Week8

Graphs

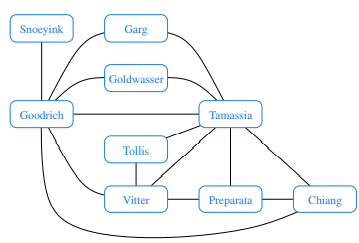
A **graph** is a way of representing relationships that exist between pairs of objects. That is, a graph is a set of objects, called **vertices**, together with a collection of pairwise connections between them, called **edges**.

[Some books use different terminology for graphs and refer to what we call **vertices** as **nodes** and what we call **edges** as **arcs**.]

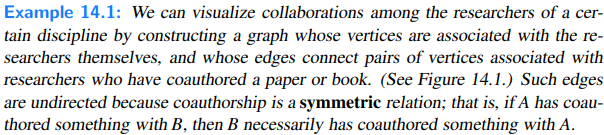
Edges in a graph are either **directed** or **undirected**.*[digraph = directed graph]*

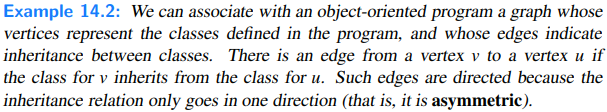
A graph that has both directed and undirected edges is often called a **mixed graph**.

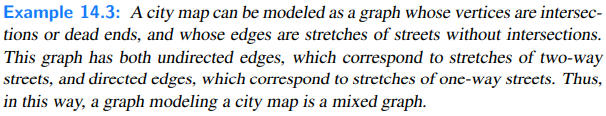
Example: Graph of co-authorship among some authors:

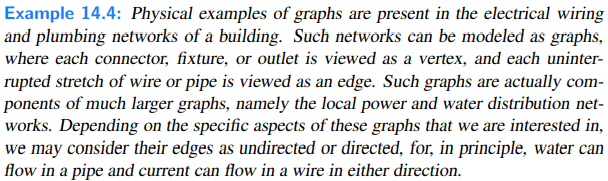


More Examples [Hogy megismerjem, ha a feladatban graph-ot kell hasznalni]:

