

Good problems

[KT] Let's consider a long, quiet country road with houses scattered very sparsely along it. (We can picture the road as a long line segment, with an eastern endpoint and a western endpoint.) Further, let's suppose that despite the bucolic setting, the residents of all these houses are avid cell phone users. You want to place cell phone base stations at certain points along the road, so that every house is within four miles of one of the base stations.

Give an efficient algorithm that achieves this goal, using as few base stations as possible.

[KT] The wildly popular Spanish-language search engine El Goog needs to do a serious amount of computation every time it recompiles its index. Fortunately, the company has at its disposal a single large supercomputer, together with an essentially unlimited supply of high-end PCs.

They've broken the overall computation into n distinct jobs, labeled J_1, J_2, \dots, J_n , which can be performed completely independently of one another. Each job consists of two stages: first it needs to be *preprocessed* on the supercomputer, and then it needs to be *finished* on one of the PCs. Let's say that job J_i needs p_i seconds of time on the supercomputer, followed by f_i seconds of time on a PC.

Since there are at least n PCs available on the premises, the finishing of the jobs can be performed fully in parallel—all the jobs can be processed at the same time. However, the supercomputer can only work on a single job at a time, so the system managers need to work out an order in which to feed the jobs to the supercomputer. As soon as the first job in order is done on the supercomputer, it can be handed off to a PC for finishing; at that point in time a second job can be fed to the supercomputer; when the second job is done on the supercomputer, it can proceed to a PC regardless of whether or not the first job is done (since the PCs work in parallel); and so on.

Let's say that a *schedule* is an ordering of the jobs for the supercomputer, and the *completion time* of the schedule is the earliest time at which all jobs will have finished processing on the PCs. This is an important quantity to minimize, since it determines how rapidly El Goog can generate a new index.

Give a polynomial-time algorithm that finds a schedule with as small a completion time as possible.

[KT] The manager of a large student union on campus comes to you with the following problem. She's in charge of a group of n students, each of whom is scheduled to work one *shift* during the week. There are different jobs associated with these shifts (tending the main desk, helping with package delivery, rebooting cranky information kiosks, etc.), but we can view each shift as a single contiguous interval of time. There can be multiple shifts going on at once.

She's trying to choose a subset of these n students to form a *supervising committee* that she can meet with once a week. She considers such a committee to be *complete* if, for every student not on the committee, that student's shift overlaps (at least partially) the shift of some student who is on the committee. In this way, each student's performance can be observed by at least one person who's serving on the committee.

Give an efficient algorithm that takes the schedule of n shifts and produces a complete supervising committee containing as few students as possible.

Example. Suppose $n = 3$, and the shifts are

Monday 4 P.M.–Monday 8 P.M.,

Monday 6 P.M.–Monday 10 P.M.,

Monday 9 P.M.–Monday 11 P.M..

Then the smallest complete supervising committee would consist of just the second student, since the second shift overlaps both the first and the third.

[KT] Consider the following variation on the Interval Scheduling Problem. You have a processor that can operate 24 hours a day, every day. People submit requests to run *daily jobs* on the processor. Each such job comes with a *start time* and an *end time*; if the job is accepted to run on the processor, it must run continuously, every day, for the period between its start and end times. (Note that certain jobs can begin before midnight and end after midnight; this makes for a type of situation different from what we saw in the Interval Scheduling Problem.)

Given a list of n such jobs, your goal is to accept as many jobs as possible (regardless of their length), subject to the constraint that the processor can run at most one job at any given point in time. Provide an algorithm to do this with a running time that is polynomial in n . You may assume for simplicity that no two jobs have the same start or end times.

Example. Consider the following four jobs, specified by (*start-time*, *end-time*) pairs.

(6 P.M., 6 A.M.), (9 P.M., 4 A.M.), (3 A.M., 2 P.M.), (1 P.M., 7 P.M.).

