Algorithms

FOURTH EDITION

PART II

ROBERT SEDGEWICK | KEVIN WAYNE

Algorithms

FOURTH EDITION

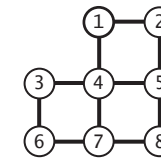
PART II

Q&A

Q. Do Prim's and Kruskal's algorithms work for *directed* graphs?

A. No, not at all. That is a more difficult graph-processing problem known as the *minimum cost arborescence* problem.

EXERCISES



4.3.1 Prove that you can rescale the weights by adding a positive constant to all of them or by multiplying them all by a positive constant without affecting the MST.

4.3.2 Draw all of the MSTs of the graph depicted at right (all edge weights are equal).

4.3.3 Show that if a graph's edges all have distinct weights, the MST is unique.

4.3.4 Consider the assertion that an edge-weighted graph has a unique MST *only* if its edge weights are distinct. Give a proof or a counterexample.

4.3.5 Show that the greedy algorithm is valid even when edge weights are not distinct.

4.3.6 Give the MST of the weighted graph obtained by deleting vertex `7` from `tinyEWG.txt` (see page 604).

4.3.7 How would you find a *maximum* spanning tree of an edge-weighted graph?

4.3.8 Prove the following, known as the *cycle property*: Given any cycle in an edge-weighted graph (all edge weights distinct), the edge of maximum weight in the cycle does not belong to the MST of the graph.

4.3.9 Implement the constructor for `EdgeWeightedGraph` that reads an edge-weighted graph from the input stream, by suitably modifying the constructor from `Graph` (see page 526).

4.3.10 Develop an `EdgeWeightedGraph` implementation for dense graphs that uses an adjacency-matrix (two-dimensional array of weights) representation. Disallow parallel edges.

4.3.11 Determine the amount of memory used by `EdgeWeightedGraph` to represent a graph with V vertices and E edges, using the memory-cost model of SECTION 1.4.

4.3.12 Suppose that a graph has distinct edge weights. Does its lightest edge have to belong to the MST? Can its heaviest edge belong to the MST? Does a min-weight edge on every cycle have to belong to the MST? Prove your answer to each question or give a counterexample.

4.3.13 Give a counterexample that shows why the following strategy does not necessarily find the MST: 'Start with any vertex as a single-vertex MST, then add `V-1` edges

EXERCISES *(continued)*

to it, always taking next a min-weight edge incident to the vertex most recently added to the MST.'

4.3.14 Given an MST for an edge-weighted graph G, suppose that an edge in G that does not disconnect G is deleted. Describe how to find an MST of the new graph in time proportional to E.

4.3.15 Given an MST for an edge-weighted graph G and a new edge e with weight w, describe how to find an MST of the new graph in time proportional to V.

4.3.16 Given an MST for an edge-weighted graph G and a new edge e, write a program that determines the range of weights for which e is in an MST.

4.3.17 Implement `toString()` for `EdgeWeightedGraph`.

4.3.18 Give traces that show the process of computing the MST of the graph defined in EXERCISE 4.3.6 with the lazy version of Prim's algorithm, the eager version of Prim's algorithm, and Kruskal's algorithm.

4.3.19 Suppose that you implement `PrimMST` but instead of using a priority queue to find the next vertex to add to the tree, you scan through all V entries in the `distTo[]` array to find the non-tree vertex with the smallest weight. What would be the order of growth of the running time for graphs with V vertices and E edges? When would this method be appropriate, if ever? Defend your answer.

4.3.20 True or false: At any point during the execution of Kruskal's algorithm, each vertex is closer to some vertex in its subtree than to any vertex not in its subtree. Prove your answer.

4.3.21 Provide an implementation of `edges()` for `PrimMST` (page 622).

Solution:

```
public Iterable<Edge> edges()
{
   Queue<Edge> mst = new Queue<Edge>();
   for (int v = 1; v < edgeTo.length; v++)
      mst.enqueue(edgeTo[v]);
   return mst;
}
```

CREATIVE PROBLEMS

4.3.22 *Minimum spanning forest.* Develop versions of Prim's and Kruskal's algorithms that compute the minimum spanning *forest* of an edge-weighted graph that is not necessarily connected. Use the connected-components API of SECTION 4.1 and find MSTs in each component.

4.3.23 *Vyssotsky's algorithm.* Develop an implementation that computes the MST by applying the cycle property (see EXERCISE 4.3.8) repeatedly: Add edges one at a time to a putative tree, deleting a maximum-weight edge on the cycle if one is formed. *Note*: This method has received less attention than the others that we consider because of the comparative difficulty of maintaining a data structure that supports efficient implementation of the "delete the maximum-weight edge on the cycle" operation.