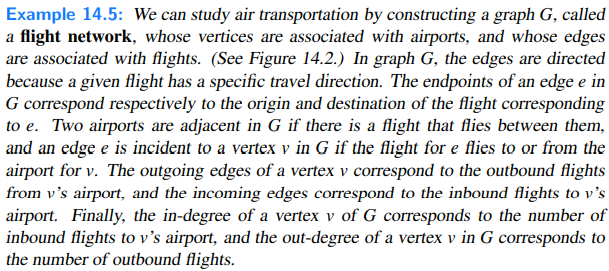


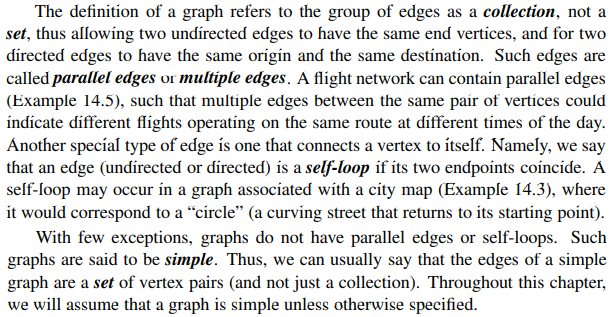


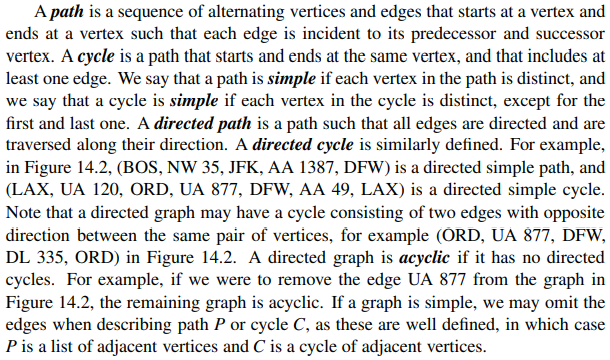


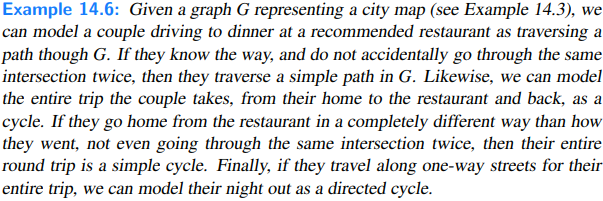


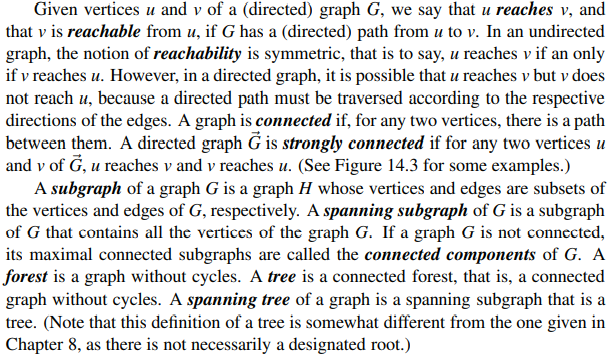
The two vertices joined by an edge are called the end vertices (or endpoints) of the edge. If an edge is directed, its first endpoint is its origin and the other is the destination of the edge. Two vertices u and v are said to be adjacent if there is an edge whose end vertices are u and v. An edge is said to be incident to a vertex if the vertex is one of the edge’s endpoints. The outgoing edges of a vertex are the directed edges whose origin is that vertex. The incoming edges of a vertex are the directed edges whose destination is that vertex. The degree of a vertex v, denoted deg(v), is the number of incident edges of v. The in-degree and out-degree of a vertex v are the number of the incoming and outgoing edges of v, and are denoted indeg(v) and outdeg(v), respectively.



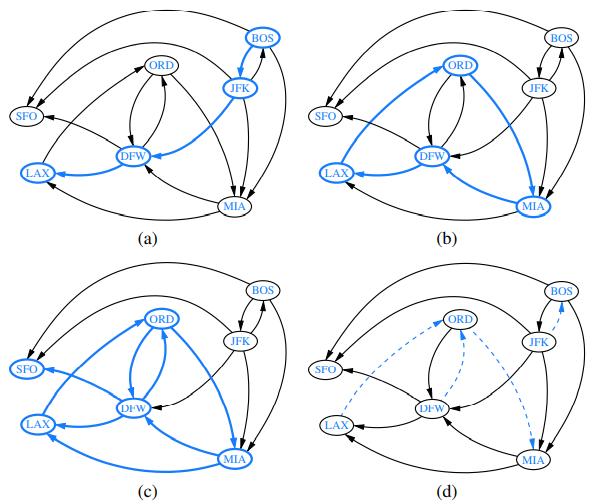


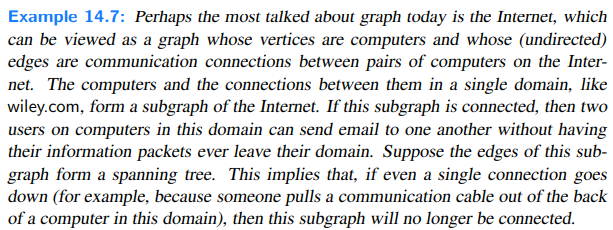




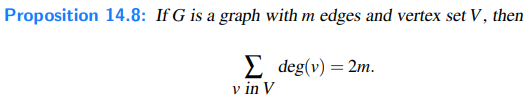


Below: Examples of reachability in a directed graph: (a) a directed path from BOS to LAX is highlighted; (b) a directed cycle (ORD, MIA, DFW, LAX, ORD) is highlighted; its vertices induce a strongly connected subgraph; (c) the subgraph of the vertices and edges reachable from ORD is highlighted; (d) the removal of the dashed edges results in a directed acyclic graph.





