II Homework Exercises

Homework 1

1. (a) Let $(S, \mathcal{I})$ be a matroid and let $A \subseteq S$. Prove that all maximal independent subsets of A have the same cardinality. (This number is called the *rank* of A. The rank of the matroid is the rank of the set S.)

   (b) Let I be a maximal independent set and let $x \notin I$. Prove that there is a unique cycle contained in $I \cup \{x\}$. (This is called the *fundamental cycle* of x and I.)

   (c) Let $G = (V, E)$ be an undirected graph, not necessarily connected. Consider the system $(E, \mathcal{I})$, where $\mathcal{I}$ consists of all subsets $E' \subseteq E$ such that the subgraph (V, E') has no cycles. Show that $(E, \mathcal{I})$ is a matroid. What is its rank? What are its maximal independent sets?

2. Let $G = (V, E)$ be a directed graph. A *transitive reduction* or *Hasse diagram* of G is a subgraph $G_H = (V, E_H)$ with minimum number of edges such that E and E_H have the same transitive closure.

   (a) Prove that if G is acyclic, then the transitive reduction of G is unique.

   (b) Give an efficient algorithm to find the transitive reduction of G in case G is acyclic. Your algorithm should have roughly the same complexity as transitive closure. (We will see later that the problem is *NP*-complete when G is cyclic.)

3. An Euler circuit in a connected undirected graph $G = (V, E)$ is a circuit that traverses all edges exactly once.

   (a) Prove that G has an Euler circuit iff G is connected and the degree of every vertex is even.

   (b) Give an $O(|E|)$ algorithm to find an Euler circuit if one exists. Give a proof of correctness and detailed complexity analysis.

Homework 2

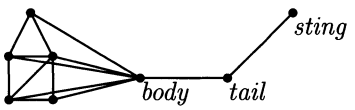
1. Give a linear-time algorithm for topological sort based on depth-first search.

2. Let $G = (V, E)$ be an undirected graph, and let θ be a circular ordering of the edges adjacent to each vertex. The ordering θ is said to be *consistent* with an embedding of G in the plane if for each vertex v, the ordering of the edges adjacent to v given by θ agrees with their counterclockwise ordering around v in the embedding.

   (a) Give a linear-time algorithm that determines whether a given θ is consistent with some plane embedding.

   (b) Consider two embeddings to be the same if one can be transformed into the other by a smooth motion in the plane without tearing or cutting. Assume that G is connected. How many distinct plane embeddings are consistent with a given θ? (*Extra credit.* Remove the assumption of connectedness.)

3. A *scorpion* is an undirected graph G of the following form: there are three special vertices, called the *sting*, the *tail*, and the *body*, of degree 1, 2, and $n - 2$, respectively. The sting is connected only to the tail; the tail is connected only to the sting and the body; and the body is connected to all vertices except the sting. The other vertices of G may be connected to each other arbitrarily.



   Give an algorithm that makes only $O(n)$ probes of the adjacency matrix of G and determines whether G is a scorpion. (This is a counterexample to an earlier version of the famous *Anderaa-Rosenberg conjecture*, which stated that any nontrivial graph property that is invariant under graph isomorphism requires $\Omega(n^2)$ probes of the adjacency matrix to test. Anderaa disproved this version in 1975 using a different counterexample (see [91]), but conjectured that it held for *monotone* properties (those that cannot change from false to true when edges are deleted). This was later verified by Rivest and Vuillemin [91].)

Homework 3

1. Verify that the family $\mathbf{Reg}_\Sigma$ of regular expressions over an alphabet Σ with the operations defined as in Lecture 6, Example 6.2 is a Kleene algebra.

2. Let R be the standard interpretation of regular expressions over $\mathbf{Reg}_\Sigma$. Prove that for regular expressions α, β, γ and interpretation I over a Kleene algebra $\mathcal{K}$,

$$I(\alpha\beta\gamma) \;=\; \sup_{x \in R(\beta)} I(\alpha x \gamma) \;.$$

   In other words, the supremum of the set

$$\{I(\alpha x \gamma) \mid x \in R(\beta)\}$$

   exists and is equal to $I(\alpha\beta\gamma)$. (*Hint.* Try induction on β, using the axioms of Kleene algebra.) Note that Lemma 7.1 of Lecture 7 is a special case with $\alpha = \gamma = 1$.

3. Modify Dijkstra's algorithm to produce the minimum-weight paths themselves, not just their weights.

Homework 4

1. The following algorithm, known as *Prim's algorithm*, produces a minimum spanning tree T in a connected undirected graph with edge weights. Initially, we choose an arbitrary vertex and let T be the tree consisting of that vertex and no edges. We then repeat the following step $n - 1$ times: find an edge of minimum weight with exactly one endpoint in T and include that edge in T.

   (a) Argue that Prim's algorithm is correct.

   (b) Give an implementation that runs in time $O(m + n \log n)$.

2. The Planar Separator Theorem gives $\frac{1}{3}$-$\frac{2}{3}$ separators of size $O(\sqrt{n})$ for arbitrary planar graphs. Show that this is the best you can do in general; *i.e.*, give a family of planar graphs whose smallest $\frac{1}{3}$-$\frac{2}{3}$ separators are of size $\Omega(\sqrt{n})$.

Homework 5

1. (a) Given a flow f on a directed graph G with positive edge capacities, show how to construct the residual graph G_f in $O(m)$ time.

   (b) Using (a), show how to calculate efficiently an augmenting path of maximum bottleneck capacity. (*Hint.* Modify Dijkstra's algorithm.)

2. Give an efficient algorithm for the s,t-*connectivity problem*: given a directed or undirected graph $G = (V, E)$ and elements $s, t \in V$, $s \neq t$, decide whether there exist k edge-disjoint paths from s to t, and find them if so. (The vertex-disjoint version of this problem is *NP*-complete; see Homework 7 Exercise 3.)

3. Give an efficient algorithm for the *min cut* problem: given an undirected graph $G = (V, E)$, elements $s, t \in V$, $s \neq t$, and edge weights $w : E \to \mathcal{R}^+$, find an s,t-cut of minimum weight; *i.e.*, find a partition A, B of V with $s \in A$, $t \in B$ minimizing

$$\sum_{(u,v) \in E \cap (A \times B)} w(u, v) \ .$$

   (Several minor variants of this problem are *NP*-complete. For example, the *max cut* problem is *NP*-complete.)

Homework 6

1. (The *stable marriage problem.*) In a group of n boys and n girls, each girl ranks the n boys according to her preference, and each boy ranks the n girls according to his preference. A *marriage* is a perfect matching between the boys and the girls. A marriage is *unstable* if there is a pair who are not married to each other but who like each other more than they like their respective spouses, otherwise it is *stable.* Prove that a stable marriage always exists, and give an efficient algorithm to find one.

2. Prove the *König-Egerváry Theorem*: in a bipartite graph, the size of a maximum matching is equal to the size of a minimum vertex cover.

3. Let $G = (U, V, E)$ be a bipartite graph. For $S \subseteq U$, let $N(S)$ be the set of neighbors of S; *i.e.*,

$$N(S) = \{v \in V \mid \exists u \in S \ (u, v) \in E\} .$$

   Prove *Hall's Theorem*: G has a matching in which every vertex of U is matched iff for every subset S of U,

$$|N(S)| \geq |S| .$$

   (*Hint.* Use 2.)

4. An undirected graph is *regular* if all vertices have the same degree. Prove that any nontrivial regular bipartite graph has a perfect matching. (*Hint.* Use 2 or 3.)

Homework 7

1. Consider a restricted version of CNFSat in which formulas may contain at most k occurrences of any variable, either negated or unnegated, where k is fixed.

   (a) Show that the problem is *NP*-complete if $k \geq 3$.

   (b) Show that the problem is solvable in polynomial time if $k \leq 2$.

2. Suppose that TSP $\in P$; that is, suppose there is a polynomial-time algorithm which, given any directed graph G with integral edge weights and positive integer k, determines whether there exists a tour of weight at most k that visits every vertex at least once. Give a polynomial-time algorithm to find such a tour of minimum weight.

3. In Homework 5 Exercise 2 we gave an efficient algorithm for the s,t-connectivity problem. Formulate a version of this problem in which the requirement "edge-disjoint" is replaced by "vertex-disjoint" (this problem is called the *disjoint connecting paths problem* in [39, p. 217]), and show that it is *NP*-complete. (*Hint.* Use 3CNFSat. Let k be the number of clauses plus the number of variables.)

Homework 8

1. Let $\mathcal{Z}_p$ denote the field of integers modulo a fixed prime p. Consider the problem of determining whether a given expression involving $+, -, \cdot, 0, 1$, and variables ranging over $\mathcal{Z}_p$ vanishes for all possible values for the variables. Show that this problem is *coNP*-complete. (*Hint.* You may want to use *Fermat's Theorem*: $a^{p-1} = 1$ for all nonzero $a \in \mathcal{Z}_p$.)

2. Consider a restricted version of the TSP such that distances are symmetric and satisfy the triangle inequality:

$$\begin{aligned} d(u,v) &= d(v,u) \\ d(u,w) &\leq d(u,v) + d(v,w) \ . \end{aligned}$$

   (a) Argue that this restricted version is still *NP*-complete.

   (b) Give a polynomial-time algorithm that finds a tour visiting all cities exactly once whose total distance is at most twice optimal. (*Hint.* Start with a minimum spanning tree.)

3. Recall that a *transitive reduction* or *Hasse diagram* of a directed graph G is a subgraph with as few edges as possible having the same transitive closure as G. Show that the problem of determining whether a given G has a transitive reduction with k or fewer edges is *NP*-complete.

Homework 9

1. Give an *NC* algorithm for obtaining a topological sort of a given directed acyclic graph. (*Hint.* Use Miscellaneous Exercise 27.)

2. Show that the problem of determining whether a given undirected graph is bipartite, *i.e.* does not have an odd cycle, is in *NC*.

3. A linear recurrence is of *order* k if it is of the form

$$\begin{aligned} x_i &= c_i, \quad 0 \leq i \leq k-1 \\ x_n &= a_1 x_{n-1} + a_2 x_{n-2} + \cdots + a_k x_{n-k} + c, \quad n \geq k \end{aligned}$$

where the c_i, a_i, and c are constants. For example, the Fibonacci sequence is of order 2 with $c_0 = c_1 = a_1 = a_2 = 1$ and $c = 0$. Show that the n^{th} term of a linear recurrence of order k can be computed in time

   (a) $O(k^2 \log n)$ with a single processor. (*Hint.* Use Miscellaneous Exercise 22.)

   (b) $O(\log k \log n)$ with $O(k^\alpha)$ processors, where $\alpha = 2.81\ldots$ is the constant in Strassen's matrix multiplication algorithm.

   (c) $O(\log k + \log n)$ with $O(kn)$ processors, assuming that the ring we are working in supports an FFT. (*Hint.* Work with the generating function

$$x(y) = \sum_{i=0}^{\infty} x_i y^i$$

   where y is an indeterminate and the coefficients x_i are the solution to the recurrence.)

Homework 10

1. In Luby's algorithm, we need to show that if we expect to delete at least a fixed fraction of the remaining edges in each stage, then the expected number of stages is logarithmic in the number of edges. We can formalize this as follows.

   Proposition *Let $m \geq 0$ and $0 < \epsilon < 1$. Let $X_1, X_2, \ldots$ and $S_0, S_1, S_2, \ldots$ be nonnegative integer-valued random variables such that*

   $$\begin{aligned} S_n &= X_1 + \cdots + X_n \leq m \\ \mathcal{E}(X_{n+1} \mid S_n) &\geq \epsilon \cdot (m - S_n) \ . \end{aligned}$$

   Then the expected least n such that $S_n = m$ is $O(\log m)$.

   In our application, m is the number of edges in the original graph, X_n is the number of edges deleted in stage n, S_n is the total number of edges deleted so far after stage n, and $\epsilon = \frac{1}{72}$.

   (a) Show that

   $$\mathcal{E}S_n \geq m(1 - (1 - \epsilon)^n) \ .$$

   (*Hint.* Using the fact $\mathcal{E}(\mathcal{E}(X_{n+1} \mid S_n)) = \mathcal{E}X_{n+1}$ shown in class, give a recurrence for $\mathcal{E}S_n$.)

   (b) Using the definition of expectation, show also that

   $$\mathcal{E}S_n \leq m - 1 + \Pr(S_n = m)$$

   and therefore

   $$\Pr(S_n = m) \geq 1 - m(1 - \epsilon)^n \ .$$

   (c) Conclude that the expected least n such that $S_n = m$ is $O(\log m)$. (*Hint.* Define the function

   $$f(x) = \begin{cases} 1 \ , & \text{if } x < m \\ 0 \ , & \text{otherwise} \end{cases}$$

   and compute the expectation of the random variable

   $$R = f(S_0) + f(S_1) + f(S_2) + \cdots$$

   that counts the number of rounds.)

2. A collection $\mathcal{A}$ of events are *d-wise independent* if for any subset $\mathcal{B} \subseteq \mathcal{A}$ of size d or less, the probability that all events in $\mathcal{B}$ occur is the product of their probabilities. Consider the following generalization of Luby's scheme. For each $u \in \mathcal{Z}_p$, let A_u be any subset of $\mathcal{Z}_p$. Randomly select $x_0, \ldots, x_{d-1} \in \mathcal{Z}_p$. Show that the p events

$$x_0 + x_1 u + x_2 u^2 + \cdots + x_{d-1} u^{d-1} \in A_u$$

for $u \in \mathcal{Z}_p$ are d-wise independent. (*Hint.* Consider $d \times d$ Vandermonde matrices over $\mathcal{Z}_p$ with rows

$$(1,\ u,\ u^2,\ \ldots,\ u^{d-1})$$

shown in class to be nonsingular.)

3. Consider the following random NC algorithm for finding a maximal (not maximum) matching in an undirected graph $G = (V, E)$. The algorithm proceeds in stages. At each stage, a matching M is produced, and the matched vertices and all adjacent edges are deleted. Each stage proceeds as follows:

   (a) In parallel, each vertex u chooses a neighbor $t(u)$ at random. Set

   $$H := \{(u, t(u)) \mid u \in V\} .$$

   (b) If there are two or more edges $(u, t(u))$ in H with $t(u) = v$, then v chooses one of them arbitrarily and deletes the rest from H.

   (c) Let U be the set of vertices with at least one incident edge in H. Each vertex in the graph (U, H) has degree 1 or 2. If 2, it randomly selects one of its two incident edges as its favorite. If 1, it selects its one incident edge as its favorite.

   (d) For each edge $e \in H$, e is included in M if it is the favorite of both its endpoints.

   Show that M is a matching, and the expected number of edges deleted is at least a constant fraction of the remaining edges. Conclude that the expected number of stages before achieving a maximal matching is $O(\log m)$.

Miscellaneous Exercises

1. Let $(S, \mathcal{I}, w)$ be a weighted matroid. Let M be the family of all maximal independent sets of minimum weight, and let $\mathcal{I}_{\max}$ be the family of all subsets of elements of M. Show that $(S, \mathcal{I}_{\max})$ is a matroid. (*Hint.* Use the blue rule to give a procedure for finding an $x \in J - I$ such that $I \cup \{x\} \in \mathcal{I}_{\max}$ whenever $I, J \in \mathcal{I}_{\max}$ and $|I| < |J|$.)

2. Let $T = (V, E)$ be a connected undirected tree such that each vertex has degree at most 3. Let $n = |V|$. Show that T has an edge whose removal disconnects T into two disjoint subtrees with no more than $\frac{2n+1}{3}$ vertices each. Give a linear-time algorithm to find such an edge.

3. Show how to solve the all-pairs shortest path problem on directed graphs when negative edge weights are allowed. (If there is no lower bound to the weights of paths from s to t, then we define the distance from s to t to be $-\infty$.)

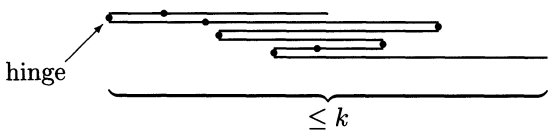
4. Suppose that we wish to schedule n unit-time jobs on a single processor, starting at time 0. Associated with each job j is a deadline $d_j \geq 1$ and a penalty $p_j \geq 0$. Job j must be completed by time d_j or the penalty p_j is incurred.

   (a) Let S be the set of jobs. Call a subset $A \subseteq S$ *independent* if all the jobs in A can be scheduled without violating their deadlines, irrespective of the jobs not in A. Thus A is independent if, when the elements of A are sorted by deadline, the i^{th} element in sorted order has deadline $d(i) \geq i$. Let $\mathcal{I}$ be the set of independent subsets of S. Show that $(S, \mathcal{I})$ is a matroid.

   (b) Give an efficient algorithm to produce a schedule that minimizes the total penalty.

5. A *pattern* is a finite-length string over the alphabet $\{0, 1, *\}$. A pattern σ *covers* a string x of 0's and 1's if x can be obtained from σ by replacing each occurrence of $*$ with either 0 or 1. For example, the pattern $0 * *1$ covers the four strings 0001, 0011, 0101, and 0111. A set A of patterns *covers* a set B of strings if every string in B is covered by some pattern in A. Show that the following problem is *coNP*-complete: given a set of patterns of length n, do they cover all strings of 0's and 1's of length n?

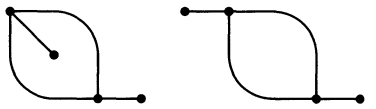
6. (The *Carpenter's Rule Problem*) Prove that the following problem is *NP*-complete: given a sequence of rigid rods of various integral lengths connected end-to-end by hinges, can it be folded so that its overall length is as most k?



7. Give a parsimonious reduction from CNFSat to 3CNFSat.

8. The standard adjacency list representation of an undirected graph G, as described for example in [3, p. 51], consists of a linked list of vertices in which the list element for vertex u contains a pointer to a linked list of pointers to all vertices adjacent to u. For plane graphs, the counterclockwise order of the vertices about u is given by the list order.

   For our purposes, this representation is inadequate for two reasons. First, in the representation of Lecture 14, the function $^-$ which reverses direction cannot be computed in constant time. In addition, the representation does not deal adequately with multiple edges; for example, it does not distinguish the following two nonisomorphic[1] plane graphs:



   Describe an enhanced adjacency list representation in which distinct undirected graphs with multiple edges and self-loops have distinct representations, each of the functions θ, $^-$, $\mathbf{h}$, and $\mathbf{t}$ can be computed in constant time, and single edges or vertices with all adjacent edges can be deleted in time proportional to the number of objects deleted. In the absence of multiple edges and self-loops, show how to obtain the new representation from the old one in linear time. All computations are to be done with pure pointer manipulation; no random access is allowed.

9. Let $G = (E, \theta, ^-)$ be an undirected graph in the formalism described in Lecture 14. Assume that G is represented by the adjacency list representation constructed in Miscellaneous Exercise 8. Show how to construct the dual G^* of G in linear time.

10. Let $G = (E, \theta, ^-)$ be an undirected graph in the formalism described in Lecture 14. Show that its characteristic $\chi(G)$ is always a nonnegative even number.

11. Prove Theorem 14.6 of Lecture 14: a graph $G = (E, \theta, ^-)$ has characteristic $\chi(G) = 0$ iff θ corresponds to the counterclockwise ordering induced by some plane embedding of G. (*Hint.* Use Exercise 10.)

[1] They are isomorphic as graphs, but not as *plane* graphs.

12. The following problem arose recently in Keith Marzullo's META project. Given an undirected graph with black and red vertices, does there exist a maximal clique with no red vertices? Show that this problem is *NP*-complete.

13. Show that the partition problem can be solved in polynomial time if the weights are restricted to be integers and bounded in absolute value by a fixed polynomial in the number of items.

14. Given $n > 0$, prove that there exists a number k such that when the binary-to-Gray operation is applied k times in succession starting with any bit string of length n, we get back the original bit string. What is the smallest such k as a function of n?

15. (a) In the proof given in Lecture 27 of the $\#P$-completeness of the permanent, show that the four-node widget can be replaced by a three-node widget.

    (b) Show that no two-node widget exists.

16. Consider the following scheduling problem. You are given positive integers m and t and an undirected graph $G = (V, E)$ whose vertices specify unit-time jobs and whose edges specify that the two jobs cannot be scheduled simultaneously. Can the jobs be scheduled on m identical processors so that all jobs complete within t time units?

    (a) Give a polynomial time algorithm for $t = 2$. (*Hint.* Use Miscellaneous Exercise 13.)

    (b) Show that the problem is *NP*-complete for $t \geq 3$.

17. Let G be a directed graph with positive and negative edge weights and let s and t be vertices of G. Recall that a path or cycle of G is *simple* if it has no repeated vertices.

    (a) Give a polynomial-time algorithm to determine whether G contains a simple cycle of negative weight.

    (b) Show that the problem of determining whether G contains a simple path from s to t of negative weight is *NP*-complete.

18. Show how matching can be used to give efficient algorithms for the following two problems.

    (a) Given an undirected graph with no isolated vertices, find an edge cover of minimum cardinality. (An *edge cover* is a subset of the edges such that every vertex is an endpoint of some edge in the subset.)

    (b) Find a vertex cover in a given undirected graph that is at most twice the cardinality of the smallest vertex cover.

19. Give a fast algorithm to determine whether a given directed graph has a cycle cover.

20. Given a bipartite graph $G = (U, V, E)$, say that $D \subseteq U$ is *deficient* if $|D| > |N(D)|$, where $N(D)$ denotes the set of neighbors of D; *i.e.*,

$$N(D) = \{v \in V \mid \exists u \in D\ (u, v) \in E\} .$$

Give a polynomial-time algorithm for finding a minimal deficient set if one exists. (*Hint.* Use Hall's Theorem; see Exercise 3, Homework 6. Grow a Hungarian tree.)

