graph, but still the algorithm and analysis is understood as the solving of a sequence of phases. Many other examples of this paradigm are contained in the chapter. Breaking down a problem and analysis into manageable pieces is clearly the most important paradigm in tackling a hard problem, but it might only be a starting point. More will be said on this later.

Another broad design paradigm illustrated in this chapter was that of exploiting special structure of properties of the problem to obtain faster algorithms. This was certainly important in the discussion on parametric flow, bipartite matching, computing sets of minimum cuts with ancestor trees, optimal greedy matching with box inequalities, and edge connectivity.

More technically and narrowly we saw many identifiable design and analysis techniques. We saw *breadth-first search* in the FF algorithm; the use of *max-min duality* to direct an algorithm and prove its correctness; algorithm termination proofs by appeal to *finite progress*; the idea of *amortized analysis* in many examples; *greedy algorithms* for matching; more general than greedy algorithms, we saw *hill climbing* or *local improvement* methods such as in the FF algorithm or *scaling* methods for minimum cost flow, where the methods move from one flow to the next in a greedy manner; algorithms that use *preprocessing*; *self-reduction* in the randomized matching method; and *recursive programming* which led to *dynamic programming*.

16.4. A thesis: Combine and prosper versus divide and conquer

In contrast to the idea of breaking problems and analyses down into small easily solved concatenable units, is the technique of grouping or combining units together and solving them more efficiently as a whole. This design technique is often accompanied and encouraged by the very important analysis technique of amortization.

Dinits saw that the work done during a series of certain augmentations in the FF method could be grouped together into a larger unit, a phase, and computed more efficiently as maximal flow in a layered graph. Hence the basic unit of the FF algorithm (a single augmentation) was expanded and optimized. By considering a larger unit, the time for a set of augmentation operations was amortized over all the operations, and hence improved. The wave algorithm then sped up the work per phase keeping the phase as the basic unit. The Goldberg–Tarjan algorithm took the next step, breaking down phases entirely and amortizing the work over the entire flow computation. This allowed further amortization over a sequence of flows in the parametric setting, with the resulting faster time bounds compared to solving each problem from scratch. There the basic unit of a single flow was broken down and all the flows analyzed together. Along similar lines, the efficiency of the connectivity algorithm presented in Section 6 was established by amortizing over all the $n - 1$ specific flow computations. Taking the worst-case of single flow, and then multiplying by the number of flows would have given much inferior time

bounds for both the parametric and connectivity problems. Computation of the ancestor tree in Section 10 illustrates the same point. The output of that computation could be gotten by independent flow computations. But instead of breaking down the problem into its $\binom{n}{2}$ natural pieces, Cheng and Hu addressed how to compute the entire constellation of flows, with the resulting speed up.

We see this same moral in dynamic programming compared to recursive programming or divide and conquer. Recursive programming and divide and conquer are *top–down* techniques. In the divide and conquer paradigm one can design an algorithm by just describing how to efficiently divide the problem into smaller pieces and how to efficiently combine the solutions to these smaller pieces to obtain the solution to the original problem. Following this paradigm, the algorithm designer is allowed to assume that the solutions to the smaller pieces will be obtained 'by recursion'. Dynamic programming can be understood as recursive programming with an implementation trick that works in highly structured settings. The trick is that of tabulating and reusing subresults, or evaluating the recursion tree bottom up. The trick is needed, because the top–down dividing method is often not efficient. But in comparison to the divide and conquer paradigm, the dynamic programming trick requires analyzing, understanding and exploiting the interrelationships between the recursively called subproblems. It requires an examination of the entire sequence of computations and recursive calls that a top–down algorithm would invoke, and reorganizing those computations to avoid redundant work.

The idea of breaking down a problem and analysis into easily solved units is an important one, and it may be the most effective way to make early progress. But the examples in this chapter, and the history generally of flow and flow related algorithms suggest that as a problem is better understood, further improvements are made by fuzzing or enlarging the boundaries of the basic unit, or combining units into larger ones, or considering the computation over the whole set of units. In other words, combine and prosper.

Acknowledgement

I would like to thank Jan Karel Lenstra for his patience, and David Shmoys for reading an early draft and making many helpful suggestions.

References

Adelson-Velski, G.M., E.A. Dinits, A.V. Karzanov (1975). *Flow algorithms*, Science, Moscow.

Ahuja, R.K., T.L. Magnanti, J.B. Orlin (1989). Network flows, in: G.L. Nemhauser, A.H.G. Rinnooy Kan, M.J. Todd (eds.), *Handbooks in Operations Research and Management Science, Vol. 1: Optimization*, North-Holland, Amsterdam, pp. 211–369.

Bar-Yehuda, R., S. Even (1981). A linear time approximation algorithm for the weighted vertex cover problem. *J. Algorithms* 2, 198–203.

Blum, A., T. Jiang, M. Li, J. Tromp, M. Yannakakis (1991). Linear approximation of shortest superstrings, *Proceedings of the twenty-third annual ACM symposium on theory of computing*, pp. 328–336.

Cheng, C.K., T.C. Hu (1989). Ancestor tree for arbitrary multiterminal cut functions, Technical Report CS88-148, Department of Computer Science, University of California, San Diego, CA.