The optimal solution would be to pick the two jobs (9 P.M., 4 A.M.) and (1 P.M., 7 P.M.), which can be scheduled without overlapping.

[DPV] Here's a problem that occurs in automatic program analysis. For a set of variables x_1, \dots, x_n , you are given some *equality* constraints, of the form " $x_i = x_j$ " and some *disequality* constraints, of the form " $x_i \neq x_j$." Is it possible to satisfy all of them?
For instance, the constraints

$$x_1 = x_2, x_2 = x_3, x_3 = x_4, x_1 \neq x_4$$

cannot be satisfied. Give an efficient algorithm that takes as input m constraints over n variables and decides whether the constraints can be satisfied.

[DPV] Shortest paths are not always unique: sometimes there are two or more different paths with the minimum possible length. Show how to solve the following problem in $O((|V| + |E|) \log |V|)$ time.

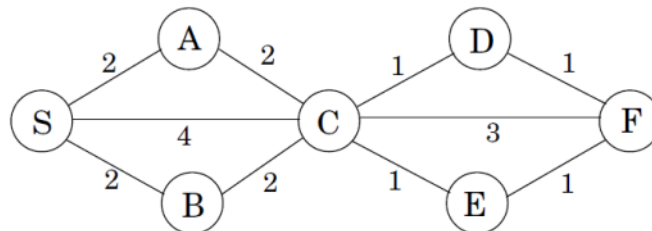
Input: An undirected graph $G = (V, E)$; edge lengths $l_e > 0$; starting vertex $s \in V$.

Output: A Boolean array $\text{usp}[\cdot]$: for each node u , the entry $\text{usp}[u]$ should be true if and only if there is a *unique* shortest path from s to u . (Note: $\text{usp}[s] = \text{true}$.)

[DPV] In cases where there are several different shortest paths between two nodes (and edges have varying lengths), the most convenient of these paths is often *the one with fewest edges*. For instance, if nodes represent cities and edge lengths represent costs of flying between cities, there might be many ways to get from city s to city t which all have the same cost. The most convenient of these alternatives is the one which involves the fewest stopovers. Accordingly, for a specific starting node s , define

$\text{best}[u] = \text{minimum number of edges in a shortest path from } s \text{ to } u$.

In the example below, the best values for nodes S, A, B, C, D, E, F are 0, 1, 1, 1, 2, 2, 3, respectively.



Give an efficient algorithm for the following problem.

Input: Graph $G = (V, E)$; positive edge lengths l_e ; starting node $s \in V$.

Output: The values of $\text{best}[u]$ should be set for *all* nodes $u \in V$.

[DPV] *Generalized shortest-paths problem.* In Internet routing, there are delays on lines but also, more significantly, delays at routers. This motivates a generalized shortest-paths problem.

Suppose that in addition to having edge lengths $\{l_e : e \in E\}$, a graph also has *vertex costs* $\{c_v : v \in V\}$. Now define the cost of a path to be the sum of its edge lengths, *plus* the costs of all vertices on the path (including the endpoints). Give an efficient algorithm for the following problem.

Input: A directed graph $G = (V, E)$; positive edge lengths l_e and positive vertex costs c_v ; a starting vertex $s \in V$.

Output: An array $\text{cost}[\cdot]$ such that for every vertex u , $\text{cost}[u]$ is the least cost of any path from s to u (i.e., the cost of the cheapest path), under the definition above.

Notice that $\text{cost}[s] = c_s$.

[DPV] 5.5. Consider an undirected graph $G = (V, E)$ with nonnegative edge weights $w_e \geq 0$. Suppose that you have computed a minimum spanning tree of G , and that you have also computed shortest paths to all nodes from a particular node $s \in V$.

Now suppose each edge weight is increased by 1: the new weights are $w'_e = w_e + 1$.

- (a) Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.
- (b) Do the shortest paths change? Give an example where they change or prove they cannot change.

