Algorithms

THIRD EDITION

IN C++

Part 5

GRAPH ALGORITHMS

ROBERT SEDGEWICK

with C++ consulting by Christopher J. Van Wyk

Algorithms THIRD EDITION in C++

PART 5
GRAPH ALGORITHMS

Robert Sedgewick
Princeton University

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Program 18.10 Generalized graph search

This graph-search class generalizes BFS and DFS and supports numerous graph-processing algorithms (see Section 21.2 for a discussion of these algorithms and alternate implementations). It maintains a generalized queue of edges called the *fringe*. We initialize the fringe with a self-loop to the start vertex; then, while the fringe is not empty, we move an edge `e` from the fringe to the tree (attached at `e.v`) and scan `e.w`'s adjacency list, moving unseen vertices to the fringe and calling `update` for new edges to fringe vertices.

This code makes judicious use of `ord` and `st` to guarantee that no two edges on the fringe point to the same vertex. A vertex `v` is the destination vertex of a fringe edge if and only if it is marked (`ord[v]` is not `-1`) but it is not yet on the tree (`st[v]` is `-1`).

```
#include "GQ.cc"
template <class Graph>
class PFS : public SEARCH<Graph>
{ vector<int> st;
  void searchC(Edge e)
  { GQ<Edge> Q(G.V());
    Q.put(e); ord[e.w] = cnt++;
    while (!Q.empty())
      {
        e = Q.get(); st[e.w] = e.v;
        typename Graph::adjIterator A(G, e.w);
        for (int t = A.beg(); !A.end(); t = A.nxt())
          if (ord[t] == -1)
            { Q.put(Edge(e.w, t)); ord[t] = cnt++; }
          else
            if (st[t] == -1) Q.update(Edge(e.w, t));
      }
  }
public:
  PFS(Graph &G) : SEARCH<Graph>(G), st(G.V(), -1)
    { search(); }
  int ST(int v) const { return st[v]; }
};
```

Program 18.11 Random queue implementation

When we remove an item from this data structure, it is equally likely to be any one of the items currently in the data structure. We can use this code to implement the generalized-queue ADT for graph searching to search a graph in a "random" fashion (*see text*).

```
template <class Item>
class GQ
  {
    private:
      vector<Item> s; int N;
    public:
      GQ(int maxN) : s(maxN+1), N(0) { }
      int empty() const
        { return N == 0; }
      void put(Item item)
        { s[N++] = item; }
      void update(Item x) { }
      Item get()
        { int i = int(N*rand()/(1.0+RAND_MAX));
          Item t = s[i];
          s[i] = s[N-1];
          s[N-1] = t;
          return s[--N]; }
  };
```

we take them off. The disadvantage of this approach, as with BFS, is that the maximum queue size has to be E instead of V. Or, we could handle updates implicitly in the ADT implementation, just by specifying that no two edges with the same destination vertex can be on the queue. But the simplest way for the ADT implementation to do so is essentially equivalent to using a vertex-indexed vector (see Exercises 4.51 and 4.54), so the test fits more comfortably into the client graph-search program.

The combination of Program 18.10 and the generalized-queue abstraction gives us a general and flexible graph-search mechanism. To illustrate this point, we now consider briefly two interesting and useful alternatives to BFS and DFS.

The first alternative strategy is based on *randomized queues* (see Section 4.6). In a randomized queue, we *remove* items randomly: Each item on the data structure is equally likely to be the one removed. Program 18.11 is an implementation that provides this functionality. If we use this code to implement the generalized queue ADT for Program 18.10, then we get a randomized graph-searching algorithm, where each vertex on the fringe is equally likely to be the next one added to the tree. The edge (to that vertex) that is added to the tree depends on the implementation of the *update* operation. The implementation in Program 18.11 does no updates, so each fringe vertex is added to the tree with the edge that caused it to be moved to the fringe. Alternatively, we might choose to always do updates (which results in the most recently encountered edge to each fringe vertex being added to the tree), or to make a random choice.

Another strategy, which is critical in the study of graph-processing algorithms because it serves as the basis for a number of the classical algorithms that we address in Chapters 20 through 22, is to use a *priority-queue* ADT (see Chapter 9) for the fringe: We assign priority values to each edge on the fringe, update them as appropriate, and choose the highest-priority edge as the one to be added next to the tree. We consider this formulation in detail in Chapter 20. The queue-maintenance operations for priority queues are more costly than are those for stacks and queues because they involve implicit comparisons among items on the queue, but they can support a much broader class of graph-search algorithms. As we shall see, several critical graph-processing problems can be addressed simply with judicious choice of priority assignments in a priority-queue–based generalized graph search.