21. Let G be a connected undirected graph. We say G is *k-connected* if the deletion of any $k-1$ vertices leaves G connected. Give a polynomial-time algorithm (ideally, $O(k^2mn)$) for testing k-connectivity, and for finding a set of $k-1$ disconnecting vertices if G is not k-connected. (*Hint.* Use *Menger's Theorem*, which states that G is k-connected if and only if any pair of vertices is connected by at least k vertex-disjoint paths, then use maximum flow. You need not prove Menger's Theorem.)

22. Let p be a monic univariate polynomial of degree k:

$$p(x) = x^k + a_{k-1}x^{k-1} + \cdots + a_1 x + a_0 .$$

The *companion matrix* of p is the $k \times k$ matrix

$$C_p = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ -a_0 & -a_1 & -a_2 & -a_3 & -a_4 & -a_5 \end{bmatrix}$$

here illustrated for the case $k = 6$. The characteristic polynomial of C_p is p itself. Show how to compute the n^{th} power of C_p in time $O(k^2 \log n)$.

23. Let k be a finite field with q elements. Let A be an $n \times n$ matrix over k of rank r. Prove that for an $n \times n$ matrix R with entries chosen independently and uniformly at random from k,

$$\Pr(\text{rank } (RA)^2 = \text{rank } RA = \text{rank } A) = \prod_{i=1}^{r} (1 - \frac{1}{q^i}) .$$

24. The following problem arose recently in Tim Teitelbaum's synthesizer generator project. Let f be a binary function symbol, a and b constant symbols, and $X = \{x, y, \ldots\}$ a set of variables. A *term* is a well-formed expression over f, a, b, and X; for example, the following are terms:

$$a \qquad (1)$$

$$b \quad (2)$$
$$x \quad (3)$$
$$f(a,b) \quad (4)$$
$$f(a,x) \quad (5)$$
$$f(b,x) \quad (6)$$
$$f(f(x,y),z) \quad (7)$$

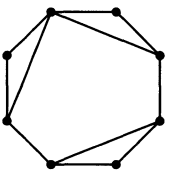
A term is a *ground term* if it contains no variables; for example, (1), (2) and (4) above are ground terms. A ground term t is a *substitution instance* of a term s if t can be obtained from s by substituting ground terms for the variables of s; for example, (4) is a substitution instance of (5) obtained by substituting b for x. A set of terms T is a *cover* if every ground term is a substitution instance of some term in T; for example, (1), (2), (5), (6), (7) form a cover.

(a) Show that determining whether a given set of terms is a cover is *coNP*-hard. (*Hint.* Encode the problem of Miscellaneous Exercise 5.)

(b) *Extra credit.* Show that the problem is *coNP*-complete.

25. Give an *NC* algorithm for finding the preorder numbering of a directed tree. That is, the algorithm should label each node of the tree with number of vertices visited before it in a preorder traversal.

26. An *outerplanar* is a graph that can be embedded in the plane so that every vertex is on the outer face. An *outerplane graph* is an outerplanar graph along with such an embedding.



An outerplane graph.

(a) Give a linear-time algorithm for testing whether a graph is outerplanar, and for finding an outerplane embedding if one exists. You may use for free the Hopcroft-Tarjan linear-time algorithm for finding a plane embedding of an arbitrary planar graph [52]. *Warning*: an arbitrary plane embedding of an outerplanar graph is not necessarily an outerplane embedding. Here are two embeddings of the same outerplanar graph, one outerplane and one not:

(b) Find the best $s(n)$ possible such that any outerplanar graph has a $\frac{1}{3}$-$\frac{2}{3}$ separator of size $s(n)$; *i.e.*, such that there exists a partition A, S, B of the vertices with $|A|, |B| \leq \frac{2n}{3}$, $|S| \leq s(n)$, and there are no edges between A and B. Give a linear time algorithm for finding the separator. (*Hint.* Use Miscellaneous Exercise 2.)

27. Assuming that comparisons between data elements take one unit of time on one processor, give parallel sorting algorithms that run on a CREW PRAM in

    (a) time $O(\log n)$ with $O(n^2)$ processors

    (b) time $O((\log n)^2)$ with $O(n)$ processors

    where n is the number of inputs. Your algorithm should produce an array of length n containing the input data in sorted order.

28. An $n \times n$ matrix is called a *circulant matrix* if the i^{th} row is obtained from the first row by a right rotation of i positions, $0 \leq i \leq n-1$. For example,

$$\begin{bmatrix} a & b & c & d \\ d & a & b & c \\ c & d & a & b \\ b & c & d & a \end{bmatrix}$$

    is a 4×4 circulant matrix.

    (a) Find an algorithm to multiply two $n \times n$ circulant matrices in time $O(\log n)$ with $O(n^2)$ processors.

    (b) Assuming that the field contains all n^{th} roots of unity and a multiplicative inverse of n, show that the processor bound in part (a) can be reduced to $O(n)$. Represent circulant matrices by their first row. (*Hint.* See [3, pp. 256–257].)

    (c) Under the assumptions of (b), find an algorithm to invert a nonsingular circulant matrix in $O(\log n)$ time with $O(n)$ processors.

29. In Lecture 40 we gave an *RNC* algorithm to test for the existence of a perfect matching in a given bipartite graph. In this exercise we extend this technique to arbitrary undirected graphs.

    The *Tutte matrix* of an undirected graph $G = (V, E)$ is an $n \times n$ matrix T with rows and columns indexed by V such that

$$T_{uv} = \begin{cases} x_{uv}, & \text{if } (u,v) \in E \text{ and } u < v \\ -x_{vu}, & \text{if } (u,v) \in E \text{ and } u > v \\ 0, & \text{if } (u,v) \notin E \end{cases}$$

where the x_{uv}, $u < v$, are indeterminates.

(a) Show that $\det T \neq 0$ iff G has a perfect matching.

(b) Use this fact to give an *RNC* test for the existence of a perfect matching in G.

(c) If G has a perfect matching, show how to compute one in random polynomial time.

III Homework Solutions

Homework 1 Solutions

1. (a) Suppose I and J are maximal independent subsets of A, but $|I| < |J|$. By property (ii) of matroids, we can find an $x \in J - I$ such that $I \cup \{x\} \in \mathcal{I}$. But then I was not a maximal independent subset of A; this is a contradiction.

   Incidentally, it can be shown that for systems $(S, \mathcal{I})$ satisfying axiom (i) of matroids that the property we have just proved is equivalent to axiom (ii) of matroids.

   (b) The set $I \cup \{x\}$ is dependent, since I is a maximal independent set. It therefore contains a minimal dependent set D. We show that D is unique. First, $x \in D$, since any subset of I is independent. Suppose there were two such cycles D and D'; assume without loss of generality that $y \in D - D'$. The set $(D \cup D') - \{x\}$ is a subset of I and is therefore independent. The set $D - \{y\}$ is also independent, since D is minimal, and its cardinality is no more than that of $(D \cup D') - \{x\}$. By property (ii) of matroids, elements of $(D \cup D') - \{x\}$ can be added to $D - \{y\}$ until the cardinality is the same as $(D \cup D') - \{x\}$, maintaining independence. Then y is not in the resulting set, since then D would be a subset, and D is dependent. Therefore the resulting set must be $(D \cup D') - \{y\}$. But this set contains D' as a subset, which is dependent; this is a contradiction.

   (c) Property (i) holds trivially, since removing an edge from a graph cannot add any cycles. To show (ii), let $E', E'' \subseteq E$ such that (V, E') and (V, E'') have no cycles. Let c' and c'' be the number of connected components of (V, E') and (V, E''), respectively, $m' = |E'|$, $m'' = |E''|$, and $n = |V|$. Suppose that $m' < m''$. By the equation $m + c = n$ proved in class, $c' > c''$. Then there must be an edge of E'' between two distinct connected components of (V, E'), otherwise all components of (V, E'') would be contained in components of (V, E'), implying that $c'' \geq c'$. Adding that edge to E' cannot give a cycle.

   The rank is $n - c$ and the maximal independent sets are the spanning forests.

   Two good references about matroids in optimization problems are [85, 70]. An excellent reference on matroid theory itself is [107].

2. (a) Since the graph is acyclic, all directed paths are of length at most $n - 1$. Define

$$\begin{aligned} E_{uv} &= \{e \in E \mid e \text{ lies on some } E\text{-path from } u \text{ to } v \\ &\qquad\qquad \text{of maximum length}\} \\ E_H &= \bigcup_{u,v \in V} E_{uv} \ . \end{aligned}$$

For any $(x, y) \in E_H$, the only E-path from x to y is the edge (x, y) itself; if there were a longer path, then (x, y) would never lie on any maximal length path and would not be in E_H. Thus

$$E_H = \{(u, v) \in E \mid \text{the length of the longest path from } u \text{ to } v \text{ is } 1\} . \quad (1)$$

Let F be an arbitrary subset of E such that $F^* = E^*$. For $e \in E_H$, if $e \notin F$, then F^* would not contain e either, by (1); this contradicts $F^* = E^*$. Thus $E_H \subseteq F$. Moreover, $E_H^* = E^*$, since any two vertices connected by a path are connected by a path of maximum length. Since $E_H^* = E^*$ and E_H is contained in every subset of E whose transitive closure is E^*, it is the unique minimal such set.

(b) Our algorithm to find E_H simply removes all edges (u, v) of E for which there exists a path of length two or more from u to v. We use the adjacency matrix representation. Let E denote this matrix. Using Boolean matrix multiplication ($\vee$ instead of $+$, $\wedge$ instead of $\cdot$), the matrix E^k has a 1 in position (u, v) iff there is a path from u to v of length exactly k. Compute the transitive closure matrix

$$\begin{aligned} E^* &= I \vee E \vee E^2 \vee \cdots \vee E^{n-1} \vee E^n \vee \cdots \\ &= I \vee E \vee E^2 \vee \cdots \vee E^{n-1} \quad \text{(why?)} \\ &= (I \vee E)^{n-1} . \end{aligned}$$

This can be done by repeated squaring in time $O(M(n) \log n)$, where $M(n)$ is the time to multiply two $n \times n$ matrices. (We can actually compute E^* in time $O(M(n))$, as we will see later.) Then compute $E^2 \cdot E^*$. This matrix has a nonzero value in position (u, v) iff there is a path of length at least 2 from u to v. Finally, the adjacency matrix of E_H is given by $E \wedge \neg(E^2 \cdot E^*)$.

In this problem the goal was to minimize the running time. If the goal were to minimize the amount of extra space used, then an in-place algorithm would be better; see [46]. The original paper showing that transitive reduction is as easy as transitive closure is [2].

3. This solution is from [85].

(a) Suppose the graph is connected and each vertex is of even degree. Starting from an arbitrary vertex v, trace an arbitrary path, traversing edges at most once (mark each edge as traversed as we encounter it), until we return to v. This must happen eventually: because each vertex is of even degree, is impossible to get stuck. Now delete the cycle we have found. Deleting a cycle maintains the invariant that all vertices are of even degree, but the graph may no longer be connected.

However, we can repeat the process on the connected components, and so on until all edges have been deleted. Now we string the cycles together to form one long cycle. Two cycles that contain a vertex v can be combined into one cycle in a figure 8, with v at the intersection point. Since the graph is connected, it is possible to string all the cycles together in this way to get one long cycle, an Euler circuit.

Conversely, if there is an Euler circuit, then each vertex v is of even degree, since each occurrence of v on the circuit accounts for two incident edges.

(b) We assume an adjacency list representation of the graph. Starting at vertex v_0, trace an arbitrary path until we return to v_0 as in part (a), giving a cycle $c = v_0, v_1, \ldots, v_{n-1}, v_0$ (the v_i's need not be distinct). As we traverse c, we delete each edge from the graph and create a doubly-linked circular list of these edges in the order they are encountered. Now for each v_i in order, we recursively find an Euler circuit c_i beginning and ending with v_i in the connected component containing v_i. We link c_i into c at v_i and then go on to v_{i+1}. In the recursive call to get an Euler circuit in the connected component of v_i, any v_j, $j > i$, in that connected component will have all remaining edges deleted, so by the time the algorithm gets to v_j, the connected component of v_j will consist only of v_j and there will be no work to do.

The algorithm runs in $O(m)$ time, because there is only a constant amount of work done for each edge.

Homework 2 Solutions

1. As shown in Theorem 4.8, a directed graph is a dag iff the DFS tree has no back edges. It was also shown that if the DFS tree is numbered in postorder, then all edges go from higher numbered vertices to lower numbered vertices. Then this numbering gives a topological sort.

2. (a) The graph G has an embedding consistent with θ iff this is true for every connected component of G, so we can assume without loss of generality that G is connected.

   Assume the adjacency lists are ordered according to θ. Then a depth-first search corresponds to traversing the edges clockwise around each face. All back edges can be drawn going up the right side of the path from the root down to the source of the back edge. The very first back edge encountered can be taken to be on the outer face. We need only maintain a stack of back edges that we are "inside", and make sure that any new back edge does not go up higher than the destination vertex of the innermost back edge that we are inside. The back edges that we are inside form a chain.

   In Lecture 14, Theorem 14.6 and Miscellaneous Exercise 11 we will see another approach using Euler's Theorem, which states that $n + n^* - m = 2$ for a connected plane graph G, where n, n^*, m are the number of vertices, faces, and edges of G, respectively. We will use the theorem in the opposite direction: given θ, we calculate the number n^* of "faces". We will show that $n + n^* - m = 2$ iff θ does indeed correspond to a plane embedding with n^* faces.

   (b) If G is not planar then it has no planar embeddings. Otherwise, G has exactly one embedding on the sphere consistent with θ. We can choose the North Pole to be in any one of the faces of the embedding and then project the graph onto the plane from the North Pole. Each choice of face for the pole gives a different embedding in the plane. Thus there are as many different embeddings as faces of G.

   Suppose now that G is not connected. Number the connected components $0, 1, \ldots, c-1$ and let f_i be the number of faces of component i. There are f_i ways to choose the outer face of the component i, so there are $f_0 f_1 \cdots f_{c-1}$ ways to choose all the outer faces. To get the number of embeddings, we must multiply this by the number of ways to place the components inside the inner faces of other components.

   For example, when $c = 2$, there are $f_1 - 1$ ways to place component 0 inside an inner face of component 1, $f_0 - 1$ ways to place component 1 inside an inner face of component 0, and one way to place the components so that neither occurs inside the other. This gives

$$(f_0 - 1) + (f_1 - 1) + 1 \; .$$

We claim that with c components this number is

$$(1+\sum_{i=0}^{c-1}(f_i-1))^{c-1}\ .$$

Each embedding determines a labeled forest whose vertices are the components. A component i is a child of another component j in this forest if it occurs inside an inner face of j with no intervening components. The component i can be placed inside any one of f_j-1 inner faces of j; let us indicate this by labeling the edge (i,j) in the forest with the factor f_j-1. Thus the number we are seeking is the sum over all labeled forests of the product of the edge labels on that forest.

We modify an argument of Prüfer that there are n^{n-2} labeled trees on n nodes (see [62]). We establish a one-to-one correspondence between labeled forests and sequences of length $c-1$ from the set

$$\{\top,0,1,\ldots,c-1\} \tag{2}$$

as follows. Given a labeled forest, start with the null sequence and repeat the following operation until there is only one vertex left. Prune off the lowest numbered leaf i and append j to the sequence, where j is the parent of i. If i has no parent, then append $\top$ to the sequence. Then each labeled forest determines a sequence of length $c-1$, and it is not difficult to reconstruct the forest uniquely from the sequence.

For each sequence $a_0,a_1,\ldots,a_{c-2}$ of length $c-1$ over the set (2), the product of the edge labels on the labeled forest corresponding to that sequence is

$$g(a_0)g(a_1)\cdots g(a_{c-2})\ ,$$

where $g(i)=f_i-1$ and $g(\top)=1$. The number we are seeking is the sum of all such products. This number is

$$(g(\top)+g(0)+g(1)+\cdots+g(c-1))^{c-1} \;=\; (1+\sum_{i=0}^{c-1}(f_i-1))^{c-1}\ .$$

The total number of embeddings is this number times the number of ways of choosing the outer faces of the components, or

$$(1+\sum_{i=0}^{c-1}(f_i-1))^{c-1}\prod_{i=0}^{c-1}f_i\ .$$

3. Here is an algorithm that takes $5n$ probes of the adjacency matrix. Let $d(v)$ be the degree of vertex v. The main difficulty is to locate one of the

interesting vertices (the body, tail, or sting); once we have done that, we can locate all the other interesting vertices with $3n$ probes and check that the graph is a scorpion. For example, if we have found a vertex v with $d(v) = n - 2$, then that vertex must be the body if the graph is a scorpion. By scanning the v^{th} row of the matrix, we can check that $d(v) = n - 2$ and determine its unique non-neighbor u, which must be the sting if the graph is a scorpion. Then by scanning the u^{th} row, we can verify that $d(u) = 1$ and find its unique neighbor w, which must be the tail; and with n more probes we can verify that $d(w) = 2$.

We start with an arbitrary vertex v, and scan the v^{th} row. If $d(v) = 0$ or $n - 1$, the graph is not a scorpion. If $d(v) = 1$, 2, or $n - 2$, then either v is interesting itself or one of its 1 or 2 neighbors is, and we can determine all the interesting vertices as above and check whether the graph is a scorpion with at most $4n$ additional probes.

Otherwise, $3 \leq d(v) \leq n-3$, and v is boring. Let B be the set of neighbors of v and let $S = V - (B \cup \{v\})$. The body must be in B and the sting and tail must be in S. Choose arbitrary $x \in B$ and $y \in S$ and repeat the following: if x and y are connected, then delete y from S (y cannot be the sting) and choose a new $y \in S$. If x and y are not connected, then delete x from B (x is not the body unless y is the sting) and choose a new $x \in B$. If the graph is indeed a scorpion, then when this process ends, B will be empty and y will be the sting. To see this, observe that B cannot be emptied without encountering the sting, because the body cannot be deleted from B by any vertex in S except the sting; and once the sting is encountered, all remaining elements of B will be deleted.

Whether or not the graph is a scorpion, the loop terminates after at most n probes of the adjacency matrix, since after each probe some vertex is discarded.

If a property is such that we have to look at *every* entry in the adjacency matrix, then that property is said to be *evasive*. Many monotone graph properties have been conjectured to be evasive. Yao [110] has shown that all monotone bipartite graph properties are evasive if we are given a bipartite adjacency matrix representation. The question for general graphs remains open. Bollobás discusses this issue in his book [13]. He gives a $6n$-probe solution to the scorpion problem there.

Homework 3 Solutions

1. The structure $(\mathbf{Reg}_\Sigma, \cup, \cdot, {}^*, \emptyset, \{\epsilon\})$ is a Kleene algebra if it satisfies the axioms of Kleene algebra. We therefore need to show, for any regular sets A, B and C,

$$\begin{aligned}
A \cup (B \cup C) &= (A \cup B) \cup C \\
A \cup B &= B \cup A \\
A \cup A &= A \\
A \cup \emptyset &= \emptyset \cup A = A \\
A \cdot (B \cdot C) &= (A \cdot B) \cdot C \\
A \cdot \{\epsilon\} &= \{\epsilon\} \cdot A = A \\
\emptyset \cdot A &= A \cdot \emptyset = \emptyset \\
A \cdot (B \cup C) &= A \cdot B \cup A \cdot C \\
(B \cup C) \cdot A &= B \cdot A \cup C \cdot A \\
A \cdot B^* \cdot C &= \sup_{n \geq 0} A \cdot B^n \cdot C \,. \qquad (3)
\end{aligned}$$

Note that the natural order on $\mathbf{Reg}_\Sigma$ is set inclusion $\subseteq$, since

$$A \subseteq B \;\leftrightarrow\; A \cup B = B \,.$$

Most of the above properties are obvious. The only one we will verify explicitly is (3). Recall the definition of the * operator in $\mathbf{Reg}_\Sigma$:

$$B^* = \{\epsilon\} \cup \{y_1 y_2 \cdots y_n \mid n \geq 1 \text{ and } y_i \in B,\ 1 \leq i \leq n\} \,.$$

Recall also that, by definition,

$$\begin{aligned}
B^0 &= \{\epsilon\} \\
B^{n+1} &= B \cdot B^n \,.
\end{aligned}$$

It follows by induction on n that

$$B^n = \{y_1 y_2 \cdots y_n \mid y_i \in B,\ 1 \leq i \leq n\} \,.$$

To be in the regular set on the left side of equation (3), a string must be of the form $x y_1 y_2 \cdots y_n z$ for some $n \geq 0$, where $x \in A$, $z \in C$, and $y_i \in B$, $1 \leq i \leq n$. Here we allow the possibility $n = 0$, in which case the string would be of the form xz. Thus the left side of (3) is equal to the set

$$\bigcup_{n \geq 0} A \cdot B^n \cdot C \,.$$

This is the least upper bound of the sets $A \cdot B^n \cdot C$, $n \geq 0$, with respect to set inclusion $\subseteq$: it is an upper bound, since it includes all the $A \cdot B^n \cdot C$ as subsets; and it is the least upper bound, since any set that includes all the $A \cdot B^n \cdot C$ must include their union.

2. Let $\mathbf{RExp}_\Sigma = \{\text{regular expressions over } \Sigma\}$, let $\mathcal{K}$ be an arbitrary Kleene algebra, and let I be an interpretation

$$I : \mathbf{RExp}_\Sigma \rightarrow \mathcal{K} .$$

The proof will proceed by induction on the structure of β. There are three base cases, corresponding to the regular expressions $b \in \Sigma$, 1, and 0. For $b \in \Sigma$, we have $R(b) = \{b\}$ and

$$\sup_{x \in R(b)} I(\alpha x \gamma) = I(\alpha b \gamma) .$$

The case of 1 is similar, since $R(1) = \{\epsilon\}$. Finally, since $R(0) = \emptyset$ and since the element $0_\mathcal{K}$ is the least element in $\mathcal{K}$ and therefore the supremum of the empty set,

$$\begin{aligned} \sup_{x \in R(0)} I(\alpha x \gamma) &= \sup \emptyset \\ &= 0_\mathcal{K} \\ &= I(0) \\ &= I(\alpha 0 \gamma) . \end{aligned}$$

There are three cases to the inductive step, one for each of the operators $+$, $\cdot$, *. We give a step-by-step argument for the case $+$, followed by a justification of each step.

$$\begin{aligned} I(\alpha(\beta_1 + \beta_2)\gamma) &= I(\alpha) \cdot (I(\beta_1) + I(\beta_2)) \cdot I(\gamma) && (4) \\ &= (I(\alpha) \cdot I(\beta_1) \cdot I(\gamma)) + (I(\alpha) \cdot I(\beta_2) \cdot I(\gamma)) && (5) \\ &= I(\alpha\beta_1\gamma) + I(\alpha\beta_2\gamma) && (6) \\ &= \sup_{x \in R(\beta_1)} I(\alpha x \gamma) + \sup_{y \in R(\beta_2)} I(\alpha y \gamma) && (7) \\ &= \sup_{z \in R(\beta_1) \cup R(\beta_2)} I(\alpha z \gamma) && (8) \\ &= \sup_{z \in R(\beta_1 + \beta_2)} I(\alpha z \gamma) . && (9) \end{aligned}$$

Equation (4) follows from the properties of the map I; (5) follows from the distributive laws of Kleene algebra satisfied by $\mathcal{K}$; (6) again follows from the properties of the map I; (7) follows from the induction hypothesis on β_1 and β_2; (8) follows from the general property of Kleene algebras that if A and B are two sets whose suprema $\sup A$ and $\sup B$ exist, then the supremum of $A \cup B$ exists and is equal to $\sup A + \sup B$ (this requires proof—see below); finally, equation (9) follows from the definition of the map R interpreting regular expressions as regular sets.

The general property used in equation (8) states that if A and B are two subsets of a Kleene algebra whose suprema $\sup A$ and $\sup B$ exist, then the supremum $\sup A \cup B$ of $A \cup B$ exists and

$$\sup A \cup B = \sup A + \sup B .$$

To prove this, we must show two things:

(i) $\sup A + \sup B$ is an upper bound for $A \cup B$; that is, for any $x \in A \cup B$, $x \leq \sup A + \sup B$; and

(ii) $\sup A + \sup B$ is the least such upper bound; that is, for any other upper bound y of the set $A \cup B$, $\sup A + \sup B \leq y$.

To show (i),

$$\begin{aligned} x \in A \cup B &\rightarrow x \in A \text{ or } x \in B \\ &\rightarrow x \leq \sup A \text{ or } x \leq \sup B \\ &\rightarrow x \leq \sup A + \sup B \ . \end{aligned}$$

To show (ii), let y be any other upper bound for $A \cup B$. Then

$$\begin{aligned} \forall x \in A \cup B \ x \leq y &\rightarrow \forall x \in A \ x \leq y \text{ and } \forall x \in B \ x \leq y \\ &\rightarrow \sup A \leq y \text{ and } \sup B \leq y \\ &\rightarrow \sup A + \sup B \leq y + y \ = \ y \ . \end{aligned}$$

We give a similar chain of equalities for the case of the operator $\cdot$, but omit the justifications.

$$\begin{aligned} I(\alpha(\beta_1\beta_2)\gamma) &= I(\alpha) \cdot I(\beta_1) \cdot I(\beta_2) \cdot I(\gamma) \\ &= I(\alpha\beta_1(\beta_2\gamma)) \\ &= \sup_{x \in R(\beta_1)} I(\alpha x(\beta_2\gamma)) \\ &= \sup_{x \in R(\beta_1)} I((\alpha x)\beta_2\gamma) \\ &= \sup_{x \in R(\beta_1)} \sup_{y \in R(\beta_2)} I(\alpha xy\gamma) \\ &= \sup_{x \in R(\beta_1),\ y \in R(\beta_2)} I(\alpha xy\gamma) \\ &= \sup_{z \in R(\beta_1\beta_2)} I(\alpha z\gamma) \ . \end{aligned}$$

Finally, for the case *, we have

$$\begin{aligned} I(\alpha\beta^*\gamma) &= I(\alpha) \cdot I(\beta)^* \cdot I(\gamma) \\ &= \sup_{n \geq 0} I(\alpha) \cdot I(\beta)^n \cdot I(\gamma) \\ &= \sup_{n \geq 0} I(\alpha\beta^n\gamma) \\ &= \sup_{n \geq 0} \sup_{x \in R(\beta^n)} I(\alpha x\gamma) \\ &= \sup_{n \geq 0,\ x \in R(\beta^n)} I(\alpha x\gamma) \end{aligned}$$

$$
\begin{aligned}
&= \sup_{x \in \bigcup_{n \geq 0} R(\beta^n)} I(\alpha x \gamma) \\
&= \sup_{x \in R(\beta^*)} I(\alpha x \gamma) \ .
\end{aligned}
$$

3. Suppose we are given a directed graph $G = (V, E, \ell)$ with nonnegative edge weights ℓ. The quantity $\ell(e)$ is called the *length* of $e \in E$. Define the *length* $\ell(p)$ of a directed path p to be the sum of the edge lengths along p; thus

$$
\ell(e_1 e_2 \cdots e_n) \ = \ \sum_{i=1}^{n} \ell(e_i) \ .
$$

In the version of Dijkstra's algorithm of Lecture 5, we use the loop invariant that the variable X contains the set of vertices whose minimal distance from s has already been determined, and $D(y)$ gives the minimum distance from s to y through only elements of X. Here we have to keep track of the paths themselves. To do this we will use pointers $P(\cdot)$. The new invariants are:

- for $v \in V$, the path $p(v) = v, P(v), P(P(v)), \ldots$ is a shortest path from v back to s such that all vertices except possibly v lie in X, or $\perp$ if no such path exists;
- $D(v) = \ell(p(v))$;
- for any $u \in X$, $v \notin X$, $D(u) \leq D(v)$.

Here is the algorithm:

```
X := {s};
P(s) := nil;
for each v ∈ V − {s} do
        if (s, v) ∈ E then
                P(v) := s;
                D(v) := ℓ(s, v)
        else
                P(v) := ⊥;
                D(v) := ∞
        end if
end for;
while X ≠ V do
        let u ∈ V − X such that D(u) is a minimum;
        X := X ∪ {u};
        for each edge (u, v) with v ∈ V − X do
                if D(u) + ℓ(u, v) < D(v) then
                        P(v) := u;
                        D(v) := D(u) + ℓ(u, v)
                end if
        end for
end while
```

Homework 4 Solutions

1. (a) The algorithm is correct because each step is an instance of the blue rule in the spanning tree matroid.

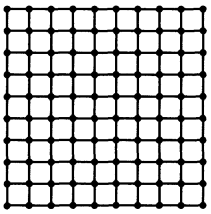
   (b) Prim's algorithm is a variant of Dijkstra's shortest-path algorithm. The implementation is very similar to the solution to Exercise 3 of Homework 3. We maintain a Fibonacci heap containing the set $V - T$, where V is the set of all vertices and T is the set of vertices in the portion of the spanning tree chosen up to that point. The value $D(v)$ for the purposes of **findmin**, **deletemin** and **decrement** is the weight of the minimum-weight edge connecting v to a vertex in T, or ∞ if no such edge exists. In addition, we maintain with each vertex v in the heap a pointer $P(v)$ to a vertex in T closest to v; *i.e.*, a $u \in T$ such that $d(u, v) = D(v)$.

   We initialize the values $D(v)$ and $P(v)$ after the first vertex u is chosen by setting $D(v) := d(u, v)$ and $P(v) := u$ for each vertex v adjacent to u and $D(v) := \infty$ and $P(v) :=$ **nil** for each vertex v not adjacent to u.

   At each step, we find the element w of the heap such that $D(w)$ is minimum, delete it from the heap, and add it to T. We also add the edge $(w, P(w))$ to the spanning tree. Then we update distances as follows: for each edge (w, v), $v \notin T$, if $d(w, v)$ is less than $D(v)$, we set $D(v) := d(w, v)$ and $P(v) := w$.

   It takes constant time for each edge or $O(m)$ time in all to do the updates. Using the Fibonacci heap, it takes $O(n \log n)$ time amortized over the entire sequence of operations to maintain the heap, find the minima and delete them, and decrement the values $D(v)$ when necessary.

2. Let C_k be the $k \times k$ checkerboard graph:



C_{10}

The number of vertices is $n = k^2$. We show that the smallest $\frac{1}{3}$-$\frac{2}{3}$ separators of C_k are of size at least $\frac{k}{\sqrt{3}} - \frac{1}{2} = \Omega(\sqrt{n})$.

Consider an arbitrary partition of the vertices of C_k into A, B, and S such that $|A|, |B| \leq 2n/3$ and there are no edges between A and B. If there is

an element of S in every row or an element of S in every column, we are done. Otherwise, there must be some row r and some column c with no S vertices. The row r must contain all A vertices or all B vertices, since if it contained some of each, it would have to have an S vertex to separate them. Without loss of generality assume it contains only A vertices. Similarly, c contains only A vertices or only B vertices, and they must be A vertices, since c intersects r. Thus there is an A vertex in every column and every row, since every row intersects c and every column intersects r. If a column or row has a B vertex, then it must also have an S vertex to separate it from the A vertices. So any column or row containing a B vertex must also contain an S vertex. Thus there are at most $|S|$ columns containing a B vertex and at most $|S|$ rows containing a B vertex; hence at the very most,

$$|B| \leq |S|^2 .$$

However, since $|A| \leq 2n/3$,

$$\begin{aligned} |B| &= n - (|A| + |S|) \\ &\geq \frac{n}{3} - |S| . \end{aligned}$$

Combining these inequalities, we get

$$|S|^2 + |S| - \frac{n}{3} \geq 0 ,$$

and $|S|$ must be at least as big as the positive root of this quadratic:

$$\begin{aligned} |S| &\geq \frac{-1 + \sqrt{1 + \frac{4n}{3}}}{2} \\ &> \frac{-1 + \sqrt{\frac{4n}{3}}}{2} \\ &= \frac{k}{\sqrt{3}} - \frac{1}{2} . \end{aligned}$$

Homework 5 Solutions

1. (a) For every edge (u, v) in G, compute the residual capacity from u to v using the formula $r(u, v) = c(u, v) - f(u, v)$, where c is the capacity function. Also compute the residual capacity $r(v, u)$ in the opposite direction. If $r(u, v)$ is zero, then do not put the edge (u, v) into G_f. Otherwise, put the edge (u, v) into G_f with capacity $r(u, v)$. Do the same for (v, u). All the relevant computations can be done in $O(m)$ time since we spend at most constant time for each edge.

   (b) A minor modification of the algorithm of Exercise 3, Homework 3 does it. The only difference is that we are seeking paths of maximum bottleneck capacity instead of minimum length. Here is the algorithm:

```
X := {s};
P(s) := nil;
for each x ∈ V − {s} do
        D(x) := r(s, x);
        P(x) := s
end for;
while X ≠ V do
        let x ∈ V − X such that D(x) is a maximum;
        X := X ∪ {x};
        for each edge (x, y) ∈ E_f such that y ∈ V − X do
                if min(D(x), r(x, y)) > D(y) then
                        P(y) := x;
                        D(y) := min(D(x), r(x, y))
                end if
        end for
end while
```

2. Let every edge in G have unit capacity and find a maximum flow f^* in G. Then there exist k edge-disjoint paths from s to t if and only if $|f^*| \geq k$: certainly if there exist k edge-disjoint paths, then $|f^*| \geq k$ by pushing one unit of flow along each path. Conversely, if $|f^*| \geq k$, then we can repeatedly find a path flow from s to t with unit flow as in the proof of Lemma 17.4, then remove the edges along this path and repeat. With each iteration the flow value decreases by one, so at least k paths are found.

3. Make G a directed graph by replacing each undirected edge $\{u, v\}$ with two directed edges (u, v) and (v, u). The capacities on these directed edges will be the same as those on the corresponding original edge. Find a maximum flow from s to t in this network and compute the residual graph of this flow. Take A to be the set of vertices reachable from s in the residual graph, and let B be the rest. By the Max Flow-Min Cut Theorem, this

cut has minimum weight in the network. It is also a minimum cut for the original undirected graph because any edge between A and B has the same weight in both graphs.

Homework 6 Solutions

1. Gale and Shapley were the first to investigate the stable marriage problem and gave this solution in 1962 [36].

   Let $r_g(b)$ be g's ranking of b, $r_b(g)$ be b's ranking of g. Let B be the set of boys and G be the set of girls, $n = |B| = |G|$. If M is a partial matching, let $B_M = \{b \mid \exists g\ (b,g) \in M\}$ and $G_M = \{g \mid \exists b\ (b,g) \in M\}$. Let $M(b) = g$ if $(b,g) \in M$, undefined if no such g exists. Let $M(g) = b$ if $(b,g) \in M$, undefined if no such b exists. Execute the following program:

   ```
   M := ∅;
   while |M| < n do
         g := arbitrary element of G − G_M;
         b := element of B with r_g(b) maximum and either
               (i) b ∉ B_M
               (ii) b ∈ B_M and r_b(g) > r_b(M(b));
         if (i), set M := M ∪ {(b, g)};
         if (ii), set M := (M − {(b, M(b))}) ∪ {(b, g)}
   end while
   ```

   The following invariants are maintained by the **while** loop:

   - M is stable;
   - if $r_g(b) \geq r_g(M(g))$ then $b \in B_M$.

   The algorithm halts in $O(n^2)$ iterations of the **while** loop because the quantity

   $$\sum_{b \in B_M} r_b(M(b))$$

   increases by at least one each time.

2. Ford and Fulkerson solved this problem by reducing it to a network flow problem and then applying the Max Flow-Min Cut Theorem [34].

   In a bipartite graph $G = (U, V, E)$, the size of any matching is at most the size of any vertex cover, since all matched edges must have at least one endpoint in the cover. To show that this bound is attained, we construct a flow network H with vertices $U \cup V \cup \{s,t\}$ and edges

   $$\{(s,u) \mid u \in U\} \ \cup \ E \ \cup \ \{(v,t) \mid v \in V\}\ .$$

   The edges of E are directed from U to V in H. We give the edges of E infinite (or sufficiently large finite) capacity, and all other edges capacity 1. Let S, T be a minimum s,t-cut in H. This cut has finite capacity, since

it is bounded by the capacity of the cut $\{s\}, (U \cup V \cup \{t\}) - \{s\}$, which is $|U|$. Therefore, no edge in E can cross from S to T, since those edges have infinite capacity. Thus each edge in E either

- has both endpoints in S,
- has both endpoints in T, or
- crosses from T to S.

In any of these three cases, the edge is incident to an element of the set

$$(T \cap U) \ \cup \ (S \cap V) \ ,$$

so this set forms a vertex cover of G. Moreover, the capacity of the cut S, T is the size of this set, which is also the maximum flow value in H by the Max Flow-Min Cut Theorem. In Lecture 18 we argued that the size of the maximum matching in G is the value of a maximum flow in H.

3. Suppose first that G has a matching in which every vertex of U is matched. Let S be a subset of U. For every vertex $s \in S$, the vertex that s is matched to appears in $N(S)$. Since no two vertices in S are matched to the same vertex in V, we have $|N(S)| \geq |S|$.

   Conversely, suppose G does not have a matching in which every vertex of U is matched. By the König-Egerváry Theorem, there exists a vertex cover C of G containing fewer than $|U|$ vertices. Let $S = U - C$. All vertices adjacent to vertices S must be in $V \cap C$, otherwise C would not be a vertex cover; *i.e.*, $N(S) \subseteq V \cap C$. Then

$$\begin{array}{rcll} |N(S)| & \leq & |V \cap C| & \\ & = & |C| - |U \cap C| & \text{since } U \text{ and } V \text{ are disjoint} \\ & < & |U| - |U \cap C| & \\ & = & |S| & \text{since } S \text{ is defined to be } U - C. \end{array}$$

   Thus there exists an $S \subseteq U$ such that $|N(S)| < |S|$.

   Hall's theorem is sometimes called the Marriage Theorem because it tells us whether all U vertices can be "married" to a V vertex of their own.

4. Let d be the degree of the vertices in the regular bipartite graph $G = (U, V, E)$. The total number of edges is

$$d \cdot |U| \ = \ d \cdot |V| \ ,$$

   thus $|U| = |V|$; therefore any matching that uses all the vertices in U will be a perfect matching in G. By Hall's Theorem, it suffices to show that $|S| \leq |N(S)|$ for any $S \subseteq U$. For an arbitrary subset S of U, consider the subgraph of G induced by $S \cup N(S)$. There are exactly $d \cdot |S|$ edges in this

subgraph, since every vertex in S has degree d. Similarly, this subgraph has no more than $d \cdot |N(S)|$ edges, because no vertex in $N(S)$ has degree larger than d. Thus

$$d \cdot |S| \leq d \cdot |N(S)| .$$

It follows that $|S| \leq |N(S)|$.

One of the reasons that matching theory is interesting is because of beautiful min-max theorems like the König-Egerváry Theorem and Hall's Theorem. Lovász and Plummer's book [75] provides extensive coverage of matchings for those who want to learn more about them. Papadimitriou and Steiglitz [85] and Lawler [70] also discuss important algorithmic and theoretical aspects of matchings.

Homework 7 Solutions

1. (a) We reduce the general CNFSat problem to the restricted problem in which variables are allowed at most three occurrences to show that the restricted problem is *NP*-hard. Given the CNF formula $\mathcal{B}$, let x be a variable with exactly m occurrences in $\mathcal{B}$. Let $x_1, x_2, \ldots, x_m$ be m new variables. Replace the i^{th} occurrence of x with x_i and append the CNF formula

$$(\neg x_1 \vee x_2) \ \wedge \ (\neg x_2 \vee x_3) \ \wedge \ \cdots \ \wedge \ (\neg x_m \vee x_1)$$

which is equivalent to the chain of implications

$$x_1 \ \rightarrow \ x_2 \ \rightarrow \ x_3 \ \rightarrow \ \cdots \ \rightarrow \ x_m \ \rightarrow \ x_1 \, .$$

This will force all the x_i to have the same truth value in any satisfying assignment, and there are exactly three occurrences of each x_i. Do this for all variables in the original formula.

The restricted problem is in *NP*, since it is a special case of the unrestricted CNFSat problem. Therefore it is *NP*-complete.

(b) If a variable x appears only positively (only negatively), then we can satisfy the clauses containing x by assigning x the value *true* (*false*), thus we might as well eliminate those clauses. The new formula is satisfiable iff the original one is.

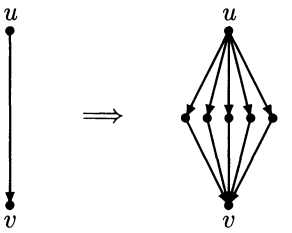
Suppose then that the variable x appears both positively and negatively. Since there are only two occurrences of x, there is one of each. If the occurrences are in the same clause, then that clause is true under any truth assignment, and we can eliminate it. If not, and if the two clauses contain no other variables, then we have reached a contradiction $x \wedge \neg x$ and there is no satisfying assignment. Otherwise, we apply the *resolution rule* of propositional logic: we combine the two clauses containing x, but throw out x itself. For example, if one clause is $x \vee y \vee z$ and the other is $\neg x \vee u \vee v$, then the new clause is $y \vee z \vee u \vee v$. Again, the new formula is satisfiable iff the original one was.

We continue applying these rules until we see a contradiction $x \wedge \neg x$ or we eliminate all of the variables. In the latter case, the formula is satisfiable.

The resolution rule is a widely used proof procedure for propositional logic. It is known to require exponential time in the worst case when there are no restrictions on the number of occurrences of variables [47, 101]. Resolution, suitably extended to handle first-order formulas, forms the basis for many PROLOG implementations.

2. If the graph is not strongly connected, there is no TSP tour. Otherwise, we calculate the weight w^* of an optimal TSP tour as follows. Let $w(e)$ be the weight of edge e and let $d = \max_{e \in E} w(e)$. There exists a tour of weight at most $n^2 d$ consisting of minimum-weight paths between pairs of vertices (u_1, u_2), $(u_2, u_3), \ldots, (u_n, u_1)$ placed end to end. Starting with $n^2 d$, perform a binary search to find w^*. This can be done in polynomial time, because we have to make only $O(\log(n^2 d)) = O(\log n + \log d)$ calls to our subroutine that tells whether there exists a TSP tour of weight k, and $\log d$ is the size of the binary representation of d.

   Next we wish to determine the number of times each edge is traversed in some optimal tour. Note that these numbers are not unique; different tours may traverse an edge different numbers of times. Consider an operation in which we replace the edge $e = (u, v)$ with the graph D_k pictured below right, where k is the number of extra vertices. In the example shown, $k = 5$.



   We give each of the edges in D_k weight $w(e)/2$. Any tour in the new graph gives rise to a tour in the old graph of the same weight. Conversely, any tour in the old graph that traverses e at least k times gives rise to a tour in the new graph of the same weight. Thus the weight of an optimal tour in the new graph is at least w^*, and strictly greater than w^* if all optimal tours in the old graph traverse e fewer than k times. Since no optimal tour of the old graph traverses e more than n times, if $k > n$ then the weight of the optimal tour in the new graph must be strictly greater than w^*.

   Now, for each edge e in turn, we replace e with D_k, where k is as large as possible such that there still exists a tour of weight w^*. We discover k by binary search in the interval $0 \leq k \leq n$. When we are done, there is a tour of weight w^* in the resulting graph that traverses each edge exactly once (if not, some D_k could have been replaced by D_{k+1}). We find an Euler circuit in this graph, which will give rise to a TSP tour in the original graph of optimum weight w^*.

3. Let $\mathcal{B}$ be a CNF formula. We wish to produce an undirected graph $G = (V, E)$, integer k, and distinguished vertices s_i, t_i, $1 \leq i \leq k$, such that there exist k vertex-disjoint paths connecting the s_i and t_i iff $\mathcal{B}$ has a satisfying truth assignment.

Let C be the set of clauses, let X be the set of variables, and let L be the set of occurrences of literals in $\mathcal{B}$. Let the set of vertices be

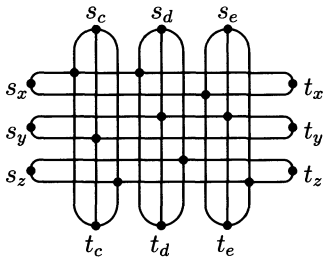
$$V = L \cup \{s_c,\ t_c \mid c \in C\} \cup \{s_x,\ t_x \mid x \in X\}$$

and take $k = |C| + |X|$. For $c \in C$, connect all occurrences of literals in c to s_c and t_c. For $x \in X$, connect all the positive occurrences of x in $\mathcal{B}$ in a single path, connect the negative occurrences of x in a single path, and connect one endpoint of each of these two paths to s_x and the other to t_x.

For example, consider the formula

$$\mathcal{B} = \underbrace{(x \vee \overline{y} \vee \overline{z})}_{c} \wedge \underbrace{(x \vee y \vee z)}_{d} \wedge \underbrace{(\overline{x} \vee y \vee \overline{z})}_{e}$$

with variables $X = \{x, y, z\}$ and clauses $C = \{c, d, e\}$. The construction produces the following graph:



If $\mathcal{B}$ has a satisfying assignment, then for each $c \in C$, take the path from s_c to t_c of length two through some true literal of c, and for each $x \in X$, take the path from s_x to t_x through the false literals. These k paths are vertex-disjoint. Conversely, suppose there are k vertex-disjoint paths. The path from s_x to t_x, $x \in X$, must go through either all and only the positive occurrences of x or all and only the negative occurrences of x; any deviation would necessarily go through some s_c or t_c. Assign x *true* if the path from s_x to t_x goes through the negative occurrences of x and *false* if it goes through the positive occurrences of x. Since the paths are vertex-disjoint, the paths from s_c to t_c, $c \in C$, go through only true literals. Thus the truth assignment satisfies all clauses.

The problem is in *NP* since the disjoint paths can be guessed and verified in polynomial time.

Homework 8 Solutions

1. This problem is in *coNP* because its complement is in *NP*: we can guess a value for each variable and verify in polynomial time that the given expression does not vanish. We show *coNP*-hardness by showing that its complement is *NP*-hard using a reduction from CNFSat. Given a Boolean formula, we transform it into an algebraic expression that vanishes mod p for all variable assignments iff the original formula is unsatisfiable. We simulate the Boolean values *true* and *false* with 1 and 0 in $\mathcal{Z}_p$, respectively. If x is a variable of the Boolean formula, replace it with x^{p-1}. Replace each expression $\neg A$ by $1 - A$, $A \wedge B$ by AB, and $A \vee B$ by $A + B - AB$.

2. (a) The restricted problem remains in *NP*, so the difficult part is to show *NP*-hardness. We do this by reducing the undirected Hamiltonian circuit problem to TSP with the triangle inequality. Given an undirected graph $G = (V, E)$, we will construct a symmetric distance function $d : V \times V \to \mathcal{N}$ satisfying the triangle inequality such that G has a Hamiltonian circuit iff it has a TSP tour of length n. Let

$$d(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E, \\ 2 & \text{if } (u, v) \notin E. \end{cases}$$

   Then d obeys the triangle inequality trivially. A TSP tour in G that has length n must contain only edges of length 1. Thus it corresponds directly to a Hamiltonian tour in G.

   (b) Actually, it is possible to find in polynomial time a tour that is no worse than 3/2 times the optimal tour by using an algorithm known as *Christofides' heuristic*. Our solution will only be guaranteed to be no more than twice the optimal solution, but it is important to understand how our solution works before looking at Christofides' algorithm, which uses matching. Papadimitriou and Steiglitz [85] discuss Christofides' heuristic for those interested in finding out more about it.

   We will assume that distances are symmetric. First, we find a minimum spanning tree T. Next, we create a directed graph G by using two copies of each edge of T, one in each direction. Finally, we find an Euler circuit in G, which gives a TSP tour of weight twice that of the minimum spanning tree. Since every TSP tour contains a spanning tree of the original graph, the length of any TSP tour is at least as large as the weight of the minimum spanning tree. Therefore, the length of our Euler circuit is no more than twice the length of the optimal TSP tour.

   Since the triangle inequality holds, we can convert the Euler circuit to a TSP tour of no greater weight in which vertices are visited only once: we merely skip over vertices previously visited.

3. Recall that if $G = (V, E)$ is not acyclic, its transitive reduction or Hasse diagram is not necessarily unique. The problem is in NP, since we can just guess a transitive reduction, verify that it is antitransitive (*i.e.*, that if (u, v) and $(v, w) \in E$ then $(u, w) \notin E$), take its transitive closure, and verify that this is the same as the transitive closure of E.

   We show that the transitive reduction problem is NP-hard by exhibiting a reduction from the directed Hamiltonian circuit problem. The reduction from the vertex cover problem to the directed Hamiltonian circuit problem given in Lecture 24 produces a strongly connected graph, whether or not it has a Hamiltonian circuit. Thus the problem of determining whether a given strongly connected graph has a Hamiltonian circuit is also NP-hard. We now argue that a strongly connected graph H has a Hamiltonian circuit if and only if it has a transitive reduction with at most n edges. Note that the transitive closure of such a graph is the complete graph. If H has a Hamiltonian circuit, then the Hamiltonian circuit itself serves as a transitive reduction, and it has exactly n edges. Conversely, any transitive reduction of H must contain at least n edges, since it must enter every vertex at least once, since H is strongly connected. If it contains exactly n edges, then it enters every vertex exactly once, thus it must be a Hamiltonian circuit.

Homework 9 Solutions

1. We will use the concurrent-read exclusive-write (CREW) PRAM model with unit cost for integer operations and comparisons.

   Represent the dag G as an adjacency matrix A. Compute its reflexive transitive closure G^* either by computing $A^* = (I \vee A)^n$ in NC using parallel prefix, or more efficiently, using the relationship between matrix multiplication and transitive closure discussed in Lecture 5. Then sort the vertices by indegree in G^* using the algorithm of Miscellaneous Exercise 27. This gives a topological sort, since if there is a path from u to v in G, then vertex u has smaller indegree than v in G^*.

2. If G has an odd cycle, then it must have an odd simple cycle (one with no repeated vertices), because any odd cycle that is not simple is composed of two smaller cycles, one of which must be odd. Therefore, to check for an odd cycle, we need only check for a path of odd length at most n from a vertex back to itself. The k^{th} power of the adjacency matrix A of G tells us the paths of length k in G. Using parallel prefix, we can compute all the odd powers of A up to n in NC and see if any of them contain a 1 on the main diagonal.

   This algorithm is NC, but it is not very efficient in its use of processors. We can save a factor of n processors by observing that we only have to check the diagonal of A^k for some odd k greater than $n-1$. If there is an odd cycle of shorter length, we can extend it to one of length k by retracing an edge backwards and forwards. Thus we can use matrix powering instead of parallel prefix.

   Another approach would compute the * of a matrix A over a Kleene algebra consisting of four elements $\bot$, 0, 1, and $\top$, where the operations $+$ and $\cdot$ are given by

$+$	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	1	$\top$
0	0	0	$\top$	$\top$
1	1	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\cdot$	$\bot$	0	1	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
0	$\bot$	0	1	$\top$
1	$\bot$	1	0	$\top$
$\top$	$\bot$	$\top$	$\top$	$\top$

   We set

$$A_{ij} = \begin{cases} 0 & \text{if } i = j \\ 1 & \text{if } (i,j) \in E \\ \bot & \text{otherwise.} \end{cases}$$

   The element 0 in position ij of A^* means that there is an even-length path between i and j; 1 means there is an odd-length path; $\top$ means there are

both (then the graph is not bipartite); and $\perp$ means i and j are in different connected components.

Here is a method due to Shiloach and Vishkin [93] that is much more efficient in its use of processors. First, find a spanning tree of G. This can be done in $O(\log n)$ time using $O(n+m)$ processors on a CRCW PRAM. Assign a parity 0 or 1 to each vertex according to its distance from the root; this too can be done using $O(n+m)$ processors in parallel, using the technique of *pointer doubling* (see Miscellaneous Exercise 25). Finally, check every edge of G to make sure that an edge does not join two vertices of the same color.

3. (a) The system is described by the following matrix-vector equation, illustrated here for the case $k = 5$.

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ a_5 & a_4 & a_3 & a_2 & a_1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{n-5} \\ x_{n-4} \\ x_{n-3} \\ x_{n-2} \\ x_{n-1} \\ c \end{bmatrix} = \begin{bmatrix} x_{n-4} \\ x_{n-3} \\ x_{n-2} \\ x_{n-1} \\ x_n \\ c \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ c \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c \end{bmatrix} \tag{11}$$

Raising the $(k+1) \times (k+1)$ matrix on the left hand side of (10) to the n^{th} power and using (11), we obtain

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ a_5 & a_4 & a_3 & a_2 & a_1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}^n \cdot \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c \end{bmatrix} = \begin{bmatrix} x_n \\ x_{n+1} \\ x_{n+2} \\ x_{n+3} \\ x_{n+4} \\ c \end{bmatrix}$$

It therefore suffices to compute the n^{th} power of the matrix in (10). Represent this matrix as the sum $C + U$, where

$$C = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ a_5 & a_4 & a_3 & a_2 & a_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

and

$$U = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} .$$

Note that $UC = U$ and $U^2 = 0$; from this it follows that

$$\begin{aligned} (C+U)^m &= C^m + C^{m-1}U + C^{m-2}U + \cdots + CU + U \\ &= C^m + D_m , \end{aligned} \tag{12}$$

where

$$D_m = (\sum_{i=0}^{m-1} C^i)U .$$

The matrix C is a block diagonal matrix consisting of a $k \times k$ companion matrix in the upper left and a 1×1 identity matrix in the lower right, thus Miscellaneous Exercise 22 applies.

Given C^m and D_m, we can obtain C^{2m}, C^{m+1}, D_{2m}, and D_{m+1} in time $O(k^2)$. For the first two we use Miscellaneous Exercise 22. For the last two, we have

$$\begin{aligned} D_{2m} &= C^m D_m + D_m \\ D_{m+1} &= CD_m + U . \end{aligned}$$

The product $C^m D_m$ is essentially a matrix-vector product, since all columns of D_m except the last are zero.

Thus we can compute D_n and C^n with at most $\log n$ matrix operations, each taking time $O(k^2)$ (the sequence of operations is determined by the binary representation of n). The desired power $(C+U)^n$ is given by (12).

For a different approach to this problem, see [109, 86, 45].

(b) With $O(k^\alpha)$ processors, we can multiply two $(k+1) \times (k+1)$ matrices in time $O(\log k)$ using a parallel version of Strassen's matrix multiplication algorithm. Thus the n^{th} power of the matrix in (10) can be computed in time $O(\log k \log n)$ by repeated squaring.

Here is how we parallelize Strassen's algorithm. Recall that Strassen multiplies 2×2 matrices as follows:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$= \begin{bmatrix} s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\ s_6 + s_7 & s_2 - s_3 + s_5 - s_7 \end{bmatrix}$$

where

$$\begin{aligned} s_1 &= (b-d)\cdot(g+h) \\ s_2 &= (a+d)\cdot(e+h) \\ s_3 &= (a-c)\cdot(e+f) \\ s_4 &= h\cdot(a+b) \\ s_5 &= a\cdot(f-h) \\ s_6 &= d\cdot(g-e) \\ s_7 &= e\cdot(c+d)\ . \end{aligned}$$

We can first compute the quantities $b-d$, $g+h$, $a+d$, $e+h$, $a-c$, $e+f$, $a+b$, $f-h$, $g-e$, and $c+d$ in parallel, then compute $s_1, \ldots, s_7$ in parallel from these, and finally the four entries of the product from the s_i in parallel.

Now we apply this technique inductively. Given a pair of $k \times k$ matrices, we wish to build an *NC* circuit to compute their product. We break each matrix up into four submatrices of size roughly $\frac{k}{2} \times \frac{k}{2}$, and assuming that we have already constructed circuits to compute the sum and product of $\frac{k}{2} \times \frac{k}{2}$ matrices, we can use those circuits in the calculation of the $k \times k$ product exactly as in the 2×2 case described above.

Let $P(k)$ and $T(k)$ be, respectively, the number of processors (size of the circuit) and the time (depth of the circuit) necessary to to multiply two $k \times k$ matrices by this method. These quantities satisfy the recurrences

$$\begin{aligned} P(k) &= 7P(\frac{k}{2}) + O(k^2) \\ T(k) &= T(\frac{k}{2}) + O(1) \end{aligned}$$

since we need $O(k^2)$ processors and $O(1)$ time to add two $k \times k$ matrices. These recurrences give

$$\begin{aligned} P(k) &= O(k^{\log 7}) = O(k^{2.81\ldots}) \\ T(k) &= O(\log k)\ . \end{aligned}$$

For more details, see [44].

(c) Let y be an indeterminate and consider the generating function

$$x(y) = \sum_{i=0}^{\infty} x_i y^i$$

where the x_i are the solution to the recurrence. Multiplying $x(y)$ by y^i shifts the coefficients i positions; using this trick, we can encode

the linear recurrence as a single equation involving shifts of $x(y)$ as follows. Let

$$\begin{aligned} p(y) &= a_1 + a_2 y + a_3 y^2 + \cdots + a_k y^{k-1} \\ q(y) &= c_0 + c_1 y + c_2 y^2 + \cdots + c_{k-1} y^{k-1} \\ r(y) &= (p(y) - a_k y^{k-1})(q(y) - c_{k-1} y^{k-1}) \bmod y^k \ . \end{aligned}$$

(The coefficients of $r(y)$ can be computed in time $O(\log k)$ with $2k$ processors using FFT.) Then

$$\begin{aligned} x(y) &= a_1 y(x(y) - (c_0 + c_1 y + \cdots + c_{k-2} y^{k-2})) \\ &\quad + a_2 y^2 (x(y) - (c_0 + c_1 y + \cdots + c_{k-3} y^{k-3})) \\ &\quad + a_3 y^3 (x(y) - (c_0 + c_1 y + \cdots + c_{k-4} y^{k-4})) \\ &\quad + \cdots \\ &\quad + a_{k-1} y^{k-1} (x(y) - c_0) \\ &\quad + a_k y^k x(y) \\ &\quad + c y^k (1 + y + y^2 + \cdots) \\ &\quad + c_0 + c_1 y + c_2 y^2 + \cdots + c_{k-1} y^{k-1} \\ &= y p(y) x(y) - r(y) + \frac{c y^k}{1-y} + q(y) \ , \end{aligned}$$

therefore

$$\begin{aligned} x(y) &= \frac{(q(y) - r(y))(1-y) + c y^k}{(1-y)(1 - y p(y))} \\ &= \frac{u(y)}{1 - y v(y)} \end{aligned} \tag{13}$$

where

$$\begin{aligned} u(y) &= (q(y) - r(y))(1-y) + c y^k \\ v(y) &= 1 + p(y) - y p(y) \ . \end{aligned}$$

Expanding the denominator of (13) in an infinite series, we get

$$x(y) = u(y) \sum_{i=0}^{\infty} y^i v(y)^i \ ,$$

and since we are only interested in computing the first n terms, we might as well truncate the series and compute instead

$$\widehat{x}(y) = u(y) \sum_{i=0}^{n-1} y^i v(y)^i \tag{14}$$

$$= \frac{u(y)(1 - y^n v(y)^n)}{1 - y v(y)} \ . \tag{15}$$

We take Fourier transforms and use componentwise operations. In particular, to get the transform of $y^n v(y)^n$, we raise the transform of $yv(y)$ to the n^{th} power componentwise, and to get the transform of $\sum_{i=0}^{n-1} y^i v(y)^i$, we divide the transform of $1 - y^n v(y)^n$ by the transform of $1 - yv(y)$ componentwise. There is one glitch: suppose that at a root of unity ω^j, we find that $\omega^j v(\omega^j) = 1$, so that we cannot do the division in (15) at the j^{th} component. In that case, we use (14) instead, observing that

$$\sum_{i=0}^{n-1} \omega^{ji} v(\omega^j)^i \quad = \quad n \; .$$

We need N^{th} roots of unity and a multiplicative inverse of N, where N exceeds the degrees of all polynomials involved, so that there will be no wrap; since $u(y)$ and $v(y)$ are each of degree at most k, the numerator of (15) is of degree at most $kn+k+n$, thus $N > kn+k+n$ suffices.

Homework 10 Solutions

1. (a) As shown in Lecture 36, the expected value of the random variable $\mathcal{E}(X_{n+1} \mid S_n)$ is

$$\mathcal{E}(\mathcal{E}(X_{n+1} \mid S_n)) \;=\; \mathcal{E}X_{n+1} \;.$$

This yields the recurrence

$$\begin{aligned} \mathcal{E}S_0 &= 0 \\ \mathcal{E}S_{n+1} &= \mathcal{E}(S_n + X_{n+1}) \\ &= \mathcal{E}S_n + \mathcal{E}X_{n+1} \\ &= \mathcal{E}S_n + \mathcal{E}(\mathcal{E}(X_{n+1} \mid S_n)) \\ &\geq \mathcal{E}S_n + \mathcal{E}(\epsilon(m - S_n)) \\ &= \epsilon m + (1-\epsilon)\mathcal{E}S_n \end{aligned}$$

whose solution gives

$$\mathcal{E}S_n \;\geq\; m(1-(1-\epsilon)^n) \;.$$

(b)

$$\begin{aligned} \mathcal{E}S_n &= \sum_{i=0}^{m} i \cdot \Pr(S_n = i) \\ &= m \cdot \Pr(S_n = m) + \sum_{i=0}^{m-1} i \cdot \Pr(S_n = i) \\ &\leq m \cdot \Pr(S_n = m) + \sum_{i=0}^{m-1} (m-1) \cdot \Pr(S_n = i) \\ &= m \cdot \Pr(S_n = m) + (m-1) \cdot (1 - \Pr(S_n = m)) \\ &= m - 1 + \Pr(S_n = m) \;. \end{aligned}$$

Combining this inequality with (a), we obtain

$$\Pr(S_n = m) \;\geq\; 1 - m(1-\epsilon)^n \;.$$

(c) Using (b),

$$\begin{aligned} \mathcal{E}f(S_n) &= 1 \cdot \Pr(S_n < m) + 0 \cdot \Pr(S_n = m) \\ &= 1 - \Pr(S_n = m) \\ &\leq m(1-\epsilon)^n \;. \end{aligned}$$

Also, by definition of f,

$$\mathcal{E}f(S_n) \;\leq\; 1 \;.$$

Then for any ℓ,

$$\begin{aligned}
\mathcal{E}R &= \sum_{n=0}^{\infty} \mathcal{E}f(S_n) \\
&\leq \sum_{n=0}^{\ell-1} 1 + \sum_{n=\ell}^{\infty} m(1-\epsilon)^n \\
&= \ell + m(1-\epsilon)^\ell \sum_{n=0}^{\infty} (1-\epsilon)^n \\
&= \ell + \frac{m}{\epsilon}(1-\epsilon)^\ell \ .
\end{aligned}$$

Taking

$$\ell = \left\lceil \frac{\log m - \log \epsilon}{-\log(1-\epsilon)} \right\rceil$$

gives the desired bound.

2. Let $a_u = |A_u|$. It will suffice to show that for any subset $\mathcal{B}$ of $\mathcal{Z}_p$ of size $k \leq d$,

$$\Pr(\bigwedge_{u\in\mathcal{B}} x_0 + x_1 u + x_2 u^2 + \cdots + x_{d-1}u^{d-1} \in A_u) = \prod_{u\in\mathcal{B}} \frac{a_u}{p} \ .$$

But

$$\begin{aligned}
&\Pr(\bigwedge_{u\in\mathcal{B}} \sum_{i=0}^{d-1} x_i u^i \in A_u) \\
&= \frac{1}{p^d} \left|\{(x_0, \ldots, x_{d-1}) \mid \bigwedge_{u\in\mathcal{B}} \sum_{i=0}^{d-1} x_i u^i \in A_u\}\right| \\
&= \frac{1}{p^d} \sum_{z_u \in A_u,\ u\in\mathcal{B}} \left|\{(x_0, \ldots, x_{d-1}) \mid \bigwedge_{u\in\mathcal{B}} \sum_{i=0}^{d-1} x_i u^i = z_u\}\right| \ .
\end{aligned}$$

Consider the $k \times d$ linear system

$$x_0 + x_1 u + x_2 u^2 + \cdots + x_{d-1}u^{d-1} = z_u \ , \quad u \in \mathcal{B} \ .$$

This can be represented in matrix form as

$$Ax = z$$

where A is a $k \times d$ submatrix of a $d \times d$ Vandermonde consisting of all rows

$$(1,\ u,\ u^2,\ \ldots,\ u^{d-1}) \ , \qquad u \in \mathcal{B} \ .$$

Since the Vandermonde is nonsingular, A is of full rank k. Its kernel is therefore a subspace of $\mathcal{Z}_p^d$ of dimension $d-k$, thus the affine subspace of

solutions to $Ax = z$ also has dimension $d - k$. In $\mathcal{Z}_p$, any such subspace has p^{d-k} elements. Thus

$$\begin{aligned}
&\frac{1}{p^d} \sum_{z_u \in A_u,\ u \in \mathcal{B}} |\{(x_0, \ldots, x_{d-1}) \mid \bigwedge_{u \in \mathcal{B}} \sum_{i=0}^{d-1} x_i u^i = z_u\}| \\
&\quad = \frac{1}{p^d} \sum_{z_u \in A_u,\ u \in \mathcal{B}} p^{d-k} \\
&\quad = \frac{p^{d-k}}{p^d} \sum_{z_u \in A_u,\ u \in \mathcal{B}} 1 \\
&\quad = \frac{1}{p^k} \prod_{u \in \mathcal{B}} a_u \ .
\end{aligned}$$

3. The solution to this problem is very similar to the analysis of Luby's algorithm given in class. Recall from there that a vertex is *good* if

$$\sum_{u \in N(v)} \frac{1}{d(u)} \ \geq \ \frac{1}{3} \ .$$

Lemma A *For all good v, $\Pr(v \in U) \geq \frac{1}{9}$.*

Proof. If v has a neighbor u of degree 2 or less, then

$$\begin{aligned}
\Pr(v \in U) \ &\geq \ \Pr(v = t(u)) \\
&\geq \ \frac{1}{2} \ .
\end{aligned}$$

Otherwise $d(u) \geq 3$ for all $u \in N(v)$, and as in the analysis of Luby's algorithm, there must exist a subset $M(v) \subseteq N(v)$ such that

$$\frac{1}{3} \ \leq \ \sum_{u \in M(v)} \frac{1}{d(u)} \ \leq \ \frac{2}{3} \ .$$

Then

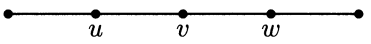
$$\begin{aligned}
&\Pr(v \in U) \\
&\quad \geq \ \Pr(\exists u \in M(v)\ v = t(u)) \\
&\quad \geq \ \sum_{u \in M(v)} \Pr(v = t(u)) - \sum_{\substack{u, w \in M(v) \\ u \neq w}} \Pr(v = t(u) \wedge v = t(w)) \\
&\qquad \text{(by inclusion-exclusion)} \\
&\quad \geq \ \sum_{u \in M(v)} \Pr(v = t(u)) - \sum_{\substack{u, w \in M(v) \\ u \neq w}} \Pr(v = t(u)) \cdot \Pr(v = t(w))
\end{aligned}$$

$$
\begin{aligned}
& \text{(by pairwise independence)} \\
\geq & \sum_{u \in M(v)} \frac{1}{d(u)} - \sum_{u,w \in M(v)} \frac{1}{d(u)} \cdot \frac{1}{d(w)} \\
= & \left(\sum_{u \in M(v)} \frac{1}{d(u)} \right) \cdot \left(1 - \sum_{w \in M(v)} \frac{1}{d(w)} \right) \\
\geq & \frac{1}{3} \cdot \frac{1}{3} = \frac{1}{9} .
\end{aligned}
$$

□

Lemma B *For all v, $\Pr(v \text{ is matched} \mid v \in U) \geq \frac{1}{2}$.*

Proof. There are several cases, depending on the number of H-neighbors of v and the number of H-neighbors of each H-neighbor of v. The situation minimizing the likelihood of v being matched is



There are eight possibilities for the choices of favorites of u, v, w, all equally likely. Of these, four give matchings for v. Thus

$$\Pr(v \text{ is matched} \mid v \in U) \geq \frac{1}{2} .$$

□

Combining Lemmas A and B, the probability that any particular good vertex is matched is at least $\frac{1}{18}$. The remainder of the argument is exactly like the analysis of Luby's algorithm given in class.

Note that the proof of Lemma A required only pairwise independence and the proof of Lemma B required only 3-wise independence, thus using Exercise 2 the algorithm can be made deterministic.

Solutions to Miscellaneous Exercises

1. The set $\mathcal{I}_{\max}$ is closed under subset, so axiom (i) of matroids is satisfied. To show (ii), let $I, J \in \mathcal{I}_{\max}$ such that $|I| < |J|$; we wish to find $x \in J - I$ such that $\{x\} \cup I \in \mathcal{I}_{\max}$.

   Let $\widehat{I}$ and $\widehat{J}$ be elements of M extending I and J, respectively. Consider the set

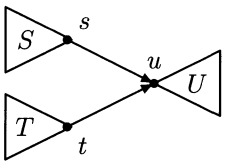
   $$E = (J \cup (S - \widehat{J})) - I .$$

   The cardinality of E is greater than that of $S - \widehat{J}$. By Exercise 1a of Homework 1, E is dependent in the dual, therefore contains a cut C. Now C must intersect every maximal independent set including $\widehat{J}$, so it must contain an element of $J - I$. Let x be such an element of minimal weight.

   We argue now that for any element $y \in C - \widehat{J}$, the weight of y is at least as great as that of some element of $C \cap J$. This will say that x is of minimal weight in C, which will allow us to apply the blue rule. Let D be the fundamental cycle of y and $\widehat{J}$. By Lemma 3.7, $C \cap D$ contains an element z of J, and by Lemma 3.8, the colors of y and z can be exchanged in the coloring $\widehat{J}, S - \widehat{J}$ to obtain an acceptable coloring. But the weight of z cannot exceed that of y, otherwise $\widehat{J}$ was not of minimal weight.

   At this point we have given a cut C disjoint from I such that $C \cap J$ contains an element x of minimal weight among all elements of C. If we color I blue, then we can apply the blue rule with x and C. Since $I \subseteq \widehat{I} \in M$, it follows from Lemma 3.9 that $I \cup \{x\}$ is also contained in an element of M, therefore $I \cup \{x\} \in \mathcal{I}_{\max}$.

2. Determine for each $e \in E$ the sizes of the two connected components obtained by deleting e. This can be done in linear time by depth-first search, computing the values recursively. Orient the edge e in the direction of the smaller component. The resulting directed graph is a directed tree, since no vertex has indegree greater than 1: if (s, u) and (t, u) were both oriented toward u, and if $|S|$, $|T|$, and $|U|$ were the subtrees pictured,

   then

   $$\begin{aligned} |S| &\geq |T| + |U| \\ |T| &\geq |S| + |U| \end{aligned}$$

from which it would follow that $|U| = 0$, a contradiction.

Since we have a directed tree, there is a unique root r. The desired edge is the one from r to its largest subtree. To see this, let A, B, and C be the maximal proper subtrees of r, and let A be the one of maximum size. By the orientation of the edge from r to A,

$$|A| \leq \frac{n}{2} \leq \frac{2n+1}{3} .$$

Now $|B| \leq |A|$ and $|C| \leq |A|$, therefore $|B| + |C| \leq 2|A|$ and

$$\begin{aligned} n-1 &= |A| + |B| + |C| \\ &\geq \frac{|B| + |C|}{2} + |B| + |C| \\ &= \frac{3(|B| + |C|)}{2} . \end{aligned}$$

It follows that

$$|B| + |C| + 1 \leq \frac{2n+1}{3} .$$

3. There are several reasonable approaches to this problem. Here is one involving Kleene algebra. Consider the structure

$$\mathcal{K} = (\mathcal{R} \cup \{\infty, -\infty\},\ \min,\ +,\ ^*,\ \infty,\ 0)$$

where we extend the usual $+$ on $\mathcal{R}$ to $\mathcal{R} \cup \{\infty, -\infty\}$ by

$$\begin{aligned} \infty + a &= a + \infty = \infty, \quad a \in \mathcal{R} \cup \{\infty, -\infty\} \\ -\infty + a &= a + (-\infty) = -\infty, \quad a \in \mathcal{R} \cup \{-\infty\} \end{aligned}$$

and define * by

$$a^* = \begin{cases} 0 , & \text{if } a \geq 0, \\ -\infty , & \text{if } a < 0. \end{cases}$$

A routine check of the axioms verifies that this structure is a Kleene algebra.

Let G be a directed graph with n vertices and real edge weights w, possibly negative. Form the $n \times n$ matrix A with $A_{uv} = w(u, v)$. If there is no edge (u, v), set $A_{uv} = \infty$. As shown in Lecture 7, the family of $n \times n$ matrices over $\mathcal{K}$ is again a Kleene algebra, and we can calculate A^* in time proportional to the time needed to multiply two $n \times n$ matrices over $\mathcal{K}$.

We claim that A^*_{uv} is the weight of the minimum-weight path from u to v, or $-\infty$ if there exists a path from u to v but no path is of minimum weight. As with the min, $+$ algebra discussed in Example 6.6, it can be argued by

induction that A^k_{uv} gives the weight of the minimum-weight path of length k from u to v. Since $A^* = \inf_k A^k$, if there exists a minimum-weight path from u to v, and that path is of length k, then $A^*_{uv} = A^k_{uv}$.

It remains to argue that if there exists a path from u to v but no minimum-weight path, then the weights of paths between u and v are unbounded below, so that $A^*_{uv} = -\infty$. This argument is necessary, since it is conceivable that the weights of the paths from u to v approach some real lower limit without ever achieving it. We show that this cannot happen: if there is a path from u to v but no minimum-weight path, then there is a path from u to v that traverses some cycle of strictly negative weight, which can be traversed arbitrarily many times.

Under our assumption, there exists an infinite sequence $p_0, p_1, \ldots$ of paths from u to v such that p_i is a shortest (in terms of number of edges) path of weight strictly less than that of $p_0, p_1, \ldots, p_{i-1}$. The number of edges in these paths is unbounded, since for each k there are only finitely many paths with k edges. Let p_i be the first path in the list with at least n edges; then some vertex x must be repeated on p_i. The cycle that is traversed between the two occurrences of x on p_i must be of strictly negative weight, otherwise that cycle could be cut out of p_i to give a path of fewer edges and weight strictly less than that of $p_0, p_1, \ldots, p_{i-1}$, contradicting the minimality of p_i.

Another approach to this problem is to identify the vertices contained in negative-weight cycles and treat them separately. The solution to Miscellaneous Exercise 17(a) would presumably be useful in this regard. See [100] for more details about this approach.

4. (a) Call a schedule for a set of jobs A *safe* if no deadlines are violated. If A has a safe schedule, then so does any subset: simply delete the jobs not in the subset.

   Let A and B be two independent sets with $|A| < |B|$. Consider separate safe schedules for A and B. Assume without loss of generality that jobs are scheduled as early as possible with no gaps in the schedules. Let j be the job occurring latest in the schedule for B that is not in A. Let C be the set of jobs occurring after j in the schedule for B. Then $C \subseteq A$. Consider the following schedule for $A \cup \{j\}$: first schedule all the jobs in $A - C$ in the same order as in the schedule for A, then schedule j, then schedule C in the same order as in the schedule for B. This schedule is safe, since all elements of $A - C$ are scheduled no later than they were in A, and all elements in $\{j\} \cup C$ are scheduled no later than they were in B.

   (b) We use the greedy algorithm with the jobs sorted by penalty in decreasing order. The greedy algorithm produces a maximal independent set of maximum weight; these jobs can be scheduled safely. The remaining jobs are all scheduled after their deadlines and incur a

penalty, but this penalty is a minimum. Since all maximal independent sets are of the same cardinality, we can do no better than this.

The sorting phase takes $O(n \log n)$ in general, or linear time if the penalties are small enough that bucket or radix sort can be used. The remainder of the algorithm can be implemented in time $O(n\alpha(n))$, where $\alpha(n)$ is the inverse of Ackermann's function, as follows.

Suppose that we are at some intermediate stage of the algorithm and have selected an independent set of jobs A. For $d \in \mathcal{N}$, define

$$\mu_A(d) = \text{the maximum number of jobs with deadline } d \text{ that could be added to } A \text{ without sacrificing independence.}$$

For example, $\mu_\emptyset(d) = d$. Then for any A,

$$\mu_A(0) = 0 \tag{16}$$
$$\mu_A(d) \leq \mu_A(d+1) \tag{17}$$
$$\leq \mu_A(d) + 1 \; . \tag{18}$$

The inequality (17) holds because if k jobs with deadline d can be safely added to A, then k jobs with any later deadline can be safely added to A. The inequality (18) holds because in any safe schedule for A and k additional jobs of deadline $d+1$, $k-1$ of the additional jobs must finish before time d, thus there is a safe schedule for A and $k-1$ additional jobs of deadline d.

Properties (16), (17), and (18) say that the disjoint sets

$$\mu_A^{-1}(k) = \{d \mid \mu_A(d) = k\}$$

are intervals (contiguous sequences of natural numbers), and if the set $\mu_A^{-1}(k)$ is nonempty, then $\mu_A^{-1}(i)$ is nonempty for any $0 \leq i \leq k$.

We will use the union-find data structure to maintain the disjoint sets $\mu_A^{-1}(k)$. Consider a new job $j \notin A$ with deadline d_j. Then $A \cup \{j\}$ is independent iff $\mu_A(d_j) > 0$ iff $d_j \notin \mu_A^{-1}(0)$. Also, if $A \cup \{j\}$ is independent, then

$$\mu_{A\cup\{j\}}(d) = \begin{cases} \mu_{A\cup\{j\}}(d) - 1, & \mu_A(d) \geq \mu_A(d_j) \\ \mu_{A\cup\{j\}}(d), & \mu_A(d) < \mu_A(d_j) \; , \end{cases}$$

thus

$$\mu_{A\cup\{j\}}^{-1}(k) = \begin{cases} \mu_A^{-1}(k), & k < \mu_A(d_j) - 1 \\ \mu_A^{-1}(k-1), & k > \mu_A(d_j) \\ \mu_A^{-1}(\mu_A(d_j)) \cup \mu_A^{-1}(\mu_A(d_j) - 1), & \text{otherwise.} \end{cases}$$

The sets $\mu_A^{-1}(k)$ will be linked in a list ℓ in order of increasing k. Initially, $\mu_A^{-1}(k) = \{k\}$, $0 \leq k \leq n$. To test for independence of $A \cup \{j\}$, where A is independent, we ask whether the set $\mathbf{find}(d_j)$ has a predecessor on the list ℓ. If not, $\mu_A(d_j) = \mu_A(0) = 0$ so $A \cup \{j\}$ is not independent. To insert the element j, we form the union of the set $\mathbf{find}(d_j)$ and its predecessor on the list ℓ.

5. The problem is in *coNP*, since its complement is in *NP*: we can guess a string of length n and check in polynomial time that it is not covered by any of the patterns.

   The problem is also hard for *coNP*, as the following reduction from (the complement of) CNFSat shows. Suppose we have a CNF formula $\mathcal{B}$ with n Boolean variables $x_1, \ldots, x_n$. Convert each clause c to a pattern σ_c of length n as follows:

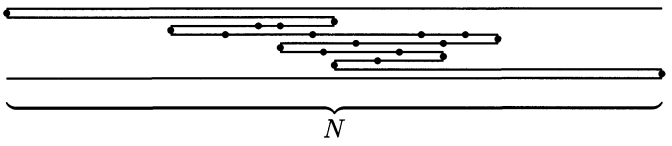
   - if x_i does not appear in c, put * in position i of σ_c.
   - if x_i appears positively in c, put a 0 in position i of σ_c.
   - if x_i appears negatively in c, put a 1 in position i of σ_c.

   A string of length n over $\{0,1\}$ represents a truth assignment to the variables $x_1, \ldots, x_n$ by assigning *true* to x_i if 1 appears in position i in the string, *false* if 0 appears in position i. Then σ_c covers exactly those strings corresponding to truth assignments that do not satisfy c. Therefore every string is covered by some σ_c iff every truth assignment falsifies some clause, *i.e.* iff $\mathcal{B}$ is unsatisfiable.

6. The problem is in *NP*, because we can guess a folding and compute its length in polynomial time. To show *NP*-hardness, we reduce the partition problem to it. Given an instance of the partition problem consisting of the weight function $w : \{1, 2, \ldots, n\} \to \mathcal{N}$, construct a ruler with $n + 4$ segments of length (in order)

$$N,\ \frac{N}{2},\ w(1),\ w(2),\ \ldots,\ w(n),\ \frac{N}{2},\ N$$

   where N is very large (actually $N \geq \sum_{i=1}^{n} w(i)$ suffices), and let $k = N$. In order to fit, the endpoints of the two end segments of length N must line up vertically, and the two segments next to them of length $\frac{N}{2}$ must be folded back in. Thus we will get a fit if and only if the remainder of the ruler can be folded so that the inner endpoints of the $\frac{N}{2}$ segments line up vertically.



This can occur if and only if there exists a subset $S \subseteq \{1, 2, \ldots, n\}$ such that

$$\sum_{i \in S} w(i) \;=\; \sum_{i \notin S} w(i) \;;$$

the sets S and $\{1, 2, \ldots, n\} - S$ correspond to the segments in the ruler pointing left and right, respectively.

7. Given a Boolean formula in CNF, perform the following operation on long clauses until every clause has at most three literals. First replace the clause

$$(x_1 \vee x_2 \vee x_3 \vee \cdots \vee x_k) \tag{19}$$

with

$$(y \leftrightarrow (x_1 \vee x_2)) \;\wedge\; (y \vee x_3 \vee \cdots \vee x_k) \tag{20}$$

where y is a new variable. The two formulas are equisatisfiable, and the number of satisfying truth assignments is the same, because the truth assignment to y is forced by the truth assignment to x_1 and x_2. The rightmost clause of (20) has one fewer literal than (19).

Next, replace the new clause

$$(y \leftrightarrow (x_1 \vee x_2))$$

with the equivalent formula

$$(\overline{y} \vee x_1 \vee x_2) \;\wedge\; (\overline{x}_1 \vee y) \;\wedge\; (\overline{x}_2 \vee y) \;.$$

This procedure gives a formula with at most three literals per clause.

It is possible to get exactly three distinct literals per clause as follows. Let x, y, and z be three new variables. Replace each deficient clause $(u \vee v)$ with $(u \vee v \vee \overline{x})$ and each deficient clause (u) with $(u \vee \overline{x} \vee \overline{y})$, and add the clauses (x), (y) and (z). The number of satisfying assignments is the same, since the new variables have to be *true* in any satisfying assignment. All clauses have exactly three literals except the three new ones. It now suffices to show how to express the conjunction $x\,y\,z$ in 3CNF. But

$$\neg(x\,y\,z) \;\leftrightarrow\; x\,y\,\overline{z} \vee x\,\overline{y}\,z \vee x\,\overline{y}\,\overline{z} \vee \overline{x}\,y\,z \vee \overline{x}\,y\,\overline{z} \vee \overline{x}\,\overline{y}\,z \vee \overline{x}\,\overline{y}\,\overline{z} \;,$$

and a 3CNF representation of $x\,y\,z$ can be obtained by negating the right hand side and applying DeMorgan's law.

8. In the new representation, there will be a doubly linked list V of vertices. By "Given u,..." we will mean, "Given a pointer to the list element for vertex u on the list V,..." Each vertex u will point to a circular linked

list $\mathbf{adj}(u)$ of edges e such that $\mathbf{t}(e) = u$. By "Given e,..." we will mean, "Given the list element for e on the list $\mathbf{adj}(\mathbf{t}(e))$,..." The order on the circular list $\mathbf{adj}(u)$ will give the order θ. The edge e will point to $\theta(e)$, $\mathbf{t}(e)$ and $\overline{e}$; that is to say, the list element for e on $\mathbf{adj}(\mathbf{t}(e))$ will contain pointers to the list element for $\theta(e)$ on $\mathbf{adj}(\mathbf{t}(e))$, the list element for $\mathbf{t}(e)$ on V, and the list element for $\overline{e}$ on $\mathbf{adj}(\mathbf{t}(\overline{e})) = \mathbf{adj}(\mathbf{h}(e))$.

Given e, in constant time we can compute $\theta(e)$, $\overline{e}$ or $\mathbf{t}(e)$ by following a single pointer; and we can compute $\mathbf{h}(e)$ by following a pointer to $\overline{e}$ and then following the pointer there to $\mathbf{t}(\overline{e}) = \mathbf{h}(e)$. We can delete a given edge e in constant time by unlinking its list element from $\mathbf{adj}(\mathbf{t}(e))$ and unlinking the list element of $\overline{e}$ from $\mathbf{adj}(\mathbf{h}(e))$. We can delete a vertex u by first deleting all the edges on $\mathbf{adj}(u)$ as above and then unlinking u from V.

We now show how to obtain the new representation from the old one in linear time. The old representation consists essentially of the list V, for each u on V a pointer to an adjacency list $\mathbf{adj}(u)$ containing all edges e with $\mathbf{t}(e) = u$, and for each e on $\mathbf{adj}(\mathbf{t}(e))$ a pointer to $\mathbf{h}(e)$ on V. To calculate the pointers from e to $\mathbf{t}(e)$ in linear time, for each u scan the list $\mathbf{adj}(u)$ and append to each list element a pointer back to u.

It remains to calculate the pointers from e to $\overline{e}$. First produce for each vertex u an auxiliary adjacency list $\mathbf{aux}(u)$ with a list element for each e with $\mathbf{t}(e) = u$ and a pointer to $\overline{e}$. The lists $\mathbf{aux}(u)$ will contain the desired pointers but will not be in the correct order. The lists $\mathbf{aux}(u)$ are produced in linear time by initializing all $\mathbf{aux}(u)$ to the empty list, then scanning through all the lists $\mathbf{adj}(u)$ and for each e encountered appending a pointer to e to the front of the list $\mathbf{aux}(\mathbf{h}(e))$.

At this point each u points to two lists, $\mathbf{adj}(u)$ and $\mathbf{aux}(u)$, each with one entry for each edge e with $\mathbf{t}(e) = u$. The former list is in the correct order θ and the latter contains the $^{-}$ pointers. We wish to consolidate them into a single list with both properties. For each u, execute the following two steps:

(a) Scan the list $\mathbf{aux}(u)$. For each e on the list, save the pointer to $\overline{e}$ in the list element for $\mathbf{h}(e)$ on V. The pointer to $\mathbf{h}(e)$ is available as $\mathbf{t}(\overline{e})$.

(b) Scan the list $\mathbf{adj}(u)$. For each e on the list, pick up the pointer to $\overline{e}$ from the list element for $\mathbf{h}(e)$ on V.

This procedure takes linear time, since each vertex and edge is visited a constant number of times.

9. Since $^{-}$ and θ can be computed in constant time, the permutation $\theta^* = \theta \circ {}^{-}$ can be computed in constant time. Construct a doubly linked list L of pointers to the list elements of all (directed) edges in the adjacency list

representation of G. While constructing L, store with each list element in G corresponding to an edge e a pointer to the list element in L corresponding to e. Repeat the following loop until L is empty. Let e be an arbitrary edge on L. Starting from e, successively compute θ^*, deleting each edge in succession from L and inserting it on a circular list C. Stop when we get back to e. Then C contains all edges in the cycle of θ^* containing e. Create a new vertex u in V^*, set $\mathbf{t}^*(e) = u$ for each edge e in C, and make C the adjacency list of u. When L is empty, set $\mathbf{h}^*(e) = \mathbf{t}^*(\bar{e})$ for each edge e. The pointers giving $^-$ are available from G.

10. As in Lecture 14, let n and n^* be the number of vertices and faces of G, respectively; let $m = |E|/2$, the number of undirected edges; let c be the number of connected components (orbits of E under the subgroup generated by θ and $^-$); and let $\theta^* = \theta \circ {}^-$. Note that the subgroup generated by θ and $^-$ is the same as the subgroup generated by θ and θ^*, since $^- = \theta^{-1} \circ \theta^*$.

    Write θ as a product of disjoint cycles. Multiplying θ on the left by an involution $(a\ b)$ acts on θ as follows: if a and b are in different cycles of θ, then those two cycles are merged, and if a and b are in the same cycle, then that cycle is split. All cycles of θ not containing a or b are left intact. For example,

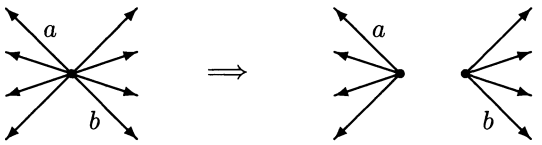
$$\begin{aligned}(1\ 5) \circ (1\ 2\ 3\ 4) \circ (5\ 6\ 7\ 8) &= (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8)\\ (1\ 5) \circ (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8) &= (1\ 2\ 3\ 4) \circ (5\ 6\ 7\ 8)\ .\end{aligned}$$

    Here $(a_0\ a_1\ \ldots\ a_{n-1})$ is the permutation that maps a_i to $a_{(i+1) \bmod n}$, $0 \le i < n$, and fixes all other elements of E.

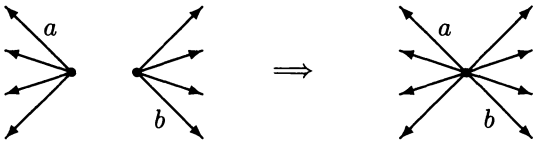
    Let $\theta' = (a\ b) \circ \theta$. In terms of the graphs

$$\begin{aligned}G &= (E,\ \theta,\ {}^-)\\ G' &= (E,\ \theta',\ {}^-)\ ,\end{aligned}$$

    if the edges a and b have a common tail in G (*i.e.*, if a and b are in the same cycle of θ), then that tail is split into two vertices in G' as shown.



    If a and b have different tails (*i.e.*, if a and b are contained in different cycles of θ), then the tails are merged in G'.



Note that

$$(\theta')^* \;=\; (\theta^*)' \;=\; (a\ b)\circ\theta^*\ ,$$

so that these comments apply to the duals G^* and G'^* as well.

Write θ as a product of disjoint cycles

$$\theta \;=\; \theta_0\circ\theta_1\circ\cdots\circ\theta_{p-1}\ .$$

Each θ_i can be expressed as a product of $|\theta_i|-1$ transpositions:

$$(a_0\ a_1\ \ldots\ a_{k-1}) \;=\; (a_0\ a_1)\circ(a_1\ a_2)\circ\cdots\circ(a_{k-2}\ a_{k-1})\ .$$

Thus θ can be expressed as a product of

$$\sum_{i=0}^{p-1}(|\theta_i|-1) \;=\; 2m-p$$

transpositions. Let σ_i denote the product of the rightmost i transpositions in this list, and let G_i be the graph $(E,\ \sigma_i,\ {}^-)$. Then σ_0 is the identity, which has $2m$ singleton cycles, and $\sigma_{2m-p} = \theta$, which has p cycles, so multiplication by the i^{th} transposition taking σ_i to σ_{i+1} must combine two cycles into one. Thus G_{i+1} has one fewer vertex than G_i; *i.e.*, n decreases by one in each step.

Let the i^{th} transposition be $(a\ b)$. Either

(i) a and b belong to different connected components of G_i;

(ii) a and b belong to the same component but different cycles of σ_i^*; or

(iii) a and b belong to the same cycle of σ_i^*.

In case (i), a and b belong to different cycles of σ_i^*, so c and n^* each decrease by one and $\chi(G_{i+1}) = \chi(G_i)$. In case (ii), n^* decreases by one and c remains the same, in which case $\chi(G_{i+1}) = \chi(G_i)+2$. In case (iii), a and b belong to the same component of G_i and n^* increases by one. In this case, we claim that c remains the same, thus $\chi(G_{i+1}) = \chi(G_i)$. To see that c does not increase, it suffices to show that for any $d\in E$ there exists an η in the subgroup generated by σ_{i+1} and ${}^-$ such that $\eta(d) = \sigma_i(d)$; thus each orbit of the subgroup generated by σ_i and ${}^-$ is contained in an orbit of the subgroup generated by σ_{i+1} and ${}^-$.

If $\sigma_i(d) \notin \{a, b\}$, then

$$\begin{aligned} \sigma_{i+1}(d) &= (a\ b) \circ \sigma_i(d) \\ &= \sigma_i(d)\ , \end{aligned}$$

so in this case we can take $\eta = \sigma_{i+1}$. Suppose now that $\sigma_i(d) = a$; the case $\sigma_i(d) = b$ is symmetric. Since a and b are in the same cycle of σ_{i+1}, there exists a k such that $\sigma_{i+1}^k(b) = a$. Let $\eta = \sigma_{i+1}^{k+1}$. Then

$$\begin{aligned} \eta(d) &= \sigma_{i+1}^k(\sigma_{i+1}(d)) \\ &= \sigma_{i+1}^k((a\ b)\ (\sigma_i(d))) \\ &= \sigma_{i+1}^k(b) \\ &= a \\ &= \sigma_i(d)\ . \end{aligned}$$

The reverse inclusion, which implies that c does not decrease, follows from a dual argument. (It is the dual argument that uses assumption (iii).)

11. We prove the harder direction ($\rightarrow$); the direction ($\leftarrow$) follows from a straightforward induction on the number of faces.

Let σ_i and G_i be as in the solution to Miscellaneous Exercise 10. In that solution it was argued that $\chi(G_{i+1}) \geq \chi(G_i) \geq 0$ for all i. Since $G_{2m-p} = G$ and $\chi(G) = 0$, we have $\chi(G_i) = 0$ for all i.

We proceed by induction on i. For the basis, σ_0 is the identity map and G_0 is the graph consisting of m connected components, each consisting of two vertices and an undirected edge. This certainly has a plane embedding consistent with σ_0.

Now suppose G_i has a plane embedding consistent with σ_i. Let $(a\ b)$ be the i^{th} transposition, so that $\sigma_{i+1} = (a\ b) \circ \sigma_i$. The tails of a and b in G_i are merged in G_{i+1}. Either

(i) a and b lie in the same face of G_i, or

(ii) a and b lie in different faces of G_i.

In case (i), the tails of a and b can be merged across the common face of a and b. It is easily checked that the resulting embedding of G_{i+1} is consistent with σ_{i+1}.

In case (ii), both n and n^* decrease by one in passing from G_i to G_{i+1}, so c must also decrease by one to maintain $\chi(G_{i+1}) = 0$. Thus a and b are in different connected components of G_i. By embedding on the sphere and changing the position of the North Pole, we can arrange the embedding of G_i so that a and b occur on the outer faces of their components, then merge the tails of a and b to obtain a plane embedding consistent with σ_{i+1}.

12. Instead of maximal cliques, take the complement graph and consider the maximal independent sets. Given a Boolean formula, construct a graph with a black vertex for each literal and a red vertex for each clause. Connect complementary literals and connect each clause to the literals it contains. Any maximal independent set M containing no red vertex contains exactly one of each pair of complementary literals and thus gives a truth assignment; moreover, it is a satisfying assignment, since every red vertex is connected to a vertex in M (otherwise M would not be maximal). Conversely, any satisfying assignment gives a black maximal independent set.

13. Given a partition problem (S, w), where S is the set of items, $n = |S|$, and $w : S \to \mathcal{Z}$ the weight function, let ℓ be an upper bound on the absolute values of weights of elements of S. Then for any subset $R \subseteq S$,

$$|w(R)| \;=\; \sum_{a \in R} |w(a)| \;\leq\; \ell |R| \; .$$

Thus, although there are $2^{|R|}$ subsets of R, there are only $2\ell|R|+1$ weights these subsets can take on.

The following algorithm is quite reasonable if ℓ is sufficiently small. We calculate recursively the set of all possible weights of subsets of $R \subseteq S$. If $R = \{a\}$, the possible weights are 0 and $w(a)$. If $|R| \geq 2$, partition R into two disjoint sets R_1, R_2 of roughly equal size and recursively produce the data for R_1 and R_2. For each pair of weights w_1 and w_2 in the sets computed for R_1 and R_2, respectively, calculate $w_1 + w_2$ and record it in the set for R. Since w_1 is the weight of some subset of R_1 and w_2 is the weight of some subset of R_2, $w_1 + w_2$ is the weight of their union, which is a subset of R; moreover, every subset of R is obtained as such a union.

Since the weights of subsets of R_i lie between $-\ell|R_i|$ and $\ell|R_i|$, the calculation of all the weights $w_1 + w_2$ requires at most $O(\ell|R_1| \cdot \ell|R_2|)$ operations. This gives rise to the recurrence

$$T(n) \;=\; 2T(\frac{n}{2}) + O(\ell^2 n^2)$$

with solution

$$T(n) \;=\; O(\ell^2 n^2) \; .$$

Thus for partition problems with $\ell = \ell(n)$ a polynomial in n, $T(n)$ is also a polynomial in n.

14. Try $k = 2^{\lceil \log n \rceil}$, the smallest power of 2 greater than or equal to n. Recall that in the representation over $\mathcal{Z}_2[x]/x^n$, the binary-to-Gray operation amounts to multiplication by $x + 1$. We wish to show that $(x+1)^k = 1$

and for all $0 \leq \ell < k$, $(x+1)^\ell \neq 1$; in other words, the order of $x+1$ in the group of invertible elements of $\mathcal{Z}_2[x]/x^n$ is k. This will suffice to prove our result, since for any p, applying the binary-to-Gray operation k times gives $(x+1)^k p = p$, and for any $0 < \ell < k$, $(x+1)^\ell (x+1)^{k-\ell} = 1 \neq (x+1)^{k-\ell}$.

To show that $(x+1)^k = 1$, note that in $\mathcal{Z}_2[x]/x^n$, squaring is a linear operation:

$$\begin{aligned}(p+q)^2 &= p^2 + 2pq + q^2 \\ &= p^2 + q^2 \,,\end{aligned}$$

since $2 = 0 \pmod 2$. Thus

$$\begin{aligned}(x+1)^k &= (x+1)^{2^{\lceil \log n \rceil}} \\ &= (\cdots((x+1)^2)^2 \cdots)^2 \\ &= x^{2^{\lceil \log n \rceil}} + 1 \\ &= 1 \,,\end{aligned}$$

since $2^{\lceil \log n \rceil} \geq n$ and therefore $x^{2^{\lceil \log n \rceil}} = 0 \bmod x^n$. To show that no smaller power works, we use the fact that the order of $x+1$ must divide k, therefore it must be a power of 2. But for $m < \lceil \log n \rceil$ we have $2^m < n$, therefore

$$(x+1)^{2^m} = x^{2^m} + 1 \neq 1 \,.$$

15. (a) Consider the weighted 4-node widget A we used in Lecture 27 with the following adjacency matrix:

$$\begin{bmatrix} 1 & 1 & -1 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 \end{bmatrix}$$

The input nodes of the widget are 1 and 3 and the output nodes are 2 and 4. Note that $A_{34} = 1$ and everything else in the row and column containing A_{34} is 0. This says that every cycle cover of nonzero weight must contain the edge from 3 to 4 of weight 1. We might just as well collapse 3 and 4 into one node. The resulting widget is

$$\begin{bmatrix} 1 & 1 & -1 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ 1 & -1 & 0 \end{bmatrix} \tag{21}$$

with input nodes 1 and 3 and output nodes 2 and 3. The desired behavior is summarized by the following equations, analogous to those presented in the lecture for the four-node widget. As before, $A(i;j)$

denotes the submatrix of A obtained by deleting row(s) i and column(s) j.

$$\text{perm } A(3;3) = \text{perm } A(2;1) = \text{perm } A(2,3;1,3) = 1$$
$$\text{perm } A(3;1) = \text{perm } A(2;3) = \text{perm } A = 0 \ .$$

A quick calculation shows that all these properties are satisfied by (21).

(b) To be used in this construction, any widget must have two distinct input vertices and two distinct output vertices, because there are good cycle covers containing two input edges and two output edges. Thus in any two-node widget, both nodes are input nodes and both are output nodes. The relevant equations are then

$$\text{perm } A(1;1) = \text{perm } A(2;2) = \text{perm } A(1,2;1,2) = 1$$
$$\text{perm } A(2;1) = \text{perm } A(1;2) = \text{perm } A = 0 \ .$$

The first line implies that $A_{11} = A_{22} = 1$. The first two equations of the second line imply that $A_{12} = A_{21} = 0$. Thus A must be the identity matrix. But then perm $A = 1$, so the last equation is violated.

16. (a) We first apply depth- or breadth-first search to determine whether the graph is 2-colorable (say with cyan and magenta) and find the connected components. If it is not 2-colorable, then no schedule exists. If it is, then we must decide for each connected component whether to schedule the cyan vertices of that component at time 0 and the magenta vertices at time 1 or vice versa so that there are at most m vertices in all scheduled at any one time. This is a partition problem with small weights and can be solved in polynomial time by the method of Miscellaneous Exercise 13.

(b) The problem is equivalent to the following graph coloring problem: given an undirected graph, can the vertices be colored with t colors so that no two adjacent vertices receive the same color and each monochromatic set is of cardinality at most m? The ordinary k-colorability problem discussed in Lecture 21 reduces to this problem trivially by taking $m = n$. Therefore the problem is NP-hard. It is also easily seen to be in NP, since a solution can be guessed and verified in polynomial time.

17. (a) The problem of detecting negative cycles is made easier by the following lemma.

Lemma *A vertex u is contained in a negative-weight cycle iff some vertex in the strong component of u is contained in a simple negative-weight cycle.*

Proof.

($\rightarrow$) If the negative-weight cycle containing u is simple, we are done. Otherwise there is a repeated vertex x on the cycle, and x is in the strong component of u. The cycle can be decomposed into two strictly smaller cycles containing x, at least one of which must have negative weight. By induction, there is a simple negative-weight cycle containing an element of the strong component of x, which is also the strong component of u.

($\leftarrow$) Let x be a vertex in the strong component of u contained in a simple negative-weight cycle. Combining the cycle through u and x with sufficiently many trips around the negative-weight cycle through x, we obtain a negative-weight cycle through u. □

Every simple cycle of negative weight is of length at most n. Thus we can test for the existence of such a cycle by testing for the existence of any negative-weight cycle of length at most n. We can do this in polynomial time by examining the diagonal elements of $(I + C)^n$, where C is the weighted adjacency matrix and I is the identity matrix over the min,+ Kleene algebra.

(b) The problem is in *NP*, since we can guess a simple path and verify that it has negative weight in polynomial time.

To show *NP*-hardness, we reduce the Hamiltonian circuit problem to this problem. Given a graph $G = (V, E)$ with $|V| = n$, pick a vertex u and split it into two vertices v, t with no edge between. Each of v, t is connected to the other vertices of G in the same way u was. Let s be a new vertex and add an edge of weight $n - 1$ from s to v. Weight the remaining edges -1. The new graph has $n + 2$ vertices. Any path from s to t must begin with the edge from s to v of weight $n - 1$, therefore must traverse at least n other edges in order to be of negative weight. In order to be simple, it can traverse at most n other edges. Thus a simple negative weight path from s to t is of length exactly $n+1$ and contains a simple path from v to t of length n. This gives rise to a Hamiltonian circuit in the original graph.

18. (a) Every maximal matching M can be extended to an edge cover C by adding an extra edge for each unmatched vertex. Each such extra edge must connect the unmatched vertex to a matched one, otherwise M was not maximal. Thus $|C| + |M| = |V|$.

Conversely, every minimal edge cover C contains a matching M by choosing one edge from each connected component of C. Each component of C consists of a vertex with edges radiating out from it; if a component had more than one vertex of degree greater than one, we could remove an edge and still have a cover. The number of components of C is $|M|$, and again, $|C| + |M| = |V|$.

We have shown that for every maximal M there is a C with $|C|+|M| = |V|$ and for every minimal C there is an M with $|C|+|M| = |V|$. Thus an edge cover of minimum cardinality can be obtained by extending a matching of maximum cardinality as described above.

(b) Find a maximum matching and take the vertex cover to be the set of matched vertices. This is a vertex cover, since any edge not covered could have been added to the matching. It is at most twice the size of a minimum vertex cover, since every vertex cover must contain at least one endpoint of each matched edge.

These algorithms can be implemented in time $O(m\sqrt{n})$ using the algorithm of Micali and Vazirani [80, 105].

19. As shown in Lecture 26, G has a cycle cover iff the permanent of its adjacency matrix is nonzero. Let G' be a bipartite graph whose bipartite adjacency matrix is the same as the (nonbipartite) adjacency matrix of G. Then G has a cycle cover iff G' has a perfect matching, so we can use any one of the fast algorithms given in class for maximum matching.

    We can also go directly from an adjacency list representation of G to an adjacency list representation of G' in linear time without constructing the common adjacency matrix: duplicate the entire adjacency list representation of G and make the pointers in the edge lists in each copy to point to the other copy.

20. First find a maximum matching M in time $O(m\sqrt{n})$ using the Hopcroft and Karp algorithm. If there is a deficient set, then Hall's Theorem says that there is at least one free vertex $x \in U$. Starting from x, build a Hungarian tree of all vertices reachable from x along alternating paths. Cut each path off when a vertex that has been seen before is encountered. Let R be the set of vertices reachable from x in the Hungarian tree. Let $D = U \cap R$. We will show that D is a minimal deficient set.

    First we claim that every element of R besides x is matched. There are certainly no free vertices in $R \cap U$ except x, since each such vertex is first seen immediately after traversing an edge in M. If there were a free vertex v in $R \cap V$, then the path from v back to x would be an augmenting path, contradicting the maximality of M. Moreover, for every matched vertex in R, its mate is also in R. Thus $|N(D)| = |D| - 1$ and D is deficient.

    To show that D is minimal, we show that for any element $y \in D$, there is a matching in which all elements of D except y are matched; thus no proper subset of D is deficient. If $y = x$, we can take the matching M. Otherwise, y is matched in M. Change the status of all edges on the alternating path from y back to x through the Hungarian tree. This unmatches y and matches x. All other elements of D remain matched.

21. First we extend Menger's Theorem slightly.

Lemma *Let S be any set of k vertices of a connected undirected graph $G = (V, E)$. The following are equivalent:*

(i) G is k-connected;

(ii) any pair of vertices is connected by at least k vertex-disjoint paths;

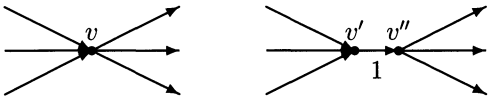
(iii) for any vertex $s \in S$ and any vertex $v \in V$, s and v are connected by at least k vertex-disjoint paths.

Proof. The implication (i) $\rightarrow$ (ii) is given by Menger's Theorem and (ii) $\rightarrow$ (iii) is trivial.

To show (iii) $\rightarrow$ (i), let U be any set of $k-1$ vertices. At least one element $s \in S$ is not in U, and for any other vertex v not in U, by (iii) there is a path from s to v avoiding U. Thus the removal of U does not disconnect G. □

We will show how to test whether a given s and t are connected by k vertex-disjoint paths in time $O(km)$. By the lemma, we only need to do this for kn pairs of vertices. This gives an overall time bound of $O(k^2mn)$.

To test whether s and t are connected by k vertex-disjoint paths, we adapt the max flow algorithm of Edmonds and Karp. First we replace each edge with two directed edges, one in each direction, then give them all capacity 1. Then we split each vertex v into two vertices v' and v'' with an edge from v' to v'' of capacity 1. All edges that were directed into v are directed into v' and all edges that were directed out of v are directed out of v''.



before after

The value of an integral max flow from s to t in this graph will be the maximum number of vertex-disjoint paths from s to t. This follows from the Max Flow-Min Cut Theorem and the splitting of vertices that prevents more than one unit of flow to pass through any interior vertex.

However, we do not need to find a max flow, but only to check whether it is at least k. This can be done by performing k augmenting steps of the Edmonds-Karp algorithm, calculating new level graphs as necessary. The time to calculate all the level graphs is at most $O(km)$ and the time to do all the augmenting steps is at most $O(km)$.

22. Let e_i be the row vector with a 1 in position i and 0 elsewhere. For $1 \le i \le k$, the i^{th} row of C_p is the first row of C_p^i; that is,

$$e_i C_p = e_1 C_p^i .$$

Then for any $m \ge 1$, the i^{th} row of C_p^m is the first row of C_p^{m+i-1}:

$$\begin{aligned} e_i C_p^m &= e_i C_p C_p^{m-1} \\ &= e_1 C_p^i C_p^{m-1} \\ &= e_1 C_p^{m+i-1} . \end{aligned}$$

For any row vector $x = (x_1, \ldots, x_k)$, xC_p can be computed in time $O(k)$:

$$xC_p = (-a_0 x_k, x_1 - a_1 x_k, x_2 - a_2 x_k, \ldots, x_{k-1} - a_{k-1} x_k) . \quad (22)$$

Thus we can compute any product AC_p in time $O(k^2)$. Now given a power C_p^m of C_p, we can square it in time $O(k^2)$ as follows. First compute the row vector

$$e_1 C_p^{2m} = (e_1 C_p^m) C_p^m .$$

This takes time $O(k^2)$ and gives the first row of C_p^{2m}. Compute the remaining rows of C_p^{2m} by successively multiplying by C_p on the right. This takes time $O(k^2)$ by (22). We can compute C_p^n with at most $\log n$ operations of squaring and multiplying by C_p; the sequence of operations is determined by the binary representation of n.

23. Let $r = \operatorname{rank} A$ and let A' be an $n \times r$ submatrix of A of full rank r. Let B be an $n \times (n-r)$ matrix spanning $\ker A$. As argued in Lecture 40, $\operatorname{rank} (RA)^2 = \operatorname{rank} RA = \operatorname{rank} A$ iff the matrix $[RA'|B]$ formed by juxtaposing RA' and B is nonsingular. Thus

$$\Pr(\operatorname{rank} (RA)^2 = \operatorname{rank} RA = \operatorname{rank} A) = \sum_{S \in \mathcal{S}} \Pr(RA' = S) ,$$

where the sum is over the set $\mathcal{S}$ of all $n \times r$ matrices of full rank r such that $[S|B]$ is nonsingular.

We claim that RA' is a random[1] $n \times r$ matrix. To see this, let D be any nonsingular $n \times n$ matrix agreeing with A on the columns A'. Then

$$RA' = (RD)' ,$$

where $(RD)'$ denotes the $n \times r$ submatrix of RD obtained by deleting the same columns that were deleted from A to get A'. Since the map $R \mapsto RD$

[1] A *random matrix* is a matrix with each entry chosen independently and uniformly at random from the field. If the matrix is $k \times m$, then all q^{km} possible matrices are equally likely.

is a bijection and R is a random matrix, so are RD and $(RD)'$. Therefore for any $n \times r$ matrix S,

$$\Pr(RA' = S) \;=\; \Pr((RD)' = S) \;=\; \frac{1}{q^{nr}} \; .$$

We now calculate the size of $\mathcal{S}$. Note that a subspace of k^n dimension m has q^m elements, since every element is a linear combination of m basis elements, and there are q^m ways to choose the coefficients of the linear combination. We wish to count the number of ways of choosing an $n \times r$ matrix S of full rank whose columns avoid the subspace of dimension $n-r$ spanned by the columns of B. There are $q^n - q^{n-r}$ ways to choose the first column of S to avoid this subspace; once that column is chosen, there are $q^n - q^{n-r+1}$ ways of choosing the second column to avoid the linear span of B and the first column already chosen; and so on. After all but one column are chosen, there are $q^n - q^{n-1}$ ways to choose the last column. Thus there are in all

$$(q^n - q^{n-r}) \cdot (q^n - q^{n-r+1}) \cdots (q^n - q^{n-1}) \;=\; \prod_{i=1}^{r} (q^n - q^{n-i})$$

ways of choosing S, and this number is $|\mathcal{S}|$. Combining all the above equations, we have

$$\begin{aligned} &\Pr(\text{rank } (RA)^2 = \text{rank } RA = \text{rank } A) \\ &= \sum_{S \in \mathcal{S}} \Pr(RA' = S) \\ &= \frac{|\mathcal{S}|}{q^{nr}} \\ &= \prod_{i=1}^{r} (1 - \frac{1}{q^i}) \; . \end{aligned}$$

24. (a) To show that the problem is *coNP*-hard, we give a reduction from the pattern covering problem of Miscellaneous Exercise 5. Let P be a given set of patterns $p \in \{0, 1, *\}^n$. We show how to construct in polynomial time a set of terms T such that T is a cover of the ground terms over $\{f, a, b\}$ iff P is a cover of $\{0,1\}^n$.

Let

$$F_n(x_1, \ldots, x_n) \;=\; f(x_1, f(x_2, \cdots f(x_{n-1}, x_n) \cdots)) \; .$$

Let $0' = a$ and $1' = b$. We will encode a string $d_1 d_2 \cdots d_n \in \{0,1\}^n$ by the term $F_n(d'_1, \ldots, d'_n)$. Let S be the set of all terms of this form; *i.e.*,

$$S \;=\; \{F_n(d'_1, \ldots, d'_n) \mid d_1 d_2 \cdots d_n \in \{0,1\}^n\} \; .$$

We will take $T = T_0 \cup T_1$, where T_0 will cover exactly the terms not in S, and T_1 will cover all terms in S iff P is a cover of $\{0,1\}^n$. The set T_0 consists of all terms of the form

- $F_i(x_1, \ldots, x_{i-1}, a)$, $1 \leq i \leq n-1$
- $F_i(x_1, \ldots, x_{i-1}, b)$, $1 \leq i \leq n-1$
- $F_n(x_1, \ldots, x_{i-1}, f(y,z), x_{i+1}, \ldots, x_n)$, $1 \leq i \leq n$.

The terms (a) and (b) cover all ground terms not covered by F_n, and the terms (c) cover all ground terms covered by F_n that are not in S. Now, for each pattern $p = p_1 p_2 \cdots p_n \in \{0,1,*\}^n$, let t_p be the term $F_n(p'_1, \ldots, p'_n)$, where

$$p'_i = \begin{cases} a, & \text{if } p_i = 0, \\ b, & \text{if } p_i = 1, \\ x_i, & \text{if } p_i = * \, . \end{cases}$$

The set of ground terms in S covered by $F_n(p'_1, \ldots, p'_n)$ is exactly the set of all ground terms $F_n(d'_1, \ldots, d'_n)$ such that $d_1 d_2 \cdots d_n \in \{0,1\}^n$ is covered by p. We therefore take

$$T_1 = \{t_p \mid p \in P\} \, .$$

(b) We would like to guess a ground term and verify in polynomial time that it is not covered by T. The major difficulty here is that there are infinitely many ground terms. We need to know that if there exists a ground term that is not covered by T, then there is one of polynomial size. This will allow us to guess that term by a sequence of polynomially many nondeterministic binary choices.

For convenience, we view terms as functions labeling the nodes of the infinite binary tree with f, a, b, variables in X, or $\perp$. Formally, a term t is a map

$$t : \{L,R\}^* \rightarrow \{f,a,b\} \cup X \cup \{\perp\}$$

such that for all $\alpha \in \{L,R\}^*$, the following are equivalent:

- $t(\alpha) = f$
- $t(\alpha L) \neq \perp$
- $t(\alpha R) \neq \perp$

and such that $t(\alpha) \neq \perp$ for at most finitely many α. For example, the term $f(f(a,b),b)$ is represented by the map

$$\begin{aligned} \epsilon &\mapsto f \\ L &\mapsto f \\ LL &\mapsto a \\ LR &\mapsto b \\ R &\mapsto b \\ \text{everything else} &\mapsto \perp \, . \end{aligned}$$

Let t/α denote the subterm of t rooted at α:

$$t/\alpha(\beta) \;=\; t(\alpha\beta)\ .$$

Now let T be the given set of terms. Let

$$\begin{aligned} A_t &= \{\alpha \mid t(\alpha) \neq \bot\} \\ A &= \bigcup_{t\in T} A_t \\ \widehat{A} &= A \cup \{\alpha L, \alpha R \mid \alpha \in A\}\ . \end{aligned}$$

The set A_t is a set of paths, but it can be thought of as the binary tree obtained by ignoring labels on the nodes of t. The set A can be thought of as the tree obtained by superimposing the trees A_t, $t \in T$, on one another. The set $\widehat{A}$ can be thought of as the tree obtained from A by sprouting two new shoots out of each leaf.

Assume $|T| \geq 2$ (the only singleton cover is $\{x\}$). Let $n = \sum_{t\in T} |A_t|$ (a measure of the size of the problem). Then $|A| < n$ and $|\widehat{A}| - |A| = |A| + 1 \leq n$. Let M be the set of ground terms t such that A_t is the complete binary tree of depth $\log\log 2n$. There are at least $2n$ terms in M, one for each assignment of a and b to the $\log 2n$ leaves, and each term in M is of size $2\log 2n$.

We claim now that if T covers all ground terms of size at most $n + 2n\log 2n$, then it covers all ground terms. To see this, suppose that T covers all ground terms of size at most $n + 2n\log 2n$, and let t be an arbitrary ground term. Associate with each $\alpha \in \widehat{A} - A$ a *unique* term $u_\alpha \in M$ such that $u_\alpha \neq t/\beta$ for any $\beta \in A$. Since $|M| \geq 2n$, $|A| \leq n$, and $|\widehat{A}| - |A| \leq n$, there are enough terms in M to do this. Let t' be a new term obtained from t by replacing all subterms t/α for $\alpha \in \widehat{A} - A$ with u_α. In other words,

- for $\alpha \in A$, set $t'(\alpha) = t(\alpha)$;
- for $\alpha \in \widehat{A} - A$ such that $t(\alpha) \neq \bot$, set $t'(\alpha\beta) = u_\alpha(\beta)$ for all $|\beta| \leq \log\log 2n$;
- for all α not assigned in (a) or (b), set $t'(\alpha) = \bot$.

From the construction of t', it is clear that

$$|A_{t'}| \;\leq\; |A| + |\widehat{A} - A| \cdot 2\log 2n \;\leq\; n + 2n\log 2n\ ,$$

so by assumption t' is a substitution instance of some term $s \in T$.

We claim that t is a substitution instance of s as well. Certainly, if all the variables of s are distinct, then t is a substitution instance of s, since t and t' agree on all elements of A. Now suppose for a contradiction that there exist strings α and β such that $s(\alpha) = s(\beta) \in$

X (thus $\alpha, \beta \in A$) and $t/\alpha \neq t/\beta$. Then there exists a string γ such that

$$t(\alpha\gamma) \;=\; t/\alpha(\gamma) \;\neq\; t/\beta(\gamma) \;=\; t(\beta\gamma)\,.$$

We may assume without loss of generality that neither $t(\alpha\gamma)$ nor $t(\beta\gamma) = \perp$; otherwise, we can find a prefix of γ such that this is true, and take γ to be that prefix instead. We have three cases:

- If both $\alpha\gamma, \beta\gamma \in A$, then $t'(\alpha\gamma) \neq t'(\beta\gamma)$, since t and t' agree on A. This contradicts the fact that t' is a substitution instance of s.
- If $\alpha\gamma \notin A$ and $\beta\gamma \in A$, let γ' be a prefix of γ such that $\alpha\gamma' \in \widehat{A} - A$. Then $\beta\gamma' \in A$. Either $t/\beta\gamma'$ is entirely contained in A (in the sense that for all $\delta \in A_{t/\beta\gamma'}$, $\beta\gamma'\delta \in A$), in which case $t'/\beta\gamma' = t/\beta\gamma'$; or there exists a δ such that $\beta\gamma'\delta \in \widehat{A} - A$ and $t(\beta\gamma'\delta) \neq \perp$, in which case $t'/\beta\gamma'\delta = u_{\beta\gamma'\delta}$. In either case $t'/\alpha\gamma' = u_{\alpha\gamma'} \neq t'/\beta\gamma'$, again contradicting the fact that t' is a substitution instance of s.
- If $\alpha\gamma \notin A$ and $\beta\gamma \notin A$, let γ' be a prefix of γ such that $\alpha\gamma' \in \widehat{A} - A$ and $\beta\gamma' \in \widehat{A}$ (or vice versa—assume the former). If $\beta\gamma' \in A$, then we revert to the previous case. Otherwise, both $\alpha\gamma', \beta\gamma' \in \widehat{A} - A$. Then

  $$t'(\alpha\gamma') \;=\; u_{\alpha\gamma'} \;\neq\; u_{\beta\gamma'} \;=\; t'(\beta\gamma')\,,$$

  again contradicting the fact that t' is a substitution instance of s.

Since all three cases lead to a contradiction, we conclude that t is a substitution instance of s. Since t was arbitrary, T is a cover.

Thus to determine whether there exists a ground term not covered by T, we need only guess one of size at most $n + 2n \log 2n$ and check that it is not covered by any $s \in T$. This can easily be done in nondeterministic polynomial time.

25. This problem can be solved with n processors in $O(\log n)$ time. We associate a processor with each vertex. The processor associated with vertex v will calculate a pointer **next**(v) to the successor of v in the preorder traversal. If v is not a leaf, then **next**(v) is its leftmost child. If v is a leaf, then **next**(v) is the leftmost right sibling of $d(v)$, where $d(v)$ is the lowest ancestor of v that is not the rightmost child of its parent. If no such $d(v)$ exists, then v is the last vertex in preorder and has no successor.

    To compute the **next** pointers of leaves in logarithmic time, we use the technique of *pointer doubling*. Initially, each vertex v sets

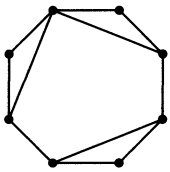
    $$\mathbf{d}(v) := \begin{cases} \text{parent}\,(v) & \text{if } v \text{ is the rightmost child of its parent;} \\ v & \text{if } v \text{ is not the rightmost child of its parent or } v \text{ is the root.} \end{cases}$$

Each vertex v then iterates the operation $\mathbf{d}(v) := \mathbf{d}(\mathbf{d}(v))$ $\log n$ times. This must be done synchronously in parallel. At this point $\mathbf{d}(v) = d(v)$ if $d(v)$ exists, or the root if not. If so, the leaf v sets $\mathbf{next}(v)$ to the leftmost right sibling of $\mathbf{d}(v)$.

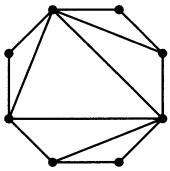
We now have a list **next** of all vertices in the correct order. It remains to compute the preorder number of v for each v. This is the number of vertices appearing on the list **next** before v. This can be done in $\log n$ stages with parallel prefix addition, using pointer doubling (*i.e.*, $\mathbf{next}(v) := \mathbf{next}(\mathbf{next}(v))$) to calculate the address to send the next message in each stage.

26. (a) To test whether G is outerplanar and to find an outerplane embedding if one exists, we add a new vertex v and an edge from v to all other vertices, then use the Hopcroft-Tarjan planarity test [52] to test whether the resulting graph G' is planar and find a plane embedding if so. This can be done in linear time. If G is outerplanar, then G can be embedded with all vertices on the outer face, so embedding v in the outer face allows v to be connected to all other vertices without crossing; thus G' is planar. Conversely, if G' is planar, then it can be embedded with v adjacent to the outer face; then deleting v and its incident edges gives an outerplane embedding of G.

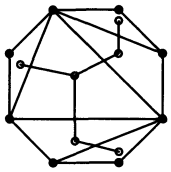
(b) There always exists a $\frac{1}{3}$-$\frac{2}{3}$ separator of size 2. To find it, we first find an outerplane embedding as in part (a).



We then triangulate the interior faces. This can be done by traversing each interior face, adding an edge from the first vertex on the face to every other vertex on the face it is not already connected to. (We know from part (a) which face is the exterior face: it is the one that contained v.)



We then compute the plane dual of G using Miscellaneous Exercise 9, but we omit the vertex corresponding to the outer face of G and all incident edges. The resulting graph is a tree T, because any cycle would contain a vertex of G not on the outer face.



Exercise 2 can then be used to find an edge e in T whose removal disconnects T into two disjoint subtrees with no more than $\frac{2f+1}{3}$ vertices each, where f is the number of vertices in T (= number of interior faces of G). The desired separator consists of the endpoints of the dual edge of e.

27. (a) Let the data elements be $a_1, \ldots, a_n$. Assign n^2 processors to compare every element with every other element. For all i, j, $1 \leq i, j \leq n$, let

$$A_{ij} = \begin{cases} 1 & \text{if either } a_i < a_j \text{ or } (a_i = a_j \text{ and } i < j) \\ 0 & \text{otherwise.} \end{cases}$$

The $n \times n$ matrix A can be produced by the n^2 processors in one step. The matrix A determines the sorted order: for all i, the number of 1's in the i^{th} row of A is the position of i in sorted order. Computing the i^{th} row sum of A takes $O(\log n)$ time and $O(n)$ processors in parallel, or $O(\log n)$ time and $O(n^2)$ processors to compute all the row sums. Once the position of an element in sorted order is computed, that element is stored in the proper element of the output array. This takes one step with n processors, assuming random access to the output array.

(b) We use parallel mergesort. Each of n processors is assigned to a different input element. The set of elements is split into two sets of roughly equal size which are then sorted recursively in parallel. We then merge the two sorted arrays in $O(\log n)$ time with n processors as described below. We obtain the recurrence

$$T(n) = T(\frac{n}{2}) + O(\log n)$$

giving a running time of

$$T(n) = O((\log n)^2) .$$

We now show how to merge two sorted arrays in time $O(\log n)$. Let S, T be the two sorted arrays of size m and n, respectively. We have $m+n$ processors at our disposal, each assigned to a different element. First we find the medians x and y of S and T, respectively. This takes one step. Then compare x and y; say without loss of generality $x \leq y$. Split each of S and T into three arrays

$$\begin{array}{lll} S_0 = \{z \in S \mid z \leq x\} & & T_0 = \{z \in T \mid z \leq x\} \\ S_1 = \{z \in S \mid x < z \leq y\} & & T_1 = \{z \in T \mid x < z \leq y\} \\ S_2 = \{z \in S \mid y < z\} & & T_2 = \{z \in T \mid y < z\} \end{array}$$

in one step. Note that each S_i is at most half the size of S and each T_i is at most half the size of T. For each $0 \leq i \leq 2$ in parallel, merge S_i and T_i recursively with the $|S_i| + |T_i|$ processors assigned to S_i and T_i. Let U_i be the array obtained by merging S_i and T_i, $0 \leq i \leq 2$. Now store U_0, U_1, and U_2 in an array end-to-end. The processor associated with the i^{th} element of U_0 stores its element in position i of the output array; the processor associated with the i^{th} element of U_1 stores its element in position $|U_0| + i$ of the output array; and the processor associated with the i^{th} element of U_2 stores its element in position $|U_0| + |U_1| + i$ of the output array. This takes constant time in parallel. We obtain the recurrence

$$T(n) = T(\frac{n}{2}) + O(1)$$

giving a parallel time bound of

$$T(n) = O(\log n)$$

for the merge.

Parallel sorting is a topic of intense current interest. There are much more efficient *NC* algorithms known for sorting than the ones given here. To mention a few: Ajtai, Komlos, and Szemeredi [5] give a sorting network of depth $O(\log n)$ and $O(n)$ linear width; Cole [20] gives a CREW PRAM sorting algorithm that runs in time $O(\log n)$ on n processors; Bilardi and Nicolau[12] give an EREW PRAM bitonic sorting algorithm that runs in time $O((\log n)^2)$ on $n/\log n$ processors.

28. (a) We first show that the product of two circulant matrices is again a circulant matrix. A matrix C is circulant iff, when the columns are rotated one position to the left and then the rows are rotated up, we get C back; in terms of permutation matrices,

$$P^{-1}CP = C\,,$$

where

$$P = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}.$$

Then AB is circulant if A and B are, since

$$P^{-1}ABP = P^{-1}APP^{-1}BP = AB\,.$$

While we are at it, let us show that the inverse of a circulant matrix, if it exists, is circulant:

$$P^{-1}A^{-1}P \;=\; (P^{-1}AP)^{-1} \;=\; A^{-1}\,.$$

We can easily compute the first row of AB in $O(\log n)$ time with n^2 processors, since each element of the first row is an inner product and can be computed in $O(\log n)$ time with n processors. Since AB is circulant, we need only rotate the first row to get the other rows. This takes constant time with n^2 processors.

(b) Let $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$. Let $c(a)$ denote the unique circulant matrix with first row a; thus $c(a)_{ij} = a_{j-i}$. In [3, pp. 256–257], it is stated that the vector $F_n^{-1}(F_n a \cdot F_n b)$ is the *positive wrapped convolution* of a and b:

$$F_n^{-1}(F_n a \cdot F_n b) \tag{23}$$

$$= \;(\sum_{i=0}^{n-1} a_i b_{-i}, \sum_{i=0}^{n-1} a_i b_{1-i}, \sum_{i=0}^{n-1} a_i b_{2-i}, \ldots, \sum_{i=0}^{n-1} a_i b_{n-1-i})\,. \tag{24}$$

(Subscripts in (24) are taken modulo n.) It can be shown by a direct calculation that this is the first row of the matrix product $c(a) \cdot c(b)$; thus

$$c(a) \cdot c(b) \;=\; c(F_n^{-1}(F_n a \cdot F_n b))\,. \tag{25}$$

The vector $F_n^{-1}(F_n a \cdot F_n b)$ can be computed in time $O(\log n)$ with n processors by doing two Fourier transforms, a componentwise vector product, and an inverse Fourier transform.

Since the proof of (24) is omitted from [3], we supply one here. Let f and g be polynomials of degree at most $n-1$ with coefficients a and b, respectively. As shown in Lecture 35, under the Fourier transform, multiplication of polynomials modulo $x^n - 1$ becomes componentwise vector product:

$$fg \bmod x^n - 1 \;=\; F_n^{-1}(F_n f \cdot F_n g)\,. \tag{26}$$

Modulo $x^n - 1$, we can equate monomials x^i and $x^{i \bmod n}$. This allows us to take superscripts as well as subscripts modulo n. Doing so, we get the following calculation:

$$\begin{aligned} fg \bmod x^n - 1 \;&=\; \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i b_j x^{i+j} \\ &=\; \sum_{i=0}^{n-1}\sum_{k=0}^{n-1} a_i b_{k-i} x^k \\ &=\; \sum_{k=0}^{n-1}(\sum_{i=0}^{n-1} a_i b_{k-i}) x^k\,. \end{aligned}$$

Thus the coefficient of x_k in $fg \bmod x^n - 1$ is $\sum_{i=0}^{n-1} a_i b_{k-i}$, the k^{th} element of the positive wrapped convolution of a and b. This and (26) give (24).