[DPV] 5.6. Let $G = (V, E)$ be an undirected graph. Prove that if all its edge weights are distinct, then it has a unique minimum spanning tree.

[DPV] 5.7. Show how to find the *maximum* spanning tree of a graph, that is, the spanning tree of largest total weight.

[KT] 10. Let $G = (V, E)$ be an (undirected) graph with costs $c_e \geq 0$ on the edges $e \in E$. Assume you are given a minimum-cost spanning tree T in G . Now assume that a new edge is added to G , connecting two nodes $v, w \in V$ with cost c .

- (a) Give an efficient algorithm to test if T remains the minimum-cost spanning tree with the new edge added to G (but not to the tree T). Make your algorithm run in time $O(|E|)$. Can you do it in $O(|V|)$ time? Please note any assumptions you make about what data structure is used to represent the tree T and the graph G .
- (b) Suppose T is no longer the minimum-cost spanning tree. Give a linear-time algorithm (time $O(|E|)$) to update the tree T to the new minimum-cost spanning tree.

[CLRS] 23.2-8

Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1 , and let E_2 be the set of edges that are incident only on vertices in V_2 . Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2) , and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Either argue that the algorithm correctly computes a minimum spanning tree of G , or provide an example for which the algorithm fails.

Interesting problems

interesting problems

[DPV] *Graphs with prescribed degree sequences.* Given a list of n positive integers d_1, d_2, \dots, d_n , we want to efficiently determine whether there exists an undirected graph $G = (V, E)$ whose nodes have degrees precisely d_1, d_2, \dots, d_n . That is, if $V = \{v_1, \dots, v_n\}$, then the degree of v_i should be exactly d_i . We call (d_1, \dots, d_n) the *degree sequence* of G . This graph G should not contain self-loops (edges with both endpoints equal to the same node) or multiple edges between the same pair of nodes.

- (a) Give an example of d_1, d_2, d_3, d_4 where all the $d_i \leq 3$ and $d_1 + d_2 + d_3 + d_4$ is even, but for which no graph with degree sequence (d_1, d_2, d_3, d_4) exists.
- (b) Suppose that $d_1 \geq d_2 \geq \dots \geq d_n$ and that there exists a graph $G = (V, E)$ with degree sequence (d_1, \dots, d_n) . We want to show that there must exist a graph that has this degree sequence and where in addition the neighbors of v_1 are $v_2, v_3, \dots, v_{d_1+1}$. The idea is to gradually transform G into a graph with the desired additional property.
 - i. Suppose the neighbors of v_1 in G are not $v_2, v_3, \dots, v_{d_1+1}$. Show that there exists $i < j \leq n$ and $u \in V$ such that $\{v_1, v_i\}, \{u, v_j\} \notin E$ and $\{v_1, v_j\}, \{u, v_i\} \in E$.
 - ii. Specify the changes you would make to G to obtain a new graph $G' = (V, E')$ with the same degree sequence as G and where $(v_1, v_i) \in E'$.
 - iii. Now show that there must be a graph with the given degree sequence but in which v_1 has neighbors $v_2, v_3, \dots, v_{d_1+1}$.
- (c) Using the result from part (b), describe an algorithm that on input d_1, \dots, d_n (not necessarily sorted) decides whether there exists a graph with this degree sequence. Your algorithm should run in time polynomial in n and in $m = \sum_{i=1}^n d_i$.

[CLRS] 16-1 Coin changing

Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

- a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.
- b. Suppose that the available coins are in the denominations that are powers of c , i.e., the denominations are c^0, c^1, \dots, c^k for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.
- c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n .
- d. Give an $O(nk)$ -time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

[CLRS] **23-1 Second-best minimum spanning tree**

Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \rightarrow \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.