All generalized graph-searching algorithms examine each edge just once and take extra space proportional to V in the worst case; they do differ, however, in some performance measures. For example, Figure 18.28 shows the size of the fringe as the search progresses for DFS, BFS, and randomized search; Figure 18.29 shows the tree computed by randomized search for the same example as Figure 18.13 and Figure 18.24. Randomized search has neither the long paths of DFS nor the high-degree nodes of BFS. The shapes of these trees and the fringe plots depend on the structure of the particular graph being searched, but they also characterize the different algorithms.

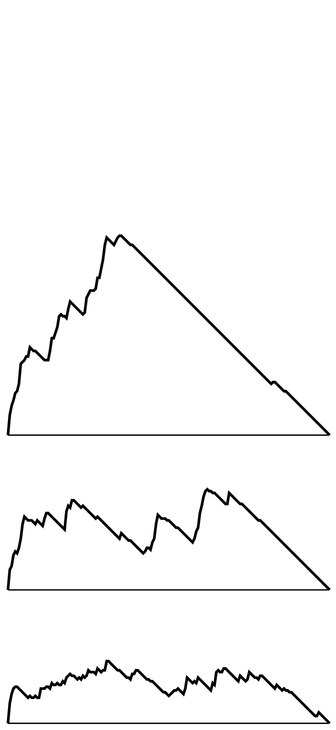
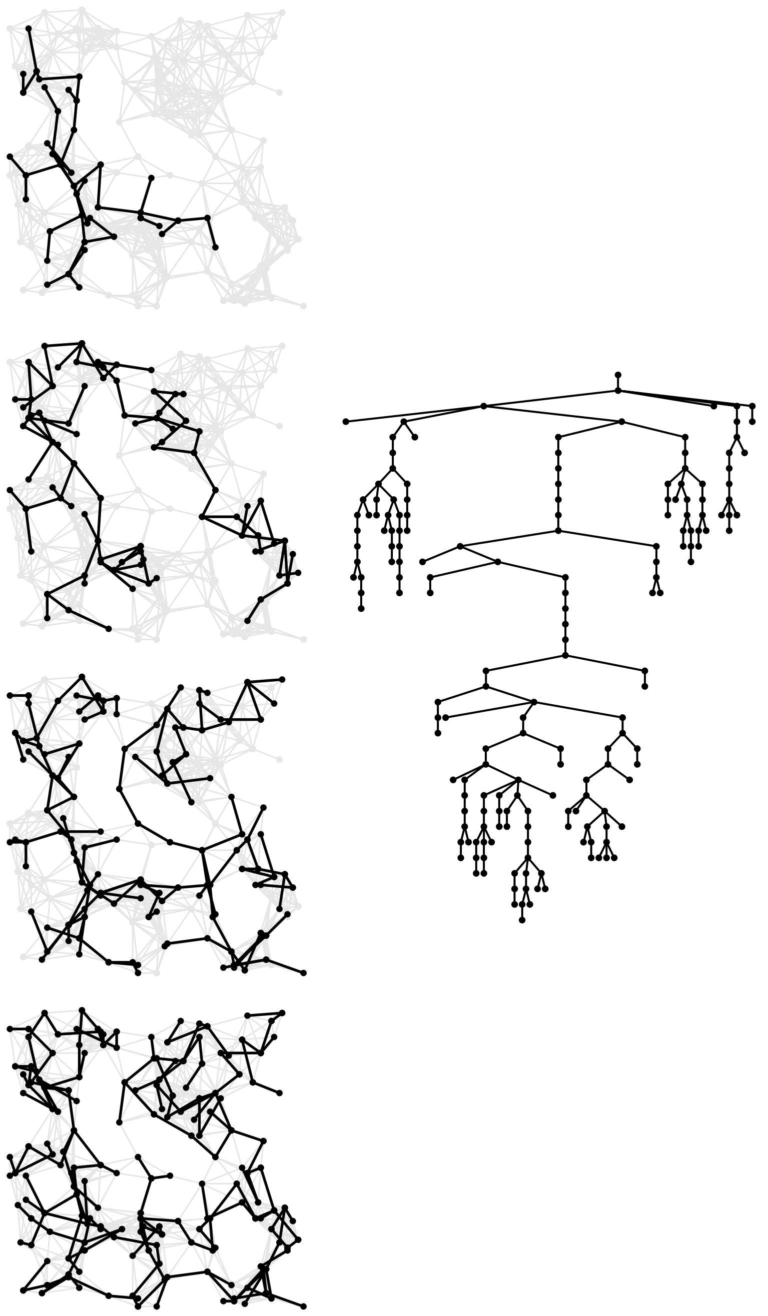