Algebraically, what is really going on here is that the circulant matrices form an n-dimensional subalgebra of the n^2-dimensional algebra of $n \times n$ matrices, and this subalgebra is isomorphic to the subalgebra of diagonal matrices via the map

$$C \mapsto F_n^{-1} C F_n \ .$$

Moreover, if $d(a)$ denotes the diagonal matrix with diagonal a, then

$$F_n^{-1} c(a) F_n \ = \ d(F_n a) \ . \tag{27}$$

This can be established by a direct calculation, using the property

$$\sum_{j=0}^{n-1} \omega^{ij} \ = \ \begin{cases} n \ , & \text{if } i \equiv 0 \bmod n \\ 0 \ , & \text{otherwise} \end{cases}$$

where ω is a primitive n^{th} root of unity: the i, ℓ^{th} element of $F_n^{-1} c(a) F_n$ is

$$\begin{aligned} \frac{1}{n} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} \omega^{-ij} a_{k-j} \omega^{k\ell} \ &= \ \frac{1}{n} \sum_{m=0}^{n-1} \sum_{k=0}^{n-1} \omega^{-i(k-m)} a_m \omega^{k\ell} \\ &= \ \sum_{m=0}^{n-1} a_m \omega^{im} \Big(\frac{1}{n} \sum_{k=0}^{n-1} \omega^{(\ell - i)k} \Big) \\ &= \ \begin{cases} \sum_{m=0}^{n-1} a_m \omega^{im} & \text{if } i = \ell \\ 0 & \text{if } i \neq \ell \end{cases} \end{aligned}$$

which is also the i, ℓ^{th} element of $d(F_n a)$.

(c) The first row of the inverse of $c(a)$ can be calculated by taking the Fourier transform of a, inverting all the elements of the resulting vector, and transforming back. In other words,

$$c(a)^{-1} \ = \ c(F_n^{-1}((F_n a)')) \ ,$$

where b' is the vector obtained from b by inverting all the elements. This follows immediately from the isomorphism $c(a) \mapsto d(F_n a)$ discussed in part (b) above, but in case you did not have the patience to wade through all that, here is a more direct argument. If $(F_n a)'$ exists, then by (25),

$$\begin{aligned} c(a) \cdot c(F_n^{-1}((F_n a)')) \ &= \ c(F_n^{-1}(F_n a \cdot F_n(F_n^{-1}((F_n a)')))) \\ &= \ c(F_n^{-1}(F_n a \cdot (F_n a)')) \\ &= \ c(F_n^{-1}(1, 1, 1, \ldots, 1)) \\ &= \ c(1, 0, 0, \ldots, 0)) \\ &= \ I \ , \end{aligned}$$

thus $c(a)$ and $c(F_n^{-1}((F_n a)'))$ are inverses. Conversely, if the inverse $c(a)^{-1}$ exists and b is its first row, then

$$\begin{aligned} c(F_n^{-1}(1,1,1,\ldots,1)) &= c(1,0,0,\ldots,0)) \\ &= I \\ &= c(a)\cdot c(b) \\ &= c(F_n^{-1}(F_n a \cdot F_n b)) \ , \end{aligned}$$

by (25). Therefore

$$F_n a \cdot F_n b = (1,1,1,\ldots,1) \ ,$$

so $(F_n a)'$ exists and is equal to $F_n b$. The entire operation can be done in time $O(\log n)$ with n processors using the fast Fourier transform.

Circulant matrices have numerous applications in geometry, differential equations, and mechanics. To find out more about them, see Davis' book [27].

29. For permutation $\sigma \in S_n$, define

$$t(\sigma) = \prod_{i=1}^{n} T_{i,\sigma(i)} \ .$$

Then

$$\det T = \sum_{\sigma \in S_n} (-1)^{\mathrm{sign}(\sigma)} t(\sigma) \ .$$

Let E_n be the set of permutations in S_n with only even cycles.

Lemma

$$\det T = \sum_{\sigma \in E_n} (-1)^{\mathrm{sign}(\sigma)} t(\sigma) \ .$$

Proof. We will show that the contributions of permutations σ containing odd cycles cancel each other out. Suppose σ contains an odd cycle ρ, and let $\tau = \sigma\rho^{-1}$. Then $\sigma = \tau\rho = \rho\tau$ (disjoint cycles commute). Consider the permutation $\tau\rho^{-1}$. Then

$$t(\tau\rho^{-1}) = -t(\tau\rho) \ ,$$

since $\tau\rho^{-1}$ changes the signs of an odd number of factors of $t(\tau\rho)$. For example, if $\rho = (1\,3\,7\,4\,6)$ and $\tau = (2\,5)$, then $\rho^{-1} = (6\,4\,7\,3\,1)$, and

$$\begin{aligned} t(\tau\rho) &= x_{25} \cdot -x_{25} \cdot x_{13} \cdot x_{37} \cdot -x_{47} \cdot x_{46} \cdot -x_{16} \\ t(\tau\rho^{-1}) &= x_{25} \cdot -x_{25} \cdot -x_{46} \cdot x_{47} \cdot -x_{37} \cdot -x_{13} \cdot x_{16} \\ &= -t(\tau\rho)\ . \end{aligned}$$

Moreover, $\operatorname{sign}(\tau\rho) = \operatorname{sign}(\tau\rho^{-1})$, since

$$\begin{aligned} (\tau\rho^{-1})^{-1}\tau\rho &= \rho\tau^{-1}\tau\rho \\ &= \rho^2 \end{aligned}$$

which is even, thus $\tau\rho$ and $\tau\rho^{-1}$ are either both even or both odd. Thus the permutations containing odd cycles ρ and ρ^{-1} can be paired up so that their contributions $(-1)^{\operatorname{sign}(\sigma)}t(\sigma)$ to $\det T$ cancel. This assignment can be repeated for other permutations σ not containing ρ but containing another odd cycle. Thus we are left with the permutations containing even cycles only. □

(a) If the multivariate polynomial $\det T$ is not identically 0, then by the Lemma there must exist a permutation σ containing even cycles only such that $t(\sigma) \neq 0$. But then σ gives a perfect matching by taking alternate edges around the cycles. Conversely, let M be a perfect matching. Assign $x_{uv} = 1$ for $(u, v) \in M$ and $x_{uv} = 0$ otherwise. Under this substitution, there is exactly one σ with $t(\sigma) \neq 0$, namely the one corresponding to M, thus $\det T$ with this substitution is nonzero. Therefore $\det T$ is not identically 0.

(b) Select a random assignment α to the x_{uv} from a set of size 2^n. By Corollary 40.2, the probability that $\det T(\alpha) = 0$ is 1 if $\det T$ is identically 0 and at most $\frac{n}{2^n}$ if not. Thus with a probability of error at most $\frac{n}{2^n}$ we can determine whether G has a perfect matching.

(c) For each edge in succession, test whether the graph with edge e removed has a perfect matching using the above procedure. If so, then delete e from G. With high probability, we are left with the edges of a perfect matching.

Bibliography

[1] L. M. Adleman and M.-D. A. Huang. Primality testing and two-dimensional Abelian varieties over finite fields. preprint, University of Southern California, February 1988.

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

[3] A. V. Aho, J. E Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1975.

[4] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM J. Comput.*, 18:939–954, 1989.

[5] M. Ajtai, J. Komlos, , and E. Szemeredi. An $O(n \log n)$ sorting network. *Combinatorica*, 3:1–9, 1983.

[6] N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *J. Algorithms*, 7:567–583, 1986.

[7] C. R. Aragon and R. G. Seidel. Randomized search trees. In *Proc. 30th Symp. Foundations of Computer Science*, pages 540–545. IEEE, 1989.

[8] E. Bach. *Analytic methods in the analysis and design of number theoretic algorithms.* MIT Press, Cambridge, Mass., 1985.

[9] P. W. Beame, S. A. Cook, and H. James Hoover. Log depth circuits for division and related problems. In *Proc. 25th Conf. Foundations of Computer Science*, pages 1–6. IEEE, October 1984.

[10] C. Berge. Two theorems in graph theory. *Proc. National Acad. Sci.*, 43:842–844, 1957.

[11] S. Berkowitz. On computing the determinant in small parallel time using a small number of processors. *Information Processing Letters*, 18:147–150, 1984.

[12] G. Bilardi and A. Nicolau. Adaptive bitonic sorting: an optimal parallel algorithm for shared-memory machines. *SIAM J. Comput.*, 18(2):216–228, April 1989.

[13] Béla Bollobás. *Extremal Graph Theory.* Academic Press, 1978.

[14] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* North Holland, 1976.

[15] A. Borodin, J. von zur Gathen, and J. Hopcroft. Fast parallel matrix and gcd computations. *Information and Control*, 52(3):241–256, 1982.

[16] I. Borosh and L. B. Treybig. Bounds on positive integral solutions of linear diophantine equations. *Proc. Amer. Math. Soc.*, 55:299–304, 1976.

[17] W. Brown and J. F. Traub. On Euclid's algorithm and the theory of subresultants. *J. Assoc. Comput. Mach.*, 18:505–514, 1971.

[18] A. L. Chistov. Fast parallel calculation of the rank of matrices over a field of arbitrary characteristic. In *Proc. Conf. Foundations of Computation Theory*, volume 199 of *Lect. Notes in Comput. Sci.*, pages 63–69. Springer-Verlag, 1985.

[19] V. Chvátal. *Linear Programming.* Freeman, 1980.

[20] R. Cole. Parallel merge sort. In *Proc. 27th Symp. Foundations of Computer Science*, pages 511–516. IEEE, October 1986.

[21] J. H. Conway. *Regular Algebra and Finite Machines.* Chapman and Hall, London, 1971.

[22] S. A. Cook. The complexity of theorem proving procedures. In *Proc. 3rd Symp. Theory of Computing*, pages 151–158. ACM, 1971.

[23] S. A. Cook. The classification of problems which have fast parallel algorithms. In Karpiński, editor, *Proc. 1983 Symp. Foundations of Computation Theory*, volume 158 of *Lect. Notes in Comput. Sci.*, pages 78–93. Springer-Verlag, 1983.

[24] J. M. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[25] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proc. 19th Symp. Theory of Computing*, pages 1–6. ACM, May 1987.

[26] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM J. Comput.*, 5:618–623, 1976.

[27] Philip J. Davis. *Circulant Matrices.* Wiley, 1979.

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

[29] E. A. Dinic. Algorithm for solution of a problem of maximal flow in a network with power estimation. *Soviet Math. Doklady*, 11:1277–1280, 1970.

[30] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network problems. *J. Assoc. Comput. Mach.*, 19:248–264, 1922.

[31] J. R. Edmonds. A combinatorial representation for polyhedral surfaces. *Notices Amer. Math. Soc.*, 7:646, 1960.

[32] J. R. Edmonds. Matroids and the greedy algorithm. *Math. Programming*, 1:127–136, 1971.

[33] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. Wiley, 1950.

[34] L. R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canad. J. Math.*, 8:399–404, 1956.

[35] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proc. 25th Symp. Foundations of Computer Science*, pages 338–346. IEEE, 1984.

[36] D. Gale and L. S. Shapley. College admissions and the stability of marriage. *Amer. Math. Monthly*, 69:9–14, 1962.

[37] Z. Galil. An $O(V^{5/3}E^{2/3})$ algorithm for the maximal flow problem. *Acta Informatica*, 14:221–242, 1980.

[38] Z. Galil and E. Tardos. An $O(n^2(m + n \log n) \log n)$ min-cost flow algorithm. *J. Assoc. Comput. Mach.*, 35:374–386, 1988.

[39] M. R. Garey and D. S. Johnson. *Computers and Intractibility: a Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[40] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified *NP*-complete graph problems. *Theor. Comput. Sci.*, 1:237–267, 1976.

[41] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

[42] L. M. Goldschlager. The monotone and planar circuit value problems are logspace complete for *P*. *SIGACT News*, 9(2):25–29, 1977.

[43] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison Wesley, 1989.

[44] A. C. Greenberg, R. E. Ladner, M. S. Paterson, and Z. Galil. Efficient parallel algorithms for linear recurrence computation. *Infor. Proc. Letters*, 15(1):31–35, 1982.

[45] D. Gries and G. Levin. Computing Fibonacci numbers (and similarly defined functions) in log time. *Infor. Proc. Letters*, 11(2):68–69, 1980.

[46] D. Gries, A. J. Martin, J. L. A. van de Snepscheut, and J. T. Udding. An algorithm for transitive reduction of an acyclic graph. *Science of Computer Programming*, 12(2):151–155, July 1989.

[47] A. Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39:297–308, 1985.

[48] F. Harary. *Graph Theory.* Addison-Wesley, 1972.

[49] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* Oxford, 1979.

[50] J. Hartmanis and J. Simon. On the power of multiplication in random access machines. In *Proc. 15th Symp. Switching and Automata Theory*, pages 13–23, 1974.

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

[52] J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *J. Assoc. Comput. Mach.*, 21:549–568, 1974.

[53] O. Ibarra, S. Moran, and L. E. Rosier. A note on the parallel complexity of computing the rank of order n matrices. *Information Processing Letters*, 11:162, 1980.

[54] S. L. Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. Technical Report YALEU/DCS/RR-361, Yale University, September 1985.

[55] S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.

[56] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.

[57] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[58] R. M. Karp. On the complexity of combinatorial problems. *Networks*, 5:45–68, 1975.

[59] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. In *Proc. 16th Symp. Theory of Computing*, pages 266–272. ACM, May 1984.

[60] L. G. Khachian. Polynomial algorithms in linear programming. *Zhurnal Vychislitelnoi Matematiki i Matematicheskoi Fiziki*, 20:53–72, 1980.

[61] S. C. Kleene. Representation of events in nerve nets and finite automata. In Shannon and McCarthy, editors, *Automata Studies*, pages 3–41. Princeton U. Press, 1956.

[62] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 2. Addison Wesley, 1973.

[63] D. C. Kozen. On induction vs. *-continuity. In Kozen, editor, *Proc. Workshop on Logics of Programs 1981*, volume 131 of *Lect. Notes in Comput. Sci.*, pages 167–176. Springer-Verlag, 1981.

[64] D. C. Kozen. On Kleene algebras and closed semirings. In Rovan, editor, *Proc. Math. Found. Comput. Sci. 1990*, volume 452 of *Lect. Notes in Comput. Sci.*, pages 26–47. Springer-Verlag, 1990.

[65] D. C. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. In *Proc. 6th Symp. Logic in Comput. Sci.*, pages 214–225. IEEE, 1991.

[66] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Soc.*, 7:48–50, 1956.

[67] R. Ladner. The circuit value problem is logspace complete for *P*. *SIGACT News*, 7(1):18–20, 1975.

[68] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *J. Assoc. Comput. Mach.*, 27(4):831–838, 1980.

[69] S. Lang. *Algebra.* Addison Wesley, second edition, 1984.

[70] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, Winston, 1976.

[71] L. A. Levin. Universal sorting problems. *Problems of Information Transmission*, 9:265–266, 1973.

[72] D. Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, 11(2):329–343, 1982.

[73] R. Lipton and R. E. Tarjan. Applications of a planar separator theorem. In *Proc. 18th Conf. Foundations of Computer Science*, pages 162–170. IEEE, 1977.

[74] L. Lovász. On determinants, matchings, and random algorithms. In Budach, editor, *Proc. Symp. on Fundamentals of Computing Theory*, pages 565–574, Berlin, 1979. Akademia-Verlag.

[75] L. Lovász and M. D. Plummer. *Matching Theory.* North Holland, 1986.

[76] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *Proc. 17th Symp. Theory of Computing*, pages 1–10. ACM, May 1985.

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

[78] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1984.

[79] A. R. Meyer and R. Ritchie. The complexity of loop programs. In *Proc. National Meeting*, pages 465–469. ACM, 1967.

[80] S. Micali and V. Vazirani. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matchings in general graphs. In *Proc. 21st Symp. Foundations of Computer Science*, pages 17–27. IEEE, 1980.

[81] G. L. Miller. Riemann's hypothesis and tests for primality. *J. Comput. Syst. Sci.*, 13:300–317, 1976.

[82] K. Mulmuley. A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. *Combinatorica*, 7(1):101–104, 1987.

[83] V. Ya. Pan. Strassen's algorithm is not optimal. In *Proc. 19th Symp. Foundations of Computer Science*, pages 166–176. IEEE, 1978.

[84] V. Ya. Pan. *How to multiply matrices faster*, volume 179 of *Lect. Notes in Comput. Sci.* Springer-Verlag, 1984.

[85] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, 1982.

[86] A. Pettorossi. Derivation of an $O(k^2 \log n)$ algorithm for computing order-k Fibonacci numbers from the $O(k^3 \log n)$ matrix multiplication method. *Infor. Proc. Letters*, 11(4):172–179, 1980.

[87] J. A. La Poutré. Lower bounds for the union-find and the split-find problem on pointer machines. In *Proc. 22nd Symp. Theory of Computing*, pages 34–44. ACM, 1990.

[88] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Comm. Assoc. Comput. Mach.*, 33(6):668–676, June 1990.

[89] M. O. Rabin. Probabilistic algorithms for testing primality. *J. Number Theory*, 12:128–138, 1980.

[90] J. Renegar. A polynomial-time algorithm based on Newton's method for linear programming. *Math. Programming*, 40:59–93, 1988.

[91] R. L. Rivest and J. Vuillemin. On recognizing graph properties from adjacency matrices. *Theor. Comput. Sci.*, 3:371–384, 1976/77.

[92] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. Assoc. Comput. Mach.*, 27:701–717, 1980.

[93] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3:57–67, 1982.

[94] D. Sleator and R. E. Tarjan. Self-adjusting binary trees. In *Proc. 15th Symp. Theory of Computing*, pages 235–245. ACM, 1983.

[95] D. D. Sleator. An $O(nm \log n)$ algorithm for maximum network flow. Technical Report STAN-CS-80-831, Stanford University, 1980.

[96] R. Solovay and V. Strassen. A fast Monte Carlo test for primality. *SIAM J. Comput*, 6:84–85, 1977.

[97] V. Strassen. Gaussian elimination is not optimal. *Numerische Math.*, 13:354–356, 1969.

[98] E. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5:247–255, 1985.

[99] R. E. Tarjan. A class of algorithms that require nonlinear time to maintain disjoint sets. *J. Comput. Syst. Sci.*, 18:110–127, 1979.

[100] R. E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *Regional Conference Series in Applied Mathematics*. SIAM, 1983.

[101] A. Urquhart. Hard examples for resolution. *J. Assoc. Comput. Mach.*, 34(1):209–219, 1987.

[102] P. M. Vaidya. An algorithm for linear programming which requires $O(((m+n)n^2+(m+n)^{1.5}n)L)$ arithmetic operations. In *Proc. 19th Symp. Theory of Computing*, pages 29–38. ACM, 1987.

[103] L. G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.

[104] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proc. 13th Symp. Theory of Computing*, pages 263–277. ACM, 1981.

[105] V. V. Vazirani. A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{|V|}\cdot|E|)$ general graph matching algorithm. Technical Report 89-1035, Cornell University, September 1989.

[106] J. Vuillemin. A data structure for manipulating priority queues. *Comm. Assoc. Comput. Mach.*, 21:309–314, 1978.

[107] D. J. A. Welsh. *Matroid Theory.* Academic Press, 1976.

[108] H. S. Wilf. *Algorithms and Complexity.* Prentice-Hall, 1986.

[109] T. C. Wilson and J. Shortt. An $O(\log n)$ algorithm for computing general order-k Fibonacci numbers. *Infor. Proc. Letters*, 10(2):68–75, 1980.

[110] A. C.-C. Yao. Monotone bipartite graph properties are evasive. *SIAM J. Comput.*, 17(3):517–520, 1988.

[111] R. E. Zippel. Probabilistic algorithms for sparse polynomials. In Ng, editor, *Proc. EUROSAM 79*, volume 72 of *Lect. Notes in Comput. Sci.*, pages 216–226. Springer-Verlag, 1979.

Index

Texts and Monographs in Computer Science

(continued from page ii)

John V. Guttag and James J. Horning, **Larch: Languages and Tools for Formal Specification**

Eric C.R. Hehner, **A Practical Theory of Programming**

Micha Hofri, **Probabilistic Analysis of Algorithms**

A.J. Kfoury, Robert N. Moll, and Michael A. Arbib, **A Programming Approach to Computability**

Dexter C. Kozen, **The Design and Analysis of Algorithms**

E.V. Krishnamurthy, **Error-Free Polynomial Matrix Computations**

Ming Li and Paul Vitányi, **An Introduction to Kolmogorov Complexity and Its Applications**

David Luckham, **Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs**

Ernest G. Manes and Michael A. Arbib, **Algebraic Approaches to Program Semantics**

Bhubaneswar Mishra, **Algorithmic Algebra**

Robert N. Moll, Michael A. Arbib, and A.J. Kfoury, **An Introduction to Formal Language Theory**

Anil Nerode and Richard A. Shore, **Logic for Applications**

Helmut A. Partsch, **Specification and Transformation of Programs**

Franco P. Preparata and Michael Ian Shamos, **Computational Geometry: An Introduction**

Brian Randell, Ed., **The Origins of Digital Computers: Selected Papers, 3rd Edition**

Thomas W. Reps and Tim Teitelbaum, **The Synthesizer Generator: A System for Constructing Language-Based Editors**

Thomas W. Reps and Tim Teitelbaum, **The Synthesizer Generator Reference Manual, 3rd Edition**

Arto Salomaa and Matti Soittola, **Automata-Theoretic Aspects of Formal Power Series**

J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg, **Programming with Sets: An Introduction to SETL**

Alan T. Sherman, **VLSI Placement and Routing: The PI Project**

Texts and Monographs in Computer Science

(continued)

Santosh K. Shrivastava, Ed., **Reliable Computer Systems**

Jan L.A. van de Snepscheut, **What Computing Is All About**

William M. Waite and Gerhard Goos, **Compiler Construction**

Niklaus Wirth, **Programming in Modula-2, 4th Edition**

Study Edition

Edward Cohen, **Programming in the 1990s: An Introduction to the Calculation of Programs**