Some Important Properties of Graphs



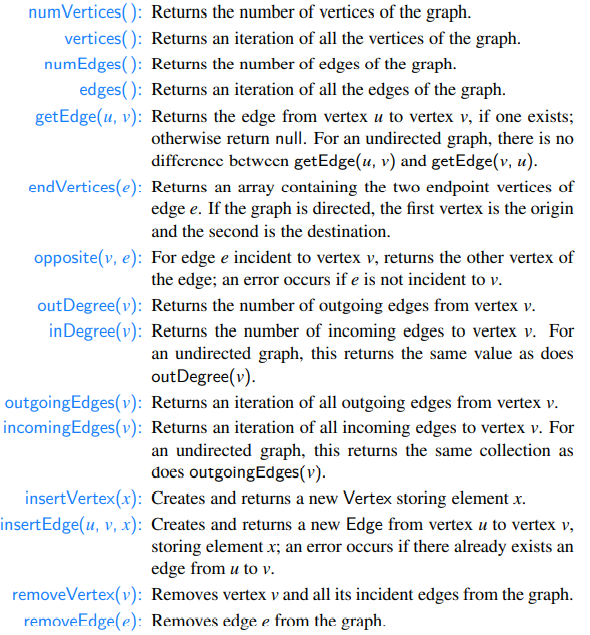






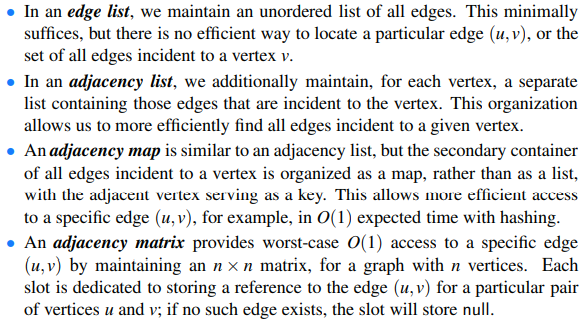
The Graph ADT

Methods:



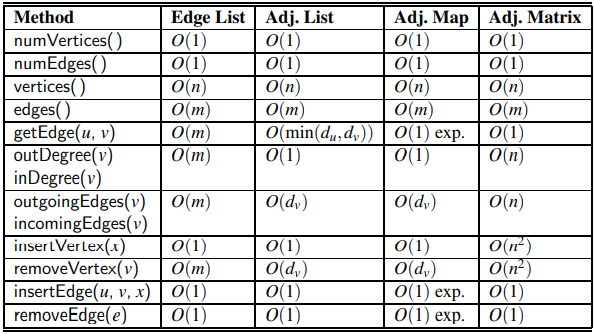
Data Structures for Graphs

In this section, we introduce four data structures for representing a graph. In each representation, we maintain a collection to store the vertices of a graph. However, the four representations differ greatly in the way they organize the edges.



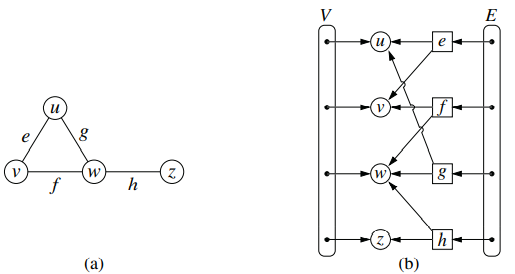
Running Times

A summary of the running times for the methods of the graph ADT, using the graph representations discussed in this section. We let n denote the number of vertices, m the number of edges, and dv the degree of vertex v. Note that the adjacency matrix uses O(n2) space, while all other structures use O(n+m) space.



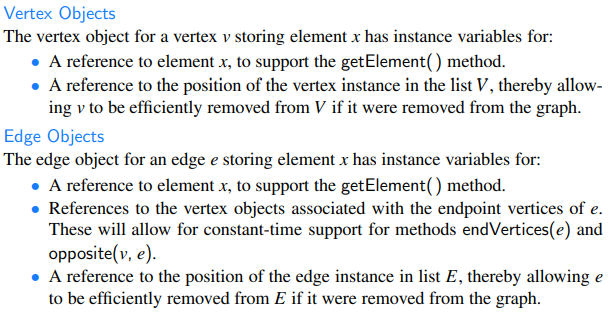
Edge List Structure

The edge list structure is possibly the simplest, though not the most efficient, representation of a graph G. All vertex objects are stored in an unordered list V, and all edge objects are stored in an unordered list E. We illustrate an example of the edge list structure for a graph G here:



1. A graph G; (b) schematic representation of the edge list structure for G. Notice that an edge object refers to the two vertex objects that correspond to its endpoints, but that vertices do not refer to incident edges.

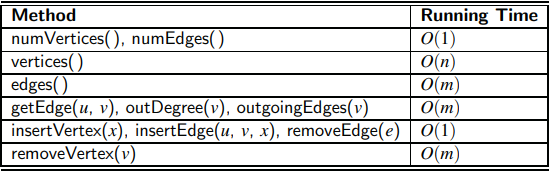
Collections V and E are represented with doubly linked lists using our LinkedPositionalList class.