Cheriyan, J., S.N. Maheshwari (1989). Analysis of preflow push algorithms for maximum network flow. *SIAM J. Comput.* 18, 1057–1086.

Cunningham, W.H. (1985). Optimal attack and reinforcement of a network. *J. ACM* 32, 549–561.

Dinits, E.A. (1970). Algorithm for solution of a problem of maximum flow in networks with power estimations. *Soviet Math. Dokl.* 11, 1277–1280.

Edmonds, J., R.M. Karp (1972). Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM* 19, 242–264.

Even, S., R.E. Tarjan (1975). Network flow and testing graph connectivity. *SIAM J. Comput.* 4, 507–508.

Ford, L.R., D.R. Fulkerson (1962). *Flows in Networks*, Princeton University Press, Princeton, NJ.

Gallo, G., M. Grigoriadis, R.E. Tarjan (1989). A fast parametric network flow algorithm. *SIAM J. Comput.* 18, 30–55.

Goldberg, A. (1987). Efficient graph algorithms for sequential and parallel computers, Ph.D. thesis, M.I.T.

Goldberg, A.V., E. Tardos, R.E. Tarjan (1990). Network flow algorithms, in: B. Korte, L. Lovász, H.J. Prömel, A. Schrijver (eds.), *Paths, Flows and VLSI-Design*, Springer, Berlin, pp. 101–164.

Goldberg, A., R.E. Tarjan (1988). A new approach to the maximum flow problem. *J. ACM* 35, 136–146.

Goldberg, A., R.E. Tarjan (1989). Finding minimum cost circulations by cancelling negative cycles. *J. ACM* 36, 873–886.

Gomory, R.E., T.C. Hu (1961). Multi-terminal network flows. *SIAM J. Appl. Math.* 9, 551–570.

Gusfield, D. (1988). A graph theoretic approach to data security. *SIAM J. Comput.* 17, 552–571.

Gusfield, D. (1990a). Faster detection of compromised cells in a 2-d table, *Proceedings of the 1990 IEEE computer society symposium on security and privacy*, pp. 86–94.

Gusfield, D. (1990b). A faster parametric minimum cut algorithm, Technical Report CSE-90-11, Computer Science Division, University of California, Davis, CA.

Gusfield, D. (1990c). Very simple methods for all pairs network flow analysis. *SIAM J. Comput.* 19, 143–155.

Gusfield, D. (1991). Computing the strength of a graph. *SIAM J. Comput.* 20, 639–654.

Gusfield, D., G. Landau, B. Schieber (1992). An efficient algorithm for the all pairs suffix-prefix problem. *Inf. Process. Lett.* 41, 181–185.

Gusfield, D., C. Martel (1989). A fast algorithm for the generalized parametric minimum cut problem and applications, Technical Report CSE-89-21, University of California, Davis, CA.

Gusfield, D., L. Pitt (1986). Equivalent approximation algorithms for node cover. *Inf. Process. Lett.* 22, 291–294.

Gusfield, D., E. Tardos (1991). A faster parametric minimum cut algorithm. *Algorithmica*, to appear.

Harel, D., R.E. Tarjan (1984). Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.* 13, 338–355.

Hopcroft, J.E., R.M. Karp (1973). An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comput.* 2, 225–231.

Karlof, H. (1986). Las Vegas RNC algorithm for maximum matching. *Combinatorica* 6(4), 387–391.

Karp, R.M. (1978). A characterization of minimum cycle mean in a digraph. *Discrete Math.* 23, 309–311.

Karzanov, A.V. (1974). Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dokl.* 15, 434–437.

Lawler, E.L. (1976). *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York.

Malhotra, V.M., M.P. Kumar, S.N. Maheshwari (1978). An o(n^3) algorithm for finding maximum flows in networks. *Inf. Process. Lett.* 7, 277–278.

Martel, C. (1989). A comparison of phase and non-phase network algorithms. *Networks* 19, 691–705.

Matula, D. (1987). Determining edge connectivity in o(nm), *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pp. 249–251.

Mulmuley, K., U.V. Vazirani, V.V. Vazirani (1987). Matching is as easy as matrix inversion. *Combinatorica* 7, 105–131.

Picard, J., M. Queyranne (1982). Selected applications of minimum cuts in networks. *INFOR J. (Can. J. Oper. Res. Inf. Process.)* 20, 394–422.

Podderyugin, B.D. (1973). An algorithm for determining edge connectivity of a graph. Soviet Radio.

Röck, H. (1980). Scaling techniques for minimal cost network flows, in: U. Pape (ed.), *Discrete Structures and Algorithms*, pp. 181–191.

Schieber, B., U. Vishkin (1988). On finding lowest common ancestors: Simplifications and parallelization. *SIAM J. Comput.* 17, 1253–1262.

Schnorr, C.P. (1979). Bottlenecks and edge connectivity in unsymmetrical networks. *SIAM J. Comput.* 8, 265–274.

Shiloah, Y., U. Vishkin (1982). An o($n^2 \log n$) parallel maximum flow algorithm. *J. Algorithms* 3, 128–146.

Stone, H.S. (1978). Critical load factors in two-processor distributed systems. *IEEE Trans. Software Engrg.* 3, 254–258.

Tardos, E. (1985). A strongly polynomial minimum cost circulations algorithm. *Combinatorica* 5, 247–255.

Tarjan, R.E. (1983). *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA.