level[$v$] − level[$w$] + 1, where the level of a vertex is its distance in the DFS tree from the root vertex. This suggests the following algorithm:

- *Do a depth-first search, keeping track of the level of each vertex.*
- *Each time a back edge is encountered, compute the cycle length and save it if it is smaller than the shortest one previously seen.*

Show that this strategy does not always work by providing a counterexample as well as a brief (one or two sentence) explanation.

4.5. Often there are multiple shortest paths between two nodes of a graph. Give a linear-time algorithm for the following task.

*Input:* Undirected graph $G = (V, E)$ with unit edge lengths; nodes $u, v \in V$.
*Output:* The number of distinct shortest paths from $u$ to $v$.

4.6. Prove that for the array `prev` computed by Dijkstra's algorithm, the edges $\{u, \texttt{prev}[u]\}$ (for all $u \in V$) form a tree.

4.7. You are given a directed graph $G = (V, E)$ with (possibly negative) weighted edges, along with a specific node $s \in V$ and a tree $T = (V, E')$, $E' \subseteq E$. Give an algorithm that checks whether $T$ is a shortest-path tree for $G$ with starting point $s$. Your algorithm should run in linear time.

4.8. Professor F. Lake suggests the following algorithm for finding the shortest path from node $s$ to node $t$ in a directed graph with some negative edges: add a large constant to each edge weight so that all the weights become positive, then run Dijkstra's algorithm starting at node $s$, and return the shortest path found to node $t$.

Is this a valid method? Either prove that it works correctly, or give a counterexample.

4.9. Consider a directed graph in which the only negative edges are those that leave $s$; all other edges are positive. Can Dijkstra's algorithm, started at $s$, fail on such a graph? Prove your answer.

4.10. You are given a directed graph with (possibly negative) weighted edges, in which the shortest path between any two vertices is guaranteed to have at most $k$ edges. Give an algorithm that finds the shortest path between two vertices $u$ and $v$ in $O(k|E|)$ time.

4.11. Give an algorithm that takes as input a directed graph with positive edge lengths, and returns the length of the shortest cycle in the graph (if the graph is acyclic, it should say so). Your algorithm should take time at most $O(|V|^3)$.

4.12. Give an $O(|V|^2)$ algorithm for the following task.

*Input:* An undirected graph $G = (V, E)$; edge lengths $l_e > 0$; an edge $e \in E$.
*Output:* The length of the shortest cycle containing edge $e$.

4.13. You are given a set of cities, along with the pattern of highways between them, in the form of an undirected graph $G = (V, E)$. Each stretch of highway $e \in E$ connects two of the cities, and you know its length in miles, $l_e$. You want to get from city $s$ to city $t$. There's one problem: your car can only hold enough gas to cover $L$ miles. There are gas stations in each city, but not between cities. Therefore, you can only take a route if every one of its edges has length $l_e \leq L$.

(a) Given the limitation on your car's fuel tank capacity, show how to determine in linear time whether there is a feasible route from $s$ to $t$.

(b) You are now planning to buy a new car, and you want to know the minimum fuel tank capacity that is needed to travel from $s$ to $t$. Give an $O((|V| + |E|) \log |V|)$ algorithm to determine this.

4.14. You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $v_0 \in V$. Give an efficient algorithm for finding shortest paths between *all pairs of nodes*, with the one restriction that these paths must all pass through $v_0$.

4.15. 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 `usp`[·]: for each node $u$, the entry `usp`[$u$] should be `true` if and only if there is a *unique* shortest path from $s$ to $u$. (Note: `usp`[$s$] = `true`.)

4.16. Section 4.5.2 describes a way of storing a complete binary tree of $n$ nodes in an array indexed by $1, 2, \ldots, n$.

(a) Consider the node at position $j$ of the array. Show that its parent is at position $\lfloor j/2 \rfloor$ and its children are at $2j$ and $2j + 1$ (if these numbers are $\leq n$).

(b) What are the corresponding indices when a complete $d$-ary tree is stored in an array?

Figure 4.16 shows pseudocode for a binary heap, modeled on an exposition by R. E. Tarjan.[2] The heap is stored as an array $h$, which is assumed to support two constant-time operations:

- $|h|$, which returns the number of elements currently in the array;
- $h^{-1}$, which returns the position of an element within the array.

The latter can always be achieved by maintaining the values of $h^{-1}$ as an auxiliary array.

(c) Show that the `makeheap` procedure takes $O(n)$ time when called on a set of $n$ elements. What is the worst-case input? (*Hint:* Start by showing that the running time is at most $\sum_{i=1}^{n} \log(n/i)$.)

(d) What needs to be changed to adapt this pseudocode to $d$-ary heaps?

[2]See: R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, 1983.

