GRAPH ALGORITHMS

J. van Leeuwen

RUU-CS-86-17
October 1986

Rijksuniversiteit Utrecht

Vakgroep Informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

CONTENTS

Contents

1 Representation

Graphs are the most common "abstract" structure encountered in computing science. [illegible] system that consists of discrete [illegible] [illegible] between [illegible] can be modeled by a graph. [illegible] between nodes of a graph [illegible] called edges (in case the [illegible] between unordered pairs of nodes), directed edges (in case the connections are between ordered pairs of nodes), or hyperedges (in case the connections are between arbitrary nonempty sets of nodes). Connections may also carry additional information as labels or weights, related to the interpretation of the graph. Consequently there are many types of graphs and many basic notions that capture aspects of the structure of graphs. In this Chapter we only consider finite graphs with undirected or directed edges.

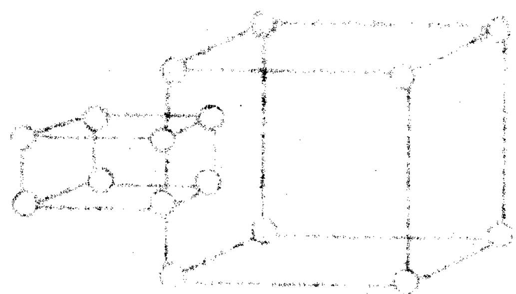
1.1 What is a Graph.

In this section we give a first introduction to the "representation" of graphs, and to the challenge of finding efficient graph algorithms. Although we omit it here, much of what we say is easily modified for weighted graphs.

Definition. A graph is a structure $G = < V, E >$ in which V is a finite set of nodes and $E \subseteq V \oplus V$ is a finite set of edges (unordered pairs). A directed graph is a structure $G = < V, E >$ in which V is again a finite set of nodes and $E \subseteq V \times V$ is a finite set of directed edges (ordered pairs).

It is common in computer science to "draw" graphs in 2- or 3- dimensional space (see figure 1). Directed edges are drawn as arcs (arrows). Three dimensional drawings of graphs

[illegible] directed graph



[illegible] graph

[illegible]

is embedded in a (3-dimensional) grid: In a 2-dimensional variant one might want to draw a graph with all nodes on a line and edges drawn as horizontal strips on the grid such that no two edges in one strip overlap (see figure 2). Define the cutwidth of a layout as the maximum number of edges that pass over any point of the line, and define the cutwidth of the graph G as the minimum of these numbers over all linear arrangements of nodes on a line. It is a simple result of graph theory that the cutwidth of a graph G equals

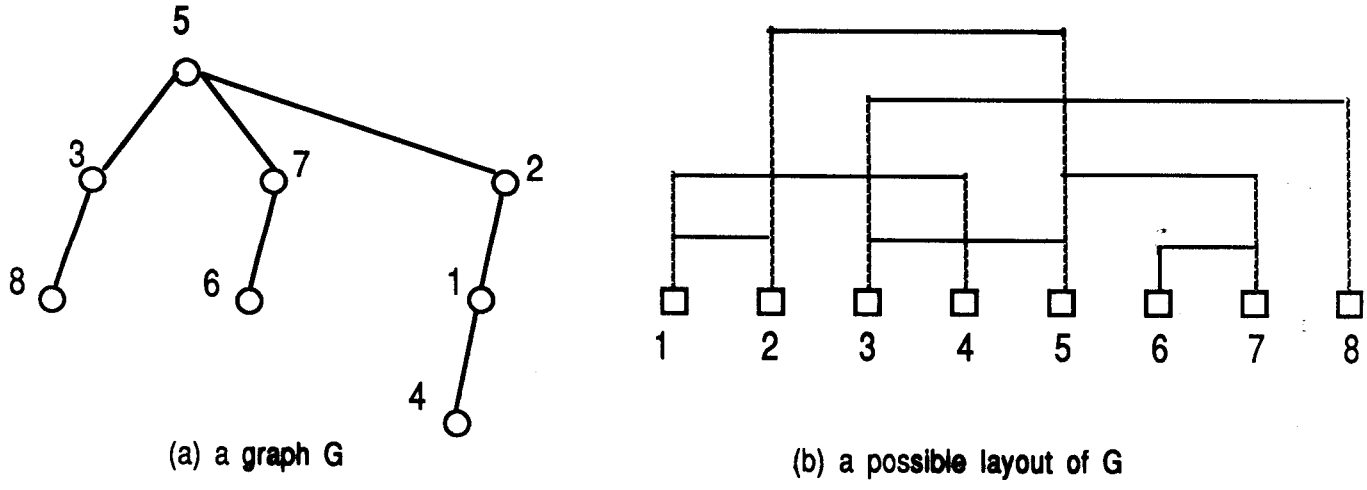
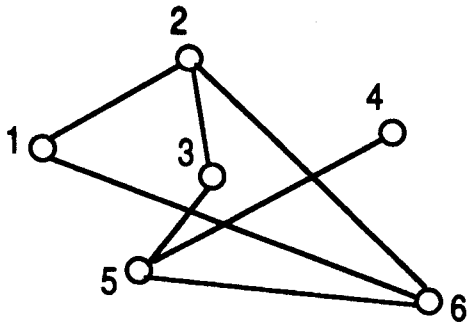


Figure 2:

the minimum number of horizontal grid lines required to layout the graph as desired (the MIN-CUT LINEAR ARRANGEMENT PROBLEM). On the other hand, determining the cutwidth of a graph is an example of a problem in the theory of graph algorithms.

While graph theory concerns itself with general results about the structure of graphs, the theory of graph algorithms is concerned with the design and analysis of effective methods to compute in graphs. The overriding concern is the design of graph algorithms of low complexity. Normally graph algorithms use results from graph theory, but very often the designer has to delve more deeply into the structure of a problem to arrive at a practical algorithm. For example, Kuratowski's characterization of planar graphs would lead to a highly inefficient planarity test when implemented on a computer, because it would require testing that no subgraph of the given graph is contractible to K_5 or $K_{3,3}$. On the other hand, it is known that the planarity of a graph can be tested by an algorithm that uses only $O(n+e)$ steps, where $n = \mid V \mid$ and $e = \mid E \mid$.

The quest for algorithms of low complexity has lead to many intriguing problems in the study of graph algorithms. For example, no efficient, i.e., polynomial time algorithm is presently known for the graph layout problem discussed above. In fact, the MIN-CUT LINEAR ARRANGEMENT problem is known to be $NP-$ complete. It is common in the theory of graph algorithms to study problems for special classes of graphs also, and to show that the special graphs are easier to deal with algorithmically. For example, the MIN-CUT LINEAR ARRANGEMENT problem is solvable by an $O(n \log n)$ algorithm for the case of $n-$node trees.



(a) a graph G

	1	2	3	4	5	6
1	0	1	0	0	0	1
2	1	0	1	0	0	1
3	0	1	0	0	1	0
4	0	0	0	0	1	0
5	0	0	1	1	0	1
6	0	1	1	0	1	0

(b) the adjacency matrix of G

Figure 3:

In this survey we will give an overview of the common paradigms and results in graph algorithms, with a bias towards the more recent developments in the field. We assume familiarity with elementary graph theory, the usual data structures and the theory of $NP-$ completeness.

1.2 Computer Representations of Graphs.

In order to compute with a graph G, it must be represented in computer storage somehow. The particular representation chosen for G can have a profound effect on the complexity of an algorithm. Thus graph representation is a form of data structuring. We discuss some representations and their impact.

1.2.1 Edge Query Representation.

The most obvious representations for a graph G use a boolean subroutine $S(x, y)$ with the property that $S(i, j) = 1$ if and only if $(i, j) \in E$. There generally is a trade-off between the time it takes S to respond to a query $S(i, j)$ ("is there an edge between i and j ") and the data storage it uses. One possibility for S is to use the adjacency matrix A of G, with A the 2-dimensional 0-1 matrix defined by $A(i, j) = 1(0)$ if and only if $(i, j) \in (not \in) E$ (see figure 3). It enables query answering in $O(1)$ time but requires $O(n^2)$ storage. For undirected graphs the adjacency matrix is symmetric, and space can be saved by storing only the $\frac{1}{2}n(n-1)$ upper-diagonal elements. In so-called loop-free (directed) graphs we have $A(i, j) \cdot A(j, i) = 0$ for all i, j and all information can be comprised in the $\frac{1}{2}n(n-1)$ upper-diagonal elements e.g. by setting $A(i, j)$ to -1 when $A(i, j) = 1\,(i < j)$.

Lemma. Any n x n adjacency matrix can be represented in $O(\frac{n^2}{\log n})$ storage while retaining $O(1)$ time access to its elements.

Proof.

Choose $k = \sqrt{\frac{1}{2}\log n}$ and observe that A can have at most 2^{k^2} different k x k submatrices. Write A as a $\frac{n}{k}$ x $\frac{n}{k}$ matrix of pointers to k x k submatrices. This requires $\frac{n^2}{k^2} + 2^{k^2} \cdot k^2 = O(\frac{n^2}{\log n})$ storage, while the access time for every element is still $O(1)$. (Observe that this does not save on the total number of bits needed to represent A.) □

Adjacency matrices are very handy when dealing with path-problems in graphs.

Lemma. Nodes i, j are connected by a path of length k if and only if $A^k(i,j) = 1$

.

Proof.

By induction on k. The induction basis ($k = 1$) is trivial. For $k > 1$, observe that $A^k(i,j) = (A^{k-1}.A)(i,j) = \sum A^{k-1}(i,x) \cdot A(x,j) = 1$ if and only if there is an $x \in V$ with $A^{k-1}(i,x) = 1$ and $A(x,j) = 1$ □

Consider undirected or loop-free directed graphs G with n nodes. Extremely interesting questions arise if we wish to know the minimum number of probes $c(P)$ of the adjacency matrix of a graph required in the worst case, in order to determine whether the graph possesses a given "non-trivial" property P. A probe is simply the inspection of one entry of the adjacency matrix (and we assume that an algorithm will forever "remember" what the entry was after it probed it), and a property is a subclass of the class of all n−node graphs that is closed under isomorphism. A property P is called "non-trivial" if P holds for some graphs but not for all. At first sight it may seen that $c(P) = \frac{1}{2}n(n-1)$ for all non-trivial graph properties, but a classical example due to Aanderaa (see Rosenberg) shows that this is not the case. The following, slightly better result is due to King and Smith-Thomas.

Lemma. If P is the property of having a sink node (for loop-free directed graphs), then $c(P) = 3n - [\log n] - 3$.

Proof.

(A sink is defined to be a node with in-degree $n-1$ and out-degree 0.) We only show that $c(P) \leq 3n - [\log n] - 3$. Arrange the n nodes at the leaves of a tournament tree (or heap). Run tournaments, with node i defeating node j when $A(i,j) = 0$ and node j defeating node i when $A(i,j) = 1$. Observe that loosers cannot be sinks. Clearly it takes $n-1$ probes to determine the winner v of the tournament, which is the only possible candidate for being a sink. By probing the $A(v,j)$ and $A(j,v)$− values for all $n-1$ nodes $j \neq v$ it is easily decided whether v is indeed a sink. We don't have to probe the $\geq [\log n]$ entries $A(v,j)$ or $A(j,v)$ that were already queried during the tournament. Thus $(n-1) + 2(n-1) - [\log n]$ probes suffice. □

Bollobas and Eldridge have established that for all non-trivial graph properties P, $c(P) \geq 2n - 4$.

On the other hand there are many graph properties for which $c(P)$ is not linear. Define a graph property P to be "elusive" if all essential entries of the adjacency matrix must be probed (in the worst case) in order to establish P, i.e., when $c(P) = \frac{1}{2}n(n-1)$. (We assume here that for loop-free directed graphs the compacted version of the adjacency matrix is used.) The following theorem combines results of Holt & Reingold, of Best, van Emde Boas & Lenstra, of Milner & Welsh, of Bollobas, of Kirkpatrick and of Yap.

Theorem. The following graph-properties are elusive: (i) having $\leq k$ edges ($0 \leq k \leq \frac{1}{2}n(n-1)$), (ii) having a vertex of degree $\geq 2 (n \geq 3)$, (iii) being a tree, (iv) having a triangle, (v) having a cycle, (vi) being connected, (vii) being Eulerian, (viii) being Hamiltonian (n prime), (ix) being bi-connected, (x) being planar ($n \geq 5$), (xi) containing a complete graph of order k ($2 \leq k \leq n$), and (xii) being $k-$chromatic.

Bollobas and Eldridge have shown that every non-elusive property of graphs must be satisfied by at least three, non-isomorphic graphs. Finally, define a graph-property P to be "monotone" when every $n-$node supergraph of an $n-$node graph G which satisfies P, also satisfies P. In 1973 Rosenberg formulated "the Aanderaa-Rosenberg conjecture" asserting that for every non-trivial, monotone property of graphs P, $c(P) = \Omega(n^2)$. Much of the results stated earlier actually arose from attempts to resolve the conjecture. The Aanderaa-Rosenberg conjecture was settled by Rivest and Vuillemin, who proved the following result.

Theorem. If P is a non-trivial, monotone property of graphs, then $c(P) \geq n^2/16$.

Kleitman and Kwiatkowski improved the bound to $c(P) \geq n^2/9$.

In a different approach one might want to represent S by a combinational circuit with two $\log n-$bit input lines and a 3-valued output line such that

$$P(x,y) = \begin{cases} ? \text{ if } x \notin V \text{ or } y \notin V, \\ \text{o if } x,y \in V \text{ and } (x,y) \notin E, \\ 1 \text{ if } x,y \in V \text{ and } (x,y) \in E. \end{cases}$$

Define a graph to have a small circuit representation if it can be represented by a combinational circuit of size $O(\log^k n)$ for some k. Many simple graph problems are surprisingly complex for graphs that are given by small circuits (see figure 4),i.e, the succinctness of the graph encoding saves space but not time.

1.2.2 Node Query Representation.

Another common representation for a graph G was uses a subroutine $S(x)$ which generates the edges incident to x for any $x \in V$. A typical example is the adjacency list of G in which for each $i \in V$, $S(i)$ produces a pointer to the "list" of edges incident to i. An edge (i,j) is represented by a simple record (or is nil when no next record in the list exists)

	Problem	Upper Bound	Lower Bound
(1)	Has a triangle	NP	NP
(2)	Has a $k-$cycle	NP	NP
(3)	Has a $k-$path	NP	NP
(4)	$\Delta(G) \geq k$	NP	NP
(5)	$\delta(G) \leq k$	Σ_2	Σ_2
(6)	Has a cycle	DSPACE(n)	NP
(7)	Has an Euler circuit	NSPACE(n)	NP
(8)	Has an $s-t$ path	NSPACE(n)	Π_2
(9)	Connectivity	NSPACE(n)	Π_2
(10)	Perfect matching	Exp.-DTIME	Π_2
(11)	Hamiltonian circuit	Exp.-DTIME	Π_2
(12)	Planar	Exp.-DTIME	Σ_2
(13)	Bipartite	Exp.-DTIME	Σ_2
(14)	$k-$colorable	Exp.-NTIME	Σ_2

Figure 4: Complexity of some graph problems for graphs given by a small circuit representation. (Note. G is a simple undirected graph, Δ and δ denote the maximum and minimum degree, respectively, and k is a fixed integer.)

(see figure 5). The adjacency list of a graph uses $O(n+e)$ storage and thus is usually a more compact representation than the adjacency matrix. The fact that the adjacency list produces the edges incident to a node in $O(1)$ time per (next) edge, makes it the primary representation of graphs in many algorithms. Note that for undirected graphs one can save space by numbering the nodes from 1 to n, and including in the adjacency list of a node i only those edges that lead to nodes $j \geq i$. Itai and Rodeh show by an information theoretic argument that in many cases this is about the tightest possible representation of a graph (in terms of the number of bits used). Sometimes it is already advantageous (from the algorithmic point of view) to have the edges (i,j) in the adjacency list of i ordered by increasing value of j. Call this the "ordered" list representation of G.

Lemma. Given any representation of G by adjacency lists, the ordered list representation of G can be constructed in $O(n+e)$ time.
Proof.

Use buckets B_1 through B_n. Go through the given adjacency lists edge by edge, and throw edge (i,j) into bucket B_j. Now purge the adjacency lists and build new ones by emptying the buckets B_1 through B_n in this order, appending any edge $(i,j) \in B_j$ to the end of the (current) adjacency list for i. □

Lemma. The representation of G by adjacency lists can be kept within $O(\frac{n^2}{\log n})$ storage while retaining the essential properties of the representation.
Proof.

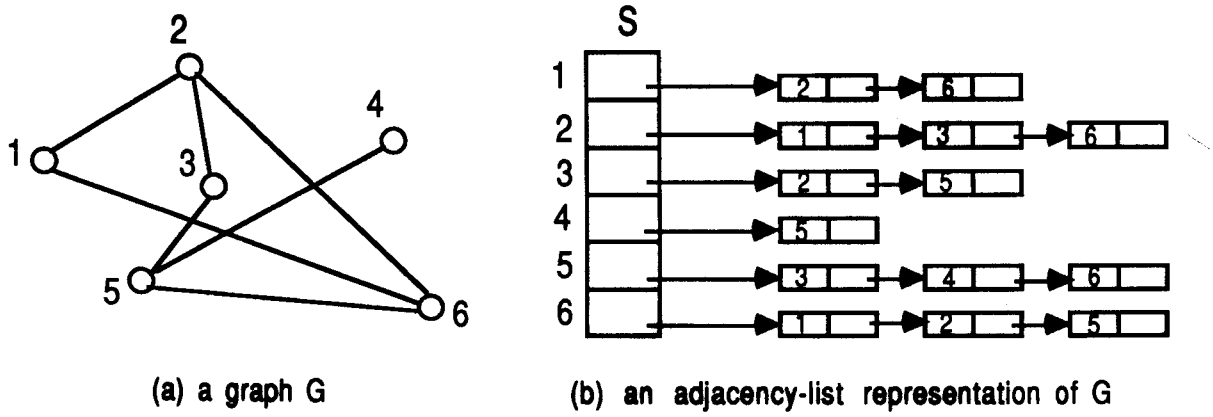


Figure 5:

Use the ordered version of representation. Choose $k = \frac{1}{2}\log n$, and observe that there can be at most 2^k different sub-lists on the adjacency lists involving the nodes $\alpha k + 1$ through $(\alpha+1)k$ for $\alpha = 0, \cdots, \frac{n}{k} - 1$. Represent the different lists separately, and include a reference to the proper sublist for every non-empty interval in the adjacency lists. This requires $n \cdot \frac{n}{k} + \frac{n}{k} \cdot k2^k = \frac{n^2}{k} + n2^k = O(\frac{n^2}{\log n})$ storage, and still allows for a simple traversal of the lists in $O(1)$ time per edge. □

Given the adjacency list representations of G, one can effectively obtain an adjacency matrix for G in $O(e)$ time despite the fact that $O(n^2)$ storage is required (cf. Aho, Hopcroft, & Ulman, p.71, exercise 2.12).

1.2.3 Structural Representation.

When a graph G has a particular structure, it may well be exploited for a suitable representation in computer storage. For example, an interval graph is defined such that every node x corresponds to an interval $I(x)$ on the real line and $(i,j) \in E$ if and only if $I(i) \cap I(j) \neq \emptyset$. The representation could simply consist of the intervals $I(x)$ for $x \in V$. It is an example of an "edge query representation" that uses only $O(n)$ storage.

When G is planar it is useful to record the edges in the adjacency list of every node i in counterclockwise order and to include in the record of every edge (i,j) a pointer to the record corresponding to (i,j) in the adjacency list of j. This representation has the added advantage that one can easily traverse the consecutive edges of any face of G in clockwise order in $O(1)$ time per edge. Itai and Rodeh have devised an ingeneous scheme to represent every $n-$node planar graph using only $1.5n \log n + O(n)$ bits.

Sometimes graphs can be hierarchically defined. In this case some nodes of G can be "expanded" and replaced by other graphs (again hierarchically defined), with suitable rules that specify the connection of the edges incident to an expansion node and the nodes

of the replacement graph. Observe that the complete expansion could lead to a graph of exponential size. Many basic graph problems like planarity testing and minimum spanning tree construction for hierarchically defined graphs can be solved in time and space linear in the size of the hierarchical description.

Acyclic directed graphs can be represented with the nodes linearly arranged in such a way that all (directed) edges run strictly from left to right. The representation is often used and known as the topological ordering or topological sort of the graph.

Theorem. A topological ordering of an acyclic directed graph can be computed in $O(n+e)$ time.

Proof.

By enumerating the edges one by one, one can compute the indegree $c(i)$ of all nodes i in linear time. Make one sweep over the list of $c(i)$ values to compile the "batch" L of nodes with $c(i)$-value 0 (the current sinks). Now execute the following code, using an auxiliary list L' that is initially empty.

```
repeat
    output the nodes of L (in any order);
    for each node i ∈ L do
        for each edge (i, j) do
          begin
              c(j) := c(j) − 1;
              if c(j) = 0 then append j to L'
          end;
        L := L';
        L' := empty
until L = empty.
```

Note that in the **for**-loop the computational effort can be accounted for by charging $O(1)$ to each node $i \in L$ and $O(1)$ to each edge (i,j). Because every node (and hence, every edge) appears precisely once in the process, the computation takes only $O(n+e)$ time total. □

For general directed graphs, Shiloach has shown that the nodes can be linearly arranged as $1,\ldots,n$ such that for every $i < j$ the minimum number of edges blocking every path from i to j is greater than or equal to the minimum number of edges blocking every path from j to i. The ordering can be computed in $O((n+e)^2)$ time.

1.3 Graph Exploration by Traversal.

Consider the following traversal of a (connected) graph G. Initially all nodes are marked unvisited and all edges are colorless. Start at a fixed but arbitrary node i_o and proceed in the following way whenever a node i is reached. When i was not reached before, mark it as visited. If there are no unexplored edges left in the edge-list of i, then backtrack to the node from which i was reached. Otherwise traverse the first unexplored edge (i,j) in the edge-list of i (and mark it explored implicitly by moving some pointer to the next element

of the list). When j was visited before, color the edge red and backtrack to i. When j was not visited before, color the edge green. Now "visit" the node reached in a similar manner. The algorithm is known as depth-first search (DFS) and is easily implemented using a stack and suitable pointer moves over the adjacency lists. The algorithm terminates when all edges incident to i_o have been explored.

Theorem. Let G be a connected graph.

(i) The edges colored green by DFS form a spanning tree of G, called a DFS tree.

(ii) The edges colored red by DFS always connect two nodes of which one is an ancestor of the other in the DFS tree.

(iii) DFS takes $O(n+e)$ time.

DFS trees are very special because of (ii), and normally reveal quite a bit of structure of a graph. For example, DFS leads to an algorithm for finding the bi-connected components of a graph in $O(n+e)$ time. Depth-first search can also be applied to directed graphs. An important application is Tarjan's algorithm for determining the strongly connected components of a directed graph in $O(n+e)$ time.

Another popular technique of traversing a (connected) graph G proceeds as follows and assigns a number $L(i)$ to every node i. Initially all nodes are marked unvisited and all edges are colorless. Start with $\{i_o\}$, mark i_o as visited, set$L(i_o) = o$, and do the following whenever a set of nodes I is reached. For every $i \in I$ and edge (i,j) on the edge-list of i, color the edge red when j was visited before. Otherwise color the edge green, mark j as visited, set $L(j)$ to $L(j)+1$, and add j to a set I'. Repeat the step until the "new" I' is empty. The algorithm is known as breadth-first search(BFS) and is easily implemented by maintaining the set I in a queue. All nodes in a set I have the same $L-$value.

Theorem. Let G be a connected graph.

(i) The edges colored green by BFS form a spanning tree of G, called a BFS tree.

(ii) For every $i \in V, L(i)$ is the shortest distance (in edges) from i_o to i.

(iii) For every green edge (i,j) one has $|L(i) - L(j)| = 1$. For all edges (i,j) one has $|L(i) - L(j)| \leq 1$.

(iv) BFS takes $O(n+e)$ time.

$L(i)$ is called the "level" of i. Observe from (ii) that edges always connect nodes that are at most one level apart. Define truncated BFS to be the BFS procedure up to and including the moment that the first red edge is encountered. Truncated BFS finishes in $O(n)$ time because it traces a tree-like environment of i_o. (The red edge closes the first cycle, if there is one). Clearly other forms of truncated BFS can be defined.

1.4 Transitive Reduction and Transitive Closure.

When one is only interested in representing the path information of a (directed) graph G, two extreme approaches can be followed:

(i) (minimum storage representation) determine a "minimal" graph $G^- =< V, E^- >$ with the property that there is a (directed) path from i to j in G if and only if there is a (directed) path from i to j in G^-.

(ii) (minimum query time representation) determine the (unique) graph $G^* =< V, E^* >$ with the property that there is a (directed) path from i to j in G if and only if there is a (directed) edge $(i,j) \in E^*$ in G^*. The graph G^* is traditionally known as the transitive closure of G.

We outline the known results for both types of representation. A simple idea for obtaining a minimum storage representation of G is to delete as many edges as possible without destroying the existing connections by directed paths. Any graph with the smallest number of remaining edges that can be obtained in this way, is called a minimum equivalent graph (or MEG). Moyles and Thompson (see also Hsu) have shown that a MEG for G can be constructed out of the MEGs of the individual strongly connected components of G and a MEG of the "condensed " graph (i.e, the acyclic graph with the strongly connected components as nodes). Sahni proved that the problem of deciding whether G has a MEG with at most k edges (for specified k) is $NP-$complete. If we drop the requirement that the "reduced" graph be a subgraph of G, then one can sometimes save more edges. Define a transitive reduction of G to be any graph G^- with the smallest possible number of edges, with the property that there is a directed path from i to j in G if and only if there is a directed path from i to j in G^-. For acyclic graphs the transitive reductions are MEGs (and vice versa). Aho, Garey and Ullman have shown that each graph has a transitive reduction that can be computed in polynomial time and that, algorithmically, computing transitive reductions is of the same time complexity as computing transitive closures and, hence, of multiplying Boolean matrices (see later).

The problem of computing the transitive closure G^* of a (directed) graph was first considered in 1959 (Roy) and a variety of algorithms have been proposed for it since. One popular approach is based on the use of the adjacency matrix A of G, considered as a Boolean matrix. Assume that ordinary matrix multiplication takes $O(n^\alpha)$ time and that the elementary arithmetric operations (no division) on $n-$bit numbers can be performed in $m(n)$ "steps". It is known that one can choose $\alpha < 2.4, m(n) = 1$ in the uniform model and $m(n) = O(n \log n \log\log n)$ if we count bit-operations. Furman observed that the adjacency matrix A^* of G^* can be written as $A^* = I \vee A \vee A^2 \cdots \vee A^{n-1} = (I \vee A)^{n-1}$. It suggests that A^* can be computed in $O(n^\alpha m(\log n) \log n)$ time using repeated squaring, by computing each necessary matrix product over the ring of integers modulo $n+1$ and recovering the true Boolean matrix product by changing nonzero entries to ones. Munro and Fischer & Meyer proved the following key result. Let $M(n)$ be a function satisfying $M(2n) \geq 4M(n)$ and $M(3n) \leq 27M(n)$.

Theorem. A^* can be computed in $O(M(n))$ time if and only if the product of two arbitrary n x n Boolean matrices can be computed in $O(M(n))$ time.

It follows that transitive closures can be computed in $O(n^{\alpha}m(\log n))$ time, because two n x n Boolean matrices can be multiplied within this time bound (by the same technique as before). Adleman et al. have improved this bound to $O(n^{\alpha}\log n)$ bit operations and even to $O(n^{\alpha}\log n(\frac{\log\log n}{\log n})^{\frac{\alpha}{2}-1})$ bit operations, by means of modular arithmetic and table look-up techniques using only $O(n^2\log n)$ bits of storage. The well-known "four Russians" algorithm (apparently due to Kronrod) is much less sophisticated and gives a bound of only $O(n^3/\log n)$ steps for Boolean matrix multiplication both in the worst case and in an average sense (van Leeuwen). O'Neill and O'Neill show that the simple technique of computing every row x column product by stepping through the 1's in the row (only) and stopping when the corresponding column position has a 1 (in which case the product will be 1), yields an algorithm for computing the product of two arbitrary Boolean matrices in $O(n^2)$ expected time.

Other approaches use the adjacency matrix in more direct terms as a problem representation. The classical algorithm of Warshall computes A^* as the "limit" of a series of matrices $A(0) = I \vee A, A(1), A(2), \ldots$ with $a_{ij}(s) = 1$ if and only if there is a directed path from i to j passing through intermediate nodes $\in \{1, \ldots, s\}$ only. $A(s)$ can be computed in $O(n^2)$ time from $A(s-1)$ by observing that $a_{ij}(s) = a_{ij}(s-1) \vee a_{is}(s-1)a_{sj}(s-1)$. As $A^* = A(n)$, Warshall's algorithm computes transitive closures in $O(n^3)$ time. Several improvements or Warshall's algorithm have been proposed that are more efficient when one analyses the number of rows that must be "paged in" , assuming that matrices are stored row-wise on disk (Warren, Martynyuk). An interesting, space-efficient variant was proposed by Thorelli. Put all directed edges in a queue Q, introduce two pointers α and β, and execute the following code. (Suppose that the queue elements are numbered $1, 2, \ldots$ for presentation and let the edge pointed at by α be (i_α, j_α).)

for $\alpha := 1$ **to** end(Q) **do**
  for $\beta := 1$ **to** end(Q) **do**
    if $j_\alpha = i_\beta$ and $i_\alpha \neq j_\beta$ **then**
      if $(i_\alpha, j_\beta) \notin Q$ **then** append (i_α, j_β) to Q.

Note that Q keeps expanding as the algorithm proceeds, until it contains the full $e^* = |E^*|$ edges of the transitive closure. Another interesting variant was considered by Ibaraki and Katoh. Suppose we have the A^* of G and wish to update it when, say, k edges are added. Clearly no 1's can change to 0's, and we can confine ourselves to considering the question which 0's of A^* must be turned into 1's. In case an edge (i, j) is added and $a^*(u, v) = 0$ then $a^*(u, v)$ must only be changed to 1 when $a^*(u, j) = 0, a^*(u, i) = 1$ and $a^*(j, v) = 1$. Process the t^{th} edge as follows. Determine the sets $U_t = \{u | a^*(u, j) = 0, a^*(u, i) = 1\}$ and $V_t = \{v | a^*(j, v) = 1\}$ (always $j \in V_t$) by straightforward inspection in $O(n)$ time and, when $U_t \neq \emptyset$, set $a^*(u, v)$ to 1 for all $u \in U_t$ and $v \in V_t$. Note that the latter costs $O(u_t v_t)$ time and creates at least u_t more 1's in A^* (because all $a^*(u, j)$ were 0 for $u \in U_t$), with $u_t = |V_t|$. Thus the addition of k edges requires a total of $O(kn + \sum_1^k u_t v_t) = O(ne^*)$ time, using that $k \leq e^*$, $v_t \leq n$ for all t and $\sum_1^t u_t \leq e^*$. The algorithm suggests an interesting, incremental approach to the construction of transitive closures. Ibaraki and Katoh also prove that the transitive closure can be updated in $O(n^3 + n^2 e)$ time when any number of edges is deleted from G.

A considerable number of transitive closure algorithms is based on the following lemma implicit in the work of Munro and Purdom. Let $l(n)$ denote the time for computing the

transitive closure of an acyclic directed graph of n nodes, and let the condensed graph of G (i.e., the induced graph with the strongly connected components of G as nodes) have n_c nodes and e_c edges.

Theorem. The transitive closure of a directed graph can be computed in $O(n + e^* + l(n_c))$ time.

Proof.

Compute the strongly connected components of G and the condensed graph G_c in $O(n+e)$ time, using DFS. The transitive closure of G can be computed in $O(n+e^*)$ time from the strongly connected components and the transitive closure of G_c by enumerating all implied pairs, i.e., all pairs (i,j) with i and j in the same strongly connected component or in components that are directly connected in G_c^* by an edge from the $i-$component to the $j-$component. Because G_c is acyclic, its transitive closure can be computed in $l(n_c)$ time. □

Eve & Kurki-Suonio, Ebert and Schmitz all show that the DFS-algorithm for computing the strongly connected components of a graph gives enough structural information to generate the transitive closure directly, typically in $O(n^2 + ne)$ time. The theorem shows that this bound may be overly pessimistic, especially when n_c is small. Consider any acyclic directed graph G of n nodes, let $G^- =< V, E^- >$ be its minimum equivalent graph (or transitive reduction) and $e^- = |E^-|$ (hence $e^- \leq e$). One can compute a topological ordering of G in $O(n+e)$ time (cf. section 1.2), with no additional overhead for having each adjacency list $L(i)$ sorted (in increasing order). The following result is due to Goralcikova and Koubek.

Theorem. $l(n) = O(n + e + ne^-)$.

Proof.

We have spend $O(n + e)$ time to represent G in topological order. Define $R(i) = \{j|$ there is a directed path from i to $j\}$. It is clear that the sets $R(i)$ represent G^*. (Observe that $j \in R(i)$ implies $j > i$.) Compute the sets $R(i)$ inductively as follows for i from n down to 1. Suppose $R(j)$ has been computed for all $j > i$. Compute $R(i) = \{i\} \cup \{R(j)|(i,j) \in E\}$ as follows, using an auxiliary bit-vector γ that is initially empty. Begin by putting i into $R(i)$ and setting the i^{th} bit of γ to 1. In fact we will use γ as the characteristic vector of $R(i)$ at all times. Now process all edges $(i,j) \in E$ (i.e., $j \in L(i)$) in order of increasing j as follows: when j already appears in $R(i)$ (which we can tell in $O(1)$ time using γ) ignore this edge and proceed, otherwise append the elements of $R(j)$ to $R(i)$ (while updating γ and, in fact, using γ to avoid adding nodes to $R(i)$ that already appear in it). Note that the edges (i,j) with $j \in R(i)$ are correctly ignored because, when j was added to $R(i)$, necessarily all nodes reachable from j must have been added to $R(i)$ as well. The edges (i,j) with $j \notin R(i)$ necessarily belong to G^-. For suppose $(i,j) \notin G^-$. Then there would be a path $i \rightarrow u \rightarrow \ldots \rightarrow j$ with $(i,u) \in E$ and (necessarily) $u < j$, and j must have been added to $R(i)$ by the time the edge (i,u) was processed. Contradiction. It follows that the time complexity of the algorithm is $O(n + e)$ plus $(\sum_{(i,j)\in E^-} |R(j)|) = O(ne^-)$. □

It can be argued that for random acyclic directed graphs the expected value of e^- is $O(n^{1.5})$. Jaumard and Minoux have given a slightly different implementation of the same algorithm, showing that $l(n) = O(n + de^*)$ for acyclic directed graphs with indegrees bounded by d. Mehlhorn and Simon have developed another algorithm showing that $l(n) = O(e+(n+e^-)p)$, where p is the number of chains (paths) in a chain decomposition of the acyclic directed graph. The expected time complexity of their algorithm is $O(n^2 \log n)$.

Following yet another approach, several algorithms try to construct the transitive closure of G by progressively searching for the "successors" of every node. Bloniarz et al. considered the following simple algorithm for determining all nodes j that are reachable by a directed path from i, for each i. We use the adjacency lists $L(j)$, an auxiliary list $R(i)$ to accumulate the $j's$ that are reachable from i, an auxiliary bit-vector γ that will serve as the characteristic vector for $R(i)$ (as before), and an auxiliary pushdown list P to record the nodes which we need to explore further in DFS-like order. For every node i, the following code is executed. Let "record(k)" denote the combined operation of appending k to $R(i)$, setting the k^{th} bit of γ to 1, and pushing k onto P.

```
record(i);
count:=1;
while P ≠ ∅ & count < n do
   begin
            pop the top-element of P and assign it to j;
            for each node k ∈ L(j) do
                if k hasn't been recorded yet (k^th bit of γ is 0) then
                   begin
                     record(k);
                     count:=count+1
                   end
   end;
output R(i).
```

For each i the algorithm may need up to $O(n + e)$ time, and its total runtime will be $O(n^2 + ne)$ in worst case. Bloniarz et al. prove that over wide classes of "random" graphs the algorithm has an average runtime of $O(n^2 \log n)$. A transitive closure algorithm with an even better expected time complexity has been devised by Schnorr. The algorithm differs from the preceding one in two ways: the DFS-like search is replaced by a BFS-like search, and advantage is taken of the fact that the sets $R(i)$ are computed for all i and thus allow for a combination of effort. More precisely, using a BFS-like search-procedure on G and G^r (the graph G with all edges reversed) respectively, the algorithm determines sets $R_+(i)$ and $R_-(i)$ for every i such that $R_+(i)$ contains the maximum number $\leq [\frac{n}{2}]+1$ of successors of i and $R_-(i)$ contains the maximum number $\leq [\frac{n}{2}]+1$ of predecessors of i. The following lemma shows that these sets are sufficient to obtain G^* in another $O(n+e^*)$ time.

Lemma. For each i, $R(i) = R_+(i) \cup \{j | i \in R_-(j)\} \cup \{j | |R_+(i)| = |R_-(j)| = [\frac{n}{2}]+1\}$

Proof.

Observe that $R_+(i)$ and $R_-(j)$ contain all successors of i and predecessors of j, respectively, whenever their cardinality is $< [\frac{n}{2}]+1$. When they both have cardinality $= [\frac{n}{2}]+1$,

$R_+(i) \cap R_-(j) \neq \emptyset$ by the pidgin hole principle and there must be a node u such that $i \rightarrow^* u \rightarrow^* j$ and (hence) j is reachable from i. □

Schnorr proves that the transitive closure algorithm has an expected time complexity of $O(n+e^*)$ over large classes of random graphs and that the probability that the algorithm will run for more than cn^2 steps (some constant c) is exponentially small in n.

1.5 Generating an arbitrary Graph.

One of the most immediate problems in dealing with graphs may be the question of generating an "arbitrary" graph with a given number of nodes (n). More specifically the problem is to generate (or select) a graph G uniformly at random from the set of unlabeled graphs with n nodes. The problem is not entirely straightforward because graphs may be written down in (usually) many isomorphic forms and different graphs can have different numbers of isomorphisms. Let G_n be the set of undirected graphs on the node set $1,\ldots,n$ and let $E_n = \{(i,j)|1 \leq i,j \leq n \text{ and } (i,j) \text{ unordered}\}$. Clearly any $G \in G_n$ has an edge set E with $E \subseteq E_n$. Consider the symmetric group S_n acting on G_n by permuting the node labels. The orbits under S_n correspond precisely to the isomorphism classes of graphs on n nodes. Dixon and Wilf have given a simple algorithm for selecting an orbit uniformly at random for the general case of a group acting on a set, and obtained the following result by specializing the algorithm to the present situation. Assume that the value $g_n = |G_n|$ has been precomputed.

Theorem. There is an algorithm for selecting an $n-$node graph G uniformly at random in $O(n^2)$ average time.

Proof.

We only sketch the algorithm. For each $\pi \in S_n$, define the permutation π^* on E_n by $\pi^*((i,j)) = (\pi(i),\pi(j))$. Considering the action of S_n on G_n , one verifies that a graph G is left fixed by π if and only if $\pi^*(E) = E$. Thus $G \in Fix(\pi)$ if and only if for every cycle C of π^* (as a permutation) either all pairs in C occur as edges of G or none of them does. Hence $|Fix(\pi)| = 2^{c(\pi)}$, where $c(\pi)$ is the number of cycles of π^*. The value of $c(\pi)$ can be computed "by formula" from the cycle structure of π. Let $[k_1,\ldots,k_n]$ denote the conjugacy class of S_n consisting of the permutations having k_i cycles of length i $(1 \leq i \leq n)$, and let W $([k_1,\ldots,k_n]) = |[k_1,\ldots,k_n]| \cdot 2^{c(\pi)} = \frac{n!2^{c(\pi)}}{\pi_i(i^{k_i}k_i!)}$ be its "weight". The following algorithm will select an $n-$node graph G uniformly at random:

1. Choose non-negative integers $k_1,\ldots,k_n$ with $\sum_1^n i.k_i = n$ such that the probability of choosing an $n-$tuple $[k_1,\ldots,k_n]$ is $\frac{1}{n!g_n}W([k_1,\ldots,k_n])$.
2. Take (any) representative π from the class $[k_1,\ldots,k_n]$ chosen in step 1, and choose a graph G uniformly at random from $Fix(\pi)$.
3. "Ignore" the node-labels of G and output it.

Steps 2 and 3 are easily implemented in $O(n^2)$ time, which is essentially the time needed for writing down G. Dixon and Wilf show that step 1 can be implemented in $O(n^2)$ average time, provided g_n is known. □

The Dixon-Wilf algorithm is particularly useful for selecting graphs "fairly", without the need for extra storage (i.e., beyond what is needed to represent the selected graph itself). The integers involved in the computation get very large for $n \geq 10$, but this seems inherent to the problem. Dixon and Wilf also prove that the $\lceil g_n \log(\frac{g_n}{\varepsilon})\rceil$−fold iteration of the algorithm, with all choices independently and uniformly at random, leads to a sequence which with probability $> 1 - \varepsilon$ includes each n−node graph at least once. This algorithm seems particularly useful for testing hypotheses for n−node graphs, without having to list all graphs and go through expensive isomorphism rejection routines.

The approach of Dixon and Wilf can be used, at least in principle, for generating arbitrary graphs of a more restrictive type as well, but much work remains to be done here. For selecting a connected n−node graph uniformly at random one could simply iterate the Dixon-Wilf algorithm until a connected graph is output. (Note that by e.g. a DFS-based routine, the connectedness of a graph can be checked in $O(n+e)$ time.) On the average no more than 2 iterations will suffice for this, by known results on the population of connected graphs among all unlabelled n−node graphs.

1.6 Recognition of Graphs.

Many different types of graphs have been distinguished in the past. This has led to the important recognition problem for every type X of interest: given a graph, is it of type X. Related questions (which we do not consider here) are: given a graph G and an integer k, can one turn G into a graph of type X by adding/deleting k nodes/edges. There are numerous versions of this question.

Interestingly enough,the recognition problem for classes of graphs can be quite tricky. Nevertheless, it should probably be required of any class of graphs that is distinguished, that its recognition problem is of polynomial time bounded complexity. We have listed the known bounds for a number of classes of graphs that are commonly encountered in the following table (see Johnson).

category	class of graphs	complexity of recognition
trees and related graphs	Trees/Forests	linear
	Almost Trees (k)	linear
	Partial $k-$trees	exponential in k (Arnborg et al.)
	Bandwidth$-k$	polynomial in k (Saxe)
	Degree$-k$	linear
planar graphs	Planar	linear (Hopcroft & Tarjan, Booth & Lueker)
	Series Parallel	linear (Valdes et al.)
	Outerplanar	linear (Mitchell)
	Halin	linear
	$k-$Outerplanar	linear
	Grid	linear
	$K_{3,3}-$Free	linear (Asano, Williamson)
	Thickness$-k$	NP complete for $k \geq 2$
	Genus$-k$	$O(n^{\alpha(k)})$ (Filotti et al.)
perfect graphs	Perfect	in co-NP
	Chordal	linear (Gavril, Tarjan & Yannakakis)
	Split	linear (Golumbic)
	Strongly Chordal	polynomial (Farber)
	Comparability	$O(\delta.e)$ with δ = max degree (Golumbic)
	Bipartite	linear
	Permutation	$O(n^3)$ (Golumbic)
	Cographs	linear
intersection graphs	Undirected Path	$O(n^4)$ (Gavril)
	Directed Path	$O(n^4)$ (Gavril)
	Interval	linear (Booth & Lueker)
	Circular Arc	$O(n^3)$ (Tucker)
	Circle	polynomial (Gabor et al.)
	Proper Circ. Arc	polynomial (Tucker)
	Edge (or line)	linear (Lehot, Syslo)
	Claw-Free	linear

There are many related questions that can be asked: given a graph of type X, is it of type Y. There is also the following range of questions: given a graph of type X, does it have property P. (Here a "property" can be any graph-theoretic property that one may wish to test like planarity, having a Hamiltonian cycle, etc.) Also, many graph-theoretic constructions can be specialized to graphs of some type X. It appears that the given list of graphs is natural hierarchy for investigating the complexity of graph problems. There is a vast body of literature resulting from this "research program", especially dealing with the study of problems that are $NP-$complete for general graphs (see Johnson).

2 Basic Structure Algorithms.

Graph theory provides us with a wealth of results about the structure op graphs. In graph algorithms the aim is to identify substructures or properties algorithmically, by a program that can be run on every admissible input graph. Thus, the theory of graph algorithms will answer questions like "does a graph G have property P" by providing an algorithm that tests graphs for property P, instead of by theorems about property P alone. The theory is sophisticated because of the concern for algorithms of a provably low worst-case or average (expected) complexity. To reach low complexity, graph algorithms usually exploit theorems from graph theory (or finds them, when they are lacking). In this Chapter we review the essential structure algorithms for general, weighted or unweighted graphs.

2.1 Connectivity.

Two nodes i, j of G are said to be $k-$connected if k is the largest integer such that there exist k node-disjoint paths from i to j in G. We denote the connectivity of i and j by $cn(i,j)$. By Menger's theorem $cn(i,j)$ is equal to the minimum size of any set of nodes S whose removal disconnects i and j. (If i and j are adjacent, we take $cn(i,j) = n-1$.) Define the connectivity $cn(G)$ of G by : $cn(G) = \min_{i,j} cn(i,j)$. It is common to study the question: given some k, is $cn(G) \geq k$. The question is relevant in networks where one wants to know for each pair of nodes a guaranteed bound on the number of alternate routes between the nodes.

For $k = 1$ the problem reduces to the question of whether G is connected. By DFS this is easily answered in $O(n+e)$ steps. One of the classical results in graph algorithms is the fact that for $k = 2$ and $k = 3$ the problem can be solved in $O(n+e)$ steps also. We only look at the case $k = 2$ and assume without loss of generality that G is connected. Do DFS starting at some i_o and number nodes consecutively the first time they are visited. Let $v(i)$ be the number assigned to node $i, v(i_o) = 1$. Let T be the DFS tree, with all green edges (see Section 1.3). For any j , define $LOW(j) = min\{v(i) | i$ is reachable from j by a downward path of green edges followed by at most one red edge$\}$. Clearly $LOW(i_o) = 1$ and $LOW(j) \leq v(j)$ for all j .

Lemma.

(i) i_o is a cutpoint of G if and only if the degree of i_o in T is at least 2.

(ii) For all $j \neq i_o$, j is a cutpoint of G if and only if it has a son j' in T with $LOW(j') \geq v(j)$.

Proof.

(i) Let i_o be a cutpoint and let x, y be two nodes such that every path from x to y passes through i_o. Then x and y cannot be in the same subtree under i_o, and i_o has degree ≥ 2. Conversely, take any two nodes x, y in separate subtrees under i_o. Because red edges cannot reach across subtrees (cf. Lemma), every path from x to y must pass through i_o. Hence i_o is a cutpoint.

(ii) Let $j \neq i_o$ be a cutpoint, and let x and y be two nodes such that every path from x to y passes through j. At least one of x, y must belong to a subtree under j, say

y. If for every subtree under j there is at least one red edge connecting it to some ancestor of j, then every subtree can be reached from x by a path that avoids j. As this is impossible, there must be one subtree with no red edges leading to an ancestor of j. Its root j' is a son of j and satisfies $LOW(j') \geq v(j)$. The converse is easy, as any path connecting i_o and j' must pass through j. □

Observe that the numbers $v(i)$ are easily assigned while DFS proceeds, and $LOW(j)$ can be computed by the time the last subtree under j has been fully explored and DFS backtracks to the father of j in T. Using the lemma, all cutpoints can be identified during DFS at only a small expense of extra work and (hence) $cn(G) \geq 2$ if in fact no cutpoint is found. The algorithm can be modified to find all biconnected components in $O(n+e)$ steps as well. Stack edges when they are traversed for the first time, and output and pop whatever edges still are on the stack and were put on it after j was visited as a component, whenever j is identified as a cutpoint. A more complex modification of DFS generates all tri-connected components in $O(n+e)$ time as well.

Testing whether $cn(G) \geq k$ for $k \geq 4$ is considerably harder. Some algorithms compute $cn(G)$ exactly using the following observation.

Lemma. For every i, j the value of $cn(i,j)$ can be computed by solving an integer maximum flow problem on a graph of $O(n)$ nodes and $O(e)$ edges.

Suppose we have an $O(n^\alpha e^\beta)$ time max-flow algorithm (see Section 3.4). Write $c = cn(G)$. Assume G is not a complete graph, hence $c \leq n-2$. Let i_o be a node that is not connected to every other node and let $i_o, i_1, \ldots$ be some fixed listing of the nodes. Consider the following algorithm:

γ_{-1} := n-2 ;
t := -1 ;
repeat
  $t := t+1$;
  compute $cn(i_t, i)$ for every $i \in \{i_{t+1}, \ldots, i_{n-1}\}$ and
  set γ_t equal to the minimum of the computed values
  and γ_{t-1}
until $t \geq \gamma_t$;

Clearly $c \leq \gamma_t \leq \gamma_{t-1} \leq n-2$ for all $t \geq 0$ and, because t increases in every iteration, the algorithm is guaranteed to terminate. Suppose it terminates when t has value k, i.e., right after node i_k was considered. Then $c \leq \gamma_k \leq k$. Let x, y be two nonadjacent nodes with $cn(x,y) = c$ and S be a set of c nodes whose removal disconnects x and y. Because $k \geq c$ there must be an i_j with $0 \leq j \leq k$ such that $i_j \notin S$ and, because S "separates" the graph, there must be a node i such that the removal of the nodes in S in fact disconnects i_j and i as well. It means that $cn(i_j, i) \leq c$ (hence $cn(i_j, i) = c$) and γ_j must have been set to c during the iteration with $t = j$. Consequently $\gamma_t = c$ when the algorithm terminates and termination occurs exactly for t with $t = c$. The jth iteration costs (at most) n maxflow computations and thus takes $O(n^{\alpha+1}e^\beta)$ time. The complete algorithm computes $c = cn(G)$ in $O(c.n^{\alpha+1}e^\beta)$ time. (Note that $c \leq \frac{2e}{n}$, because every node must have degree $\geq c$.) By a different algorithm the question "is $cn(G) \geq k$" can be solved in $O(k^3 + kne)$ time, for every fixed k.

An interesting probabilistic approach to determining $c = cn(G)$ was suggested by Becker et al. It is based on the following observation.

Lemma. Let $c = cn(G) \leq n-2$, $\varepsilon > o$ and $r = \frac{log\frac{1}{\varepsilon}}{\log n - \log c}$. Draw r random nodes $i_1,...,i_r$ of G and determine $c' = \min_{1\leq j\leq r} \min_v cn(i_j, v)$, where the inner "min" is taken over all v not adjacent to i_j. Then Prob $(c' > c) \leq \varepsilon$.

Proof.

Let S be any node separator with $|S| = c$. For all nodes $i \in V - S$ one has $\min_v cn(i, v) = c$. (Note that S separates the graph in at least two components. For any node v in a component different from the one containing i one has $cn(i, v) \leq |S| = c$, hence $cn(i, v) = c$.) Thus for c' to be strictly greater than c, all random nodes that were drawn must belong to S. It follows that Prob $(c' > c) \leq (\frac{c}{n})^r \leq \varepsilon$. □

The lemma shows that c can be computed with error probability $\leq \varepsilon$ by minimizing over the values $\min_v cn(i, v)$ for a sufficiently large random sample of nodes i . By solving $O(n)$ maximum flow problems (Lemma), each $\min_v cn(i, v)$ value can be computed in $O(n^{\alpha+1}e^{\beta})$ time. Yet the minimum sample size (r) we need is hard to determine, because we need c itself to compute it. A (poor) upperbound can be obtained by setting c' to $n-2$ and substituting c' for c in the formula for r . Now observe that by increasing the sample size, the value for c' can only decrease (approaching c with high probability for a sample of size approaching r) and the corresponding estimate for r can only decrease as well, approaching the correct value of r with high probability. One can show that the expected number of random drawings needed for the sample to reach the size estimated as the algorithm goes on is $O(r + \varepsilon n)$. Thus for $\varepsilon \leq \frac{1}{n}$ the algorithm computes $cn(G)$ with error probability $\leq \varepsilon$ in only $O(rn^{\alpha+1}e^{\beta}) = O(\frac{\log\frac{1}{\varepsilon}}{\log n - \log c} n^{\alpha+1}e^{\beta})$ time, which can be significantly faster than the $O(cn^{\alpha+1}e^{\beta})$ algorithm discussed above.

2.2 Minimum Spanning Tree.

Let $G =< V, E, w >$ be a weighted graph, with w a function that assigns a weight to every edge. A minimum spanning tree (MST) is a spanning tree of G of which the sum of all edge weights is minimum, over all spanning trees. The problem of computing a MST has an interesting history that goes back to 1926. The problem is interesting in many networking contexts. Weights might be distances, traffic densities, costs, etc.

Definition.

(i) A MST-family in G is a collection of disjoint subtrees $F = \{T_1, \cdots, T_k\}$ for which there exists a MST T with $T_i \subseteq T$ for every i, $1 \leq i \leq k$, and $\bigcup_1^k T_i$ spans G.

(ii) Let $F = \{T_1, \cdots, T_k\}$ be a MST-family in G. The shrinking of G modulo F, denoted by G mod F, is the graph $< V', E', W' >$ with $V' = \{1, \cdots, k\}, E' = \{(i, j) |$there is an edge connecting T_i and $T_j (i \neq j)\}$ and for every edge $(i, j) \in E'$, $w'(i, j)$ is the weight of the least weighted edge connecting T_i and T_j.

For every MST-family F, G mod F can be constructed in $O(n + e)$ time. All MST-algorithms start from the trivial MST-family consisting of all single nodes and try to reduce

it to a one-element MST-family by a suitable process of combining trees and shrinking, i.e., reducing the problem to G mod F. The known algorithms differ by the organisation of the combining and shrinking steps, and by the data structures employed to keep the complexity low. The following two lemmas are essential to all MST-algorithms.

Combining Lemma. Let $F = \{T_1, \cdots, T_k\}$ be a MST-family in G and $1 \leq i \leq k, k \geq 2$. Let the least weighted edge from T_i to another member of F be (x, y) with $x \in T_i$ and $y \in T_j$ $(i \neq j)$. Let T_{ij} be the tree obtained by connecting T_i and T_j by the edge (x, y). Then the collection F' obtained from F by deleting T_i and T_j and adding T_{ij} is another MST-family in G (with one element less than F).

Shrinking Lemma. Let $F = \{T_1, \cdots, T_k\}$ be a MST-family in G, and T' a MST of G mod F. Expand T' by replacing every node i by the subtree T_i and every edge $(i, j) \in T'$ by the least weighted edge connecting T_i and T_j. The resulting tree T is a MST of G.

Several algorithms only use the Combining Lemma:

(A) Boruvka (1926) : apply the lemma for every $1 \leq i \leq k$ simultaneously (assuming all edge weights are different),

(B) Kruskal (1956) : take the least weighted edge connecting two distinct subtrees and let i, j be accordingly,

(C) Jarnik (1930), Prim (1957), Dijkstra (1959) : take $i = 1$ always.

Variant A is of interest for distributed computing. Variant B runs very smoothly after all edges are sorted by increasing weight, in $O(e \log n)$ time. Note that every tree T_i can (also) be represented as a set, and deciding whether (x, y) connects two different subtrees amounts to comparing the outcome of a FIND(x) and a FIND(y). Combining the two trees and joining the corresponding sets is a UNION operation. Thus the non-sorting phase of variant B can be implemented by a sequence of $O(e)$ FINDs and $n - 1$ UNIONs, which can be achieved in $O(e\alpha(n, e))$ time where $\alpha(n, e)$ is a very slowly growing function (practically a constant). Variant C is usually implemented by maintaining a datastructure Q which contains for every node $x \notin T_1$ the least weighted edge connecting x to T_1. Say x is joined to T_1, in the combining step. Then the edge for x must be deleted from Q, and for every neighbor y of x with $y \notin T_1$ we must consider whether its entry in Q must be updated to (y, x) (which leads to at most $\deg(x)$ update operations on Q, hence, to e updates alltogether). Implementing Q by any priority queue, leads to an $O(e \log n)$ time bound for variant C. A more clever priority queue leads to an $O(e \log n / \max\{1, \log \frac{e}{n}\})$ algorithm, which is linear in e whenever $e = \Omega(n^\alpha)$.

Now consider the following variant. Keep the subtrees of a MST-family in a queue with a marker # at the end. Apply the combining lemma to whatever T_i appears up front and pull the desired T_j from the row, put the resulting T_{ij} at the end of the queue and proceed until the # appears up front. Now move the marker to the end of the queue and continue with another phase, unless there is only one tree left (which must be a MST). Since every node occurs precisely once before the # at the beginning of a phase, the process of finding the least cost edge out of T_i (for every T_i that appears up front in this phase) inspects every node once. Let the edges incident to every node x

be divided into $\lceil \frac{\deg(x)}{f} \rceil$ groups of size f and sort every group, at a total cost of about $O(\sum_x (1 + \frac{\deg(x)}{f}) f \log f) = O(nf \log f + e \log f)$. Clearly this is done before phase 1. The least weight edge incident to x can now be found by inspecting every group, throwing away the edges that do not lead out of T_i, and minimizing over $\leq 1 + \frac{\deg(x)}{f}$ edges that do. This leads to a cost of $O(n + \frac{e}{f})$ per phase, and a total term of $O(e)$ for eliminating edges. By induction one easily proves that in phase l, every T_i in the queue has $\geq 2^l$ nodes. Thus there are (at most) $\log n$ phases, and the complexity of the algorithm after s phases is bounded by $O(nf \log f + ns + e \log f + e.\frac{s}{f})$. Here is how the Shrinking Lemma comes in. After $\log \log n$ phases we have spend $O(e \log \log n)$ time (take $f = 1$) and the MST-family F has trees of size $\geq 2^{\log \log n} = \log n$ each. G mod F is a graph of $\leq n/\log n$ nodes and $\leq e$ edges, and by running the same algorithm on it for the full $O(\log n)$ phases we obtain a MST for G mod F in $O(e \log \log n)$ time as well (take $f = \log n$). Using the Shrinking Lemma, it follows that a MST is obtained in $O(e \log \log n)$ time.

For special graphs a better bound may be obtained. For example, by shrinking after every phase a MST of a planar graph can be obtained in $O(n)$ steps. Also, there is still a lot of freedom left in the algorithm for the general case. By using another organization of the steps and refined data structures, the complexity of the MST problem can be reduced to $O(e\beta(n,e))$ with $\beta(n,e) = min\{i | \log^i n \leq \frac{e}{n}\}$ (a very slowly growing function) and even to $O(e \log \beta(n,e))$.

An interesting problem concerns the "maintenance" of minimum spanning trees. Spira & Pan showed that a minimum spanning tree can be updated in $O(n)$ time when a new node is inserted in the graph. Chin & Houck developed an $O(n^2)$ algorithm for handling deletions. Frederickson improved the bound to $O(\sqrt{e})$, and to $O(\log^2 e)$ for planar graphs.

2.3 Shortest Paths.

Let $G =< V, E, w >$ be a weighted, directed graph. For $s, t \in V$ we let $l(s,t)$ be the weight of a least weight path from s to t and π_{st} a path of weight $l(s,t)$ from s to t, provided such a path exists. (The weight of a path is the sum of the weights of the edges of the path.) If $w(x,y) = 1$ for every $(x,y) \in E$, then a least weight path is a "shortest" path in the traditional sense, but we will use the phrase for the general case as well. For $A, B \subseteq V$ the (A,B)–shortest path problem asks for the shortest path (and its length) from s to t, for very s $\in A$ and $t \in B$. The problem is well-defined if the following assumptions are in effect:

(*) for every $s \in A$ and $t \in B$ there is a path from s to t, and

(**) no path from s to t contains a cycle of negative weight (a "negative" cycle).

Traditionally the ($\{s\}$,V) and (V,V) shortest path problems have received most attention, and we will restrict ourselves to it here. The problems are known as the "single source" and the "all pairs" shortest path problem, respectively.

2.3.1 Single Source Shortest Paths.

Let s be a fixed source node and let $(*), (**)$ be in effect. Write $l(t)$ for $l(s,t)$, $l(s) = o$. Define a shortest path tree (SP tree) to be any spanning tree rooted at s such that the tree-path from s to t is a shortest path in G, for every $t \in$ V.

Theorem. With $(*), (**)$ in effect, there exist $SP-$trees for every s. If the values $l(t)$ are known $(t \in V)$, then a $SP-$tree can be constructed in $O(e)$ time.

The theorem shows that we can restrict the problem to computing the weights $l(t)$. In order to do so, it is common to view the $l(t)$ as solutions to a set of equations, known as Bellman's equations:

$$\begin{aligned} u_s &= o, \\ u_t &= min_{x \neq t}\{u_x + w(x,t)\} \text{ for } t \neq s. \end{aligned}$$

(the u_t for $t \neq s$ are the unknowns). It can be shown that when G has no cycles of weight $\leq o$, then the solution $l(t)$ to Bellman's equations is unique. Otherwise it is unique under the additional assumption that $\sum_t u_t$ is maximal, over all solutions. Thus the shortest path lengths are the solution to the following linear programming problem (set $w(x,y) = \infty$ for $(x,y) \notin E$):

$$\begin{aligned} \text{maximize} &: \textstyle\sum_t u_t \\ \text{subject to} &: u_s = o, \\ &\quad u_t \leq u_x + w(x,t) \text{ for } t \neq s, x \end{aligned}$$

We will not explore this connection to linear programming, but several solution methods of Bellman's equation directly fit in with this theory. Nevertheless, the characterization suggests the following general strategy for computing a solution and a $SP-$tree (with $f(t)$ pointing to the father of t):

initialize : set $u_s = o, u_t = \infty$ for $t \neq s, f(t) =$ **nil** for all t.
algorithm : **while** not all inequalities are satisfied **do**
begin
scan : determine a $t \neq s$ for which there is an $x \neq t$ with $u_t > u_x + w(x,t)$;
label : set u_t to $u_x + w(x,t)$ and $f(t)$ to x
end

The algorithm is due to Ford and known as the "labeling (and scanning) method". It can be shown that the algorithm converges regardless of the choice made in the scanning step, provided $(*)$ and $(**)$ are in effect. Some scanning orders may be better than others however, and this distinguishes the various algorithms known for the problem. (Some scanning orders may lead to exponential time computations.) A simple example is provided by the acyclic graphs. Let an acyclic graph be given in topologically sorted order. Considering the nodes t in this order in Ford's algorithm, must produce the (final) value of u_t when its turn is there. The algorithm will finish in $O(e)$ steps.

Another efficient scanning order exists when $w(x,y) \geq o$, although it takes more work per step to pull the right node t out (i.e, out of a suitable data structure). There will be a set F of nodes t for which u_t already has a final value, and a set I of nodes t for which u_t is not known to be final. Initially $F = \{s\}$ and $I = V - \{s\}$. The algorithm is slightly changed as follows:

initialize : set $u_s = o, u_t = w(s,t)$ for $t \neq s, f(t) = s$ for all t;
algorithm : **while** I is not - $\emptyset$ **do**
begin
scan : choose $t \in I$ for which u_t is minimal ;
$F := F \cup \{t\}$; $I := I - \{t\}$;
update: for every neighbor x of t, $x \in I$, set u_x to
$min\{u_x, u_t + w(x,t)\}$; set $f(x)$ to t
when $u_t + w(x,t)$ was smaller than u_x
end

The algorithm is due to Dijkstra, and operates on the observation that node closest to s by a path through the current fragment of the final $SP-$tree(F) must be exposed in the scanning step. The algorithm is easily seen to require $O(e)$ steps, except for the operations for maintaining a priority queue of the u_x-values (selection and deletion of an element in the scanning step, and deg(t) priority updates in the update step). A simple list will lead to $O(n)$ steps per iteration, thus $O(n^2)$ total time, and a standard priority queue to $O(\deg(t).\log n)$ steps per iteration, hence $O(e \log n)$ steps total. By using a refined data structure this can be improved to $O(e \log n / \log \max\{2, \frac{e}{n}\})$ steps, and even to $O(e + n \log n)$ steps. When the edge-weights all belong to a fixed range o..K, the algorithm can be implemented to run in $O(\min\{(n+e)\log\log M, nM + e\})$.

Assuming (*) and (**), there is a good scanning order for general graphs as well: assume a fixed ordering of the nodes $\neq s$, and make consecutive passes over the list. Observe that it takes $O(\deg(t))$ time to see if a node t must be selected and its u_t updated accordingly, and one pass takes $O(e)$ time. By induction one sees that in the k^{th} pass all nodes t for which a shortest path from s exists with k edges, must get their final u_t-value. Thus in $O(n)$ passes all $u-$values must converge, and the algorithm finishes in $O(ne)$ time. Observe that for planar graphs this gives an $O(n^2)$ algorithm, because $e \leq 3n - 6$ (for $n \geq 3$). By a partitioning technique based on the planar separator theorem, this can be improved to $O(n^{1.5} \log n)$ time for planar graphs (Mehlhorn & Schmidt). For general graphs an $O(hn^2)$ algorithm has been shown, where h is the minimum of n and the number of edges with negative weight (Yap).

2.3.2 All Pairs Shortest Paths.

The all pairs shortest path problem has been subjected to a wide variety of solution methods, and illustrates better than any other graph problem the available techniques in the field. We assume throughout that the conditions (*) and (**) are in effect, in particular we assume that there are no negative cycles in the graph. One approach to the all pairs shortest path problem is to solve n single source shortest path problems, one problem for every possible source. The resulting complexity bounds are $O(ne + n^2 \log n)$ for graphs with non-negative edges and $O(n^2 e)$ for general graphs, using the results from the previous Section. By the Theorem it takes another $O(ne)$ time to construct all $SP-$trees.

By an ingenious device, Edmonds & Karp observed that the all pairs shortest path problem for general graphs can be reduced to the problem for graphs with non-negative edge-weights. Let $f : V \to \mathbf{R}$ be a function such that for all $x, y \in V : f(x) + w(x,y) \geq f(y)$ and consider the graph $G' =< V, E, w' >$ with $w'(x,y) = f(x) + w(x,y) - f(y)$. One easily verifies that $l'(s,t) = l(s,t) + f(s) - f(t)$ for $s, t \in V$, where $l'(s,t)$ is the length of a shortest path from s to t in G'. As G' has only non-negative weights, the all pairs shortest

path problem is solved in $O(ne + n^2 \log n)$ steps for G. A suitable function f can be obtained by considering the graph G'' obtained from G by adding a (new) source z and edges (z, x) for all $x \in V$, and taking $f(x) = l''(x)$ (the length of the shortest path from z to x in G''). Computing f takes one single source shortest path problem computation on a general graph, and (thus) no more than $O(ne)$ time. The total computation still takes no more than $O(ne + n^2 \log n)$ time.

A variety of methodes for the all pairs shortest path problem is based on the use of a variant of the adjacency matrix A_G defined by $A_G(i,j) = w(i,j)$ for $i, j \in V$, with $w(i,i) = 0$. (We assume that the nodes are numbered $1, \cdots, n$ here.) Consider matrices $A = (a_{ij}), B = (b_{ij})$ over the reals and define $A \times B = C = (c_{ij})$ to be the matrix with $c_{ij} = \min_{1 \leq k \leq n}(a_{ik} + b_{kj})$, a term reminiscent of matrix product with operations min and + for + and $\cdot$. It can be shown that all algebraic laws for matrix multiplication hold for the (min, +) product. Observe that $A_G^k = A_G \times \cdots \times A_G$ (k times) is the matrix with $A_G^k(i,j)$ equal to the length of the shortest path from i to j with $\leq k$ edges. Define the transitive closure of any matrix A as $A^* = \min\{I, A, A, A^2, A^3, \cdots\}$ with I the matrix with o on the main diagonal and ∞ elsewhere, and min taken component-wise. Clearly A_G^* is the matrix with $A_G^*(i,j) = l(i,j)$ and, in the absence of negative cycles, $A_G^*(i,j) = A_G^{n-1}(i,j)$.

The conclusions in the preceeding paragraph give rise to several interesting algorithms and complexity considerations. For example, the Floyd-Warshall algorithm for computing transitive closures can be adapted to computed the all pairs shortest paths in $O(n^3)$ steps. A beautiful result of Romani shows that every $O(n^\alpha)$ matrix multiplication algorithm (over rings) can be adapted to compute the transitive closure of any A_G in $O(n^\alpha)$ steps, provided A_G has elements from $\mathbf{Z} \cup \{\infty\}$. Watanabe (with corrections) has shown that in this case a suitable representation of the actual shortest paths can be computed in $O(n^\alpha)$ time as well. The representation is a matrix $\tilde{A}_G$ with $\tilde{A}_G(i,j)$ equal to a node $x \neq i, j$ on a shortest path from i to j.

Several algorithms have been proposed that compute the all pairs shortest path information probabilistically fast, over a large class of random graphs. The best complexity bound so far is $O(n^2 \log n)$ expected time (Moffat and Takaoka).

2.4 Paths and Cycles.

For most of this Section we assume that G is an (unweighted) undirected graph, although for most problems discussed there also is a "directed" version. Whenever we speak of a path and a cycle, we shall mean a simple path and a simple cycle (or circuit) respectively. Paths and cycles have been a predominant issue in the analysis of graphs for ages. Consequently we can touch only on a number of topics under this category.

2.4.1 Paths of Length k.

Probably the simplest question is, given k, whether there exists a (simple) path of length k from i and j. Algorithmically the problem can be solved in $O(k(n-2)^{k-1})$ time by considering all possible choices of $k-1$ intermediate nodes, and verifying that a sequence is a path. An efficient solution is surprisingly hard to obtain. In fact, with k arbitrary the problem is NP–complete and (thus) probably not polynomial time computable. Monien has proved the following result.

Theorem. Let G be (directed or undirected) graph, $k \geq o$. Then a path of length k

can be computed for all pairs i, j (λ for the pairs for which no path of length k exists) in time $O(k!ne)$.

By applying the theorem, one can find a longest path in G in $(\mu! \log \mu . ne)$ with μ the length of the longest path: search for paths of length k for $k = 1, 2, 4, \cdots$ and use binary search between 2^r and 2^{r+1} if $k = 2^r$ was the last value for which a path was found.

2.4.2 Disjoint Paths.

We shall only consider vertex-disjoint paths, although the versions for edge-disjoint paths are interesting as well. The problem of finding a maximum set of vertex-disjoint paths between two nodes i and j is of obvious interest in many network problems. The (few) algorithms for the problem make use of the connection to a $0-1$ maximum flow problem, which makes it polynomial in n and e.

Lemma. The number of vertex-disjoint paths between i and j may be found as the maximum flow in an augmented (directed) graph G^+ with $2n$ nodes, $2e + n$ edges, and edge capacities 1.

Proof.

Let $G^+ =< V^+, E^+, c >$ be the (directed) graph defined as follows. For every $x \in V$ let there be two nodes $x', x'' \in V^+$ and an edge $(x'', x') \in E^+$. For every (undirected) edge $(x, y) \in E$ let there be two edges $(x', y''), (y', x'') \in E^+$. All edges $\alpha \in E^+$ have capacity $c(\alpha) = 1$. If there are k vertex-disjoint paths from i to j in G, then it is seen that there can be a flow of k from i to j in G^+. Consider a maximum flow in G^+. By the integer max-flow theorem there is a maximum flow that is integral on all edges, hence 0 or 1 in this case. By tracing the 1's that flow out of i, one necessarily obtains max-flow vertex-disjoint traces in G^+, which translate to vertex-disjoint paths in G. □

Frisch has exploited the Lemma for the construction of a maximum set of vertex-disjoint paths, see also Steiglitz & Bruno. Clearly the algorithm is too complex e.g. if one only wants to find k vertex-disjoint paths between i and j for some k (provided they exist), but by exploiting the connection to integer flow it should be clear that in this case one only needs to find (and trace) k augmenting paths to get a flow of $\geq k$. Suurballe extended the connection to a minimal cost flow problem (see Section 3.5.2) and derived an algorithm to find a set of k vertex-disjoint paths with minimal total length. No complexity analysis was given. Menger's theorem gives a necessary and sufficient condition for the existence of k vertex-disjoint paths.

A simple and practical generalisation of the problem is the following. Let l pairs $(i_1, j_1), \cdots, (i_l, j_l)$ be given. Does G contain l vertex-disjoint paths, one connecting i_k and j_k for every $1 \leq k \leq l$. (There are many variants, e.g. one may wish to have more than one connecting vertex-disjoint path for every pair.) With G and l arbitrary, the problem is $NP-$complete. (It is a very simple instance of the multicommodity flow problem.) For fixed $l \geq 2$ little is known about the complexity of the problem. For $l = 2$ Shiloach obtained an $O(ne)$ algorithm, by a tedious analysis. The problem is an interesting example where graph theory is advanced because of the need of an efficient algorithm. Shiloach was able to conclude that in every 4-connected non-planar graph there always are 2 connecting

vertex-disjoint paths, for every choice of i_1 and j_1 and i_2 and j_2. Seymour observed that the general problem is equivalent to the following. Define a set of edges $E' \subseteq E$ to be $G-$separable if for each $\alpha \in E'$ there is a (simple) cycle C_α containing α and $C_\alpha \cap C_\beta = \emptyset$ for $\alpha \neq \beta$. Under what conditions is a set $G-$separable. Cypher proved that for $l \leq 5$ the problem is solvable in polynomial time for graphs that are $l+2$- connected.

2.4.3 Cycles.

Throughout this Section we only consider undirected graphs. The cycle structure of graphs has inspired many studies and analyses, and a variety of types of cycles have been considered. Every cycle can be expressed as a sum of so-called fundamental cycles in G, where "addition($\oplus$)" is defined as follows : for $G_1 =< V_1, E_1 >$ and $G_2 =< V_2, E_2 >$, $G_1 \oplus G_2 =< V_1 \cup V_2, (E_1 \cup E_2) - (E_1 \cap E_2) >$. The cycle space of a graph has dimension $e - n + 1$, and it is known that for every spanning tree T the cycles obtained by adding an arbitrary non-tree edge to T are a fundamental set. A fundamental set of cycles can be generated easily during a DFS of a graph. Each time a red edge (i,j) is encountered while exploring node i, another fundamental circuit is closed and can easily be output (because j is an ancester of i) in $O(n)$ steps. The method is due to Paton and runs in $O(n(e-n+1))$ time. Finding specific cycles by enumerating the $2^\gamma - 1$ non-trivial combinations of fundamental cycles ($\gamma = e-n+1$) is usually inefficient. Dixon & Goodman use a branch-and-bound search in the cycle space to find a longest cycle in the graph. (See later.)

The most common problem is that of finding cycles with a length constraint : specified length (k), shortest, longest, odd length, even length. Consider first the simplest case of all, namely, finding a cycle of length 3 (a triangle). An $O(ne)$ algorithm is obtained by inspecting every combination of an edge (i,j) and a node x, and deciding in $O(1)$ time using the adjacency matrix A_G whether (i,x) and (j,x) are edges. A possibly better worst-case bound results by observing that triangles of G correspond to off-diagonal ones in the (boolean) matrix $A_G^2 \wedge A_G$, where $\wedge$ denotes the element-wise "and". The complexity is bounded by $O(n^\alpha)$, with α the exponent of a fast matrix multiplication algorithm. More intriguing is the following observation. Let T be any rooted spanning tree of G (or, a spanning forest in case G is not connected).

Lemma. *G has a triangle which contains a tree-edge if and only if G has a non-tree edge (x,y) for which $(father(x), y) \in E$.*

The Lemma suggests the following algorithm : find a spanning tree (forest) of G, test every non-tree edge in $O(1)$ time per edge, and delete the tree-edge and repeat the same procedure on the resulting graph if no triangle was found. Itai & Rodeh show that at most $2\sqrt{e}$ iterations will lead to a graph of isolated vertices, and (thus) the algorithm is $O(e^{\frac{3}{2}})$ time bounded. Observing that $e \leq 3n-6$ ($n \geq 3$) for planar graphs, each iteration removes at least $\frac{1}{3}$ of the edges of G in the planar case and the total algorithm finishes in $O(n)$ time (in this case). For planar graphs Richards gives an $O(n)$ algorithm for finding a 4-cycle, as well as $O(n \log n)$ algorithms for finding a 5-cycle and a 6-cycle.

Algorithms to find a shortest cycle in general, usually rely on a form of "truncated BFS". Let i be the first node during BFS from x where a red edge is created. Let the edge be (i,j) and define $L_x = L(i)$. Observe that necessarily $L(j) = L(i)$ or $L(j) = L(i)+1$.

Let $l_o = \min_x L_x$ and x_o be such that $l_o = L_{x_o}$. Let k be the length of a shortest cycle in G, with $k = \infty$ if no cycle exist. l_o follows in $O(n^2)$ time (when defined).

Lemma. Let $k < \infty$. The red edge in the truncated BFS started at x_o closes a cycle of length k' with $k \leq k' \leq k+1$ (and $2l_o + 1 \leq k' \leq 2l_o + 2$).

Note that a cycle of length $2l_o + 1$ containing x_o can exist only if level l_o of the BFS tree contains two nodes connected by a (red) edge. Thus it is easily decided within $O(e)$ steps what the smallest cycle through x_o is. If G is bipartite and (thus) has cycles of even length only, this test can be skipped. Considering all possible choices for x_o, the smallest cycle of G is found in $O(ne)$ steps. When G is bipartite, $O(n^2)$ steps suffice. Monien has shown that the technique can be modified to find the smallest cycle of odd length in $O(ne)$ time and the smallest cycle of even length in $O(n^2 \min\{\alpha(n,n), \lambda\})$ where $\alpha(m,n)$ is an "almost constant" function (cf. UNION-FIND programs) and λ the length of the shortest even cycle.

Note that deciding whether level l_o of the truncated BFS tree of x_o contains two nodes i and j connected by an edge can be formulated as the problem of finding a triangle $\triangle(x_o, i, j)$ in a collapsed version of the tree. Itai & Rodeh show that in $O(n^2)$ time a graph G' of at most $2n$ nodes can be built with the property that G contains a cycle of length $2l_o + 1$ (and thus has a shortest cycle of this length) if and only if G' contains a triangle. Triangle finding was discussed above.

Little is known about the complexity of finding cycles of a specified length except for Monien's result (see 2.4.1). His techniques lead to an algorithm for finding a longest cycle in $O((2\mu)!ne)$ time, where μ is the length of the longest cycle of G. Note that the longest cycle problem in graphs is $NP-$complete. Another problem asks for the existence of a cycle though k specified vertices. The problem is believed to be polynomial for fixed k, but this has been proved only for $k = 2$ and $k = 3$ (LaPaugh). In directed graphs the problem is $NP-$complete already for $k = 2$. Yet another problem asks for determining a set F of at most k nodes such that F contains at least one vertex from every cycle in G. (F is called a feedback vertex set.) The problem is $NP-$complete, but for fixed k it can be solved in $O(n^{k-1}e)$ time. (Note that an algorithm of $O(n^k e)$ time results if one considers every subset of k vertices and tests whether removing these vertices yields an a cyclic graph.)

2.4.4 Weighted Cycles.

Next we let $G =< V, E, w >$ be a weighted graph and consider the problem of finding cycles with a length and/or a weight constraint. If all edge weights are $\geq o$, then it is not hard to modify any of the existing shortest path algorithms to find a cycle of least (total) weight in polynomial time. Monien shows that the least cost cycle of odd length can be computed in $O(n^3)$ time. If edge weights $< o$ are allowed, the problem of finding a least weight cycle is $NP-$complete. However, in this case there are polynomial time algorithms for finding just a negative cycle (if one exists). The problem of finding a negative cycle in a graph is interesting e.g. in the context of shortest path finding or applications in optimization theory. An early heuristic is due to Florian & Robert and based on the following observation. (We now assume that G is directed.)

Lemma. Every negative cycle C contains a vertex i such that the partial sums of the edge weights along the cycle starting and ending at i are all negative.

By applying a branch-and-bound strategy at every node i, one hopes to be able to "guide" the search quickly to negative cycles (if there are). The method seems to perform well in practice, but Yen proved that on certain graphs the algorithm does very badly.

More interesting techniques are based on shortest path algorithms for general graphs (see 3.4). Recall that these algorithms assume that no negative cycles exist only for the sake of valid termination. Thus, if e.g. in Ford's algorithm applied to a source s no termination has occurred after $O(ne)$ steps, then there must be a negative cycle accessible from s (and vice versa). In fact, a negative cycle is detected as soon as some node x is added into the (current fragment of the) SP tree that appears to be its own ancestor (i.e., x already occurs earlier on the path towards the root). A direct and (hence) more efficient approach was developed by Maier, by building a purported SP tree in breadth-first fashion as follows. Suppose we have a tree T which is an SP tree of paths from s of length $\leq k-1$. In stage k we look for paths of length k to nodes that were visited in earlier stages that are less costly than the paths of length $\leq k-1$ discovered so far. To this end, consider the "frontier" nodes i at depth $k-1$: if $(i,j) \in E$ and $u_j > u_i + w(i,j)$ then we clearly have a "better" path via i to j. If j is an ancestor of i, then we have a negative cycle. Otherwise we purge whatever subtree was attached to j, and make j a son of i. (The nodes of the subtree will automatically come in again with better paths at later stages.) Repeat this for all i, except those for which a better path was found (they're moved to the next stage). The algorithm requires "fast" tree-grafting and ancestor-query techniques. Tsakalidis has shown that this can all be handled in $O(n+e)$ time and $O(n+e)$ space.

2.4.5 Eulerian (and other) Cycles.

Define an Eulerian cycle to be a (non-simple) cycle that contains each edge of G exactly one. An Eulerian path (or covering trail) is defined to be a (non-simple) path with the same property. An excellent survey of results was given by Fleischner. The characterization of Eulerian graphs is classical.

Theorem. Let G be connected. The following conditions are equivalent.

(i) G contains an Eulerian cycle,

(ii) each node of G has even degree,

(iii) G can be covered by (edge-disjoint) simple cycles.

Testing G for the existence of an Eulerian cycle is simple by (ii), but efficient algorithms to find Eulerian cycles all seem to be based on (iii) and the following observation. Define a (maximal) random trail of untraversed edges from x to be any trail that starts at x and takes a previously untraversed edge each time a node is reached, unless no such edge exists (in which case the trail ends at this node).

Lemma. Let G be connected. The following conditions are equivalent:

(i) G contains an Eulerian cycle,

(ii) for each node x, every random trail from x ends at x (even if at some nodes an even number of edges is eliminated at the beginning).

Proof.

(i) $\Rightarrow$ (ii). Because all degrees are even, every node $\neq x$ that is reached over an edge can be left again. Thus eventually the trail must return to x. If there are untraversed edges incident to x, the trail is continued. Eventually the trail must end, necessarily at x.

(ii) $\Rightarrow$ (i). It is not hard to argue (by induction on $|G|$) that (ii) implies that G is covered by edge-disjoint cycles. Apply theorem (iii). □

The simplest approach to finding an Eulerian cycle is to pick an node x and follow a random trail. With some luck the trail will be Eulerian, but this need not be so. Ore has shown that every random trail from x is an Eulerian cycle if and only if x is a feedback vertex (i.e., x lies on every cycle). He also noted that graphs with this property are necessarily planar. If the random trail is not Eulerian, it must contain a node y which still has an (even) number of untraversed edges. One can now enlarge the random trail as follows: insert a random trail of untraversed edges from y at some point in the random trail from x where y is reached. Repeating this will eventually yield an Eulerian cycle (or a proof that none exists). The algorithm is due to Hierholzer, and is easily implemented in $O(n + e)$ steps by linked list techniques. The algorithm still has the (minor) defect that it does not produce an Eulerian cycle in traversal order immediately. But a simple modification of DFS will. Start from a node i_o and follow previously untraversed edges (i.e., a random trail) until a node (i_o) is reached where one cannot proceed. Backtrack to the last node visited on the path which still has untraversed edges and proceed again. Repeat it until one backtracks to the very beginning of the path. One can observe that, when G is Eulerian, outputting edges in the order in which they are traversed the second time (i.e., when backtracking) produces an Eulerian cycle in traversal order. Rather similar results are known for Eulerian paths, and for the case of directed graphs.

In case a graph is not Eulerian, one might ask for a shortest non-simple cycle containing every edge (necessarily, some edges more than once). Any cycle of this sort is called a "postman's walk" and the problem is known as the Chinese postman problem, after Kwan Mei-Ko.

Theorem. Let G be connected. The following conditions are equivalent:

(i) P is a postman's walk,

(ii) the set of edges that appear more than once in P contains precisely a minimum number of edges that need to be doubled in order to obtain an Eulerian graph,

(iii) no edge occurs more than twice in P and for every cycle C, the number of edges of C that occur twice in P is at most $\frac{1}{2}|C|$.

More generally one can consider weighted graphs $G =< V, E, w >$ and ask for a postman's walk of minimum weight (or "length"). The following result is due to Edmonds & Johnson. (It is assumed that weights are non-negative.)

Lemma. The Chinese postman problem can be solved by means of an all-pairs shortest

path computation, solving a minimum weight perfect matching problem, and tracing an Eulerian cycle in an (Eulerian) graph.

Proof.

In analogy to theorem (ii) the minimum cost set of edges that must be doubled in G to make G "chinese postman optimal" must be the union of minimum cost paths connecting nodes of odd degree, and (conversely) any union of this kind added to G will yield an Eulerian graph. Thus, let A be the set of nodes of odd degree and solve the (A, A) shortest path problem in G. To select a minimum cost set of paths, design a complete graph G' on node-set A with $w'(i,j)$ equal to $d(i,j)$ for $i,j \in A$ (the shortest distance between i and j in G), and determine a minimum weight perfect matching in G'. (Note that necessarily $|A|$ even.) For every edge (i,j) of the matching, double the edges of the shortest path from i to j in G. Tracing an Eulerian cycle in the resulting graph will yield an (optimal) postman walk. □

It follows (see Chapter 3) that the Chinese postman problem is solvable in polynomial time. A similar result holds in the directed case, but quite surprisingly the problem is $NP-$complete for mixed graphs.

Considering theorem (iii) one might wish to approach non-Eulerian graphs as follows. One problem is to determine a maximum set of edge-disjoint cycles in G (a "cycle packing"). In the weighted case one asks for a cycle packing of maximum total weight. Meigu proves that this problem is equivalent to the Chinese postman problem (hence polynomially solvable). Another problem asks for a minimum set of cycles such that each edge is contained in at least one cycle from the set (a "cycle covering"). In the weighted case one asks for a cycle covering of minimum total weight. Itai et al. have shown that every 2-connected graph has a cycle covering of total length at most $min\{3e, 6n+e\}$, and that this covering can be found in $O(n^2)$ time. (No results seem to be known for the weighted case.)

2.4.6 Hamiltonian (and other) Cycles.

Define a Hamiltonian cycle to be a (simple!) cycle that contains each node of G exactly once. Define a Hamiltonian path to be any (simple) path with the same property. Unlike the "Eulerian" case, there appears to be no easily recognised characterisation of Hamiltonian graphs. Thus all traditional algorithms to find a Hamiltonian cycle in a graph are based on exhaustive search from a (random) starting point. When a partial cycle $i_o, i_1, \cdots, i_{j-1}$ has been formed and i_j (a neighbour of i_{j-1}) is the next node tried, then the path is extended by i_j and one of its neighbours is considered next if i_j did not already occur on the partial cycle. If this neighbour happens to be i_o and all nodes have been visited, then a Hamiltonian cycle is formed. Otherwise we just consider the neighbour like we did i_j. If i_j did occur on the partial cycle, then we back-up to i_{j-1} and consider another untried neighbor. If there are no more untried neighbors, then we back-up to i_{j-2} and repeat. The algorithm ends with a Hamiltonian cycle if one exists, otherwise it ends with all neighbors of i_o "tried". Rubin discusses a variety of techniques to design a more efficient branch-and-bound algorithm for the problem.

Lemma. For every graph G there is a graph G' with $n+1$ nodes and $e+n$ edges such

that G has a Hamiltonian path if and only if G' has a Hamiltonian cycle.

Proof.

Let s be a new node, and let $G' =< V \cup \{s\}, E \cup \{(s,i)|i \in V\} >$. The lemma is now easily verified. □

There is an interesting analogy between the lack of an easily tested criterion for the existence of a Hamiltonian cycle and the lack of an efficient (viz. polynomial time) algorithm for finding a Hamiltonian cycle in a graph. In fact, even knowing that a Hamiltonian cycle exists does not seen to be of much help in finding one. Of course the Hamiltonian cycle is a "classical" example of an $NP-$complete problem. The problem is even $NP-$complete for rather restricted classes of graphs and demonstrates the "thin" dividing line that sometimes exists between tractable and untractable problems. We mention only two examples.

(i) It is not hard to see that the Hamiltonian cycle problem is polynomial for directed graphs with all out-degrees or all in-degrees at most 1. Plesnik has shown that the problem is $NP-$complete already for planar directed graphs with in-degrees and out-degrees at most 2.

(ii) Garey, Johnson & Tarjan have shown that the Hamiltonian cycle problem is $NP-$-complete for 3-connected planar graphs. On the other hand a beautiful theorem of Tutte asserts that every 4-connected planar graph must be Hamiltonian (although this does not imply in itself that the cycle is easy to find!). Gouyou - Beauchamps has shown that in the later case the problem is indeed polynomial-time bounded.

Like in the "Eulerian" case (see Section 2.4.5), several approaches have been considered for non-Hamiltonian graphs. Define the Hamiltonian completion number $hc(g)$ to be the smallest number of edges that must be added to G to make it Hamiltonian. Goodman, Hedetniemi, & Slater prove that the Hamiltonian completion number can be computed in $O(n)$ time for graphs with (at most) one cycle. (Of course the problem is $NP-$complete in general.) Rather more interesting perhaps is the approach in which one tries to minimize the number of "double" visits. Define a Hamiltonian walk (or shortest closed spanning walk) to be any minimum length cycle that contains each node of G at least once. If G admits a closed spanning walk of length $n+h$, then we say that G has Hamiltonian excess bounded by h. (Clearly h is equal to the number of nodes visited twice in the walk.) One can show that $h \leq 2n - \lfloor \frac{k}{2} \rfloor (2d-2) - 2$ for $k-$connected graphs of diameter d.

Lemma. Let W be a Hamiltonian walk. Then

(i) no edge of G appears more than twice in W, and

(ii) for every cycle C, the number of edges that appear twice in W is at most $\frac{1}{2}|C|$.

Proof.

(i) Suppose (i,j) occurs ≥ 3 times in W. Then there are (possibly empty) subwalks W_1, W_2, W_3 and W_4 such that W can be written in one of following forms: $(a) W_1 ij W_2 ij W_3 ij W_4$, $(b) W_1 ij W_2 ji W_3 ij W_4$, and $(c) W_1 ij W_2 ij W_3 ji W_4$. In case $(a) W_1 i \overline{W}_2 j W_3 ij W_4$ would be a shorter walk, contradicting the minimality of W. $\overline{W}_2$ denotes reverse of $W2$.) Similar contradictions are obtained in the other cases.

(ii) Let $G_W =< V, E_W >$ be the (multi-)graph obtained from G by only including the edges traversed by W, taking an edge "twice" if it is doubled by W. G_W is Eulerian. Suppose C is a cycle such that more than $\frac{1}{2}|C|$ edges of C appear twice in W (hence in G_W). Delete one copy of every "doubled" edge of C from G_W, but add every non-traversed edge of C to it. G_W is still connected and has all even degrees, and hence is again connected. The Eulerian cycle in the (modified) graph G_W traces a closed spanning walk of G that is shorter than W. Contradiction. □

The following result due to Takamizawa, Nishizeki & Saito gives a sufficient condition for the Hamiltonian excess to be bounded by $n - c$, for given c. (By traversing e.g. a spanning tree one observes that the Hamiltonian excess is always bounded by n.) The result is remarkable, because for $c = n$ (excess o, i.e., the case of a Hamiltonian graph) it subsumes several known sufficient conditions for the existence of Hamiltonian cycles.

Theorem. Let G be connected, $n \geq 3$ and $o \leq c \leq n$. Suppose there exists a labeling $i_1, \cdots, i_n$ of the nodes such that for all j, k:

$j < k, (i_j, i_k) \notin E, \deg(i_j) \leq j$, and $\deg(i_k) \leq k - 1 \Rightarrow \deg(i_j) + \deg(i_k) \geq c$

Then G contains a Hamiltonian walk of length $\leq 2n - c$ (i.e, excess $\leq n - c$).

Takamizawa et al. prove that if the conditions of the theorem are satisfied, then a closed spanning walk of length $\leq 2n - c$ can be found in $O(n^2 \log n)$ time. (For $c = n$ this would necessarily be a Hamiltonian cycle.)

While the Hamiltonian cycle problem is $NP-$complete (thus "hard") in general, there may be algorithms for it that do quite well in practice. There are two ways to make this more precise: (i) by considering the problem for random graphs (with a certain edge probability) or, (ii) by averaging over all graphs with N edges (taking each graph with equal probability). The starting point are results of the following form: if $N \geq cn \log n$, then the probability that G is Hamiltonian tends to 1 for $n \to \infty$. (The sharpest bound for which this holds is $N \geq \frac{1}{2}n \log n + \frac{1}{2}n \log \log n + w(n)$ for $w(n) \to \infty$, cf. Bollobás.) Angluin & Valiant proposed the following randomized algorithm for finding a Hamiltonian cycle with high probability (averaged over all graphs with $N \geq cn \log n$). In the course of the algorithm explored edges will be deleted ("blocked") from G. Let s be a specified starting node and suppose we succeeded building a partial cycle C from s to x, some x. If the partial cycle contains all nodes of G and if we previously deleted (x, s) from G, then add the edge (x, s) to C and report success ($C$is now a Hamiltonian cycle). Otherwise, select (and delete) a random edge $(x, y) \in E$. If no edges (x, y) exist, then stop and report failure. If $y \neq s$ and y does not already occur on C, then add (x, y) to C and continue by exploring y by the same algorithm. If $y \neq s$ but y does occur on C (see figure 6), then locate the neighbour z of y on C in the direction of x, delete the edge (y, z) from C and add (y, x), modify the traversal order of the edges so z becomes the "head" of C, and continue exploring z. (The case $y = s$ is handled as in the opening clause of the algorithm.) By using suitable data structures each "step" of the algorithm can be executed in $O(\log n)$ time. It can be shown that for $N \geq cn \log n$, the probability that the algorithm reports success within $O(n \log n)$ steps tends to 1 for $n \to \infty$. The result implies that the randomized algorithm "almost certainly" finds a Hamiltonian cycle within $O(n \log^2 n)$ time, for $N \geq cn \log n$ (and averaged over all graphs with N edges). More

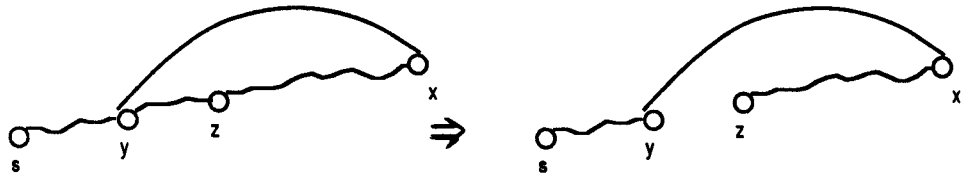


Figure 6:

recent results have lowered the threshold for N to $\frac{1}{2}n\log n + \frac{1}{2}n\log\log n + w(n)$ while still guaranteeing a polynomial time algorithm that is successful almost always. Similar results with rather more complicated algorithms hold for the directed case.

2.5 Decomposition of Graphs.

"Decomposition" is a broad term, and is generally used to indicate a technique for unraveling a graph in terms of simpler (smaller) structures. Traditionally graphs are decomposed into components with a certain degree of connectivity, or in a (small) number of subgraphs of a specified kind. For example, a connected graph can be decomposed as a "tree" of bi-connected components, and this decomposition can be computed in $O(n+e)$ time and space. The bi-connected components are the (unique) maximal connected subgraphs without cutpoints. The bi-connected components can be decomposed further into triconnected components (roughly speaking, the components without non-trivial 2-element separating sets) which can also be computed in $O(n+e)$ time and space by a rather more complicated DFS-based algorithm. Decompositions can be helpful in reducing problems for general graphs. As an example we mention Maclane's result that a graph is planar if and only if its triconnected components are.

Directed graphs can be decomposed as an "acyclic graph" of strongly connected components, again in $O(n+e)$ time by a suitable version of DFS. One can group strongly connected components into "weak components" W, which have the property that for any x, y either x and y belong to the same strongly connected component or there is no (directed) path from x to y nor from y to x. The weak components of a graph can again be identified in $O(n+e)$ time in a rather elegant fashion from the topologically sorted arrangement of the strongly connected components. Yet another notion is the following. Define a unilaterally connected component as a maximal subgraph U with the property that for any $x, y \in U$ there is a path from x to y or from y to x (or both). It is easily seen that the unilaterally connected components are maximal chains of strongly connected components. These chains can be identified in $O(n+e+nc)$ time when c is the number of chains to be output, provided one works with the transitive reduction of the underlying acyclic graph (otherwise non-maximal chains must be explicitly removed at extra cost). Finally, strongly connected components can be decomposed into a hierarchy of (sub)components as follows. In its general form the process assumes that edges have distinct, non-negative weights. Begin with the isolated nodes of C, and add the edges in order of increasing

weight one-at-a-time. Gradually strongly connected clusters are formed and merged into larger strongly connected components, until one strongly connected component (namely C itself) results. The process is represented by a tree in which every internal node v corresponds with a strongly connected cluster that is formed, its sons are the strongly connected clusters of which it is the immediate merge (after a number of edge insertions), and a label is assigned that is equal to the weight of the "last" edge that formed the component associated with v. Tarjan shows that strong component decomposition trees can be constructed in $O(e \log n)$ time.

Lemma. Any algorithm for constructing strong component decomposition trees can be used to construct minimum spanning trees of undirected graphs within the same time bound.

Proof.

Let $G =< V, E, w >$ be an undirected graph with distinct edge weights. Let G' be the directed graph obtained by taking a directed version in both directions of every edge of G. (G' is strongly connected.) The internal labels of any strong component decomposition tree correspond precisely to the edges of a minimum spanning tree of G. □

Another classical approach to decomposition involves the notion of a factor of a graph. A factor F is any non-trivial (i.e., not totally disconnected) spanning subgraph of G. If F is connected, then it is called a connected factor. A decomposition (or, factorization) of G is any expression of G as the union of a number of factors no two of which have an edge in common. Tutte proved that G can be decomposed into c connected factors if and only if $c \cdot (q(V, E - E') - 1) \leq |E'|$ for every $E' \subseteq E$, where $q(V, E)$ denotes the number of components of the graph $< V, E >$. Computational results on factors are few. Define an $H-$factor to be any factor that consists of a number of disjoint copies of H. The question whether G contains an $H-$factor is $NP-$complete for any fixed H that is not a disjoint union of $K_1's$ (single nodes) and $K_2's$ (complete graphs on 2 nodes). Note that for the existence of an $H-$factor it is required that $|H|$ divides $|G|$. The special case that $H = K_2$ can be recognized as the perfect matching problem, which is solvable in polynomial time. Define a k-factor ($k \geq 1$) to be any factor that is regular of degree k. The problem of computing a 1-factor is again another formulation of the perfect matching problem. 2−factors can also be identified by means of the perfect matching algorithm as follows.

Lemma. For every graph G there exists a (bipartite) graph G' with $2n$ nodes and e edges such that G has a 2−factor if and only if G' has a perfect matching.

Proof.

Define $G' =< V \cup V', E' >$ with $(i, j') \in E'$ iff $(i, j) \in E$ and no other edges in E'. (V' consists of nodes i' for $i \in V$.) Clearly, any 2-factor of G translates into a perfect matching of G' and vice versa. □

Note that the problem of finding a connected 2-factor is just the Hamiltonian cycle problem and (thus) $NP-$complete. Next, define an odd (even) factor to be any factor in which all nodes have odd (even) degree. Ebert showed that a maximal odd (even) factor of a graph (when it exists) can be found in $O(n + e)$ time by an application of DFS. Finally, a result due to Lovász asserts that for any $k-$connected graph G and partition

of n into $a_1, \cdots, a_k \geq 1, G$ admits a factor with k connected components $C_1, \cdots, C_k$ and $|C_i| = a_i\ (1 \leq i \leq k)$. For $k = 2$ an $O(ne)$ algorithm is known to compute a factor of this form.

In a related approach to decomposition one tries to partition the nodes rather than the edges of G. Define a (node-disjoint) covering of G by graphs of type X to be any collection of subgraphs $G_1, \cdots, G_k$ that are node-disjoint and of type X and together contain all nodes of G. (k is called the size of the covering.) In many cases the problem of determining whether a graph has a covering of size $\leq K$ is $NP-$complete, e.g. when G is a general undirected graph and X is the collection of triangles, circuits or cliques. Some interesting results are known when X is the collection of (directed) paths. The minimum size of a covering with node-disjoint paths is known as the path-covering number of G. The following result is quite straightforward, except part (ii) which is due to Noorvash.

Theorem.

(i) Unless G is Hamiltonian, the path covering number of G is equal to the Hamiltonian completion number of G.

(ii) A path-covering of G has minimum size if and only if it contains the maximum number of edges among all path-coverings of G.

(iii) If $g(n, K)$ is the smallest number such that every $n-$node graph with $\geq g(n, K)$ edges has a path-covering number $\leq K$, then $\frac{1}{2}(n-K)(n-K-1)+1 \leq g(n, K) \leq \frac{1}{2}(n-1)(n-K-1)+1$.

One concludes that the path-covering number of a graph is computationally hard to determine in general. Misra and Tarjan prove that it can be computed in $O(n \log n)$ time for rooted trees (with all edges directed away from the root), even in a weighted version. The following, more general result is due to Boesch and Gimpel.

Theorem. The path-covering number of a directed acyclic graph can be computed by means of maximum matching algorithm in a suitable bipartite graph.

Proof.

Construct a graph G' by replacing each node i by a pair of nodes i' and i'' such that all edges directed into i are directed into i' and all edges directed out of i are directed out of i''. One easily verifies that a minimum path-covering of G corresponds to a maximum matching of G', and vice versa. □

(In section 3.2 we will see that the maximum matching problem in bipartite graphs is efficiently solvable in polynomial time.)

The last approach we discuss is based on pulling a graph apart or "separating" it by cutting a small number of edges (or, deleting a few nodes). The approach is crucial in order to apply a divide-and-conquer strategy on graphs that admit this kind of decomposition. One approach is the following. Let's say $G =< V, E >$ "decomposes" if there is a non-trivial partition $V_1 \cup V_2 \cup V_3 \cup V_4$ of V such that for every $\alpha \in E$ there is an i such that α

connects two nodes in V_i or two nodes in V_i and V_{i+1}, and furthermore V_2 x $V_3 \subseteq E$ ("all nodes in V_2 are connected to all nodes in V_3"). When G "decomposes" it is completely determined by its "components" $G/(V_1 \cup V_2 \cup \{i_3\})$ and $G/(\{i_2\} \cup V_3 \cup V_4)$ for some $i_3 \in V_3$ and some $i_2 \in V_2$. Clearly one may try to decompose the components again. When a graph does not decompose this way, it is called "prime". A decomposition of a graph into prime components can be computed in $O(ne)$ time. Prime graphs with ≥ 5 nodes have several useful properties, e.g. they have no cutpoints and every two nodes i and j in it are non-similar (i.e., there is a node x that is adjacent to one but not to the other). Another approach is explicitly based on separation. Let G be connected, and define a $k-$separator to be any set $S \subseteq V$ with the property that the components of $G-S$ have at most k nodes each $(1 < k \leq n)$. The celebrated planar separator theorem asserts that planar graphs have "small" separators. Ungar proved that for every k there exists a planar $k-$separator of size $O(\frac{n}{\sqrt{k}} \log^{\frac{3}{2}} k)$, Lipton & Tarjan proved that there always exists a planar $\frac{2}{3}n$-separator of size $\leq \sqrt{8n}$ (which, moreover, can be determined in linear time). Djidjev improved the result slightly and showed that there always exists a planar $\frac{2}{3}n$-separator of size $\leq \sqrt{6n}$. Similar separation results exist for node-weighted graphs, in which the "weight" of the components after separation in bounded. Miller has shown that every 2-connected planar graph of which every face has at most d edges, has a $\frac{2}{3}n$-separator that is a simple cycle of size $\leq O(\sqrt{dn})$. Moreover, this separating cycle can be found in $O(n)$ time. There are several other classes of graphs besides the planar ones that be separated evenly by deleting only a "small" number of edges. Chordal graphs admit a $\frac{1}{2}n-$separator of size $O(\sqrt{e})$ that can be determined in linear time (Gilbert et al.). Graphs of genus g admit separators of no more than $O(\sqrt{gn})$ edges, which can be completed in $O(n+g)$ time.

2.6 Isomorphism Testing.

In many applications of graphs one or both of the following two problems arise : (i) given graphs G and H, determine whether G is isomorphic to a subgraph of H (the "subgraph isomorphism problem") and (ii) given graphs G and H, determine whether G is isomorphic to H (the "graph isomorphism problem"). Both problems are notoriously hard in general, and no polynomial time bounded algorithm is presently known for either of them. We will outline some of the more recent results.

2.6.1 Subgraph Isomorphism Testing.

The subgraph isomorphism problem is well-known to be $NP-$complete. Even in the restricted cases that G is an $n-$node circuit and H an $n-$node planar graph of degree ≤ 3 or G is a forest and H is a tree, the problem is NPcomplete. Matula has shown that the subgraph isomorphism problem can be solved in polynomial time when G is a tree and H is a forest, by the following simple technique. Assume that G is a tree with n nodes, H is a tree with m nodes $(n \leq m)$, and both G and H are rooted (all without loss of generality). Let the root of G have degree p and the root of H have degree q. Now execute the following (recursive) procedure **TEST.**

procedure TEST (G, H):
begin

1. delete the roots of G and H, and isolate the (rooted) subtrees $G_1, \cdots, G_p$ of G and $H_1, \cdots, H_q$ of H.

2. form a bipartite graph with nodes corresponding to G_1 through G_p and H_1 through H_q, and draw an edge between "G_i" and "H_j" if and only if **TEST** (G_i, H_j) returns true (meaning that G_i is a rooted subtree of H_j).

3. compute a maximum matching in the bipartite graph, to determine whether the rooted subtrees of G can be matched (mapped) to distinct, rooted subtrees of H.

4. if the outcome of step 3 is successful then return true, otherwise return false.

end.

The procedure is easily modified to actually give a concrete isomorphic embedding of G into H when it returns the value true. Suppose we have an $O(nm^\alpha)$ algorithm for computing a maximum matching in a (n, m)-bipartite graph ($n \leq m, \alpha > 1$). (Such algorithms exists e.g,. for $\alpha = 1.5$, see Chapter 3)

Theorem. The subgraph isomorphism problem for rooted trees can be solved in $O(nm^\alpha)$ time.

Proof.

Assume inductively that the problem can be solved in $\leq cnm^\alpha$ time for small n and m (c sufficiently large). The running time of **TEST** can be estimated by $cpq^\alpha + \sum_{i=1}^{p}\sum_{j=1}^{q} cn_i m_i^\alpha$, where $n_i = |G_i|$ and $m_j = |H_j|$. (We assume that the linear amount of time for step 1 of **TEST** is subsumed by the first term.) Using that $\sum_{i=1}^{p} n_i = n - 1$ and $\sum_{j=1}^{q} m_j = m - 1$, this is easily bounded by cnm^α. □

By using a faster maximum matching algorithm, the subtree isomorphism test can be improved as well (Reyner). Lingas has extended the result by showing that for $k-$trees G and H with n and m nodes, respectively, the subgraph isomorphism problem can be solved in $O(k \cdot k! n^{1.5} m)$ time.

For the subgraph isomorphism problem for more general classes of graphs very little is known. Lingas has studied the problem for classes of graphs that are $s(N)-$separable, for some function s (see section 2.5). A graph is called $s(n)-$separable if it either consists of one node or has a $\frac{2}{3}n-$separator of size $\leq s(n)$ whose removal disconnects the graph into two parts of (say) n_1 and n_2 nodes that are $s(n_1)-$separable and $s(n_2)-$separable, respectively. Even for $1-$separable graphs the subgraph isomorphism problem remains $NP-$complete.

Theorem. If G and H are $n-$node graphs that are $s(n)-$separable and have degrees $\leq d(n)$, then the subgraph isomorphism problem can be solved in $2^{O(\gamma(\log n + d(n))}$ time for $\gamma = \sum_{i=o}^{\lceil \log_{\frac{3}{2}} n \rceil} s((\frac{2}{3})^i n)$.

By using the planar separator theorem (see section 2.5) it follows, for example, that for planar graphs that are $\log n$-degree bounded the subgraph isomorphism problem can be

solved in $2^{O(\sqrt{n.\log^2 n})}$ time. Lingas has shown that for bi-connected outerplanar graphs the subgraph isomorphism problem can be solved in $O(nm^2)$ time.

In general there is hardly an alternative to the brute-force approach of enumerating all $n-$node subgraphs of H and testing for isomorphism with G or enumerating and testing all feasible embeddings of G into H in some suitable representation (see e.g. Bertziss). Ullman describes an interesting technique for pruning the systematic enumeration process.

2.6.2 Graph Isomorphism.

The problem of finding an efficient (i.e., polynomial time) algorithm for testing whether two $n-$node graphs G and H are isomorphic is of fundamental importance in the theory of graphs, but has withstood all attempts at a solution to date. The graph asomorphism problem is especially tantalizing because it is neither known to be $NP-$complete nor known to be polynomially solvable. There is at least some theoretical evidence that the graph isomorphism problem is not $NP-$complete because, if it were $NP-$complete, the Meyer-Stockmeyer polynomial hierarchy would collapse to its second level, which seems an unlikely event (Schöning). In this section we will sketch the main developments concerning the graph isomorphism problem only. For more background see Read and Corneil and the book of Hoffmann.

Let us look first at some "easy" cases. It is well-known that for rooted $n-$node trees the isomorphism problem can be solved in $O(n)$ time (see Aho, Hopcroft and Ullman, theorem 3.3). Weinberg first studied the isomorphism problem for planar graphs and obtaining an $O(n^3)$ algorithm for the case of tri-connected planar graphs, heavily relying on a theorem of Whitney asserting that a tri-connected graph has a unique embedding on the sphere. It was eventually shown that any two planar $n-$node graphs the isomorphism problem can be solved in $O(n)$ time (Hopcroft & Wong, Fontet). Lueker and Booth give an $O(n+e)$ algorithm for the isomorphism problem for interval graphs.

For most other classes of graphs the graph isomorphism problem quickly becomes very hard. It has been shown that the problem is polynomially equivalent to the problem of deciding for any graph G and a node v of G whether G admits an automorphism that "moves" v (Lubiw). Interestingly the problem of deciding whether G admits an automorphism that moves every node is $NP-$complete. To solve the general graph isomorphism problem several approaches have been followed. The older techniques are often based on a type of brute-force backtracking procedure (see e.g. Schmidt and Druffel, McKay). An interesting technique has been proposed by Deo et al., using a very natural incremental approach. Suppose the nodes of G are numbered from 1 to n (based on any reasonable scheme). At the kth level of the algorithm we always consider the subgraph $G(k)$ of G induced by the nodes 1 through k and a subgraph H' of H isomorphic to $G(k)$. Essentially the algorithm now proceeds as follows, exploiting a call of the (recursive) procedure **TEST** which we describe very informally.

```
procedure TEST (k, H'):
begin
    if k = n then return true
        else
        begin
            b := false ;
            {consider all possible extensions of H'..}
```

```
            while ¬b and there is an unexplored node v ∉ H' left do
            begin
                extend H' by v to obtain an (induced) graph H'';
                if the isomorphism between G(k) and H' can be
                extended to an isomorphism between G(k+1) and
                H'' then assign to b the value returned by TEST(k+1, H'')
                else "try the next v"
            end;
            return b
        end
    end
```

Deo et al. argue that for random graphs the algorithm has an expected computation time of $O(n \log n)$. Apparently the best worst-case algorithm for the graph isomorphism problem runs in $O(c^{n^{\frac{1}{2}+o(1)}})$ time (Babai, and Luks). For graphs of genus $\leq g$ the graph isomorphism problem can be solved in $O(n^{O(g)})$ time (Filotti & Mayer, Miller).

A second approach that has considerably advanced the understanding of the graph isomorphism problem has been based on the study of various types of automorphism groups associated with graphs. The connection has spurred considerably interest in the aspects of polynomial computability in groups (see e.g. Hoffmann). In fact the graph isomorphism problem can be reduced entirely to computational problems for suitable groups. Let Aut(G) denote the automorphism group of G. Consider the following series of problems.

GRAPH ISOMORPHISM : given two $n-$node graphs G and H, decide whether they are isomorphic.

REGULAR GRAPH ISOMORPHISM : given two regular $n-$node graphs G and H, decide whether they are isomorphic.

LABELLED GRAPH ISOMORPHISM : given two labelled $n-$node graphs G and H, decide whether they are isomorphic by means of a "label-preserving" isomorphism.

COMPLEMENT ISOMORPHISM : given an $n-$node graph G, is G isomorphic to its complement.

GRAPH ISOMORPHISM CONSTRUCTION : given two $n-$node graphs G and H, decide whether they are isomorphic and if so, construct an isomorphism from G to H.

GRAPH ISOMORPHISM COUNTING : given two $n-$node graphs G and H, determine the number of isomorphisms from G to H.

GRAPH AUTOMORPHISM WITH RESTRICTION : given an $n-$node graph G and a node v, decide whether G has an automorphism that moves v.

GRAPH AUTOMORPHISM GROUP : given an $n-$node graph G, determine a generating set for Aut(G).

GRAPH AUTOMORPHISM GROUP ORDER : given an $n-$node graph G, determine the order of $\mathrm{Aut}(G)$.

GRAPH AUTOMORPHISM PARTITION : given an $n-$node graph G, determine the orbits of $\mathrm{Aut}(G)$ on G.

The intriguing fact is that when any of these ten problems is solvable in polynomial time, then so are all the others. Combining results of especially Mathon, Babai, Booth, Lubiw and Colbourn & Colbourn one has the following fact.

Theorem. GRAPH ISOMORPHISM through GRAPH AUTOMORPHISM PARTITION are all polynomially equivalent.

While the theorem has set the scene for many studies in group-theoretic complexity theory, it does not in itself lead to all the benefits of the study of automorphism groups. A crucial type of result is the following. Let $\mathrm{Aut}_e(G)$ be the group of automorphisms of G that leave a particular edge e fixed.

Theorem. The isomorphism problem for trivalent graphs is polynomially reducible to the problem of determining a set of generators for $\mathrm{Aut}_e(X)$ where X is a trivalent connected graph and e a distinguished edge.

Luks succeeded in finding a polynomial time algorithm for the group-theoretic problem mentioned in the theorem, resulting in an $O(n^6)$ algorithm for the isomorphism problem for trivalent grapjs. More generally he proved that for every fixed d, the isomorphism problem for graphs of degree $\leq d$ can be solved in polynomial time. Currently an $O(n^3)$ probabilistic algorithm and $O(n^3 \log n)$ deterministic algorithm are known for the isomorphism problem for trivalent graphs (Galil et al.). Babai et al. have shown that for $n-$node graphs with eigenvalues of multiplicities $\leq k$, the isomorphism problem can be solved by an $O(n^{4k+O(1)})$ deterministic algorithm and by an $O(n^{2k+O(1)})$ probabilistic algorithm.

A third, and computationally very intriguing approach to the graph isomorphism problem is based on the use of "signatures". A signature is a (partial) mapping s defined on the set of $n-$node graphs such that for all graphs G : (i) if $s(G)$ is defined then so is $s(H)$ for all graphs H isomorphic to G and $s(G) = s(H)$, and (ii) if $s(H)$ is defined for some graph H and $s(G) = s(H)$ then G and H are isomorphic. Assume that all $n-$node graphs are defined on a standard node-set $\{v_1, \cdots, v_n\}$.

Lemma. Signature functions which are defined for all graphs exist.
Proof.

A classical example is due to Heap. For each graph G let $s(G)$ be the lexicographically largest $o-1$ vector that can be obtained by listing the nodes in some permuted order and concatenating the rows of the adjacency matrix of G corresponding to this order. □

Clearly Heap's signature function can be exceedingly hard to compute in general, although it may be reasonable for small graphs (Proskurowski). An interesting problem is to find efficiently computable, possibly partial signature functions that apply to large classes of graphs.

A number of studies have shown considerable success with signature functions that merely label the nodes of a given graph in some "canonical" way. Babai, Erdös and Selkow define an $O(n^2)$ time computable canonical labelling scheme s with the property that for random graphs $G, s(G)$ is defined with probability $\geq 1 - O(n^{-\frac{1}{7}})$. Karp has given an $O(n^2 \log n)$ time computable canonical labelling scheme s such that for random graphs G, $s(G)$ is even defined with a probability of at least $1 - O(n^{\frac{3}{2}}2^{-\frac{n}{2}})$. Babai and Kučera have shown that there are $O(n)$ time computable labelling schemes s with exponentially small failure probability. They also show the following remarkable result.

Theorem. There exists a canonical labelling scheme (a signature) defined for all $n-$node graphs that is computable in $O(n^2)$ expected time for random graphs.

Again group-theoretic considerations have crept into the study of signatures. For any fixed d, the class of graphs of degree $\leq d$ has a signature function that is polynomial time computable. For general graphs the best signature function known to date is computable in $O(c^{n^{\frac{2}{3}+o(1)}})$ time (Babai & Luks).

3 Combinatorial Optimization on Graphs.

The richest source of computational problems on graphs probably is the theory of combinatorial optimization, where the underlying structures usually are **networks**. Roughly speaking, a network is a graph in which the edges are labeled by (positive) edge-weights or capacities. The labels have a natural interpretation when certain transports are carried out over the edges of the graph. Traditionally two main areas have manifested themselves here:

(a) matching problems: determine a (maximum cardinality, maximum weight) set of edges $E' \subseteq E$ such that no two edges of E' are incident.

(b) flow problems: one or more commodities must be transported through the network, under constraints of maximum throughput and (perhaps) minimum cost.

Both matching and network flow are important graph-theoretic principles: many other problems can often be solved by restating ("reducing") these problems in terms of matching or flow problems. In all cases the task consists of maximizing a goal function, under suitable constraints.

3.1 Maximum Matching.

An edge belonging to a given matching M is called matched, the other edges are called free. If (x, y) is matched, then x and y are called "mates". A node incident to a matched edge is called matched as well, the other nodes are free or "exposed". There are two types of maximum matching:

(i) maximum cardinality matching, which asks for a matching of maximum size (and called perfect when all nodes are matched),

(ii) maximum weight matching, which asks for a matching of maximum total weight (assuming that fixed weights ≥ 0 are assigned to the edges).

A useful generalization is the notion of a "$b-$matching". Suppose (fixed) integers b_i are assigned to the nodes i such that $0 \leq b_i \leq \deg(i)$. A $b-$matching is a set of edges $E' \subseteq E$ such that at most b_i edges of E' are incident to node i, for all i. (Taking $b_i = 1$ results in an "ordinary" matching.) One can now ask for maximum cardinality and maximum weight $b-$matchings as before.

Lemma. For every graph G there is a (polynomial-sized) graph G' such that $b-$matchings in G correspond exactly to certain matchings in G'.

Proof.

Construct G' as follows. For each node $i \in V$, include $\deg(i)$ nodes $\alpha_i^{(j_1)}, \cdots$ corresponding to the edges $(i, j,), \cdots$ incident to i and $\deg(i) - b_i$ nodes $\beta_i^{(1)}, \cdots$. Connect all $\alpha_i^{(j)}$ nodes to all $\beta_i^{(k)}$ nodes, for every i, and connect $\alpha_i^{(j)}$ and $\alpha_j^{(i)}$ for all $(i, j) \in E$. We claim that $b-$matchings in G correspond precisely to matchings in G' that leave no β-node exposed. □

For computational purposes we like to have a "direct" reduction to maximum matching.

Construct the following graph G''. For each node $i \in V$ include b_i nodes $i^{(1)}, \cdots, i^{(b_i)}$ and for each edge $e \in E$ include two nodes x_e and y_e. For each $e = (i,j) \in E$ connect the nodes according to the following schema $S(e)$:

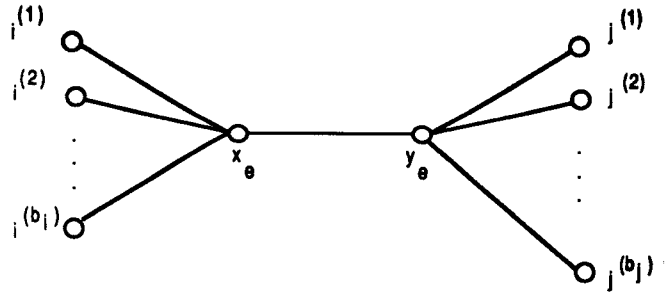


Figure 7:

Theorem. Let E' be a maximum cardinality $b-$matching in G, M a maximum cardinality matching in G''.

(a) $|M| = |E'| + e$,

(b) a maximum cardinality $b-$matching E_0 can be obtained from M by including every edge e for which $|S(e) \cap M| = 2$.

Proof.

Observe that a maximum matching M of G'' contains 1 or 2 edges from every $S(e)$. Let $E_0 = \{e \in E| \ |S(e) \cap M| = 2\}$. It follows that $|M| = |E_0| + e$. On the other hand, if we "choose" the edges from E_0, then we obtain a $b-$matching, hence $|E_0| \leq |E'|$ and $|M| \leq |E'| + e$. Conversely, given E', we can construct a matching M_0 of G'' as follows. If $e \notin E'$, then $(x_e, y_e) \in M_0$. If $e = (i,j) \in E'$ then include $(i^{(k)}, x_e)$ and $(y_e, j^{(l)})$ in M_0 for suitable k, l. Because E' is a $b-$matching, one can always choose k, l such that $i^{(k)}$ and $j^{(l)}$ are still exposed. It follows that $|M_0| = |E'| + e$. As $|M_0| \leq |M|$, we have $|M| = |E'| + e$, and the $b-$matching E_0 was maximum. □

A similar reduction applies to the weighted case. (Let all edges in $S(e)$ have weight $w(e)$.) The theorem shows that the techniques for maximum matching can be used to solve the more general maximum $b-$matching problem as well. We will see later that there are algorithms of low polynomial time complexity for it. (Note that G'' has $e + \sum_1^n b_i \deg(i) \leq 2ne$ edges and $2e + \sum_1^n b_i \leq 4e$ nodes.) $b-$matchings can be computed in linear time when G is a tree.

An interesting variant of $b-$matching requires to find a (maximum) set of edges E' such that at least a_i and at most b_i edges are incident to node i for a given assignment of values a_i and b_i with $a_i \leq b_i$ to the nodes. This probably is the most general version of what is usually called a "degree-constrained subgraph construction problem". (Note that

matching is a very special case of this problem.) Again, this general version can be solved in polynomial time.

$b-$matching is a useful intermediate step towards the theory of coverings.

Definition. A node(edge-) cover of G is any set of nodes V' (edges E') such that each edge (node) of G is incident to at least one of the nodes in V' (or edges in E', respectively).

Observe that every edge-covering is the complement (in E) of a $b-$matching with $b_i = \deg(i)-1$ for all $i \in V$. Thus the problem of computing a minimum cardinality or minimum weight edge-cover is equivalent to computing a maximum cardinality or maximum weight $b-$matching, respectively, and hence both problems are polynomial time computable. For the case of bipartite graphs, the correspondence takes a particularly attractive form, known as the König-Egervary theorem (the "duality theorem" of matching).

Theorem. For any bipartite graph G, the number of edges in a maximum matching is equal to the number of edges in a minimum edge-covering.

There is an analogous result for general graphs due to Edmonds. We note that the problem of computing minimum node-covers is NP-complete.

Of related interest is the question what fraction of the nodes is actually covered by a maximum matching. In general no particular bound can be given, and the following example shows that the fraction can be arbitrarily close to 0:

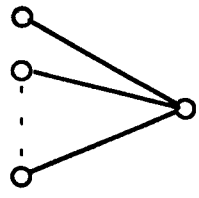


Figure 8:

Papadimitriou & Yannakakis proved that in any planar graph with minimum node-degree ≥ 3, any maximum matching will contain $\geq \frac{n}{3}$ edges. (For minimum node-degree ≥ 4 and ≥ 5 they prove a bound of $\geq \frac{2}{5}n$ and $\geq \frac{5}{11}n$ edges, respectively.)

Finally we need the following crucial concept (apparently due to Berge).

Definition. An alternating path or cycle is a simple path or cycle whose edges are alternately matched and free. An alternating path is called augmenting if both its end nodes are exposed. The weight of an alternating path or cycle is equal to the weight of the free edges minus the weight of the matched edges.

If M admits an augmenting path, then it cannot be maximum: reversing the roles of the matched and free edges in the path, results in a new matching of size $|M| + 1$. A central

result of matching theory states that repeated augmentation must result in a maximum matching.

Theorem.

(i) A matching is of maximum cardinality if and only if it has no augmenting path.

(ii) A matching is of maximum weight if and only if it has no alternating path or cycle of weight > 0.

Proof.

(i) $\Rightarrow$ is trivial. To prove $\Leftarrow$, let M not be of maximum cardinality. Let M' be a matching with $|M'| > |M|$. Consider the graph G' on V with edge set $E' = M \triangle M'$ (the symmetric difference of M and M'). Clearly each node of G' is incident to at most one edge of M and at most one edge of M'. It easily follows that the connected components of G' will be paths or circuits of even length. In all circuits we have the same number of edges from M as from M'. Because $|M'| > |M|$ there must be a path with more edges from M' than from M. This path necessarily is an augmenting path for M.

(ii) $\Rightarrow$ is trivial. (Reversing the roles of matched and free edges in an alternating path or cycle of weight > 0 would result in a "heavier" matching.) To prove $\Leftarrow$, we proceed as before, by a very similar argument. □

Theorem (ii) does not have the same constructive flavor as (*i*). In 1967, White proved the following more useful result.

Theorem. Let M be a maximum weight matching of size $q = |M|$ (i.e., maximum among all matchings of size q), and let L be an augmenting path of maximum weight. Then the matching M' obtained from M by reversing the roles of the edges along L is of maximum weight among all matchings of size $q + 1$.

Proof.

Let M'' be a maximum weight matching among all matchings of weight $q+1$. Consider the graph $\tilde{G}$ on V with edge set $M \triangle M'$. As before, the component of $\tilde{G}$ are paths or cycles of even length. Any even length path or (even length) cycle must have weight 0 with respect to M: if not, then reversing the roles of the edges would lead to a matching of larger weight than M or M''. Consider the paths of odd length. Because $M \triangle M''$ has precisely one more edge in M'' than in M, the number of odd length paths is odd and we can combine all but one of the paths in pairs such that each pair has an equal number of edges in M and in M''. Each pair of paths must have total weight 0 with respect to M (otherwise the choice of M or M'' as maximum weight matchings is contradicted). The one remaining path must have precisely one more edge in M' than in M, and is augmenting with respect to M. Augmenting along this path gives a matching of $q+1$ edges necessarily of the same weight of M''! The theorem now follows. □

The theorem is intriguing because it shows that in the weighted case the "augmenting

path method" can be used to construct maximum weight matchings of all possible sizes $(0, 1, 2, \cdots)$ if we always augment using maximum weight augmenting paths. While the theory is strongly suggestive of the types of algorithms we need to use, more details are needed to show that maximum matchings can be computed in small polynomial time.

In one case maximum matchings can be characterised in another way, namely when G is a tree.

Definition. Let G be a tree rooted at r, and let $f(i)$ denote the father of i. A matching $M \subseteq E$ is termed proper if for every exposed node i there is a brother j of i such that $< j, f(i) > \in M$.

Lemma. Let G be a tree. If M is a proper matching, then it is a maximum matching.

(The easy proof proceeds by showing that no proper matching can have an augmenting path.) One can show that a proper matching in a tree can be constructed in $O(n)$ time.

3.2 Computing Maximum Matchings.

We consider the simplest case first, namely maximum cardinality matching. All algorithms are based on the augmenting path idea, with special techniques to find augmenting paths efficiently. We outline some of the underlying ideas.

Consider the case that G is bipartite. One can search for an augmenting path as follows. (We assume that a matching M is given as a start.) Build a BFS tree starting from an exposed node i_0. (If there is no exposed node, M is a perfect matching.) Because G is bipartite, there can be no edges connecting nodes within the same level. If an exposed node j appears for the first time (necessarily in an odd level), then the "green" path from i_0 to j is an augmenting path. If no odd level contains exposed nodes, then the BFS-tree is called "Hungarian" and we can forget i_0. If the trees from all exposed nodes are Hungarian, the matching we have built clearly is maximum. Clearly there can be at most $n/2$ augmentations (as the number of matched nodes increases by 2 in every round) and every round takes $O(e)$ time. Thus the algorithm computes a maximum matching in $O(ne)$ time. A better result can be obtained by doing more augmentations "in one step". In particular, the idea is to compute a maximal set of vertex-disjoint augmenting paths $L_1, \cdots, L_r$ that are all of equal, shortest length (under the current matching M), and to augment along all paths simultaneously. Hopcropt & Karp showed that this must result in a maximum matching after $O(\sqrt{n})$ steps for all graphs, and that for bipartite graphs each step can be implemented in $O(e)$ time. Thus maximum cardinality matching on bipartite graphs needs only $O(\sqrt{n}.e)$ time.

There is another way to see this, by observing a simple connection between maximum matching and flow theory (even in the weighted case).

Theorem. For every bipartite graph G there exists a network G' with integer edge-capacities such that maximum matchings in G correspond to maximum flows in G'.

Proof.

Let $V = X \cup Y (X \cup Y = \emptyset)$ such that $E \subseteq X \times Y$. Design G' by adding a source node s, a target node t, and edges (s, x) and (y, t) for all $x \in X, y \in Y$. Let the edges

(s, x) and (y, t) have capacity 1 (cost 0), and the other edges $(x, y) \in E$ have capacity ∞ (cost - $w(x, y)$). By the integer flow theorem, maximum flows in G' are integral. Clearly every flow of size f must identify f matching edges for G and vice versa. Cost $-c$ of a maximum flow corresponds to weight $-c$ of the corresponding matchings. □

As maximum flows are computable in $O(\sqrt{n} \cdot e)$ time, so can maximum cardinality matchings.

Maximum cardinality matching for general (connected, non-bipartite) graphs is considerably more difficult and requires an additional technique due to Edmonds. Once again we try to find an augmenting path from an exposed node i_0, by building a "tree" of alternating paths starting at i_0 very much like in the bipartite case. Note that we can assume that odd level nodes are always extended by a matched edge (otherwise an augmenting path is found) and, hence, that the "tree" is grown by building on the even level nodes. The "Hungarian" case (no augmenting path from i_0 exists) is special, and implies that we can effectively delete the "tree" from the graph.

Lemma. If there is no augmenting path from node i_0 (at some stage), then there never will be an augmenting path from i_0.

Proof.

We show that the lemma must hold after every augmentation of a matching M for which there is no augmenting path from i_0. (The lemma then follows by induction.) Let L be an augmenting path from x to y, necessarily with x and y exposed and $x, y \neq i_0$. Suppose there is an augmenting path L' from i_0 with respect to the matching $M \triangle L$. If L and L' are node-disjoint, then L' would already have been augmenting in M. Contradiction. Thus let $L \cap L' \neq \emptyset$, and let z be the first node on L' that is also on L. Consider

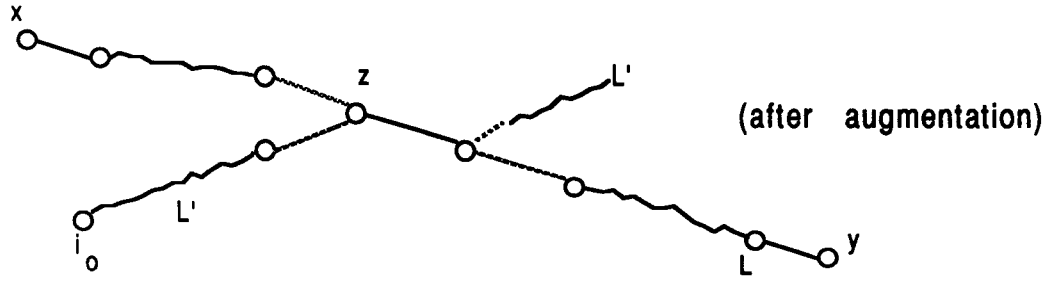
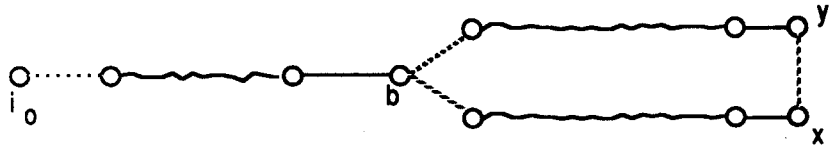


Figure 9:

the situation before augmenting along L. The path from i_0 to x via z (along a section of L' and of L) must have been an augmenting path in M, contrary to our assumption. □

Consider the process of building a "tree" of alternating paths from i_0. We are especially interested in what happens when we examine even level nodes x. (It is not necessary to build the tree from one node, one can just work on matched or exposed nodes, orienting the incident matched edges such that the nodes are even.) If $< x, y > \in E$ and y is not

matched, then there is an augmenting path from i_0 to y. If y is matched, there are two cases. If y is odd, there is nothing special happening (like in the bipartite case). If y is even, there is a special situation. After all, there is an alternating path of even length from i_0 to x and one from i_0 to y, and the edge $< x, y >$ closes a cycle of odd length. If we let b be the last node common to both paths (the base node), then the resulting structure is like this:



The cycle is a special case of a substructure known as a blossom, which can be formed if we do the building process in a distributed manner over many clusters simultaneously.

Definition. Let M be a matching, B a set of edges connecting nodes in $V_B \subseteq V, |B| = 2r + 1\ (r \geq 1)$. B is called a blossom of M if the following conditions hold:

(i) $|M \cap B| = r$ (the unique node of B left exposed in $M \cap B$ is called the base b of the blossom),

(ii) there is an alternating path L of even length with $L \cap B = \emptyset$ from an exposed node (the "root") to the base of the blossom, and

(iii) for each node $i \in V_B$ there is an alternating path of even length from the base of the blossom to i.

Lemma. Let B be a blossom. There exists an augmenting path in G mod B with respect to $M - B$ if and only if there exists an augmenting path in G with respect to M.

The construction of G mod B is known as "blossom-shrinking". Algorithmically, the lemma is of crucial importance for finding augmenting paths. The different algorithms that exist are usually based on (very) efficient techniques of finding and manipulating blossoms. The algorithm of Micali & Vazirani follows the Hopcroft-Karp strategy, but succeeds in handling blossoms so a maximal set of augmenting paths is found in each phase in $O(e)$ time. See figure 3.2 for the known complexity results.

The construction of maximum weight matchings roughly follows similar techniques and is based on the theorem on page 45.

Once again the bipartite case is the easiest to handle. In this case the problem is also known as the assignment problem, because it can be interpreted as the problem of finding an optimal cost assignment of tasks to agents (machines, workers). For an explanation

Kameda & Munro (1974)	$O(ne)$
Edmonds (1965)	$O(n^4)$
Balinski (1967)	$O(n^3)$
Lawler (1976)	$O(n^3)$
Gabow (1976)	$O(n^3)$
Gabow & Tarjan (1983)	$O(ne)$
Even & Kariv (1975)	$O(\min\{n^{\frac{5}{2}}, \sqrt{n}.e.\log\log n\})$
Micali & Vazirani (1980)	$O(\sqrt{n}.e)$

Figure 10: Maximum cardinality matching algorithms for general (non-bipartite) graphs.

of the so-called Hungarian method to solve the assignment problem, see Papadimitriou & Steiglitz. (It runs in $O(n^3)$ time.) By theorem (page 58) the problem can be formulated as a minimum cost maximum flow problem, which can be solved in $O(ne.^{\log n}/\log(2+\frac{m}{n}))$ time. The general case is solved by a technique that exploits the formulation as a linear programming problem (due to Edmonds). Define the variables x_{ij} to mean: $x_{ij} = 1$ if $< i, j > \in M$, and 0 otherwise. Let B_k be any set of $2r_k + 1$ nodes (i.e., an odd set). The fact that we want the $x-$variables to represent a matching leads to the constraint

$$\sum_{i,j \in B_k} x_{ij} = r_k$$

for all k and sets B_k. Now maximum weight matching can be formulated as the following linear programming problem:

$$\begin{aligned}
\text{maximize} \quad &: \sum_{i,j} w_{ij} x_{ij} \\
(*)\ \text{subject to} \quad &: \sum_{<i,j> \in E} x_{ij} \leq 1 \text{ for all } i \in V, \\
&\sum_{i,j \in B_k} x_{ij} \leq r_k \text{ for all sets } B_k, |B_k| = 2r_k + 1, \\
&x_{ij} \geq 0 \text{ for all } i, j.
\end{aligned}$$

Theorem. Every solution to the linear programming problem (*) has $x_{ij} \in \{0, 1\}$ for all i, j and hence can be interpreted as a matching.

All that is "left" is to solve (*)! It turns out that the dual problem holds important clues to the efficient solution of (*). The dual problem has a variable u_i for every vertex i and a variable z_k for every set B_k, and reads as follows (by applying the duality theorem of linear programming):

$$\begin{aligned}
\text{minimize} \quad &: \sum_i u_i + \sum_k r_k z_k \\
\text{subject to} \quad &: u_i + u_j + \sum_{k : i,j \in B_k} z_k \geq w_{ij} \text{ for all } i, j \\
&u_i \geq o \text{ for all } i, \\
&z_k \geq o \text{ for all } k.
\end{aligned}$$

One can redefine the computational problem as follows. Let $\pi_{ij} = u_i + u_j + \sum_{k:i,j \in B_k} z_k - w_{ij}$.

Lemma. A matching M has maximum weight if and only if the following conditions are satisfied for all nodes i, edges $< i,j > \in E$, and odd sets $B_k \subseteq V$ with $|B_k| = 2r_k + 1$:

(a) $u_i, \pi_{ij}, z_k \geq 0$,

(b) if $< i,j >$ is matched, then $\pi_{ij} = 0$,

(c) if i is exposed, then $u_i = 0$,

(d) if $z_k > 0$, then B_k is maximally matched (i.e., $|\{< i,j > | i,j \in B_k, < i,j > \in M\}| = R_k$ and B_k is a blossom).

Proof.

"Only if" follows from duality. The "if"- part is seen as follows. Let M satisfy (a) through (d) and let M' be another matching. Observe

$$\sum_{<i,j> \in M'} w_{ij} \leq \sum_{<i,j> \in M'} (u_i + u_j + \sum_{k:i,j \in B_k} z_k) \leq \sum_i u_i + \sum_k r_k z_k = \sum_{<i,j> \in M} w_{ij},$$

by applying known facts for M (and the constraints for M'). □

The algorithms for maximum weight matching start with an empty matching $M, z_k = 0$ for all k, and $u_i = G$ for all i and a suitably large G (e.g. $G = \frac{1}{2} \max w_{ij}$). This satisfies (a), (b) and (d) but not (c). One now tries to make changes such that (a), (b) and (d) are preserved but the number of violations of (c) is reduced (this is done by finding augmenting paths between nodes i,j with $u_i = u_j > 0$). Again one uses blossom-shrinking to find the augmenting paths. The further details are explained in Lawler. The most efficient implementations of maximum weighted matching run in time $O(ne \log n)$ and $O(ne \log\log \log_d n + n^2 \log n)$ for $d = \max\{2, e/n\}$. For planar graphs, a different algorithm based on the planar separator theorem yields a maximum weight matching in $O(n^{1.5} \log n)$ time.

The maximum matching algorithms are still fairly complex and invite alternative approaches. One approach is to drop the strict requirement that a matching is maximum, and to construct a matching that is "approximately" maximum. A simple, but rather crude bound is obtained as follows: construct a DFS tree T of G (in linear time), and construct a maximum matching in T (in linear time, cf. section 3.1). Let the matching be M'.

Theorem. $|M'| \geq \frac{1}{2}|M|$, for any maximum matching M of G.

A different approach aims at finding f^* probabilistic algorithms for maximum matchings. The basic starting point is a result of Erdös & Renyi that asserts that the probability that an n−node graph (n even) with N edges has a perfect matching tends to 1 for $n \to \infty$, provided $N \geq \frac{1}{2} n \log n + w(n)n$ (for some $w(n)$ with $w(n) \to \infty$ for $n \to \infty$). Angluin & Valiant proposed the following randomized algorithm for finding a perfect matching

high probability (averaged over all graphs with $N \geq cn \log n$ for c sufficiently large). Suppose we succeeded building a (partial) matching M. If M has $n/2$ edges we can stop and report success (M is perfect). Otherwise, let x be the least-numbered exposed node in M and select (and delete) a random edge $(x, y) \in E$. If no edge exists, then stop and report failure. If y is exposed in M, then (x, y) is added to M and we repeat the entire procedure for the new matching. If y is matched in M, with mate z, then delete (y, z) from M but add (x, y) to it, and repeat the procedure at node z. By using a suitable data-structure

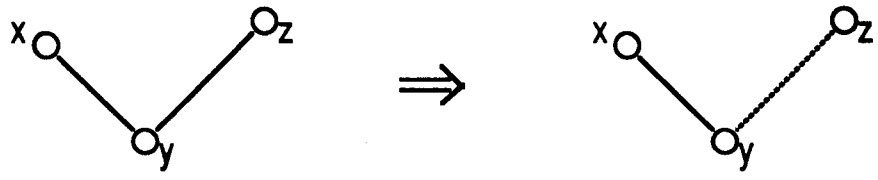


Figure 11:

the overhead in each step of the algorithm takes $O(1)$ time. It can be shown that for $N \geq cn \log n$ the probability that the algorithm reports success within $O(n \log n)$ steps tends to 1 for $n \to \infty$. It follows that the algorithm "almost certainly" finds a perfect matching in $O(n \log n)$ time.

3.3 Maximum Flow.

Let G be a directed graph with a non-negative capacity $c(e)$ assigned to every edge e. (We shall henceforth refer to G as a "network".) We are interested in solving the $s - t$ maximum flow problem, normally referred to as "the maximum flow problem", defined as follows: determine a maximum (legal) flow f from s to t, where s and t are specified nodes of G called the source and the target of the flow, respectively. The maximum flow problem is probably one of the most classical problems in combinatorial optimization on graphs. We will outline the basic theory and some very efficient, polynomial time algorithms for it. The results are usually refined for special sub-cases of the problem, like

- the network is of a special type (e.g. planar),
- the edge capacities are integral, or
- the edge capacities are all 0 or 1.

We will usually mention the results for these cases, but omit the details. Let $\overleftarrow{E} = \{(j, i) | (i, j) \in E\}$, and $\overleftarrow{e} = (j, i)$ for $e = (i, j)$.

Definition. A flow on G is a function $f : E \cup \overleftarrow{E} \to \mathbf{R}$ satisfying the following constraints:

(i) for every $e \in E, 0 \leq f(e) \leq c(e)$ and $f(\overleftarrow{e}) = -f(e)$, and

(ii) for every $i \neq s, t, \sum_{j:(j,i)\in E} f(j,i) = \sum_{j:(i,j)\in E} f(i,j)$.

The latter constraint is the basic conservation law, which requires that what flows in flows out for every node i different from s and t. The value $|f|$ of a flow is defined as $\sum_{i:(s,i)\in E} f(s,i)$. It is seen to be equal to $\sum_{i:(i,t)\in E} f(i,t)$: what flows out of s is precisely what flows into t.

Definition. An $s-t$ cut is a partition $X;\overline{X}$ of V such that $s \in X$ and $t \in \overline{X}$. The capacity of an $s-t$ cut $X;\overline{X}$ is defined to be $c(X,\overline{X}) = \sum_{(i,j)\in E: i\in X, j\in\overline{X}} c(i,j)$, and the flow across the cut is defined to be $f(X,\overline{X}) = \sum_{(i,j)\in E: i\in X, j\in\overline{X}} f(i,j)$.

The following lemma can be shown by induction on $|X|$.

Lemma. $|f| = f(X,\overline{X}) - f(\overline{X},X) \leq c(X,\overline{X})$ for any $s-t$ cut $X;\overline{X}$.

The methods of finding a maximum flow mostly use the basic technique of successively augmenting a given flow. An augmenting path from s to t is any "path" π of edges $\in E \cup \overleftarrow{E}$ with the following properties:

- for any edge $e \in \pi$ $(e \in E)$, $f(e) < c(e)$
- for any edge $\overleftarrow{e} \in \pi$ $(e \in E)$, $f(e) > 0$.

The edges $e \in \pi$ are sometimes called "forward edges", and the edges $\overleftarrow{e} \in \pi$ "backward edges". It should be clear why π is called an augmenting path. Define $\varepsilon_1 = \min\{c(e) - f(e) | e \in \pi (e \in E)\}$, $\varepsilon_2 = \min\{f(e) | \overleftarrow{e} \in \pi (e \in E)\}$ and $\varepsilon = \min\{\varepsilon_1, \varepsilon_2\}$. Adding ε flow to every "edge" in π results in a flow f' with $|f'| = |f| + \varepsilon$, i.e., a flow of larger value! (Note that the process of augmentation effectively increases the flow by ε on every forward edge, and decreases the flow by ε on every backward edge.) The following result is crucial and due to Ford & Fulkerson.

Theorem.

(i) ("the augmenting path theorem") f is a maximum flow if and only if there is no augmenting path for f.

(ii) ("the max-flow min-cut theorem") the maximum value of an $s-t$ flow is equal to the minimum capacity of an $s-t$ cut.

Proof.

(i) Let f be a maximum flow. It is clear that there can be no augmenting path, otherwise a flow augmentation along the path would yield a larger flow (contradiction). Next suppose there is no augmenting path from s to t. Any attempt to build a path of edges $\in E \cup \overleftarrow{E}$ from s to t must end at a node where all next edges $e \in E$ are saturated (i.e., $f(e) = c(e)$) and all next edges $\overleftarrow{e}$ $(e \in E)$ have $f(e) = 0$. Define X to be the set of nodes v (including s) such that there is an augmenting path from s to v. It follows that $X;\overline{X}$ is an $s-t$ cut and $|f| = f(X,\overline{X}) = c(X,\overline{X})$. As this is the largest value a flow can have, f must be maximum.

(ii) Let f be a maximum $s-t$ flow. By the preceding argument there is an $s-t$ cut $X;\overline{X}$ such that $|f| = c(X,\overline{X})$. As $|f| \leq c(Y,\overline{Y})$ for any $s-t$ cut $Y;\overline{Y}$ it follows that $X;\overline{X}$ is an $s-t$ cut of minimum capacity. □

From the proof it follows that for any maximum $s-t$ flow and minimum capacity $s-t$ cut $X;\overline{X}$ the edges leading "across the cut" from X to $\overline{X}$ are saturated and the edges leading from $\overline{X}$ to X carry flow 0. The max-flow min-cut theorem actually implies that every network admits a maximum flow.

Corollary. ("the integral flow theorem") If all capacities are integers, there is a maximum flow which is integral (i.e., with an integral flow through every edge).

Proof.

Consider any feasible, integral flow f. If f is not maximum, there must be an augmenting path. Augmenting the flow along this path yields another feasible, integral flow f' with $|f'| \geq |f| + 1$. By repeating this we must arrive at an integral flow that is maximum. □

While these are the main theorems on which the construction of maximum flows are based, the theory can be ramified further. For example, one may want to impose limits on the capacity of the various nodes and wish to determine a maximum flow f subject to the (additional) requirement that $\sum_{j:(j,i)\in E} f(i,j) \leq c_i$, i.e., the total flow into every node i does not exceed its capacity c_i $(i \neq s,t)$. This problem can be reduced to an ordinary maximum flow problem as follows: split each node $i \neq s,t$ into two nodes i' and i'' connected by an edge (i',i''), let the edge (i',i'') have capacity c_i, and lead all incoming edges of i to i' and all outgoing edges from i''. By virtue of this transformation the previous theorems go through virtually unchanged.

Another common constraint is to impose a lowerbound on the flow through each edge, in addition to the upperbound (or: capacity) for each edge. The results here require that we change our view from flows to circulations. Let $l(e)$ and $c(e)$ denote the lower- and upperbound, respectively, of the flow through edge $e \in E$.

Definition. A circulation in G is a function ("flow") $f : E \cup \overleftarrow{E} \rightarrow \mathbf{R}$ satisfying the following constraints:

(i) for every edge $e \in E, 0 \leq f(e) \leq c(e)$ and $f(\overleftarrow{e}) = -f(e)$, and

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

Thus circulations simply are flows that observe the basic conservation law at all nodes. The maximum flow problem can easily be converted to circulation form as follow: add an edge (t,s) to the flow network with $l(t,s) = 0$ and $c(t,s) = \infty$, and ask for a circulation for which $f(t,s)$ is maximum. A typical scenario for determining a maximum flow "with lowerbounds" is the following: first determine a feasible circulation, next augment the implicit (feasible) $s-t$ flow to a maximum flow. The following result is due to Hoffman. A circulation is called feasible if $l(e) \leq f(e) \leq c(e)$.

Theorem. ("the circulation theorem") In a network with lowerbounds and capacities,

a feasible circulation exists if and and only if $\sum_{i\in Y, j\in\overline{Y}} l(j,i) \leq \sum_{i\in Y, j\in\overline{Y}} c(i,j)$ for every cutset $Y;\overline{Y}$.

Proof.

Let f be a feasible circulation. One can prove by induction that the effective flow across any cut must be zero, i.e., $f(Y,\overline{Y}) - f(\overline{Y},Y) = 0$. Now $\sum_{i\in Y, j\in\overline{Y}} l(j,i) \leq f(\overline{Y},Y) = f(Y,\overline{Y}) \leq \sum_{i\in Y, j\in\overline{Y}} c(i,j)$ as claimed, for any cut $Y;\overline{Y}$.

Conversely, assume that $\sum_{i\in Y, j\in\overline{Y}} l(j,i) \leq \sum_{i\in Y, j\in\overline{Y}} c(i,j)$ for every cut $Y;\overline{Y}$. Suppose that the network admits no feasible circulation. Consider any circulation f and let u, v be two nodes with $(u,v) \in E$ and $f(u,v) < l(u,v)$. We claim that there must be an augmenting path from v to u, i.e., a path in which each forward edge e has $f(e) > l(e)$ and each backward edge $\overleftarrow{e}$ has $f(e) < c(e)$. For suppose this is not the case. Let Y be the set of nodes which can be reached from v by an augmenting path. Clearly $Y;\overline{Y}$ is a $v-u$ cut, every edge from Y to $\overline{Y}$ must be saturated and every edge e from $\overline{Y}$ to Y must have $f(e) \leq l(e)$. Assuming (as we may) that f does satisfy the basic conservation law, we have

$$\sum_{i\in Y, j\in\overline{Y}} c(i,j) = f(Y,\overline{Y}) = f(\overline{Y},Y) < \sum_{i\in Y, j\in\overline{Y}} l(j,i)$$

(with strict inequality because of the edge (u,v)). This contradicts the assumption. Thus consider any augmenting path from v to u, and let δ be the smallest surplus on any edge (i.e., $f(e) - l(e)$ on a forward edge and $c(e) - f(e)$ on a backward edge). We can now "augment" f by redistributing flow as follows: add a flow of δ to edge (u,v) and subtract a flow of δ from the edges in the augmenting path (i.e., subtract δ from every forward edge and add δ to every backward edge in it). Apply this procedure to any network with a circulation f for which $\min_{e\in E} |f(e) - l(e)|$ is smallest, and it follows that we obtain a circulation for which this minimum is smaller yet unless it is zero. Contradiction. □

In a flow network with lower- and upperbounds, it may be of interest to determine both a maximum and a minimum flow that satisfy the constraints. The following result is again due to Ford and Fulkerson.

Theorem. Let G be a flow network with lowerbounds and capacities, which admits a feasible $s-t$ flow. Define the capacity of a cutset $Y;\overline{Y}$ as $\sum_{i\in Y, j\in\overline{Y}} c(i,j) - \sum_{i\in Y, j\in\overline{Y}} l(j,i) = cap(Y;\overline{Y})$.

(i) ("the generalized max-flow min cut theorem") The maximum value of an $s-t$ flow is equal to the minimum capacity of an $s-t$ cut.

(ii) ("the min-flow max-cut theorem") The minimum value of an $s-t$ flow is equal to the maximum of $\sum_{i\in Y, j\in\overline{Y}} l(j,i) - \sum_{i\in Y, j\in\overline{Y}} c(i,j)$ over all $s-t$ cuts $Y;\overline{Y}$, i.e., minus the minimum of the capacity of a $t-s$ cut.

Proof.

(i) Convert the flow network into a circulation network by adding an edge (t,s) with $l(t,s) = l$ and $c(t,s) = \infty$, for a "parametric" value of l. Because G admits a feasible flow f, the modified network admits a feasible circulation for any $l \leq |f|$. Clearly, the maximum $s-t$ flow in G has a value equal to the maximum value of l for which

the modified network admits a feasible circulation. By the circulation theorem a feasible circulation exists if and only if $l \leq cap(Y; \overline{Y})$. The result follows.

(ii) Similar, by considering the circulation network obtained by adding an edge (t, s) with $l(t, s) = 0$ and $c(t, s) = l$. □

An important type of flow problem arises when a cost (or weight) $b(i, j) \geq 0$ is involved with each unit of flow through an edge (i, j). The cost of a flow f is defined as $\sum_{e \in E} b(e)f(e)$. The minimum cost flow problem asks for the $s - t$ flow of some value v and (among all flows of this value) of least cost. Usually the minimum cost flow problem is understood to be the problem of finding a maximum flow of least cost. The methods for finding a minimum cost flow typically start with a feasible flow (i.e., a flow of value v or a maximum flow) and transform it into another flow of some value and e.g. lesser cost by working on "cost-reducing augmenting paths". The cost of an augmenting path or cycle is defined as the sum of the costs of the flow through the forward edges minus the sum of the costs of the flow through the backward edges.

Theorem. ("the minimum cost flow theorem")

(i) A flow of value v has minimum cost if and only if it admits no augmenting cycle of negative cost.

(ii) Given a minimum cost flow of value v, the augmentation by δ along an $s - t$ augmenting path of minimum cost yields a minimum cost flow of value $v + \delta$.

Proof.

(i) Suppose f has value v and minimum cost. Clearly no negative cost augmenting cycle can exist, otherwise we could introduce a (small) amount of extra flow around the cycle without changing the value of the flow but reducing its cost. Conversely, let f have value v and no negative cost augmenting cycle. Suppose f does not have minimum cost, but let f^* be a flow of value v that has minimum cost. Now $f^* - f$ is a "flow" of value 0 and negative cost. The "flow" can be decomposed into cycles of flow (by reasoning backwards from every edge carrying non-zero flow) at least one of which apparently must have negative cost. Contradiction.

(ii) Let f be a minimum cost flow of value v, and suppose the augmentation by δ along the minimum cost $s - t$ augmenting path π introduced an augmenting cycle C of negative cost. Thus C includes one or more edges affected by the augmentation along π in reversed direction, to account for the negative cost of C. Define $\pi \oplus C$ to be the set of edges of $\pi \cup C$ except those which occur in π and reversed on C. The set $\pi \oplus C$ partitions into an $s - t$ path π' and a number of cycles, and the total flow-cost in the edges of $\pi \oplus C$ is less than the cost of π. Clearly π' must be an $s - t$ augmenting path of cost less than the cost of π, as the cycles cannot have nonnegative cost by (i) and the assumption on f. Contradiction. □

The minimum cost flow theorem is implicit in the work of Jewell and of Busacker and Gowen.

Corollary. ("the integrality theorem for minimum cost flows") If all capacities are integer, there is a maximum flow which is integral and has minimum cost (over all maximum flows).

Proof.

Similar to the proof of the integrality theorem, by using (ii) of the minimum cost flow theorem. □

In the subsequent sections we will see that many versions of the maximum flow problem can be solved by polynomial time algorithms. It has been shown that the maximum flow problem (i.e., the problem to determine whether there exists a flow of value $\geq K$ for specified K) is log-space complete for the class P of all polynomially computable problems.

3.4 Computing Maximum Flows.

3.4.1 Augmenting Path Methods

Many algorithms to determine a maximum flow in a network start from Ford and Fulkerson's augmenting path theorem. Given a flow f and an augmenting path π, let the maximum amount by which we can augment f along π be called the residual capacity $res(\pi)$ of π. (Thus $res(\pi) = \min\{res(e)|e \in \pi\}$ with $res(e) = c(e) - f(e)$ if $e \in E$ and $res(e) = -f(e)$ if $e \in \overleftarrow{E}$). Augmenting path methods are based on the iteration of one of the following types of steps, starting from any feasible flow (e.g. $f \equiv 0$).

clairvoyant augmentation : augment the flow along some $s-t$ augmenting path π by an amount $\leq res(\pi)$,

Ford-Fulkerson augmentation : find an augmenting path π and augment the flow by $res(\pi)$ along π,

maximum capacity augmentation : find an augmenting path π of maximum residual capacity and augment the flow by $res(\pi)$ along π,

Edmonds-Karp augmentation : find a shortest augmenting path π and augment the flow by $res(\pi)$ along π.

Clairvoyant augmentation is only of theoretical interest, but the following observation will be useful.

Proposition. A maximum flow can be constructed using at most e clairvoyant augmentation steps (along augmenting paths without backward edges).

Proof.

Let f be a (maximum) non-zero flow in G, and let $e \in E$ be an edge with the smallest non-zero amount of flow $f(e)$. One easily verifies that there must be an $s-t$ path π containing only forward edges including e. Decrease the flow in every edge along π by $f(e)$,

obtaining a flow f' with $|f'| < |f|$. (Note that f is obtained from f' by a clairvoyant augmentation along π.) Repeat this process as long as a non-zero flow is obtained. Because in each step at least one more edge is given a zero flow, the process goes on for at most e steps (and thus leads to at most e paths) and ends with the zero flow. By reversing the process and augmenting the flow by the right amount along the paths (taken in reverse order) one obtains f again. □

The other augmenting path methods are more interesting and will lead us to a variety of polynomial time algorithms for computing maximum flows. The main problem in a Ford-Fulkerson augmentation step is finding an augmenting path. The usual procedure for this is reminiscent of a breadth-first search algorithm and inductively generates for every node u (beginning with the neighbors of s) a "label" that indicates whether there exists an augmenting path from s to u. The label will contain a pointer to the predecessor on the augmenting path, if one exists, in order to be able to trace an augmenting path when the labeling procedure terminates at t. If the labeling procedure gets stuck before t is reached, then no augmenting path from s to t exists and the flow must be maximum. (Observe that in this case $X; \overline{X}$ with X consisting of s and the labeled vertices, must be an $s-t$ cut of minimum capacity.) When suitably implemented a Ford-Fulkerson augmentation step takes only $O(e)$ time. It follows that in a network with all integer capacities, a maximum flow can be constructed by Ford-Fulkerson augmentation in $O(ef^*)$ time, where f^* is the value of the maximum flow. On the other hand, Ford and Fulkerson have shown that Ford-Fulkerson augmentation on an arbitrary network does nog necessarily converge to the maximum flow. (Their example shows that the algorithm may get bogged down on a never ending sequence of augmenting paths with residual capacity converging to zero, without ever coming close to the maximum flow.) Thus at each step of an augmenting path method the augmenting path to work on must be properly chosen in order to obtain a viable maximum flow algorithm on general networks. We will first look at maximum capacity augmentation, suggested by Edmonds and Karp.

Theorem.

(i) Maximum capacity augmentation produces a sequence of flows that always converges to a maximum flow.

(ii) If the capacities are all integers, maximum capacity augmentation finds a maximum flow in $O(\min\{e \log c^*, 1 + \log_{M/M-1} f^*\})$ iterations where c^* is the maximum edge capacity, f^* the value of a maximum flow and $M > 1$ is the maximum number of edges across any $s - t$ cut.

Proof.

(i) Let f be a flow and f^* a maximum flow, chosen such that $f^* \geq f$ (by the augmenting path theorem). One easily verifies that $f^* - f$ is a (maximum) flow of value $|f^*| - |f|$ in the residual graph, i.e., the graph G in which each edge is given its current residual capacity. By the lemma there must be an augmenting path over which one can augment the flow by at least $\frac{1}{e}(|f^*| - |f|)$, which thus is a lowerbound on the residual capacity of the current maximum capacity augmenting path. On the other hand, over the next $2e$ augmentations (as long as the maximum flow is not reached) the residual capacity of a maximum augmenting path must go down to $\frac{1}{2e}(|f^*| - |f|)$

or less. Thus after $2e$ or fewer augmentations, the residual capacity of the maximum augmenting path is reduced by at least a factor of two.

(ii) In the special case that all capacities are integers, the residual capacity of a maximum augmenting path is initially at most c^* and will never be less than 1. Thus a maximum flow must be obtained after at most $O(e \log c^*)$ maximum capacity augmentations. To obtain another bound, let $f_0, f_1, \cdots, f_k = f^*$ be the sequence of flows obtained by iterating maximum capacity augmentation and let $\varepsilon_i = |f_{i+1}| - |f_i| (1 \leq i < k)$ be the i^{th} flow increment. Let X_i consist of s and all points reachable from s by an augmenting path of residual capacity $> \varepsilon_i$. Clearly $X_i; \overline{X}_i$ is an $s - t$ cut and all edges across the cut have either a residual capacity $\leq \varepsilon_i$ (for edges from X_i to $\overline{X}_i$) or capacity $\leq \varepsilon_i$ (for edges from $\overline{X}_i$ to X_i). Thus $|f^*| - |f_i| \leq c(X, \overline{X}) - f(X, \overline{X}) + f(X, \overline{X}) \leq \varepsilon_i M = (|f_{i+1}| - |f_i|)M$ or, equivalently, $|f^*| - |f_{i+1}| \leq (|f^*| - |f_i|)(1 - \frac{1}{M})$. By induction we have $|f^*| - |f_i| \leq |f^*|(1 - \frac{1}{M})^i$ and, hence, $1 \leq |f^*|(1 - \frac{1}{M})^{k-1}$. The bound on k follows. □

Finding an augmenting path of maximum residual capacity can be done by a procedure that is reminiscent of Dijkstra's shortest path algorithm, within $O(e \log n)$ or even less time. Thus, if all capacities are integers, maximum capacity flow augmentation yields a maximum flow in $O(m^2 \log n \log c^*)$ or less time.

Polynomial time algorithms for general networks (with a complexity independent of c^* or f^*) require a different approach. A classical starting point is Edmonds-Karp augmentation, in view of the following result.

Theorem. Edmonds-Karp augmentation yields a maximum flow after no more than $\frac{1}{2}ne$ iterations.

As a shortest augmenting path is easily found in $O(e)$ time (by a suitable version of breadth-first search), Edmonds-Karp augmentation yields an $O(ne^2)$ time algorithm for determining a maximum flow in any network. By extending the idea, Dinic observed that a more efficient algorithm results if one can construct all shortest augmenting paths first in one step (or phase) before performing the necessary augmentations on the subgraph. Define the distance $d(u)$ of a node u as the length of the shortest augmenting path from s to u. (Set $d(u)$ to ∞ if no such path exists.) Define the critical graph N as the subgraph of G containing only the edges $e = (u, v) \in E \cup \overleftarrow{E}$ with $d(v) = d(u) + 1$, with all nodes not on a path from s to t omitted. Every edge is assigned the residual capacity with respect to f. Clearly $s \in N$ and, provided f is not maximum, $t \in N$ as well. Observe that N contains precisely the shortest augmenting paths from s to t. N is also called the level graph or the layered network of f. Finally, let's call g a blocking flow if every path from s to t contains an edge of residual capacity zero.

Dinic augmentation : construct the citical graph N, find a blocking flow g in N, and "augment" f by replacing it by the flow $f + g$.

Theorem. Dinic augmentation yields a maximum flow after at most $n - 1$ iterations.
Proof.

The idea of the proof is that $d(t)$, i.e., the length of the shortest augmenting path from s to t, increases with every Dinic augmentation. Let f be the current flow, d the distance function, and N the critical graph. Let f', d', and N' be the corresponding entities after one Dinic augmentation (assuming that f is not maximum). If $t \notin N'$, the flow f' is maximum, $d(t) = \infty$ and we are done. Thus assume f' is not maximum (hence $t \in N'$). Consider a path $s = u_0, u_1, \cdots, u_k = t$ from s to t in N' with $d'(u_i) = i$. We distinguish two cases:

case I: $u_i \in N$ for $0 \leq i \leq k$.

We argue by induction that $d'(u_i) \geq d(u_i)$. For $i = 0$ this is obvious. Suppose the claim holds for i, and consider u_{i+1}. If $d(u_{i+1}) \leq d(u_i) + 1$, the induction step follows immediately. Thus let $d(u_{i+1}) > d(u_i) + 1$. Then the edge (u_i, u_{i+1}) is not in N despite the fact that it has a non-zero residual capacity (because it was not affected by the augmentation and is an edge in N'). Thus contradicts that $d(u_{i+1}) > d(u_i) + 1$, thus completing the induction. It follows in particular that $d'(t) \geq d(t)$. Suppose that $d(t') = d(t)$. Then $d'(u_i) = d(u_i)$ for $0 \leq i \leq k$, and the entire path must have been a path in N. This contradicts the fact that all paths in N were "blocked" after the Dinic augmentation. Hence $d'(t) > d(t)$.

case II: $u_j \notin N$, for some j.

Choose u_j to be the first node on the path with $u_j \notin N$. Clearly $j > 0$ and, by the preceding argument, $d'(u_i) \geq d(u_i)$ for $0 \leq i < j$. Consider the edge $(u_{j-1}, u_j) \in N'$. Because it has non-zero residual capacity now, it must have had so before the augmentation. The fact that $u_{j-1} \in N$ but $u_j \notin N$ implies that necessarily $d(u_j) = d(t)$ and (hence) $d(t) \leq d(u_{j-1}) + 1$. We conclude that $d'(t) > d'(u_{j-1}) + 1 \geq d(u_{j-1}) + 1 \geq d(t)$. As the distance of t is at least one at most $n - 1$, and increases by at least one with each Dinic augmentation, the number of Dinic augmentations required to obtain a maximum flow is at most $n - 1$. □

approx.year	author(s)	complexity
1956	Ford and Fulkerson	-
1969	Edmonds and Karp	$0(ne^2)$
1970	Dinic	$0(n^2e)$
1974	Karzanov	$0(n^3)$
1977	Cherkasky	$0(n^2\sqrt{e})$
1978	Malhotra, Pramodh Kumar and Maheshwari	$0(n^3)$
1978	Galil	$0(n^{5/3}e^{2/3})$
1980	Sleator and Tarjan	$0(ne \log n)$
1984	Tarjan	$0(n^3)$
1985	Goldberg	$0(n^3)$
1986	Goldberg and Tarjan	$0(ne \log^{n^2}/e)$

Figure 12: Some maximum flow algorithms and their complexity.

Given a flow f, its critical graph can be determined in $0(e)$ time by means of breadth-first search. Thus implementations of Dinic augmentation for computing maximum flows require $0(ne+nF)$ time, where $F = F(n, e)$ is a bound on the time to determine a blocking flow in a network of (at most) n nodes and e edges. Dinic's original algorithm determines

a blocking flow by saturating some edges in a path from s to t (in N) path after path, and achieves $F = 0(ne)$ and (hence) a total running time of $0(n^2e)$. Later algorithms greatly improved the efficiency of a Dinic augmentation, by constructing a blocking flow in a different manner and/or using special data structures. The most efficient implementation to date is due to Sleator and Tarjan and achieves $F = 0(e \log n)$, which thus yields an $0(ne \log n)$ time algorithm to determine maximum flows. Figure 3.4.1 shows the interesting sequence of maximum flow algorithms that developed over the years. (Goldberg's algorithm will be outlined below.) If all capacities are integers, an approach due to Gabow yields an $0(ne \log c^*)$ algorithm.

There are a number of results for more specialized networks that deserve mentioning at this point. For a network with all edge capacities equal to one, Even and Tarjan prove that a maximum flow is obtained after no more than $0(\min\{n^{2/3}, e^{1/2}\})$ Dinic augmentations. In a network in which all edge capacities are integers and each node $u \neq s, t$ has either one single incoming edge, of capacity one, or one single outgoing edge, of capacity one, at most $2\lceil\sqrt{n-2}\rceil$ Dinic augmentations are needed.

Next observe that for planar (directed) networks the best maximum flow algorithms we have seen so far still require $0(n^2 \log n)$ time or more (take $e = 0(n)$). Using a divide-and-conquer approach Johnson and Venkatesan have obtained an $0(n^{3/2} \log n)$ algorithm for the problem. But there is at least one case in which the maximum flow in a planar network can be obtained much easier, by an intuitively appealing technique. Define a flow network to be $s - t$ planar if it is planar and s and t are nodes on the same face. (The notion is due to Ford and Fulkerson, on a suggestion of G.B. Dantzig.) Let G be an $s - t$ planar network, and assume without loss of generality that s and t lie on the boundary of the exterior face. Embed G such that it lies in a vertical strip with s located on the left bounding line of the strip and b on the right. Let π be the topmost path from s to t, obtained by always choosing the leftmost edge in a node on the way from s to t and backtracking when one gets stuck. The following result is due to Ford and Fulkerson.

Lemma. The path π contains precisely one edge "across the cut" from X to $\overline{X}$, for every $s - t$ cut $X; \overline{X}$ of minimum capacity.

Proof.

Consider any minimum capacity $s - t$ cut $X; \overline{X}$. Clearly π must contain at least one edge across the cut from X to $\overline{X}$. Suppose it contains more than one edge across the cut, say $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$, with e_1 occurring before e_2 on the path. Because the $s - t$ cut has minimum capacity there must be directed paths π_1 and π_2 from s to t which contain e_1 and e_2, respectively, as the only edge across the cut from X to $\overline{X}$. Let π_1' be the part of π_1' from v_1 to t, π_2' the part of π_2 from a to u_2. Because π runs "above" π_1 and π_2, π_1' and π_2' must intersect at a node u. But now the path from s to t obtained by following π_2' from s to u and π_1' from u to t has no edge across the cut from X to $\overline{X}$! Contradiction. □

The lemma suggests a simple way of computing a maximum flow due to Dantzig (but sometimes referred to as Berge's algorithm).

Dantzig augmentation (for $s - t$ planar networks): find the topmost path π (forward edges only) with no saturated edges, and augment the flow by $res(\pi)$ along π.

Theorem. Given an $s-t$ planar network, Dantzig augmentation yields a maximum flow after at most $e = 0(n)$ iterations.

Proof.

Each Dantzig augmentation saturates at least one more edge, and hence there can be no more than e iterations. By the lemma it follows that whenever no further Dantzig augmentation can be carried out, the edges across the cut from X to $\overline{X}$ for any minimum capacity $s-t$ cut $X;\overline{X}$ must all be saturated. By the max-flow min-cut theorem, the flow must be maximum. □

A straightforward implementation of Dantzig augmentation requires a total of $0(e)$ edge inspections, and $0(n)$ time to augment the flow per iteration. Thus one can construct maximum flows in $s-t$ planar networks in $0(n^2)$ time. Itai and Shiloach have developed an implementation of the algorithm that requires only $0(n \log n)$ time. Frederickson claims an $0(n\sqrt{\log n})$ algorithm for the problem. As an interesting generalization Johnson and Vekatesan prove that a maximum flow in a planar network can be determined in $0(pn \log n)$ time, where p is the fewest number of faces that needs to be traversed to get from s to t.

A closely related problem is the determination of a minimum $s-t$ cut in a planar (undirected) graph. Itai and Shiloach gave an $0(n^2 \log n)$ algorithm for it. Reif has shown an algorithm for it that runs in $0(n \log n \log^* n)$ implementation.

3.4.2 Maximum Flow Algorithms Cont'd.

In 1985 A.V. Goldberg presented an entirely new approach to the construction of maximum $s-t$ flows in directed networks. The approach uses several insights from the preceding algorithms, but abandons the idea of flow augmentations along entire paths and of maintaining legal flows at all times. The generic version of Goldberg's algorithm consists of two phases. In phase I an infinite amount of flow is put on s, and an iterative scheme is called upon to let each node push as much flow as it can according to the residual capacities of the links "in the direction of the shortest path (in the residual graph)" to t. The result will be that usually more flow is entering a node than is leaving it, but that eventually all edges across any minimum cut get saturated. In phase II of the algorithm all excess flow that is entering the nodes is removed without altering the overal (maximum) value of the flow, thus resulting in a legal flow that is maximum. We describe the algorithm in more detail to appreciate some of the underlying mechanisms.

Definition. A preflow on G is a function $g : E \cup \overleftarrow{E} \rightarrow \mathbf{R}$ satisfying the following constraints:

(i) for every $e \in E, 0 \leq g(e) \leq c(e)$ and $g(\overleftarrow{e}) = -g(e)$, and

(ii) for every $i \neq s, t, \sum_{j:(j,i)\in E} g(j,i) \geq \sum_{j:(i,j)\in E} g(i,j)$.

Throughout phase I of Goldberg's algorithm, every node v maintains two items of information: (i) the current flow excess $\Delta(v)$ defined by $\Delta(s) = \infty$ and $\Delta(i) = \sum_{j:(j,i)\in E} g(j,i) - \sum_{j:(i,j)\in E} g(i,j)$ for all i, and (ii) a lowerbound $d(v)$ on the length of the shortest path from v to t in the residual graph. For the estimates $d(v)$ we only require that $d(t) = 0$ and $d(w) \geq d(v) - 1$ for every residual edge (v,w). At the start of phase I we set out with

the zero flow for g and the ordinary distance $d(v,t)$ from v to t for $d(v)$. The initialisation only requires $0(n+e)$ time. In the subsequent algorithm a node v will be called active if $\Delta(v) > 0$ and $0 < d(v) < n$. The idea of phase I is to let active nodes release as much flow as possible towards t, according to some iterative scheme, until no more active nodes remain. In the generic version of the algorithm, one of the following actions can be carried out in every step:

PUSH : Select an active node v and a residual edge (v,w) with $d(w) = d(v) - 1$. Let the edge have residual capacity r. Now send a flow of $\delta = \min\{\Delta(v), r\}$ from v to w, updating g and the flow excesses in v and w accordingly. (The push is called saturating if $\delta = r$ and non-saturating otherwise.)

RELABEL : Select a node v with $0 < d(v) < n$, and replace $d(v)$ by the minimum value of $d(w) + 1$ over all nodes w for which (v,w) is a residual edge (∞ if no such nodes exist).

Of course RELABEL steps are only useful if they actually change (i.e., increase) a $d(v)$-value. The following lemma is easily verified.

Lemma. Every step of the generic algorithm maintains the following invariants:

(i) g is a preflow,

(ii) d is a valid labeling, and

(iii) if $\Delta(v) > 0$ then there is a directed path from s to v in the residual graph.

Furthermore, the $d(v)$-values never decrease.

We now show that the generic algorithm always terminates within polynomially many steps regardless of the regime of PUSH and RELABEL instructions, provided we ignore RELABEL instructions that do not actually change a label ("void relabelings").

Theorem. Phase I terminates within $0(n^2e)$ steps, provided no void relabelings are counted.

Proof.

Every counted RELABEL step increments a $d(v)$-value with $v \neq t$ and $d(v) < n$ by at least one. Thus there can be at most $n-1$ RELABEL steps that affect a given $d(v)$-value for $v \neq t$, thus $0(n^2)$ RELABEL steps in all.

Next we estimate the number of saturating pushes involving a given edge $(u,v) \in E$. If $v = t$, the edge can be saturated at most once (because $d(t)$ remains 0 forever). For $v \neq t$, a saturating PUSH from u to v can only be followed (eventually) by another saturating PUSH over the same edge, necessarily from v to u, if $d(v)$ has increased by at least 2 in the meantime. A similar observation holds for $d(u)$ if we start with a saturating PUSH from v to u. (Note that $d(u)$ and $d(v)$ can only increase.) Thus the edge can be involved in saturating PUSH steps at most $n-2$ times, and the total number of saturating PUSH steps is bounded by $0(ne)$.

To estimate the number of non-saturating PUSH steps, we use the entropy function $\Phi = \sum_{v \text{ active}} d(v,t)$ with distances being measured in the residual graph of the current preflow. A saturating PUSH causes Φ to increase by at most $n-2$. A non-saturating

PUSH causes Φ to decrease by at least 1 (because the node sending flow pushes all its excess flow and thus turns non-active). As the total increase of Φ is thus bounded by $0(ne) \cdot (n-2) = 0(n^2e)$ and the initial value is bounded by $0(n^2)$, the number of non-saturating PUSH steps is bounded by $0(n^2e)$. It follows that Phase I terminates within $0(n^2e)$ steps, provided no void relabelings are counted as steps. □

One can implement Phase I to run in $0(n^2e)$ time, using a queue A to maintain the set of active nodes. It is clear from the theorem that we can do better only if we change the regime of instructions so as to get by with fewer non-saturating PUSH steps. Note that RELABEL steps are best executed when we visit an active node, and thus add only 0(1) overhead to a PUSH step. Assume that at every node the outgoing and incoming edges are stored in a circular list, with a pointer to the "next" edge that must be considered. (Initially it is a random first edge.) Consider the following type of step:

DISCHARGE : Select the next active node v from the front of A (and delete it from A). Consider the edges incident to v in circular order starting from the "current" edge, and apply PUSH/RELABEL steps until $\Delta(v)$ becomes 0 or $d(v)$ increases. (Note that at least one of the two must arise before a full cycle around the list is completed.) If a PUSH from v to some node u causes $\Delta(u) > 0$ (necessarily implying that u turns active), then add u to the rear of the queue A. After handling v, add it to the rear of A also, provided it is still active.

Define a "pulse" as follows. The first pulse consists of applying DISCHARGE to the initial queue $A = \{s\}$. For $i > 1$, the i^{th} pulse consists of applying the DISCHARGE steps to all node that were put into the queue during the $(i-1)^{st}$ pulse. Consider applying DISCHARGE steps until $A = \emptyset$.

Lemma. Phase I terminates after at most $0(n^2)$ pulses.

Proof.

Consider the entropy function $\Phi = \max\{d(v) | v \text{ active}\}$. If Φ does not decrease during a pulse, then the $d(v)$-value of some node must increase by at least one. Thus there can be at most $(n-1)^2$ pulses in which Φ does not decrease. As $0 < \Phi < n$ until there are no more active nodes, the total number of pulses in which Φ decreases can be no more than $(n-1)^2+(n-1)$. Thus Phase I terminates after $2(n-1)^2+(n-1) = 0(n^2)$ pulses. □

Observe that by the way a DISCHARGE step is defined, there can be at most one non-saturating PUSH step for each node v (namely the one that makes $\Delta(v)$ 0) in every pulse. Thus the number of non-saturating PUSH steps in this implementation of phase I is bounded by $0(n^2) \cdot (n-1) = 0(n^3)$. Given that there are at most $0(n^2)$ RELABEL steps and $0(ne)$ saturating PUSH steps as proved in the theorem, it easily follows that this implementation of phase I has a total runtime bounded by $0(n^3)$. Goldberg and Tarjan have shown that by exploiting very special data structures and yet another regime to cut down on the number of non-saturating PUSH steps, the complexity of phase I can be further reduced to $0(ne \log^{n^2}/e)$.

When phase I terminates, it does not necessarily terminate with a maximum flow! (Indeed g need not be a legal flow yet.) But the following lemma confirms the intuition that the current preflow saturates at least one $s-t$ cut. Let $X = \{u \in G | d(u) \geq n\}$ (the set of

nodes from which t is not reachable in the residual graph) and $\overline{X} = \{v \in G | 0 < d(v) < n\}$ (the set of nodes from which t is reachable in the residual graph). Because phase I has terminated and (thus) no more active nodes exist, it should be clear that $X; \overline{X}$ indeed is an $s - t$ cut.

Lemma. The g-flow across the cut $X; \overline{X}$ equals the capacity of the cut.
Proof.

Consider any edge (u, v) or (v, u) with $u \in X$ and $v \in \overline{X}$. Because d is a valid labeling w.r.t. the residual graph at all times, we can have $d(u) \geq n$ only if (u, v) and (v, u) have no residual capacity left. This means that the flow through (u, v) is maximum and the flow through (v, u) is 0. □

Phase II of Goldberg's algorithm consists of eliminating the excess flow in all nodes except s and t, without affecting the value of the flow across the cut $X; \overline{X}$ just defined. It follows that the resulting legal flow must be maximum and ipso facto that $X; \overline{X}$ must have been a minimum capacity $s - t$ cut.

Algorithmically phase II can be implemented in several ways. In one approach all circulations are eliminated first by means of an algorithm of Sleator and Tarjan in $0(e \log n)$ time. (Note that no circulation or "cycle of flow" can go across the $X; \overline{X}$ cut and that their elimination does not change the value of the flow.) Next the nodes of G are processed in reverse topological order, and for every node v with $\Delta(v) > 0$ the inflow is reduced by a total of $\Delta(v)$. This increases the flow excess in the predecessors, but they are handled next anyway. (Note that the nodes in $\overline{X}$ have flow excess 0 and thus remain unaffected, as does the flow across the $X; \overline{X}$ cut.) The entire procedure remains bounded by $0(m \log n)$ time. Originally Goldberg formulated a very similar algorithm to phase I for taking away excess flow. Its complexity was again $0(n^3)$.

Goldberg's approach has led to the most efficient maximum flow algorithm to date, with a running time of $0(ne \log^{n^2} / e)$ time when a suitable implementation is made. At the same time his algorithm is ideally suited for parallel or distributed computation models.

3.5 Related Flow Problems.

There is a large variety of optimization problems on graphs that can be formulated in terms of flows in networks. We briefly review the main results in this direction.

3.5.1 Networks with Lowerbounds.

It is common to also impose a lowerbound on the amount of flow through each edge, in addition to the given upperbound ("capacity") of each edge. The main problem now is to find a feasible flow, i.e., a flow that satisfies the given constraints in each edge. Taking a feasible flow as the starting point, any of the maximum flow algorithms based on flow augmentation can be called upon to construct a maximum feasible flow. In a similar way the minimum feasible flow can be constructed. For both types of flow there are results relating the value of the flow to the capacity of cuts (see section 3.2).

Lawler describes the following procedure for finding a feasible flow (if one exists). Add the edge (t, s) with lowerbound 0 and capacity ∞, and start with a suitable initial circulation f that satisfies the capacity constraints (e.g. $f \equiv 0$). If f is not feasible, there

will be an edge (u, v) for which $f(u, v) < l(u, v)$. Find a flow augmenting path from v to u in which each backward edge carries an amount of flow strictly larger than its lowerbound. Now augment the flow from v to u along the path (and, of course, along the edge (u, v)) by the maximum amount δ that keeps the flow within the constraints. Repeat this until $f(u, v) \geq l(u, v)$ and likewise for every other edge. The procedure will construct a feasible flow provided the conditions of Hoffman's circulation theorem are satisfied (see section 3.2). When suitably implemented, the algorithm finds a feasible flow in $0(n^3)$ time or less.

Curiously the problem of finding a feasible flow seems to be much harder for undirected networks. In this case a (feasible) flow is defined to be any function $f : V \times V \rightarrow \mathbf{R}$ with the following properties:(i) $f(i, j) \neq 0$ only if $(i, j) \in E$ and $f(i, j) = -f(j, i)$, (ii) $l(i, j) \leq |f(i, j)| \leq c(i, j)$ and (iii) $\sum_{u \in V} f(u, j) = 0$ for all $j \neq s, t$. (Interpret $f(i, j) > 0$ as the amount of flow from i to j.) It is assumed that $l(i, j) = l(j, i)$ and $c(i, j) = c(j, i)$. Itai has shown that the problem of determining whether there exists a feasible flow is an undirected network with lower and upper bounds is NP-complete.

3.5.2 Minimum Cost Flow.

Next suppose that each unit of flow through an edge (i, j) takes a cost $b(i, j) \geq 0$, and consider the problem of constructing a minimum cost flow of value v (if it exists). The problem is discussed at length in Ford and Fulkerson. A classical result of Wagner shows that the minimum cost flow problem is polynomially equivalent to the so-called transportation problem (or "Hitchcock problem") in linear programming, which has been extensively studied. Assuming integer capacities, Edmonds and Karp developed an algorithm for finding a minimum cost maximum flow exploiting this connection, achieving a running time polynomial in n, e and $\log |f^*|$ (with f^* the value of the maximum flow).

The minimum cost flow theorem suggests the following algorithm for finding a minimum cost flow of value v. Set f° equal to the zero flow and construct a sequence of (minimum cost) flows $f^1, f^2, \cdots$ as follows, for as long as $|f^k| < v$ and f^k isn't maximal: find an $s - t$ augmenting path π of minimum cost for f^k, and construct a new flow (f^{k+1}) by augmenting the flow by $\min\{res(\pi), v - |f^k|\}$ along π. The algorithm is known as the minimum cost augmentation algorithm. Zadeh has shown that in general the algorithm is not polynomially bounded in the parameters of the network. If all capacities are integers, the algorithm clearly is polynomial, as there can be at most $v + 1$ stages in the algorithm (each augmentation increments the flow value by an integer ≥ 1) and each stage requires the computation of a minimum cost path from s to t in the residual graph (which takes $0(ne)$ time when a general single source shortest path algorithm is used, where we note that the residual graph of any minimum cost flow is known to have no negative cost cycles). The shortest path computation in every stage can be considerably simplified by the following obervation due to Edmonds and Karp.

Lemma. The minimum cost augmenting path computation required in each stage can be carried out in a network with all weights (costs) non-negative.

Using this fact one can show that for integer capacity networks the minimum cost augmentation algorithm produces a minimum cost flow of value v in about $0(vS(n, e))$ time, with $S(n, e)$ the time needed for solving a single-source shortest path problem in a network with all weights non-negative. The best bound known to date is $S(n, e) = e + n \log n$ (see

section 2.3).

For general networks the minimum cost augmentation algorithm may behave very poorly. Zadeh showed that even convergence of the algorithm is not guaranteed. Consider the following variant

Edmonds-Karp minimum cost augmentation : find a shortest minimum cost augmenting path π and augment the flow by $\min\{res(\pi), v - |f^k|\}$ along π.

(Here f^k denotes the flow in the k^{th} iteration of the algorithm.) Assume that Edmonds-Karp minimum cost augmentation is used in every stage of the minimum cost flow algorithm.

Theorem. Edmonds-Karp minimum cost augmentation produces a minimum cost flow of value v in finitely many steps.

Proof.

We need the following property of the general minimum cost augmentation algorithm. Consider the k^{th} flow (f^k) and let $\pi^k(v)$ denote the minimum cost of a path from s to v in the residual graph R^k. For $(u,v) \in R^k$ we have $\pi^k(u) + b \geq \pi^k(v)$, where b is the cost of the edge ($b(u,v)$ when $(u,v) \in E$ and $-b(u,v)$ when $(u,v) \in \overleftarrow{E}$) and equality holds if (u,v) is on a minimum cost path from s to v. Consider the flow f^{k+1} obtained after augmenting the flow along a minimum cost $s-t$ path (in R^k) and its corresponding residual graph R^{k+1}. R^{k+1} consists of a subset of the edges of R^k and, possibly, some new edges. One verifies that the only edges $(v,u) \in R^{k+1} - R^k$ are edges such that $(u,v) \in R^k$ and $\pi^k(u) + b = \pi^k(v)$ with b as before (using that the augmentation was done along a minimum cost path). It easily follows that $\pi^{k+1}(v) \geq \pi^k(v)$ for all nodes v. Taking $v = t$ this shows that the cost of the successive augmenting paths used in the minimum cost augmentation algorithm is non-decreasing.

Now consider the Edmonds-Karp minimum cost augmentation algorithm. We know that $\pi^k(t)$ is non-decreasing in k. During any period i which $\pi^k(t)$ remains constant, Edmonds-Karp minimum cost augmentation behaves exactly like Edmonds-Karp augmentation in a maximum flow algorithm. By theorem (page 58) it follows that $\pi^k(t)$ remains constant for at most $\frac{1}{2}ne$ iterations in a row before it necessarily must increase. But $\pi^k(t)$ can only increase a finite number of times because, for each k, $\pi^k(t)$ is the cost of some loop-free path from s to t and the number of such paths is finite. □

In 1985 Tardos obtained a polynomial time minimum cost flow algorithm for the general case, with the additional property that the size of the numbers occurring during the computation remains polynomially bounded in the size of the problem as well. Fujishige gave a different approach, leading to an $0(e^3 \log n)$ algorithm. Galil and Tardos improved the result further to obtain an $0(n^2 e \log n + n^3 \log^2 n)$ algorithm for the minimum cost flow problem.

3.5.3 Multiterminal Flow.

The multiterminal flow problem (i.e., the "analysis" version of it) requires that we compute the maximum flow between every two of p given nodes in a network G, for some $p > 1$. Clearly this can be accomplished in polynomial time, by solving $\frac{1}{2}p(p-1)$ maximum flow problems (one for each pair of distinct nodes). Sometimes a much more efficient

algorithm can be found. A classical case is provided by the "symmetric" networks, which have $(v,u) \in E$ whenever $(u,v) \in E$ (i.e., the network is essentially undirected) and $c(v,u) = c(u,v)$. We outline the results of Gomory and Hu that show that for symmetric networks the multiterminal flow problem can be answered by solving only $p-1$ maximum flow problems. Several ramifications of the problem will be discussed afterwards.

Let G be symmetric, and let v_{ij} be the value of the maximum flow from i to j ($1 \leq i,j \leq n$). Let $v_{ii} = \infty$ for all i, for consistency.

Lemma.

(i) For all $i,j (1 \leq i,j \leq n), v_{ij} = v_{ji}$.

(ii) For all $i,j,k (1 \leq i,j,k \leq n), v_{ik} \geq \min\{v_{ij}, v_{jk}\}$.

(iii) There exists a tree T on the nodes 1 through n such that for all $i,j (1 \leq i,j \leq n) v_{ij} = \min\{v_{ij_1}, v_{j_1 j_2}, \cdots, v_{j_k j}\}$, where $i - j_1 - \cdots - j_k - j$ is the (unique) path from i to j in T.

Proof.

(i) By symmetry of the network.

(ii) Let $X; \overline{X}$ be a minimum capacity $i-k$ cut saturated by a maximum flow from i to k, i.e., $v_{ik} = c(X, \overline{X})$. If $j \in \overline{X}$ then necessarily $v_{ij} \leq c(X, \overline{X}) = v_{ik}$. If $j \in X$ then $v_{jk} \leq c(X, \overline{X}) = v_{ik}$.

(iii) Consider the complete graph on n nodes with the edges (i,j) labeled by the "weight" $v_{ij} (= v_{ji})$. Let T be a maximum weight spanning tree. By an inductive argument using (ii) one verifies that for each pair of nodes $i,j (1 \leq i,j \leq n)$ $v_{ij} \geq \min\{v_{ij_1}, v_{j_1 j_2}, \cdots, v_{j_k j}\}$, where $i - j_1 - \cdots - j_k - j$ is the path from i to j in T. Suppose for some i,j there is strict inequality. Then there is an edge (i',j') on the path between i and j with $v_{ij} > v_{i'j'}$. This means that the tree T' obtained by replacing the edge (i',j') of T by the edge (i,j) will have a larger weight than T. Contradiction. □

Part (iii) of the lemma has the interesting consequence that among the $\frac{1}{2}n(n-1)$ maximum flow values v_{ij} there exist at most $n-1$ different values (namely, the values associated with the edges of T). It also leads to the intuition that perhaps $n-1$ maximum flow computations suffice to build a tree T with the desired property, in order to determine all v_{ij}-values. An algorithm due to Gomory and Hu indeed achieves this.

The crucial observation of Gomory and Hu is expressed in the following lemma. Given two nodes i,j and a minimum capacity $i-j$ cut $X; \overline{X}$, define the "condensed" network G^c as the network obtained from G by contracting the nodes in $\overline{X}$ to a single (special) node $u_{\overline{X}}$ and replacing all edges leading from a node $v \in X$ across the cut by a single edge between v and $u_{\overline{X}}$ having the total capacity (for all $v \in X$).

Lemma. For every two ordinary nodes i',j' of G^c (i.e., i' and $j' \in X$), the maximum flow from i' to j' in G^c has the same value as the maximum flow from i' to j' in the original network.

(Note that for nodes $i', j' \in \overline{X}$ a similar statement holds after contracting X in G.) An important consequence of the lemma is that for every two nodes $i', j' \in X$ (or $\overline{X}$) there is a minimum capacity $i' - j'$ cut such that all nodes of $\overline{X}$ (or X, respectively) are on one side of the cut. Given two nodes i, j and a minimum capacity $i-j$ cut $X; \overline{X}$, we represent the current situation by a tree with two nodes, one representing X and the other $\overline{X}$, and one edge with label v_{ij}. (Thus the edge "represents" the cut $X; \overline{X}$.) Of course we begin by choosing i and j from among the p nodes between which the maximum flow value must be computed. We will now show how to expand the tree. For convenience we denote the nodes of the tree by capital letters. Let A be the set of designated nodes, $|A| = p$.

Definition. A tree T is called a semi-cut tree if it satisfies the following properties:

(i) every node U of T corresponds to a subset of the nodes in G and contains at least one node of A,

(ii) every edge (U, V) carries one label v, and there are nodes $i, j \in A$ with $i \in U$ and $j \in V$ such that the maximum flow between i and j has value v,

(iii) every edge (U, V) represents a minimum capacity $i - j$ cut with $i, j \in A$ and i contained in one of the nodes of T in the subtree headed by U and j contained in one of the nodes of T in the subtree headed by V (and the cut consisting of the two, collective sets of nodes).

If every node of a semi-cut tree T contains precisely one node of A, then T is called a cut tree for A. (If A consists of all nodes of G, then a cut tree for A is simply called a cut tree.)

Theorem. Let T be a cut tree for A. Then for every $i, j \in A$ one has: $v_{ij} = \min\{v_1, \cdots, v_{k+1}\}$, where v_1 through v_{k+1} are the edge labels in T on the path from the (unique) node containing i to the (unique) node containing j.

Proof.

Let $i - j_1 - \cdots - j_k - j$ be the "path" of A-nodes leading from i to j in T. (Each A-node listed is in fact the unique A-node contained in the corresponding node on the path in T.) By the properties of the cut tree the labels on the edges are simply $v_{ij_1}, \cdots, v_{j_kj}$. By the lemma we have $v_{ij} \geq \min\{v_{ij_1}, \cdots, v_{j_kj}\}$. On the other hand each edge corresponds to a cutset separating i and j, with a capacity equal to the corresponding v-label. Thus $v_{ij} \leq \min\{v_{ij_1}, \cdots, v_{j_kj}\}$ and the desired equality follows. □

The algorithm of Gomory and Hu simply begins with the 2-node semi-cut tree for A that we outlined above and shows how it can be transformed in a step-by-step fashion into a cut tree for A. By the theorem a cut tree for A has precisely the property we want.

Theorem. A cut tree for A exists and can be constructed by means of only $p - 1$ maximum flow computations.

Proof.

Suppose we have a semi-cut tree T for A. (This is certainly true at the beginning of the algorithm.) Suppose T still has a node U which contains two nodes $i, j \in A$. By the lemma the maximum flow between i and j in G has the same value as the maximum flow

between i and j in the condensed graph G^c obtained by contracting the sets of nodes in each subtree attached to U to a single (special) node and for each $v \in U$ replacing the edges that lead from v to a node in a particular subtree by one edge with the combined total capacity between v and the corresponding special node. Let $X; \overline{X}$ be a minimum capacity $i-j$ cut in G^c (necessarily with capacity v_{ij}). Construct a tree T' as follows: split U into the nodes X and $\overline{X}$, connect X and $\overline{X}$ by an edge labeled v_{ij}, and (re-) connect every neighbor V of U to either X or $\overline{X}$ depending on which side of the cut the special node corresponding to $V's$ subtree was in G^c (with original edge label).

We claim that T' is again a semi-cut tree for A. The only non-trivial property to verify is part (ii) of the definition. Thus let V be any neighbor of U in T and let the label on the edge (U, V) corespond to the maximum flow value between $r \in U$ and $s \in V (r, s \in A)$. Assume without loss of generality that $j, r \in \overline{X}$. Now the following two cases can arise in the construction:

case I. V gets connected to $\overline{X}$.
This trivially preserves the properties of te semi-cut tree.

case II. V gets connected to X.
This situation is more subtle (see figure 13).

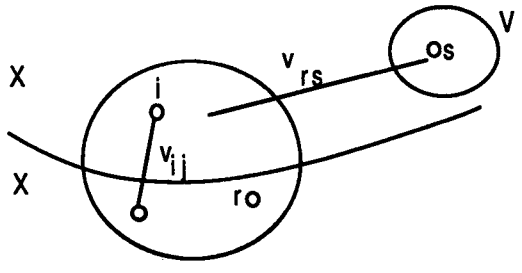


Figure 13:

Clearly the label v_{ij} on the edge $(X, \overline{X})$ is appropriate to satisfy the definition for the new edge. Now consider the label (v_{rs}) on the edge (X, V). We show that, in fact, $v_{is} = v_{rs}$. The argument is as follows. By the lemma we know $v_{is} \geq \min\{v_{ij}, v_{jr}, v_{rs}\}$. Because i and s are on one side of the cut, we know that the flow values between nodes in $\overline{X}$ do not affect v_{is} and we have $v_{is} \geq \min\{v_{ij}, v_{rs}\}$. On the other hand v_{is} must be bounded by the capacity of the minimum cut coresponding to the edge $(U, V) \in T$, i.e., $v_{is} \leq v_{rs}$. By the lemma $v_{ij} \geq \min\{v_{is}, v_{rs}\}$, hence $v_{ij} \geq v_{rs}$. It follows that $v_{is} = v_{rs}$, and the edge (X, V) corresponds to the maximum flow between i and s (with the same value and the same cut-set corresponding to it as in the original tree T).