Figure 18.28
Fringe sizes for DFS, randomized graph search, and BFS

These plots of the fringe size during the searches illustrated in Figures 18.13, 18.24, and 18.29 indicate the dramatic effects that the choice of data structure for the fringe can have on graph searching. When we use a stack, in DFS (top)*, we fill up the fringe early in the search as we find new nodes at every step, then we end the search by removing everything. When we use a randomized queue* (center)*, the maximum queue size is much lower. When we use a FIFO queue in BFS* (bottom)*, the maximum queue size is still lower, and we discover new nodes throughout the search.*

Figure 18.29
Randomized graph search

This figure illustrates the progress of randomized graph searching (left)*, in the same style as Figures 18.13 and 18.24. The search tree shape falls somewhere between the BFS and DFS shapes. The dynamics of these three algorithms, which differ only in the data structure for work to be completed, could hardly be more different.*



We could generalize graph searching still further by working with a forest (not necessarily a tree) during the search. Although we stop short of working at this level of generality throughout, we consider a few algorithms of this sort in Chapter 20.

Exercises

○ **18.61** Discuss the advantages and disadvantages of a generalized graph-searching implementation that is based on the following policy: "Move an edge from the fringe to the tree. If the vertex that it leads to is unvisited, visit that vertex and put all its incident edges onto the fringe."

18.62 Develop an adjacency-lists ADT implementation that keeps edges (not just destination vertices) on the lists, then implement a graph search based on the strategy described in Exercise 18.61 that visits every edge but destroys the graph, taking advantage of the fact that you can move all of a vertex's edges to the fringe with a single link change.

• **18.63** Prove that recursive DFS (Program 18.3) is equivalent to generalized graph search using a stack (Program 18.10), in the sense that both programs will visit all vertices in precisely the same order for all graphs if and only if the programs scan the adjacency lists in opposite orders.

18.64 Give three different possible traversal orders for randomized search through the graph

```
3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4.
```

18.65 Could randomized search visit the vertices in the graph

```
3-7 1-4 7-8 0-5 5-2 3-8 2-9 0-6 4-9 2-6 6-4
```

in numerical order of their indices? Prove your answer.

18.66 Use the STL to build a generalized-queue implementation for graph edges that disallows edges with duplicate vertices on the queue, using an ignore-the-new-item policy.

○ **18.67** Develop a randomized graph search that chooses each *edge* on the fringe with equal likelihood. *Hint*: See Program 18.8.

○ **18.68** Describe a maze-traversal strategy that corresponds to using a standard pushdown stack for generalized graph searching (see Section 18.1).

○ **18.69** Instrument generalized graph searching (see Program 18.10) to print out the height of the tree and the percentage of edges processed for every vertex to be seen.

• **18.70** Run experiments to determine empirically the average values of the quantities described in Exercise 18.69 for generalized graph search with a random queue in graphs of various sizes, drawn from various graph models (see Exercises 17.64–76).

• 18.71 Implement a derived class that does dynamic graphical animations of generalized graph search for graphs that have (x, y) coordinates associated with each vertex (see Exercises 17.55 through 17.59). Test your program on random Euclidean neighbor graphs, using as many points as you can process in a reasonable amount of time. Your program should produce images like the snapshots shown in Figures 18.13, 18.24, and 18.29, although you should feel free to use colors instead of shades of gray to denote tree, fringe, and unseen vertices and edges.

18.9 Analysis of Graph Algorithms

We have for our consideration a broad variety of graph-processing problems and methods for solving them, so we do not always compare numerous different algorithms for the same problem, as we have in other domains. Still, it is always valuable to gain experience with our algorithms by testing them on real data, or on artificial data that we understand and that have relevant characteristics that we might expect to find in actual applications.

As we discussed briefly in Chapter 2, we seek—ideally—natural input models that have three critical properties:

- They reflect reality to a sufficient extent that we can use them to predict performance.
- They are sufficiently simple that they are amenable to mathematical analysis.
- We can write generators that provide problem instances that we can use to test our algorithms.

With these three components, we can enter into a design-analysis-implementation-test scenario that leads to efficient algorithms for solving practical problems.

For domains such as sorting and searching, we have seen spectacular success along these lines in Parts 3 and 4. We can analyze algorithms, generate random problem instances, and refine implementations to provide extremely efficient programs for use in a host of practical situations. For some other domains that we study, various difficulties can arise. For example, mathematical analysis at the level that we would like is beyond our reach for many geometric problems, and developing an accurate model of the input is a significant challenge for many string-processing algorithms (indeed, doing so is an essential part of the computation). Similarly, graph algorithms take us to a