4.3.24 *Reverse-delete algorithm.* Develop an implementation that computes the MST as follows: Start with a graph containing all of the edges. Then repeatedly go through the edges in decreasing order of weight. For each edge, check if deleting that edge will disconnect the graph; if not, delete it. Prove that this algorithm computes the MST. What is the order of growth of the number of edge-weight compares performed by your implementation?

4.3.25 *Worst-case generator.* Develop a reasonable generator for edge-weighted graphs with V vertices and E edges such that the running time of the lazy version of Prim's algorithm is nonlinear. Answer the same question for the eager version.

4.3.26 *Critical edges.* An MST edge whose deletion from the graph would cause the MST weight to increase is called a *critical edge*. Show how to find all critical edges in a graph in time proportional to $E \log E$. *Note*: This question assumes that edge weights are not necessarily distinct (otherwise all edges in the MST are critical).

4.3.27 *Animations.* Write a client program that does dynamic graphical animations of MST algorithms. Run your program for `mediumEWG.txt` to produce images like the figures on page 621 and page 624.

4.3.28 *Space-efficient data structures.* Develop an implementation of the lazy version of Prim's algorithm that saves space by using lower-level data structures for `EdgeWeightedGraph` and for `MinPQ` instead of `Bag` and `Edge`. Estimate the amount of memory saved as a function of V and E, using the memory-cost model of SECTION 1.4 (see EXERCISE 4.3.11).

CREATIVE PROBLEMS *(continued)*

4.3.29 *Dense graphs.* Develop an implementation of Prim's algorithm that uses an eager approach (but not a priority queue) and computes the MST using V^2 edge-weight comparisons.

4.3.30 *Euclidean weighted graphs.* Modify your solution to EXERCISE 4.1.37 to create an API `EuclideanEdgeWeightedGraph` for graphs whose vertices are points in the plane, so that you can work with graphical representations.

4.3.31 *MST weights.* Develop implementations of `weight()` for `LazyPrimMST`, `PrimMST`, and `KruskalMST`, using a *lazy* strategy that iterates through the MST edges when the client calls `weight()`. Then develop alternate implementations that use an *eager* strategy that maintains a running total as the MST is computed.

4.3.32 *Specified set.* Given a connected edge-weighted graph G and a specified set of edges S (having no cycles), describe a way to find a minimum-weight spanning tree of G among those spanning trees that contain all the edges in S.

4.3.33 *Certification.* Write an `MST` and `EdgeWeightedGraph` client `check()` that uses the following *cut optimality conditions* implied by PROPOSITION J to verify that a proposed set of edges is in fact an MST: A set of edges is an MST if it is a spanning tree and every edge is a minimum-weight edge in the cut defined by removing that edge from the tree. What is the order of growth of the running time of your method?

EXPERIMENTS

4.3.34 *Random sparse edge-weighted graphs.* Write a random-sparse-edge-weighted-graph generator based on your solution to EXERCISE 4.1.41. To assign edge weights, define a random-edge-weighted graph ADT and write two implementations: one that generates uniformly distributed weights, another that generates weights according to a Gaussian distribution. Write client programs to generate sparse random edge-weighted graphs for both weight distributions with a well-chosen set of values of V and E so that you can use them to run empirical tests on graphs drawn from various distributions of edge weights.

4.3.35 *Random Euclidean edge-weighted graphs.* Modify your solution to EXERCISE 4.1.42 to assign the distance between vertices as each edge's weight.

4.3.36 *Random grid edge-weighted graphs.* Modify your solution to EXERCISE 4.1.43 to assign a random weight (between 0 and 1) to each edge.

4.3.37 *Real edge-weighted graphs.* Find a large weighted graph somewhere online—perhaps a map with distances, telephone connections with costs, or an airline rate schedule. Write a program `RandomRealEdgeWeightedGraph` that builds a weighted graph by choosing V vertices at random and E weighted edges at random from the subgraph induced by those vertices.

Testing all algorithms and studying all parameters against all graph models is unrealistic. For each problem listed below, write a client that addresses the problem for any given input graph, then choose among the generators above to run experiments for that graph model. Use your judgment in selecting experiments, perhaps in response to results of previous experiments. Write a narrative explaining your results and any conclusions that might be drawn.

4.3.38 *Cost of laziness.* Run empirical studies to compare the performance of the lazy version of Prim's algorithm with the eager version, for various types of graphs.

4.3.39 *Prim versus Kruskal.* Run empirical studies to compare the performance of the lazy and eager versions of Prim's algorithm with Kruskal's algorithm.

4.3.40 *Reduced overhead.* Run empirical studies to determine the effect of using primitive types instead of `Edge` values in `EdgeWeightedGraph`, as described in EXERCISE 4.3.28.