**Figure 4.16** Operations on a binary heap.

```
procedure insert($h, x$)
bubbleup($h, x, |h| + 1$)

procedure decreasekey($h, x$)
bubbleup($h, x, h^{-1}(x)$)

function deletemin($h$)
if $|h| = 0$:
    return null
else:
    $x = h(1)$
    siftdown($h, h(|h|), 1$)
    return $x$

function makeheap($S$)
$h =$ empty array of size $|S|$
for $x \in S$:
    $h(|h| + 1) = x$
for $i = |S|$ downto 1:
    siftdown($h, h(i), i$)
return $h$

procedure bubbleup($h, x, i$)
(place element $x$ in position $i$ of $h$, and let it bubble up)
$p = \lceil i/2 \rceil$
while $i \neq 1$ and key($h(p)$) $>$ key($x$):
    $h(i) = h(p)$;  $i = p$;  $p = \lceil i/2 \rceil$
$h(i) = x$

procedure siftdown($h, x, i$)
(place element $x$ in position $i$ of $h$, and let it sift down)
$c =$ minchild($h, i$)
while $c \neq 0$ and key($h(c)$) $<$ key($x$):
    $h(i) = h(c)$;  $i = c$;  $c =$ minchild($h, i$)
$h(i) = x$

function minchild($h, i$)
(return the index of the smallest child of $h(i)$)
if $2i > |h|$:
    return 0 (no children)
else:
    return $\arg\min\{\text{key}(h(j)) : 2i \le j \le \min\{|h|, 2i + 1\}\}$
```

4.17. Suppose we want to run Dijkstra's algorithm on a graph whose edge weights are integers in the range $0, 1, \ldots, W$, where $W$ is a relatively small number.

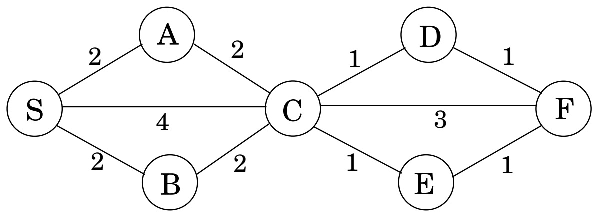
(a) Show how Dijkstra's algorithm can be made to run in time $O(W|V| + |E|)$.

(b) Show an alternative implementation that takes time just $O((|V| + |E|) \log W)$.

4.18. 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

`best`$[u]$ = minimum number of edges in a shortest path from $s$ 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 `best`$[u]$ should be set for *all* nodes $u \in V$.

4.19. *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 `cost`$[\cdot]$ such that for every vertex $u$, `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 `cost`$[s] = c_s$.

4.20. There is a network of roads $G = (V, E)$ connecting a set of cities $V$. Each road in $E$ has an associated length $l_e$. There is a proposal to add one new road to this network, and there is a list $E'$ of pairs of cities between which the new road can be built. Each such potential road $e' \in E'$ has an associated length. As a designer for the public works department you are asked to determine the road $e' \in E'$ whose addition to the existing network $G$ would result in the maximum decrease in the driving distance between two fixed cities $s$ and $t$ in the network. Give an efficient algorithm for solving this problem.

4.21. Shortest path algorithms can be applied in currency trading. Let $c_1, c_2, \ldots, c_n$ be various currencies; for instance, $c_1$ might be dollars, $c_2$ pounds, and $c_3$ lire. For any two currencies $c_i$ and $c_j$, there is an exchange rate $r_{i,j}$; this means that you can purchase $r_{i,j}$ units of currency $c_j$ in exchange for one unit of $c_i$. These exchange rates satisfy the condition that $r_{i,j} \cdot r_{j,i} < 1$, so that if you start with a unit of currency $c_i$, change it into currency $c_j$ and then convert back to currency $c_i$, you end up with less than one unit of currency $c_i$ (the difference is the cost of the transaction).

(a) Give an efficient algorithm for the following problem: Given a set of exchange rates $r_{i,j}$, and two currencies $s$ and $t$, find the most advantageous sequence of currency exchanges for converting currency $s$ into currency $t$. Toward this goal, you should represent the currencies and rates by a graph whose edge lengths are real numbers.

The exchange rates are updated frequently, reflecting the demand and supply of the various currencies. Occasionally the exchange rates satisfy the following property: there is a sequence of currencies $c_{i_1}, c_{i_2}, \ldots, c_{i_k}$ such that $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdots r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$. This means that by starting with a unit of currency $c_{i_1}$ and then successively converting it to currencies $c_{i_2}, c_{i_3}, \ldots, c_{i_k}$, and finally back to $c_{i_1}$, you would end up with more than one unit of currency $c_{i_1}$. Such anomalies last only a fraction of a minute on the currency exchange, but they provide an opportunity for risk-free profits.

(b) Give an efficient algorithm for detecting the presence of such an anomaly. Use the graph representation you found above.

4.22. *The tramp steamer problem.* You are the owner of a steamship that can ply between a group of port cities $V$. You make money at each port: a visit to city $i$ earns you a profit of $p_i$ dollars. Meanwhile, the transportation cost from port $i$ to port $j$ is $c_{ij} > 0$. You want to find a cyclic route in which the ratio of profit to cost is maximized.

To this end, consider a directed graph $G = (V, E)$ whose nodes are ports, and which has edges between each pair of ports. For any cycle $C$ in this graph, the profit-to-cost ratio is

$$r(C) = \frac{\sum_{(i,j)\in C} p_j}{\sum_{(i,j)\in C} c_{ij}}.$$

Let $r^*$ be the maximum ratio achievable by a simple cycle. One way to determine $r^*$ is by binary search: by first guessing some ratio $r$, and then testing whether it is too large or too small.

Consider any positive $r > 0$. Give each edge $(i, j)$ a weight of $w_{ij} = rc_{ij} - p_j$.

(a) Show that if there is a cycle of negative weight, then $r < r^*$.

(b) Show that if all cycles in the graph have strictly positive weight, then $r > r^*$.

(c) Give an efficient algorithm that takes as input a desired accuracy $\epsilon > 0$ and returns a simple cycle $C$ for which $r(C) \geq r^* - \epsilon$. Justify the correctness of your algorithm and analyze its running time in terms of $|V|$, $\epsilon$, and $R = \max_{(i,j)\in E}(p_j/c_{ij})$.