By repeating the construction it is clear that a cut tree for A is obtained, at the expense of a total of $p-1$ maximum flow (i.e., minimum cut) computations. □

The Gomory-Hu construction of a cut tree clearly solves the multiterminal flow problem in an elegant fashion and saves a large number of computations over the naive algorithm. Using the maximum flow algorithm of Goldberg and Tarjan, the construction needs essentially $0(pne \log^{n^2}/e)$ time. Hu and Shing have shown that a cut tree for a network with t

triconnected components can be computed in $0(\frac{n^4}{i^3}+n^2+t)$ time. For outerplanar graphs a cut tree can be constructed in linear time, for planar graphs in $0(\frac{n^2 \log^* n}{\log n})$ time.

Several ramifications of the multiterminal flow problem can be considered. Gupta has shown that the results of Gomory and Hu hold for the larger class of so-called pseudosymmetric networks, which are defined as the networks in which there exists a circulation that saturates all edges. The multiterminal flow problem can also be studied for e.g. minimum cost flows, but few results along these lines have been reported. Some attention has been given to the "synthesis" version of the multiterminal flow problem. In its simplest form the question is to determine a network (i.e., a graph and suitable capacities for the edges) that realizes a given set of maximum flow values. For the restriction to symmetric networks again some results are known. The following result is due to Gomory and Hu.

Theorem. The values $v_{ij}(1 \leq i,j \leq n)$ are the maximum flow values of an n-node symmetric network if and only if the values are non-negative, for all $i,j(1 \leq i,j \leq n)\, v_{ij} = v_{ji}$ and for all $i,j,k(1 \leq i,j,k \leq n)\, v_{ik} \geq \min\{v_{ij}, v_{jk}\}$.

Proof.

The necessity follows from the lemma on page 67. The sufficiency follows by considering the tree T constructed in part (iii) of the same lemma. Let the edge labels in T serve as edge capacities. One verifies that the v_{ij}-values actually arise as maximum flow values in the network so obtained. □

In its more interesting form, the synthesis version of the multiterminal flow problem asks for a network that realizes a given set of maximum flow values (or a set of flows with these values as lowerbounds) with minimum total edge capacity. For the symmetric case Gomory and Hu have again devised an efficient algorithm for this problem. Gusfield has shown that one can construct an optimal symmetric network G in $0(\max\{e, n \log n\})$ time, where the network G that is obtained has the additional properties that it is planar, has degrees bounded by 4 and the smallest possible number of edges (in a well-defined sense).

3.5.4 Multicommodity Flow.

In the discussion of (maximum) flow problems we have hitherto assumed that the flow concerns a single "commodity" that must be transported from a source node s to a target node t. The $m-$commodity (maximum) flow problem is the natural generalization in which there are m commodities ($m \geq 2$) and m source-target pairs (s_i, t_i). The problem is to simultaneously transport any required number of units of the i^{th} commodity from s_i to t_i for $1 \leq i \leq m$ through the same network, i.e., under the constraint that the total flow of commodities 1 to m through each edge is bounded by the capacity of the edge. Let f_i denote the flow function for the i^{th} commodity ($1 \leq i \leq m$). Traditionally one seeks to maximize the total flow of commodities $\sum_1^m |f_i|$, or even to maximize the flow and at the same minimize the total cost of all flows (assuming that cost-functions are defined for the edges as in the minimum cost flow problem). Both problems are easily formulated as linear programming problems, and early studies by Ford and Fulkerson and by Tomlin have attempted to exploit the special structure of these linear programs to obtain practical algorithms. Because of the discovery of several polynomial time-bounded linear programming algorithms (e.g. Khachian's ellipsoid method), we know that both the maximum multicommodity flow problem and the minimum cost multicommodity flow

problem an be solved in polynomial time when all network parameters are rational. Kapoor and Vaidya have adapted Karmarkar's linear programming algorithm to show that both multicommodity flow problems can be solved in $0(m^{3.5}n^{2.5}eL)$ steps, where each step involves an arithmetic operation to a precision of $0(L)$ bits. (L is the size of the network specifiation in bits.) Assad has given an excellent account of the linear programming aspects of the many variants of the multicommodity flow problem. Itai has shown that the maximum m-commodity flow problem is in fact polynomially equivalent to linear programming, for every $m \geq 2$.

In this section we restrict ourselves to the (maximum) m-commodity flow problem in undirected networks. The i^{th} commodity flow ($1 \leq i \leq m$) is to be a function $f_i : V \text{ x } V \rightarrow \mathbf{R}$ that satisfies the following properties:(i) $f_i(u,v) \neq 0$ only if $(u,v) \in E$ and $f_i(u,v) = -f_i(u,v)$, (ii) $0 \leq |f_i(u,v)| \leq c(u,v)$ and (iii) $\sum u \in V f_i(u,v) = 0$ for all $v \neq s_i, t_i$. It is assumed that $c(u,v) > 0$ only if $(u,v) \in E$ and that $(u,v) = c(v,u)$ for all u, v. (Interpret $f_i(u,v) > 0$ as the flow of the i^{th} commodity from u to v). The value of f_i is defined to be $|f_i| = f_i(s_i, t_i)$. An m-commodity flow is called feasible if all $f_i(1 \leq i \leq m)$ are legal flows and for every $(u,v) \in E : |f_i(u,v)| + \cdots + |f_m(u,v)| \leq c(u,v)$. An m-commodity flow is called maximum if it is feasible and the total flow $|f_1| + \cdots + |f_m|$ is maximum (over all feasible flows).

The m-commodity flow problem in undirected networks is especially interesting for $m = 2$, because in this case there is an analog to the max-flow min-cut theorem that leads to a fairly efficient algorithm. Let $c(s_1, s_2; t_1, t_2)$ be the minimum capacity of any cut separating s_i and $t_i (1 \leq i \leq 2)$. Let $\tau(u_1, u_2; v_1, v_2)$ denote the minimum capacity of any cut $X; \overline{X}$ such that $u_1, u_2 \in X$ and $v_1, v_2 \in \overline{X}$ (i.e., u_1 and u_2 are on the same side of the cut, as are v_1 and v_2). The following fact is due to Hu.

Lemma. In undirected networks $c(s_1, s_2; t_1, t_2) = \min\{\tau(s_1, s_2; t_1, t_2), \tau(s_1, t_2; s_2, t_1)\}$.

Next define $c(s_i; t_i)$ as the minimum capacity of any $s_i - t_i$ cut (for $1 \leq i \leq 2$). Observe that for every feasibly flow: $|f_i| \leq c(s_i; t_i)(1 \leq i \leq 2)$, by virtue of the max-flow min-cut theorem. The theory of the 2-commodity flow problem is governed by the following results due to Hu and complemented by Itai.

Theorem.

(i) ("the 2-commodity feasible flow theorem") There exists a feasible 2-commodity flow with $|f_i| = v_i (1 \leq i \leq 2)$ if and only if $v_1 \leq c(s_1; t_1), v_2 \leq c(s_2; t_2)$ and $v_1 + v_2 \leq c(s_1, s_2; t_1, t_2)$.

(ii) ("the 2-commodity max-flow min-cut theorem") A 2-commodity flow is maximum if and only if it is feasible and $|f_1| + |f_2| = c(s_1, s_2; t_1, t_2)$.

Theorem.

(i) ("the 2-commodity maximum component flow theorem") For each $i, 1 \leq i \leq 2$, there exists a maximum 2-commodity flow in which $|f_i|$ is maximum as well (as a single commodity flow).

(ii) ("the even integer theorem") If all capacities are even integers, there is a maximum 2-commodity flow f_1, f_2 which is integral (i.e., with all flow values integer).

(iii) ("the half units theorem") If all capacities are integers, there is a maximum 2-commodity flow f_1, f_2 in which all flow values are integer multiples of $\frac{1}{2}$.

Hu has shown that most of the results do not generalize to maximum m-commodity flows for $m > 2$. Rothschild and Whinston have shown that an integral maximum 2-commodity flow exists also when all capacities are integers and for each node v $\sum_{u \in V} c(u,v)$ is even.

The proofs of the (maximum) 2-commodity flow theorems essentially all follow from an algorithm due to Hu for constructing a maximum 2-commodity flow. It will be useful to keep the analogy with the ordinary maximum flow problem in mind. The idea of Hu's algorithm is to start with a maximum flow for the first commodity and a zero flow for the second, and then to use a suitable augmentation device to increase the flow of the second commodity while recirculating some flow of the first (without changing its value). Define a bi-path to be any pair of loop-free paths (π_α, π_β) such that π_α leads from s_2 to t_2 and π_β from t_2 to s_2. Given a feasible 2-commodity flow f_1, f_2 define for each (directed) edge $(u,v) \in E$: $A(u,v) = \frac{1}{2}\{c(u,v) - f_1(u,v) - f_2(u,v)\}$ and $B(u,v) = \frac{1}{2}\{c(u,v) - f_1(u,v) - f_2(v,u)\}$. (Note that $A(u,v)$ is half the "residual capacity" of the flow from u to v, and that $B(u,v)$ is half the "residual capacity" for augmenting the first flow from u to v and the second from v to u.) For paths π_α and π_β as above define $A(\pi_\alpha) = \min\{A(u,v) | (u,v) \in \pi_\alpha\}$ and $B(\pi_\beta) = \min\{B(u,v) | (u,v) \in \pi_\beta\}$. Define the residual capacity of a bi-path $\pi = (\pi_\alpha, \pi_\beta)$ as $res(\pi) = \min\{A(\pi_\alpha), B(\pi_\beta)\}$. A bi-path π is called an augmenting bi-path if $res(\pi) > 0$. Given an augmenting bi-path $\pi = (\pi_\alpha, \pi_\beta)$ we can augment the current 2-commodity flow by "bi-augmentation" as follows: increase the flow of both commodities by $res(\pi)$ along π_α ("going from s_2 to t_2"), and increase the flow of the first commodity and decrease the flow of the second commodity by $res(\pi)$ along π_β ("going from t_2 to s_2").

Lemma. Bi-augmentation preserves the feasibility of 2-commodity flows.

Theorem. ("the augmenting bi-path theorem") A 2-commodity flow with f_1 a maximum flow is maximum if and only if it is feasible and admits no augmenting bi-path.

Proof.

Let f_1, f_2 be a maximum 2-commodity flow. Suppose there was an augmenting bi-path π. Bi-augmentation along π leaves the value of f_1 unchanged but increments the value of f_2 by $2res(\pi)$. This contradicts that f_1, f_2 is maximum. Next let f_1, f_2 be a feasible with f_1 maximum, and suppose there exists no augmenting bi-path. By the nature of the problem we must have $|f_1| \leq c(s_1; t_1), |f_2| \leq c(s_2; t_2)$ and $|f_1| + |f_2| \leq c(s_1, s_2; t_1, t_2)$. We show that $|f_1| + |f_2| = c(s_1, s_2; t_1, t_2)$, which means that the 2-commodity flow f_1, f_2 must be maximum. We distinguish two cases:

case I. There exists no "forward" path π_α with $A(\pi_\alpha) > 0$.

Let X be the set of all nodes reachable from s_2 by a loop-free path of edges (u,v) with $A(u,v) > 0$. Clearly $X; \overline{X}$ is an $s_2 - t_2$ cut and for all $(u,v) \in E$ with $u \in X$ and $v \in \overline{X}$ we must have $A(u,v) = 0$, i.e., $f_1(u,v) + f_2(u,v) = c(u,v)$. Because $|f_1(u,v)| + |f_2(u,v)| \leq c(u,v)$ by feasibility, it follows that $f_1(u,v)$ and $f_2(u,v)$ are both non-negative and (also) that (u,v) is "saturated".

If $f_1(u,v)$ is zero for all edges across the cut, then the cut is entirely saturated by f_2 and $|f_2| = c(s_2;t_2)$ (by the max-flow min-cut theorem). It follows that f_1, f_2 must be a maximum 2-commodity flow in this case, with $|f_1| + |f_2| = c(s_1;t_2) \geq \tau(s_1,s_2;t_1,t_2)$ and (hence) $|f_1| + |f_2| = c(s_1,s_2;t_1,t_2)$. Next, consider the case that $f_1(u,v) > 0$ for some edge (u,v) across the cut. Now $s_1 \in X$ and $t_1 \in \overline{X}$, otherwise there would be an edge across the cut with negative f_1-flow (to conserve flow) and this is impossible. Thus $X;\overline{X}$ also is an $s_1 - t_1$ cut, and we have $|f_1| + |f_2| = c(X,\overline{X}) \geq \tau(s_1,s_2;t_1,t_2)$. Thus $|f_1| + |f_2| = c(s_1,s_2;t_1,t_2)$ and the flow is again maximum.

case II. There is no "backward" path π_β with $B(\pi_\beta) > 0$.

Similar to case I, now proving that $|f_1| + |f_2| \geq \tau(s_1,t_2;s_2,t_1)$. Using Hu's lemma, the maximality of f_1, f_2 again follows. □

The augmenting bi-path theorem provides exactly the tool we need to construct maximum 2-commodity flows, in perfect analogy to the single commodity ase. Augmenting bi-path methods are based on the iteration of one of the following types of steps, starting with a maximum flow for f_1 and any flow f_2 such that f_1, f_2 is a feasible 2-commodity flow (e.g. $f_2 \equiv 0$). (To obtain a feasible 2-commodity flow with pre-assigned values for $|f_1|$ and $|f_2|$ the steps should be suitably modified.)

Hu bi-augmentation : find a bi-augmenting path π and augment the 2-commodity flow by $res(\pi)$ along π.

Itai bi-augmentation : find a bi-augmenting path $\pi(\pi_\alpha,\pi_\beta)$ with π_α and π_β of shortest possible length, and augment the 2-commodity flow by $res(\pi)$ along π.

Both types of bi-augmentation can be implemented in $0(e)$ time per step.

Clearly Hu bi-augmentation yields a maximum 2-commodity flow with all the desired properties in finitely many steps in case the capacities are integers. For arbitrary (real) capacities Hu bi-augmentation may not converge to a maximum 2-commodity flow, very much like Ford-Fulkerson augmentation may not converge to a maximum flow in the single commodity flow problem. Itai bi-augmentation is reminiscent of Edmonds-Karp augmentation in the ordinary maximum flow case, and can be shown to lead to a maximum 2-commodity flow in $0(n^2e)$ time, for general capacities. Itai has shown that an $0(n^3)$ algorithm can be obtained by suitably adapting Karzanov's maximum flow algorithm to the 2-commodity case. Note that the fact that Itai bi-augmentation terminates proves the 2-commodity max-flow min-cut theorem for general undirected networks. (Seymour has given a proof that does not rely on this fact, solely based on the circulation theorem.) One easily verifies that the maximum 2-commodity flow obtained satisfies the additional properties of the theorem. The half-units theorem is perhaps most interesting of all. Even, Itai and Shamir have shown that the undirected 2-commodity flow problem becomes NP-complete once we insist on both flows being integral!

Hu's theory for the 2-commodity flow problem does not seem to generalize for $m > 2$. In particular there seems to be no suitable version of an m-commodity max-flow min-cut theorem. (Okamura and Seymour have shown that such a theorem does exist for planar graphs with all source and target nodes on the boundary of the infinite face.) Even the maximum m-commodity integral flow problem with all capacities equal to 1 is NP-complete: it can be recognized as the problem of finding m pairwise disjoint paths, with the i^{th} path connecting s_i and $t_i (1 \leq i \leq m)$.

4 References.

4.1 General.

[1] C. Berge. *Graphs and Hypergraphs.* North Holland Publ. Comp., Amsterdam, 1973.

[2] C. Berge. *The Theory of Graphs and Its Applications.* J. Wiley and Sons, New York, NY, 1962.

[3] B. Bollobás. *Graph Theory: An Introductory Course.* Volume 63 of *Graduate Texts in Mathematics*, Springer Verlag, New York, NY, 1979.

[4] J.A. Bondy and U.S.R. Murty. *Graph Theory with Applications.* Amer. Elsevier, NY (MacMillan, London), 1976.

[5] N. Christofides. *Graph Theory: An Algorithmic Approach.* Acad. Press, New York, NY, 1975.

[6] N. Deo. *Graph Theory with Applications to Engineering and Computer Science.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1974.

[7] J. Ebert. *Effiziente Graphenalgorithmen.* Akad. Verlagsgesellschaft, Wiesbaden, 1981.

[8] S. Even. *Graph Algorithms.* Computer Sci. Press, Potomac, Md, 1979.

[9] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, Cambridge, UK, 1985.

[10] M. Gondran and M. Minoux. *Graphs and Algorithms.* J. Wiley and Sons, Chichester, 1984.

[11] F. Harary. *Graph Theory.* Addison Wesley Publ. Comp., Reading, Mass., 1969.

[12] E. Lawler. *Combinatorial Optimisation: Network and Matroids.* Holt, Rinehart and Winston, New York, NY, 1976.

[13] W. Mayeda. *Graph Theory.* J. Wiley and Sons, New York, NY, 1972.

[14] K. Mehlhorn. *Graph Algorithms and NP-Completeness.* Volume 2 of *Data Structures and Algorithms*, Springer Verlag, Berlin, 1984.

[15] E. Minieka. *Optimisation Algorithms for Network and Graphs.* M. Dekker, New York, NY, 1978.

[16] R.T. Rockafellar. *Network Flows and Monotropic Optimization.* J. Wiley and Sons, New York, NY, 1984.

[17] R.E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, Pa., 1983.

[18] H.P. Yap. *Some Topics in Graph Theory.* Volume 108 of *London Math. Soc. Lecture Note Series*, Cambridge University Press, Cambridge, 1986.

4.2 Section 1.1 What is a graph.

[1] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. Freeman and Comp., San Francisco, Ca., 1979.

[2] J.E. Hopcroft and R.E. Tarjan. Efficient planarity testing. *J. ACM*, 21:549–568, 1974.

[3] M. Yannakakis. A polynomial time algorithm for the min-cut linear arrangement of trees. *J. ACM*, 32:950–988, 1985.

4.3 Section 1.2 Computer representations of graphs.

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publ. Comp., Reading, Mass., 1974.

[2] M.R. Best, P. van Emde Boas, and H.W. Lenstra Jr. *A Sharpened Version of the Aanderaa-Rosenberg Conjecture.* Technical Report ZW 30/74, Math. Centre, Amsterdam, 1974.

[3] B. Bollobás. Complete subgraphs are elusive. *J. Combin. Theory*, B-21:1–7, 1976.

[4] B. Bollobás and S.E. Eldridge. Packings of graphs and applications to computational complexity. *J. Combin. Theory*, B-25:105–124, 1978.

[5] H. Galperin and A. Wigderson. Succinct representations of graphs. *Inf. and Contr.*, 56:183–198, 1983.

[6] R.C. Holt and E.M. Reingold. On the time required to detect cycles and connectivity in graphs. *Math. Syst. Th.*, 6:103–106, 1972.

[7] N. Illies. A counterexample to the generalized Aanderaa-Rosenberg conjecture. *Inf. Proc. Lett.*, 7:154–155, 1978.

[8] A. Itai and M. Rodeh. Representation of graphs. *Acta Inf.*, 17:215–219, 1982.

[9] A.B. Kahn. Topological sorting of large networks. *C. ACM*, 5:558–562, 1962.

[10] K.N. King and B. Smith-Thomas. An optimal algorithm for sink-finding. *Inf. Proc. Lett.*, 14:109–111, 1982.

[11] D. Kirkpatrick. Determining graph properties from matrix representations. In *Proc. 6th Annual ACM Symp. Theory of Computing*, pages 84–90, Seattle, 1974.

[12] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[13] D. Kleitman and D.J. Kwiatkowski. Further results on the Aanderaa-Rosenberg conjecture. *J. Combin. Th.*, B-28:85–95, 1980.

[14] D.E. Knuth. *Fundamental Algorithms.* Volume 1 of *The Art of Computer Programming*, Addison-Wesley Publ. Comp., Reading, Mass., 1968.

[15] T. Lengauer. *Efficient Algorithms for Finding Minimum Spanning Forests of Hierarchically Defined Graphs.* Bericht nr. 26, Fachbereich Informatik, Gesamthochschule Paderborn, Paderborn, 1985.

[16] T. Lengauer. *Efficient Solution of Connectivity Problems on Hierarchically Defined Graphs.* Bericht 24, Fachbereich Informatik, Gesamthochschule Paderborn, Paderborn, 1985.

[17] T. Lengauer. *Hierarchical Graph Algorithms.* Technical Report SFB 124, Fachbereich 10, Angew. Mathematik und Informatik, Universität des Saarlandes, Saarbrücken, 1984.

[18] T. Lengauer and K. Wagner. *The Correlation between the Complexities of the Nonhierarchical and Hierarchical Versions of Graph Problems.* Preprint, Fachbereich Informatik, Gesamthochschule Paderborn, Paderborn, 1986.

[19] E.C. Milner and D.J.A. Welsh. On the computational complexity of graph theoretical properties. In C.St.J.A. Nash-Williams and J. Sheenan, editors, *Proceedings Fifth British Combin. Conference*, pages 471–487, Util. Math., Winnipeg, 1976.

[20] C.H. Papadimitriou and M. Yannakakis. A note on succinct representations of graphs. *Inf. and Contr.*, 71:181–185, 1986.

[21] R.L. Rivest and J. Vuillemin. On recognizing graph properties from adjacency matrices. *Theor. Comput. Sci.*, 3:371–384, 1976.

[22] A.L. Rosenberg. On the time required to recognize properties of graphs. *SIGACT NEWS*, 5(4):15–16, 1973.

[23] Y. Shiloach. Strong linear orderings of a directed network. *Inf. Proc. Lett.*, 8:146–148, 1979.

[24] A. Slisenko. Context-free grammars as a tool for describing polynomial time subclasses of hard problems. *Inf. Proc. Lett.*, 14:52–56, 1982.

[25] K. Wagner. The complexity of combinatiorial problems with succinct input representation. *Acta Inf.*, 23:325–356, 1986.

[26] K. Wagner. The complexity of problems concerning graphs with regularities. In *Proc. 11th Symp. Math. Foundations of Computer Science, Lecture Notes in Computer Science 176*, pages 544–552, Springer Verlag, Heidelberg, 1984.

[27] H.P. Yap. Computational complexity of graph properties. In K.M. Koh and H.P. Yap, editors, *Graph Theory, Singapore 1983, Lecture Notes in Mathematics 1073*, pages 35–54, Springer Verlag, Heidelberg, 1984.

4.4 Section 1.3 Graph exploration by traversal.

[1] E.F. Moore. The shortest path through a maze. In *Proc. Int. Symp. Switching Th. 1957, Part II*, pages 285–292, Harvard University Press, 1959.

[2] J.H. Reif and W.L. Sherlis. *Deriving Efficient Graph Algorithms.* TR-30-82, Aiken Computation Laboratory, Harvard University, Cambridge (Mass.), 1982.

[3] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

[4] G. Tarry. Le problème des labyrinthes. *Nouv. Ann. de Math.*, 14:187, 1895.

4.5 Section 1.4 Transitive reduction and transitive closure

[1] L. Adleman, K.S. Booth, F.P. Preparata, and W.L. Ruzzo. Improved time and space bounds for Boolean matrix multiplication. *Acta Inf.*, 11:61–75, 1978.

[2] A.V. Aho, M.R. Garey, and J.D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1:131–137, 1972.

[3] D. Angluin. The four Russians' algorithm for Boolean matrix multiplication is optimal in its class. *SIGACT News*, 8:29–33, 1976.

[4] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, and I.A. Faradžev. On economical construction of the transitive closure of an oriented graph. *Soviet Math. Dokl.*, 11:1209–1210, 1970.

[5] J. Baker. A note on multiplying Boolean matrices. *C. ACM*, 5:102, 1962.

[6] P.A. Bloniarz, M.J. Fischer, and A.R. Meyer. A note on the average time to compute transitive closures. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming (3rd Internat. Colloq.)*, pages 425–434, Edinburgh University Press, Edinburgh, 1976.

[7] J. Ebert. A sensitive transitive closure algorithm. *Inf. Proc. Lett.*, 12:255–258, 1981.

[8] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Inf.*, 8:303–314, 1977.

[9] M.J. Fischer and A.R. Meyer. Boolean matrix multiplication and transitive closure. In *Conf. Rec. 12th Annual IEEE Symp. on Switching and Automata Theory*, pages 129–131, 1971.

[10] M.E. Furman. Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Soviet Math. Dokl.*, 11:1252, 1970.

[11] A. Goralcikova and V. Konbek. A reduct and closure algorithm for graphs. In J. Bečvář, editor, *Mathematical Foundations of Computer Science, Proceedings 1979, Lecture Notes in Computer Science 74*, pages 301–307, Springer Verlag, Heidelberg, 1979.

[12] H.T. Hsu. An algorithm for finding a minimal equivalent graph of a digraph. *J. ACM*, 22:11–16, 1975.

[13] I. Ibaraki and N. Katoh. On-line computation of transitive closures of graphs. *Inf. Proc. Lett.*, 16:95–97, 1983.

[14] B. Jaumard and M. Minoux. An efficient algorithm for the transitive closure and a linear worst-case complexity result for a class of sparse graphs. *Inf. Proc. Lett.*, 22:163–169, 1986.

[15] V.V. Martynyuk. On economical construction of the transitive closure of binary relations. *Ž. Vyčisl. Mat. i. Mat. Fiz*, 2:723–725, 1962.

[16] D.M. Moyles and G.L. Thompson. An algorithm for finding a minimum equivalent graph of a digraph. *J. ACM*, 16:455–460, 1969.

[17] I. Munro. Efficient determination of the transitive closure of a directed graph. *Inf. Proc. Lett.*, 1:56–58, 1971.

[18] P.E. O'Neill and E.J. O'Neill. A fast expected time algorithm for Boolean matrix multiplication and transitive closure. *Inf. and Contr.*, 22:132–138, 1973.

[19] P. Purdom Jr. A transitive closure algorithm. *BIT*, 10:76–94, 1970.

[20] B. Roy. Transitivité et connexité. *Compt. Rend. Acad. Sci. Paris*, 249:216–219, 1959.

[21] S. Sahni. Computationally related problems. *SIAM J. Comp.*, 3:262–279, 1974.

[22] L. Schmitz. An improved transitive closure algorithm. *Comput.*, 30:359–371, 1983.

[23] C.P. Schnorr. An algorithm for transitive closure with linear expected time. *SIAM J. Comput.*, 7:127–133, 1978.

[24] K. Simon. *Ein neuer Algorithmus für die Transitive Hülle von Gerichteten Graphen.* Diplom-Arbeit, Fachbereich 10, Angewandte Mathematik und Informatik, Universität des Saarlandes, Saarbrücken, 1983.

[25] M.M. Syslo and J. Dzikiewicz. Computational experiences with some transitive closure algorithms. *Comput.*, 15:33–39, 1975.

[26] L.E. Thorelli. An algorithm for computing all paths in a graph. *BIT*, 6:347–349, 1966.

[27] J. van Leeuwen. Efficiently computing the product of two binary relations. *Int. J. Computer Math.*, 5:193–201, 1976.

[28] H.S. Warren Jr. A modification of Warshall's algorithm for the transitive closure of binary relations. *C. ACM*, 18:218–220, 1975.

[29] S. Warshall. A theorem on Boolean matrices. *J. ACM*, 9:11–12, 1962.

4.6 Section 1.5 Generating an arbitrary graph

[1] J.D. Dixon and H.S. Wilf. The random selection of unlabeled graphs. *J. Algor*, 4:205–213, 1983.

[2] A. Nijenhuis and H.S. Wilf. The enumeration of connected graphs and linked diagrams. *J. Combin. Th*, A-27:356–359, 1979.

[3] H.S. Wilf. The uniform selection of free trees. *J. Algor.*, 2:204–207, 1981.

4.7 Section 1.6 Recognition of graphs

[1] S. Arnborg, D.G. Corneil, and A. Pruskurowski. *Complexity of Finding Embeddings in a k-Tree.* Report TRITA-NA-8407, Dept. of Num. Anal. and Computer Sci., Royal Institute of Technology, Stockholm, 1984.

[2] T. Asano. *A Linear Time Algorithm for the Subgraph Homeomorphism Problem for the fixed Graph* $K_{3,3}$. Report AL 83-20, Dept. of Math. Engin. and Instrum. Physics, University of Tokyo, Tokyo, 1983.

[3] K.S. Booth and G.S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using *PQ*-trees. *J. Comp. Syst. Sci.*, 13:335–379, 1976.

[4] M. Farber. Characterizations of strongly chordal graphs. *Discr. Math.*, 43:173–189, 1983.

[5] I.S. Filotti. An efficient algorithm for determining whether a cubic graph is toroidal. In *Proc. 10th Annual ACM Symp. Theory of Computing*, pages 133–142, San Diego, 1978.

[6] I.S. Filotti, G.L. Miller, and J. Reif. On determining the genus of a graph in $O(v^{O(g)})$ steps. In *Proc. 11th Annual ACM Symp. Theory of Computing*, pages 27–37, Atlanta, GA, 1979.

[7] C.P. Gabor, W.L. Hsu, and K.J. Supowit. Recognizing circle graphs in polynomial time. In *Proc. 26th Annual IEEE Symp. Found. of Computer Science*, pages 106–116, Portland, 1985.

[8] F. Gavril. An algorithm for testing chordality of graphs. *Inf. Proc. Lett.*, 3:110–112, 1974.

[9] F. Gavril. A recognition algorithm for the intersection graphs of directed paths in directed trees. *Discr. Math.*, 13:237–249, 1975.

[10] F. Gavril. A recognition algorithm for the intersection graphs of paths in trees. *Discr. Math.*, 23:211–227, 1978.

[11] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Acad. Press, NY, 1980.

[12] J.E. Hopcroft and R.E. Tarjan. Efficient planarity testing. *J. ACM*, 21:549–568, 1974.

[13] D.S. Johnson. The NP-completeness column: An ongoing guide. *J. Algor.*, 6:434–451, 1985.

[14] P.G.H. Lehot. An optimal algorithm to detect a line graph and output its root graph. *J. ACM*, 21:569–575, 1974.

[15] A. Mansfield. Determining the thickness of graphs is NP-hard. *Math. Proc. Cambridge Phil. Soc.*, 93:9–23, 1982.

[16] S.L. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Inf. Proc. Lett.*, 9:229–232, 1979.

[17] J.B. Saxe. Dynamic programming algorithms for recognizing small-bandwidth graphs in polynomial time. *SIAM J. Algebr. and Discr. Meth.*, 1:363–369, 1980.

[18] M.M. Syslo. A labeling algorithm to recognize a line graph and output its root graph. *Inf. Proc. Lett.*, 15:28–30, 1982.

[19] R.E. Tarjan and M. Yannakakis. Simple linear time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13:566–579, 1984.

[20] A. Tucker. An efficient test for circular-arc graphs. *SIAM J. Comput.*, 9:1–24, 1980.

[21] A. Tucker. Matrix characterization of circular-arc graphs. *Pacific J. Math.*, 39:535–545, 1971.

[22] J. Valdes, R.E. Tarjan, and E.L. Lawler. The recognition of series parallel graphs. *SIAM J. Comput.*, 11:298–313, 1982.

[23] S.G. Williamson. Depth-first search and Kuratowski subgraphs. *J. ACM*, 31:681–693, 1984.

4.8 Section 2.1 Connectivity.

[1] M. Becker, W. Degenhardt, J. Doenhart, S. Hertel, G. Kaninke, W. Keber, K. Mehlhorn, S. Näher, H. Rohnert, and T. Winter. A probabilistic algorithm for vertex connectivity of graphs. *Inf. Proc. Lett.*, 15:135–136, 1982.

[2] S. Even. An algorithm for determining whether the connectivity of a graph is at least k. *SIAM J. Comput.*, 4:393–396, 1975.

[3] S. Even and R.E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[4] Z. Galil. Finding the vertex connectivity of a graph. *SIAM J. Comput.*, 9:197–199, 1980.

[5] J.E. Hopcroft and R.E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2:135–158, 1973.

[6] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.

4.9 Section 2.2 Minimum Spanning Tree.

[1] O. Borůvka. O jistém problému minimálním. *Práca Moravske Přírodovědecké Společnosti*, 3:37–58, 1926.

[2] D. Cheriton and R.E. Tarjan. Finding minimum spanning trees. *SIAM J. Comput.*, 5:724–742, 1976.

[3] F. Chin and D. Houck. Algorithms for updating minimal spanning trees. *J. Comput. Syst. Sci*, 16:333–344, 1978.

[4] E.W. Dijkstra. A note on two problems in connexion with graphs. *Num. Math*, 1:269–271, 1959.

[5] G.N. Frederickson. Data structures for on-line updating of minimum spanning trees. In *Proc. 15th Annual ACM Symp. on Theory of Computing*, pages 252–257, Boston, MA, 1983.

[6] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimalization problems. In *Proc. 25th Annual IEEE Symp. Found. of Computer Science*, pages 338–346, Singer Island, 1984.

[7] C.P. Gabor, W.L. Hsu, and K.J. Supowit. Recognizing circle graphs in polynomial time. In *Proc. 26th Annual IEEE Symp. Found. of Computer Science*, pages 106–116, Portland, 1985.