We define a second-best minimum spanning tree as follows. Let \mathcal{T} be the set of all spanning trees of G , and let T' be a minimum spanning tree of G . Then a **second-best minimum spanning tree** is a spanning tree T such that $w(T) = \min_{T'' \in \mathcal{T} - \{T'\}} \{w(T'')\}$.

- a. Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.
- b. Let T be the minimum spanning tree of G . Prove that G contains edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G .
- c. Let T be a spanning tree of G and, for any two vertices $u, v \in V$, let $\max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T . Describe an $O(V^2)$ -time algorithm that, given T , computes $\max[u, v]$ for all $u, v \in V$.
- d. Give an efficient algorithm to compute the second-best minimum spanning tree of G .

[CLRS] **23-3 Bottleneck spanning tree**

A **bottleneck spanning tree** T of an undirected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G . We say that the value of the bottleneck spanning tree is the weight of the maximum-weight edge in T .

- a. Argue that a minimum spanning tree is a bottleneck spanning tree.

Part (a) shows that finding a bottleneck spanning tree is no harder than finding a minimum spanning tree. In the remaining parts, we will show how to find a bottleneck spanning tree in linear time.

- b. Give a linear-time algorithm that given a graph G and an integer b , determines whether the value of the bottleneck spanning tree is at most b .
- c. Use your algorithm for part (b) as a subroutine in a linear-time algorithm for the bottleneck-spanning-tree problem. (*Hint:* You may want to use a subroutine that contracts sets of edges, as in the MST-REDUCE procedure described in Problem 23-2.)

Challenging problems (optional)

[CLRS] **16-5 Off-line caching**

Modern computers use a cache to store a small amount of data in a fast memory. Even though a program may access large amounts of data, by storing a small subset of the main memory in the **cache**—a small but faster memory—overall access time can greatly decrease. When a computer program executes, it makes a sequence $\langle r_1, r_2, \dots, r_n \rangle$ of n memory requests, where each request is for a particular data

16-5 Off-line caching

Modern computers use a cache to store a small amount of data in a fast memory. Even though a program may access large amounts of data, by storing a small subset of the main memory in the *cache*—a small but faster memory—overall access time can greatly decrease. When a computer program executes, it makes a sequence $\langle r_1, r_2, \dots, r_n \rangle$ of n memory requests, where each request is for a particular data element. For example, a program that accesses 4 distinct elements $\{a, b, c, d\}$ might make the sequence of requests $\langle d, b, d, b, d, a, c, d, b, a, c, b \rangle$. Let k be the size of the cache. When the cache contains k elements and the program requests the $(k + 1)$ st element, the system must decide, for this and each subsequent request, which k elements to keep in the cache. More precisely, for each request r_i , the cache-management algorithm checks whether element r_i is already in the cache. If it is, then we have a *cache hit*; otherwise, we have a *cache miss*. Upon a cache miss, the system retrieves r_i from the main memory, and the cache-management algorithm must decide whether to keep r_i in the cache. If it decides to keep r_i and the cache already holds k elements, then it must evict one element to make room for r_i . The cache-management algorithm evicts data with the goal of minimizing the number of cache misses over the entire sequence of requests.

Typically, caching is an on-line problem. That is, we have to make decisions about which data to keep in the cache without knowing the future requests. Here, however, we consider the off-line version of this problem, in which we are given in advance the entire sequence of n requests and the cache size k , and we wish to minimize the total number of cache misses.

We can solve this off-line problem by a greedy strategy called *furthest-in-future*, which chooses to evict the item in the cache whose next access in the request sequence comes furthest in the future.

- a.* Write pseudocode for a cache manager that uses the furthest-in-future strategy. The input should be a sequence $\langle r_1, r_2, \dots, r_n \rangle$ of requests and a cache size k , and the output should be a sequence of decisions about which data element (if any) to evict upon each request. What is the running time of your algorithm?
- b.* Show that the off-line caching problem exhibits optimal substructure.
- c.* Prove that furthest-in-future produces the minimum possible number of cache misses.