[8] H.N. Gabow, Z. Galil, and T.H. Spencer. Efficient implementation of graph algorithms using contraction. In *Proc. 25th Annual IEEE Symp. Found. of Computer Science*, pages 347–357, Singer Island, 1984.

[9] V. Jarnik. O jistém problému minimálním. *Práca Moravske Přírodovědecké Společnosti*, 6:57–63, 1930.

[10] D.B. Johnson. Priority queues with update and finding minimum spanning trees. *Inf. Proc. Lett.*, 4:53–57, 1975.

[11] R.C. Prim. Shortest connection networks and some generalizations. *Bell Syst. Tech. J.*, 36:1389–1401, 1957.

[12] P.M. Spira and A. Pan. On finding and updating spanning trees and shortest paths. *SIAM J. Comput.*, 4:375–380, 1975.

[13] A. Yao. An $O(|E| \log\log |V|)$ algorithm for finding minimum spanning trees. *Inf. Proc. Lett.*, 4:21–23, 1975.

4.10 Section 2.3 Shortest Paths.

4.10.1 Section 2.3.1 Single Source Shortest Paths.

[1] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numer Math.*, 1:269–271, 1977.

[2] L.R. Ford and D.R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NY, 1962.

[3] G. Frederickson. Shortest path problems in planar graphs. In *Proc. 24th Annual IEEE Symp. Foundations of Computer Science*, pages 242–247, Tucson, 1983.

[4] M.L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, 5:83–89, 1976.

[5] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimalization problems. In *Proc. 25th Annual IEEE Symp. Found. of Computer Science*, pages 338–346, Singer Island, 1984.

[6] D.B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24:1–13, 1977.

[7] K. Mehlhorn and B.H. Schmidt. A single source shortest path algoritm for graphs with separators. In *Proceedings FCT 83, Lecture Notes in Computer Science158*, pages 302–309, Springer Verlag, Berlin, 1983.

[8] C.K. Yap. A hybrid algorithm for the shortest path between two nodes in the presence of few negative arcs. *Inf. Proc. Lett.*, 16:181–182, 1983.

4.10.2 Section 2.3.2 All Pairs Shortest Paths.

[1] P. Bloniarz. A shortest-path algorithm with expected time $O(n^2 \log n \log^* n)$. *SIAM J. Comput.*, 12:588–600, 1983.

[2] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19:248–264, 1972.

[3] A. Moffat and T. Takaoka. An all pairs shortest path algorithm with expected running time $O(n^2 \log n)$. In *Proc 26th Annual IEEE Symp. Foundations of Computer Science*, pages 101–105, Portland, 1985.

[4] F. Romani. Shortest path problem is not harder than matrix multiplication. *Inf. Proc. Lett.*, 11:134–136, 1980.

[5] O. Watanabe. A fast algorithm for finding all shortest paths. *Inf. Proc. Lett.*, 13:1–3, 1981.

4.11 Section 2.4 Paths and Cycles.

4.11.1 Section 2.4.1 Paths of Length k.

[1] U. Derigs. An efficient Dijkstra-like labeling method for computing shortest odd/even paths. *Inf. Proc. Lett.*, 21:253–258, 1985.

[2] B. Monien. The complexity of determining paths of length k. In M. Nagl and J. Perl, editors, *Proceedings Int. Workshop on Graph Theor. Concepts in Comput. Sci.*, pages 241–251, Trauner Verlag, Linz, 1983.

4.11.2 Section 2.4.2 Disjoint Paths.

[1] A. Cypher. An approach to the k paths problem. In *Proc. 12th Annual ACM Symp. Theory of Computing*, pages 211–217, Los Angeles, 1980.

[2] L.R. Ford and D.R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NY, 1962.

[3] I.T. Frisch. An algorithm for vextex-pair connectivity. *Intern. J. Control*, 6:579–593, 1967.

[4] Y. Perl and Y. Shiloach. Finding two disjoint paths between two pairs of vertices in a graph. *J. ACM*, 25:1–9, 1978.

[5] N. Robertson and P.D. Seymour. Disjoint path — A survey. *SIAM J. Algebr. and Discr. Meth.*, 6:300–305, 1985.

[6] P.D. Seymour. Disjoint paths in graphs. *Discr. Math.*, 29:293–309, 1980.

[7] Y. Shiloach. A polynomial solution to the undirected two paths problem. *J. ACM*, 27:445–456, 1980.

[8] K. Steiglitz and J. Bruno. A new derivation of Frisch's algorithm for calculating vertex-pair connectivity. *BIT*, 11:94–106, 1971.

[9] J.W. Suurballe. Disjoint paths in a network. *Networks*, 4:125–145, 1974.

4.11.3 Section 2.4.3 Cycles.

[1] E.T. Dixon and S.E. Goodman. An algorithm for the longest cycle problem. *Networks*, 6:139–149, 1976.

[2] M.R. Garey and R.E. Tarjan. A linear time algorithm for finding all feedback vertices. *Inf. Proc. Lett.*, 7:274–276, 1978.

[3] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. *SIAM J. Comput*, 7:413–423, 1978.

[4] A.D. Jovanovich. Note on a modification of the fundamental cycles finding algorithm. *Inf. Proc. Lett.*, 3:33, 1974.

[5] A.S. LaPaugh and R.L. Rivest. The subgraph homeomorphism problem. In *Proc. 10th Annual ACM Symp. Theory of Computing*, pages 40–50, San Diego, 1978.

[6] B. Monien. The complexity of determining a shortest cycle of even length. *Computing*, 31:355–369, 1983.

[7] K. Paton. An algorithm for finding a fundamental set of cycles of a graph. *C. ACM*, 12:514–518, 1969.

[8] D. Richards. Finding short cycles in planar graphs using separators. *J. Algor.*, 7:382–394, 1986.

[9] C. Thomassen. Even cycles in directed graphs. *Europ. J. Combin.*, 6:85–89, 1985.

4.11.4 Section 2.4.4 Weighted Cycles.

[1] W. Domschke. Zwei Verfahren zur Suche negativer Zyklen in bewerteten Digraphen. *Computing*, 11:125–136, 1973.

[2] M. Florian and P. Robert. A direct search method to locate negative cycles in a graph. *Manag. Sci*, 17:307–310, 1971.

[3] D. Maier. A space efficient method for the lowest common ancestor problem and an application to finding negative cycles. In *Proc. 18th Annual IEEE Symp. Found. of Computer Science*, pages 132–141, Providence, 1977.

[4] R.L. Tobin. Minimal complete matchings and negative cycles. *Networks*, 5:371–387, 1975.

[5] A.K. Tsakalidis. *Finding a Negative Cycle in a Directed Graph.* Technical Report A85/05, Angew. Mathematik u. Informatik, Fb-10, Universität des Saarlandes, Saarbrücken, 1985.

[6] J.Y. Yen. On the efficiency of a direct search method to locate negative cycles in a network. *Manag. Sci.*, 19:335–336, 1972. (Rejoinder by M. Florian and P. Robert, ibid., 335-336).

4.11.5 Section 2.4.5 Eulerian (and other) Cycles.

[1] N. Alon and M. Tarsi. Covering multigraphs by simple circuits. *SIAM J. Algebr. and Discr. Meth.*, 6:345–350, 1985.

[2] J. Edmonds and E.L. Johnson. Matching, Euler tours and the Chinese postman. *Math. Progr.*, 5:88–124, 1973.

[3] H. Fleischer. Eulerian graphs. In L.W. Beineke and R.J. Wilson, editors, *Selected Topics in Graph Theory 2*, pages 17–53, Acad. Press, London, 1983.

[4] S. Goodman and S. Hedetniemi. Eulerian walks in graph. *SIAM J. Comput.*, 2:16–27, 1973.

[5] Meigu Guan. The maximum weighted cycle-packing problem and its relation to the Chinese postman problem. In J.A. Bondy and U.S.R. Murty, editors, *Progress in Graph Theory*, pages 323–326, Acad. Press, Toronto, 1984.

[6] C. Hierholzer. Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Math. Ann.*, 6:30–32, 1873.

[7] A. Itai, R.J. Lipton, C.H. Papadimitriou, and M. Rodeh. Covering graphs by simple circuits. *SIAM J. Comput.*, 10:746–750, 1981.

[8] Kwan Mei-Ko. Graphic programming using odd or even points. *Acta Math. Sinica*, 10:263–266, 1960. (Chin. Math. 1 (1962) 273-277).

[9] O. Ore. A problem regarding the tracing of graphs. *Elem. Math.*, 6:49–53, 1951.

[10] C.H. Papadimitriou. On the complexity of edge traversing. *J. ACM*, 23:544–554, 1976.

4.11.6 Section 2.4.6 Hamiltonian (and other) Cycles.

[1] D. Angluin and L.G. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *J. Comput. Syst. Sci.*, 18:155–193, 1979.

[2] B. Bollobás. *Random Graphs.* Acad. Press, London, 1985.

[3] G.H. Danielson. On finding simple paths and circuits in a graph. *IEEE Trans. Circuit Theor.*, 15:294–295, 1968.

[4] M.R. Garey, D.S. Johnson, and R.E. Tarjan. The planar Hamiltonian circuit problem is NP-complete. *SIAM J. Comput.*, 5:704–714, 1976.

[5] S.E. Goodman and S.T. Hedetniemi. On Hamiltonian walks in graphs. *SIAM J. Comput.*, 3:214–221, 1974.

[6] S.E. Goodman, S.T. Hedetniemi, and P.J. Slater. Advances on the Hamiltonian completion problem. *J. ACM*, 22:352–360, 1975.

[7] D. Gouyou-Beauchamps. The Hamiltonian circuit problem is polynomial for 4-connected planar graphs. *SIAM J. Comput.*, 11:529–539, 1982.

[8] S. Kundu. A linear algorithm for the Hamiltonian completion number of a tree. *Inf. Proc. Lett.*, 5:55–57, 1976.

[9] J. Plesńik. The NP-completeness of the Hamiltonian cycle problem in planar digraphs with degree bound two. *Inf. Proc. Lett.*, 8:199–201, 1979.

[10] F. Rubin. A search procedure for Hamilton paths and circuits. *J. ACM*, 21:576–580, 1974.

[11] P.J. Slater, S.E. Goodman, and S.T. Hedetniemi. On the optional Hamiltonian completion problem. *Networks*, 6:35–51, 1976.

[12] K. Takamizawa, T. Nishizeki, and N. Saito. An algorithm for finding a short closed spanning walk of a graph. *Networks*, 10:249–263, 1980.

4.12 Section 2.5 Decomposition of Graphs.

[1] J. Akiyama and M. Kano. Factors and factorizations of graphs — A survey. *J. Graph Th.*, 9:1–42, 1985.

[2] E. Arjomandi. On finding all unilaterally connected components of a digraph. *Inf. Proc. Lett.*, 5:8–10, 1976.

[3] F.T. Boesch and J.F. Gimpel. Covering the points of a digraph with point-disjoint paths and its application to code optimalization. *J. ACM*, 24:192–198, 1977.

[4] G.A. Cheston. A correction to a unilaterally connected components algorithm. *Inf. Proc. Lett.*, 7:125, 1978.

[5] W.H. Cunningham. Decomposition of directed graphs. *SIAM J. Alg. Discr. Meth.*, 3:214–228, 1982.

[6] H.M. Djidjev. On the problem of partitioning planar graphs. *SIAM J. Discr. and Appl. Meth.*, 3:229–240, 1982.

[7] J. Ebert. A note on odd and even factors of undirected graphs. *Inf. Proc. Lett.*, 11:70–72, 1980.

[8] J.R. Gilbert, J.P. Hutchinson, and R.E. Tarjan. *A Separator Theorem for Graphs of Bounded Genus.* Technical Report 82-506, Dept. of Computer Science, Cornell University, Ithaca, 1982.

[9] J.R. Gilbert, D.J. Rose, and A. Edenbrandt. A separator theorem for chordal graphs. *SIAM J. Discr. and Appl. Meth.*, 5:306–313, 1984.

[10] J.E. Hopcroft and R.E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2:135–158, 1973.

[11] D.G. Kirkpatrick and P. Hell. On the complexity of general graph factor problems. *SIAM J. Comput.*, 12:601–609, 1983.

[12] R.J. Lipton and R.E. Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9:615–627, 1980.

[13] R.J. Lipton and R.E. Tarjan. A separator theorem for planar graphs. *SIAM J. Appl. Math.*, 36:177–189, 1979.

[14] L. Lovász. A homology theory for spanning trees of a graph. *Acta Math. Acad. Sci. Hungary*, 30:241–251, 1977.

[15] S. MacLane. A structural characterization of planar combinatorial graphs. *Duke Math. J.*, 3:460–472, 1937.

[16] S.B. Maurer. Vertex colorings without isolates. *J. Combin. Th.*, B-27:294–319, 1979.

[17] G.L. Miller. Finding small simple cycle separators for 2-connected planar graphs. In *Proc. 16th Annual ACM Symp. Theory of Computing*, pages 376–382, Washington DC, 1984. Revised version in: *J. Comp. Syst. Sci.*, 32:265–279, 1986.

[18] J. Misra and R.E. Tarjan. Optimal chain partitions of trees. *Inf. Proc. Lett.*, 4:24–26, 1975.

[19] S. Noorvash. Covering the vertices of a graph by vertex-disjoint paths. *Pacif. J. Math.*, 58:159–168, 1975.

[20] J.F. Pacault. Computing the weak components of a directed graph. *SIAM J. Comput.*, 3:56–61, 1974.

[21] R. Philipp and E.J. Prauss. Ueber Separatoren in Planaren Graphen. *Acta Inform.*, 14:87–106, 1980.

[22] R.E. Tarjan. Decomposition by clique separators. *Discr. Math.*, 55:221–232, 1985.

[23] R.E. Tarjan. An improved algorithm for hierarchical clustering using strong components. *Inf. Proc. Lett.*, 17:37–41, 1983.

[24] R.E. Tarjan. A new algorithm for finding weak components. *Inf. Proc. Lett.*, 3:13–15, 1974.

[25] W.T. Tutte. On the problem of decomposing a graph into n connected factors. *J. London Math. Soc.*, 36:221–230, 1961.

[26] P. Ungar. A theorem on planar graphs. *J. London Math. Soc.*, 26:256–262, 1951.

4.13 Section 2.6 Isomorphism Testing.

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publ. Comp., Reading, Mass., 1974.

[2] L. Babai. Moderately exponential bound for graph isomorphism. In F. Gécseg, editor, *Fundamentals of Computation Theory, Proceedings 1981, Lecture Notes in Computer Science 117*, Springer Verlag, Berlin, 1981.

[3] L. Babai, P. Erdös, and S.M. Selkow. Random graph isomorphism. *SIAM J. Comput.*, 9:628–635, 1980.

[4] L. Babai, D.Yu. Grigoryev, and D.M. Mount. Isomorphism of graphs with bounded eigenvalue multiplicity. In *Proc. 14th Annual ACM Symp. Theory of Computing*, pages 310–324, San Francisco, 1982.

[5] L. Babai and L. Kučera. Canonical labelling of graphs in linear average time. In *Proc. 20th Annual IEEE Symp. Foundations of Computer Science*, pages 39–46, San Juan (Puerto Rico), 1979.

[6] L. Babai and E.M. Luks. Canonical labeling of graphs. In *Proc. 15th Annual ACM Symp. Theory of Computing*, pages 171–183, Boston, 1983.

[7] A.T. Bertziss. A backtrack procedure for isomorphism of directed graphs. *J. ACM*, 20:365–377, 1973.

[8] K.S. Booth. Isomorphism testing for graph, semigroups and finite automata are polynomially equivalent problems. *SIAM J. Comput.*, 7:273–279, 1978.

[9] C.J. Colbourn and K.S. Booth. Linear time automorphism algorithms for trees, interval graphs, and planar graphs. *SIAM J. Comput.*, 10:203–225, 1981.

[10] M.J. Colbourn and C.J. Colbourn. Graph isomorphism and self-complementary graphs. *SIGACT NEWS*, 10(1):25–29, 1978.

[11] D.G. Corneil and C.C. Gotlieb. An efficient algorithm for graph isomorphism. *J. ACM*, 17:51–64, 1970.

[12] D.G. Corneil and D.G. Kirkpatrick. Theoretical analysis of various heuristics for the graph isomorphism problem. *SIAM J. Comput.*, 9:281–297, 1980.

[13] N. Deo, J.M. Davis, and R.E. Lord. A new algorithm for digraph isomorphism. *BIT*, 17:16–30, 1977.

[14] I.S. Filotti and J.N. Mayer. A polynomial-time algorithm for determining the isomorphism of graphs of fixed genus. In *Proc. 12th Annual ACM Symp. Theory of Computing*, pages 236–243, Los Angeles, 1980.

[15] M. Fontet. Linear algorithms for testing isomorphism of planar graphs. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming — Proceedings 3rd Colloquim*, pages 411–423, Edinburgh, Edinburgh University Press, 1976.

[16] M. Fürer, W. Schnyder, and E. Specker. Normal forms for trivalent graphs and graphs of bounded valence. In *Proc. 15th Annual ACM Symp. Theory of Computing*, pages 161–170, Boston, 1983.

[17] Z. Galil, C.M. Hoffmann, E.M. Luks, C.P. Schnorr, and A. Weber. An $O(n^3 \log n)$ deterministic and an $O(n^3)$ probabilistic isomorphism test for trivalent graphs. In *Proc. 23rd Annual IEEE Symp. Foundations of Computer Science*, pages 118–125, Chicago, 1982.

[18] B.R. Heap. The production of graphs by computer. In R.C. Read, editor, *Graph Theory and Computing*, pages 47–62, Acad. Press, New York, NY, 1972.

[19] C.M. Hoffman. *Group-Theoretic Algorithms and Graph Isomorphism.* Volume 136 of *Lecture Notes In Computer Science*, Springer Verlag, Berlin, 1982.

[20] J.E. Hopcroft and C.K. Wong. Linear time algorithms for isomorphism of planar graphs. In *Proc. 6th Annual ACM Symp. Theory of Computing*, pages 172–184, Seattle, 1974.

[21] D.S. Johnson. The NP-completeness column: An ongoing guide. *J. Algor*, 4:393–405, 1981.

[22] R.M. Karp. Probabilistic analysis of a canonical numbering algorithm for graphs. In D.K. Ray-Chaudhuri, editor, *Relations Between Combinatorics and Other Parts of Mathematics*, pages 365–378, Proc. Symp. Pure Math. vol. XXXIV, Amer. Math. Soc., Providence (RI), 1979.

[23] A. Lingas. Subgraph isomorphism for biconnected outerplanar graphs in cubic time. In B. Monien and G. Vidal-Naquet, editors, *STACS 86 — 3rd Annual Symp. Theor. Aspects of Computer Science, Proceedings, Lecture Notes in Computer Science 210*, pages 98–103, Springer Verlag, Berlin, 1986.

[24] A. Lingas. Subgraph isomorphism for easily separable graphs of bounded valence. In H. Noltemeier, editor, *Proceedings WG'85 (Intern. Workshop on Graphtheoretic Concepts in Computer Science)*, pages 217–229, Trauner Verlag, Linz, 1985.

[25] A. Lubiw. Some NP-complete problems similar to graph isomorphism. *SIAM J. Comput.*, 10:11–21, 1981.

[26] G.S. Lueker and K.S. Booth. A linear time algorithm for deciding interval graph isomorphism. *J. ACM*, 26:183–195, 1979.

[27] E.M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. In *Proc. 21st Ann. IEEE Symp. Foundations of Computer Science*, pages 42–49, Syracuse, 1980. Extended Version in: *J. Comput. Syst. Sci.*, 25:42–65,1982.

[28] B.D. MacKay. Practical graph isomorphism. In *Proc. 10th Manitoba Conf. Numer. Math. and Computing*, 1980. Vol. 1 (1980), Congr. Numer. 30 (1981) 45-87.

[29] R. Mathon. A note on the graph isomorphism counting problem. *Inf. Proc. Lett.*, 8:131–132, 1979.

[30] D.W. Matula. Subtree isomorphism in $O(n^{5/2})$. *Ann. Discr. Math.*, 2:91–106, 1978.

[31] G. Miller. Isomorphism testing for graphs of bounded genus. In *Proc. 12th Annual ACM Symp. Theory of Computing*, pages 225–235, Los Angeles, 1980.

[32] G.L. Miller. Graph isomorphism, general remarks. *J. Comput. Syst. Sci.*, 18:128–142, 1979.

[33] A. Proskurowski. Search for a unique incidence matrix of a graph. *BIT*, 14:209–226, 1974.

[34] R.C. Read and D.G. Corneil. The graph isomorphism disease. *J. Graph Th.*, 1:239–363, 1977.

[35] S.W. Reyner. An analysis of a good algorithm for the subtree problem. *SIAM J. Comput.*, 6:730–732, 1977.

[36] D.C. Schmidt and L.E. Druffel. A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *J. ACM*, 23:433–445, 1976.

[37] U. Schöning. *Graph Isomorphism is in the Low Hierarchy.* Preprint, Fachbereich Informatik, EWH Koblenz, Koblenz, 1986.

[38] J.R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23:31–42, 1976.

[39] L. Weinberg. A simple and efficient algorithm for determing isomorphism of planar triply connected graphs. In *IEEE Trans. on Circuit Theory CT-13*, pages 142–148, 1966.

4.14 General.

[1] E. Lawler. *Combinatorial Optimisation: Network and Matroids.* Holt, Rinehart and Winston, New York, NY, 1976.

[2] L. Lovász and M.D. Plummer. *Matching Theory.* Vol. 29 of *Annals of Discrete Math., North Holland Math. Studies, 121*, North Holland Publishing Comp., Amsterdam, 1986.

[3] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1982.

4.15 Section 3.1 Maximum Matching.

[1] C. Berge. Two theorems in graph theory. In *Proc. Natl. Acad. Sci. 43*, pages 842–844, 1957.

[2] J. Edmonds. Paths, trees, and flowers. *Can. J. Math.*, 17:449–467, 1965.

[3] S.E. Goodman, S. Hedetniemi, and R.E. Tarjan. b-Matchings in trees. *SIAM J. Comput.*, 5:104–108, 1976.

[4] R.Z. Norman and M.O. Rabin. An algorithm for a minimum cover of a graph. *Proc. Amer. Math. Soc.*, 10:315–319, 1959.

[5] C.H. Papadimitriou and M. Yannakakis. Worst-case ratios for planar graphs and the method of induction on faces. In *Proc. 22 Annual IEEE Symp. Foundations of Computer Science*, pages 358–363, Nashville, 1981.

[6] C. Savage. Maximum matchings and trees. *Inf. Proc. Lett.*, 10:202–205, 1980.

[7] Y. Shiloach. Another look at the degree constrained subgraph problem. *Inf. Proc. Lett.*, 12:89–92, 1981.

[8] R.J. Urquhart. *Degree Constrained Subgraphs of Linear Graphs.* Ph.D thesis, University of Michigan, Ann Arbor, 1967.

[9] L.J. White. *A Parametric Study of Matchings and Coverings in Weighted Graphs.* Ph.D thesis, Dept. of Electr. Engin., University of Michigan, Ann Arbor, 1967.

4.16 Section 3.2 Computing Maximum Matchings.

[1] D. Angluin and L.G. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *J. Comput. Syst. Sci.*, 18:155–193, 1979.

[2] M.L. Balinski. Labelling to obtain a maximum matching. In *Proc. Combin. Math. and its Applic.*, pages 585–602, North Carolina Press, Chapel Hill, NC, 1967.

[3] J. Edmonds. Maximum matching and a polyhedron with 0,1 vertices. *J. Res. NBS*, 69B:125–130, 1965.

[4] J. Edmonds. Paths, trees, and flowers. *Can. J. Math.*, 17:449–467, 1965.

[5] P. Erdös and A. Renyi. On the existence of a factor of degree one of connected random graphs. *Acta Math. Acad. Sci. Hungar.*, 17:359–, 1966.

[6] S. Even and O. Kariv. An $O(n^{2.5})$ algorithm for maximum matching in general graphs. In *Proc. 16th Annual IEEE Symp. Found. of Computer Science*, pages 100–112, Berkeley, 1975.

[7] S. Even and R.E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[8] H.N. Gabow. An efficient implementation of Edmonds' maximum matching algorithm. *J. ACM*, 23:221–234, 1976.

[9] H.N. Gabow and R.E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proc. 15th Annual ACM Symp. Theory of Computing*, pages 246–251, Boston, 1983.

[10] Z. Galil. Efficient algorithms for finding maximum matchings in graphs. *Comp. Surv.*, 18:23–38, 1986.

[11] Z. Galil, S. Micali, and H.N. Gabow. An $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs. *SIAM J. Comput.*, 15:120–130, 1986.

[12] J.E. Hopcroft and R.M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 4:225–231, 1973.

[13] T. Kameda and I. Munro. An $O(|V|.|E|)$ algorithm for maximum matching of graphs. *Computing*, 12:91–98, 1974.

[14] R.M. Karp and M. Sipser. Maximum matchings in sparse random graphs. In *Proc. 22nd Annual IEEE Symp. Found. of Computer Science*, pages 364–375, Nashville, 1981.

[15] E. Lawler. *Combinatorial Optimisation: Network and Matroids.* Holt, Rinehart and Winston, New York, NY, 1976.

[16] R.J. Lipton and R.E. Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9:615–627, 1980.

[17] S. Micali and V.V. Vazirani. An $O(\sqrt{V}.E)$ algorithm for finding maximum matching in general graphs. In *Proc. 21st. Annual IEEE Symp. Found. of Computer Science*, pages 17–27, Syracuse, 1980.

[18] C. Savage. Maximum matchings and trees. *Inf. Proc. Lett.*, 10:202–205, 1980.

4.17 Section 3.3 Maximum Flow.

[1] R.G. Busacker and P.J. Gowen. *A Procedure for Determining a Family of Minimal-Cost Network Flow Patterns.* O.R.O. Technical paper 15, 1961.

[2] L.R. Ford and D.R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NY, 1962.

[3] L.M. Goldschlager, R.A. Shaw, and J. Staples. The maximum flow problem is log-space complete for P. *Theor. Comp. Sci.*, 21:105–111, 1982.

[4] A.J. Hoffman. Some recent applications of the theory of linear inequalities to extremal combinatorial analysis. In *Proc. Symp. Appl. Math., volume X: Combinatorial Analysis*, pages 113–127, Amer. Math. Soc., Providence, RI, 1960.

[5] W.S. Jewell. *Optimal Flow Through Networks.* Interim Techn. Rep. 8, MIT, 1958.

[6] E. Lawler. *Combinatorial Optimisation: Network and Matroids.* Holt, Rinehart and Winston, New York, NY, 1976.

4.18 Section 3.4 Computing Maximum Flows.

4.18.1 Section 3.4.1 Augmenting Path Methods.

[1] A.E. Baratz. *The Complexity of the Maximum Network Flow Problem.* Technical Report MIT/LCS/TR-230, Lab. for Computer Science, MIT, Cambridge (Mass.), 1980.

[2] C. Berge and A. Ghouila-Houri. *Programmes, Jeux et Réseaux de Transport.* Dunod, Paris, 1962.

[3] R.V. Cherkasky. Algorithm of construction of maximal flow in networks with complexity of $O(V^2\sqrt{E})$ operations. *Math. Methods of Solution of Econ. Problems*, 7:112–125, 1977. (In Russian).

[4] E.A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[5] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19:248–264, 1972.

[6] S. Even and R.E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4:507–518, 1975.

[7] L.R. Ford Jr. and D.R. Fulkerson. Maximal flow through a network. *Canad. J. Math.*, 8:399–404, 1956.

[8] G. Frederickson. Shortest path problems in planar graphs. In *Proc. 24th Annual IEEE Symp. Foundations of Computer Science*, pages 242–247, Tucson, 1983.

[9] H.N. Gabow. Scaling algorithms for network problems. In *Proc. 24th Annual IEEE Symp. Foundations of Computer Science*, pages 248–258, Tucson, 1983.

[10] Z. Galil. On the theoretical efficiency of various network flow algorithms. *Theor. Comp. Sci.*, 14:103–111, 1981.

[11] Z. Galil. An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem. *Acta Informatica*, 14:221–242, 1980.

[12] Z. Galil and A. Naamad. An $O(EV\log^2 V)$ algorithm for the maximal flow problem. *J. Comput. Syst. Sci.*, 21:203–217, 1980.

[13] R. Hassin. Maximum flow in (s,t) planar networks. *Inf. Proc. Lett.*, 13:107, 1981.

[14] A. Itai and Y. Shiloach. Maximum flow in planar networks. *SIAM J. Comput.*, 8:135–150, 1979.

[15] D.B. Johnson and S.M. Venkatesan. Partition of planar flow networks. In *Proc. 24th Annual IEEE Symp. Foundations of Computer Science*, pages 259–263, Tucson, 1983.

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

[17] V.M. Malhotra, M. Pramodh Kumar, and S.N. Maheswari. An $O(|V|^3)$ algorithm for finding maximum flows in networks. *Inf. Proc. Lett.*, 7:277–278, 1978.

[18] M. Queyranne. Theoretical efficiency of the algorithm 'capacity' for the maximum flow problem. *Math. Oper. Res.*, 5:258–266, 1980.

[19] J.H. Reif. Minimum s-t cut of a planar undirected network in $O(n \log^2 n)$ time. *SIAM J. Comput.*, 12:71–81, 1983.

[20] D.D. Sleator. *An $O(nm \log n)$ Algorithm for Maximum Network Flow.* Technical Report STAN-CS-80-831, Dept. of Computer Science, Stanford University, Stanford, 1980.

[21] D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 24:362–391, 1983.

[22] R.E. Tarjan. A simple version of Karzanov's blocking flow algorithm. *Oper. Res. Lett.*, 2:265–268, 1984.

4.18.2 Section 3.4.2 Maximum Flow Algorithms Cont'd.

[1] A.V. Goldberg. *A New Max-flow Algorithm.* Technical Memorandum MIT/LCS/TM-291, Lab. for Computer Science, MIT, Cambridge (Mass.), 1985.

[2] A.V. Goldberg and R.E. Tarjan. A new approach to the maximum flow problem. In *Proc. 18th Annual ACM Symp. Theory of Computing*, pages 136–146, Berkeley, 1986.

[3] D.D. Sleator and R.E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32:652–686, 1985.

4.19 Section 3.5 Related Flow Problems.

4.19.1 Section 3.5.1 Networks with Lowerbounds.

[1] E.M. Arkin and C.H. Papadimitriou. On the complexity of circulations. *J. Algor.*, 7:134–145, 1986.

[2] A. Itai. Two-commodity flow. *J. ACM*, 25:596–611, 1978.

[3] E. Lawler. *Combinatorial Optimisation: Network and Matroids.* Holt, Rinehart and Winston, New York, NY, 1976.

4.19.2 Section 3.5.2 Minimum Cost Flow.

[1] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19:248–264, 1972.

[2] L.R. Ford and D.R. Fulkerson. *Flows in Networks.* Princeton Univ. Press, Princeton, NY, 1962.

[3] S. Fujishige. A capacity-rounding algorithm for the minimum-cost circulation problem: A dual framework of the Tardos algorithm. *Math. Progr.*, 35:298–308, 1986.

[4] D.R. Fulkerson. An out-of-kilter method for minimal-cost flow problems. *J. SIAM*, 9:18–27, 1961.

[5] Z. Galil and E. Tardos. *An $O(n^2(m+n\log n)\log n)$ Min-cost Flow Algorithm.* Preprint, 1986.

[6] E. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5:247–255, 1985.

[7] H.M. Wagner. On a class of capacitated transportation problems. *Manag. Sci.*, 5:304–318, 1959.

[8] N. Zadeh. A bad network for the simplex method and other minimum cost flow algorithms. *Math. Progr.*, 5:255–266, 1973.

[9] N. Zadeh. More pathological examples for network flow problems. *Math. Progr.*, 5:217–224, 1973.

4.19.3 Section 3.5.3 Multiterminal Flow.

[1] R.E. Gomory and T.C. Hu. An application of generalized linear programming to network flows. *J. SIAM*, 10:260–283, 1962.

[2] R.E. Gomory and T.C. Hu. Multi-terminal network flows. *J. SIAM*, 9:551–570, 1961.

[3] R.P. Gupta. On flows in pseudosymmetric networks. *J. SIAM*, 14:215–225, 1966.

[4] D. Gusfield. Simple constructions for multi-terminal network flow synthesis. *SIAM J. Comput.*, 12:157–165, 1983.

[5] T.C. Hu. *Integer Programming and Network Flows.* Addison-Wesley Publ. Comp., Reading, Mass., 1969.

[6] T.C. Hu and M.T. Shing. *A Decomposition Algorithm for Multi-Terminal Network Flows.* Technical Report TRCS 84-08, Dept. of Computer Science, University of California, Santa Barbara, Ca., 1984.

[7] T.C. Hu and M.T. Shing. Multi-terminal flow in outerplanar networks. *J. Algor.*, 4:241–261, 1983.

[8] Y. Shiloach. Multi-terminal 0-1 flow. *SIAM J. Comput.*, 8:422–430, 1979.

[9] M.T. Shing and P.K. Agarwal. *Multi-terminal Flows in Planar Networks.* Technical Report TRCS 86-07, Dept. of Computer Science, University of California, Santa Barbara, Ca., 1986.