Performance of the Edge List Structure

The space usage is O(n + m) for representing a graph with n vertices and m edges. Each individual vertex or edge instance uses O(1) space, and the additional lists V and E use space proportional to their number of entries.

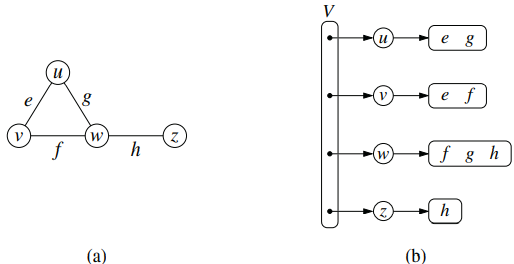
Running Times:



Adjacency List Structure

The adjacency list structure for a graph adds extra information to the edge list structure that supports direct access to the incident edges (and thus to the adjacent vertices) of each vertex. Specifically, for each vertex v, we maintain a collection I(v), called the **incidence collection** of v, whose entries are edges incident to v. In the case of a directed graph, outgoing and incoming edges can be respectively stored in two separate collections, Iout(v) and Iin(v). Traditionally, the incidence collection I(v) for a vertex v is a list, which is why we call this way of representing a graph the **adjacency list** structure.

The primary benefit of an adjacency list is that the collection I(v) (or more specifically, Iout(v)) contains exactly those edges that should be reported by the method outgoingEdges(v). Therefore, we can implement this method by iterating the edges of I(v) in O(deg(v)) time, where deg(v) is the degree of vertex v. This is the best possible outcome for any graph representation, because there are deg(v) edges to be reported.

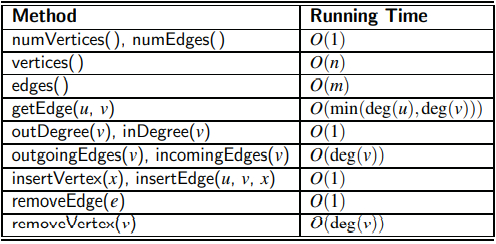


1. An undirected graph G; (b) a schematic representation of the adjacency list structure for G. Collection V is the primary list of vertices, and each vertex has an associated list of incident edges. Although not diagrammed as such, we presume that each edge of the graph is represented with a unique Edge instance that maintains references to its endpoint vertices, and that E is a list of all edges.

Performance of the Adjacency List Structure

Asymptotically, the space requirements for an adjacency list are the same as an edge list structure, using O(n + m) space for a graph with n vertices and m edges.

Running Times:



[Csak eszembe jutott: Mi a kulombseg, h ha azt kerdezik, h mi a time complexity asymptotically vagy nem-asymptotically?]

Adjacency Map Structure

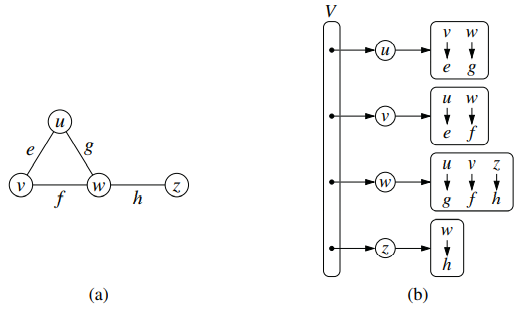
In the adjacency list structure, we assume that the secondary incidence collections are implemented as unordered linked lists. Such a collection I(v) uses space proportional to O(deg(v)), allows an edge to be added or removed in O(1) time, and allows an iteration of all edges incident to vertex v in O(deg(v)) time. However, the best implementation of getEdge(u, v) requires O(min(deg(u),deg(v))) time, because we must search through either I(u) or I(v).

We can improve the performance by using a hash-based map to implement I(v) for each vertex v. Specifically, we let the opposite endpoint of each incident edge serve as a key in the map, with the edge structure serving as the value. We call such a graph representation an adjacency map.

The space usage for an adjacency map remains O(n+ m), because I(v) uses O(deg(v)) space for each vertex v, as with the adjacency list.

The advantage of the adjacency map, relative to an adjacency list, is that the getEdge(u, v) method can be implemented in expected O(1) time by searching for vertex u as a key in I(v), or vice versa. This provides a likely improvement over the adjacency list, while retaining the worst-case bound of O(min(deg(u),deg(v))).

In comparing the performance of adjacency map to other representations, we find that it essentially achieves optimal running times for all methods, making it an excellent all-purpose choice as a graph representation.

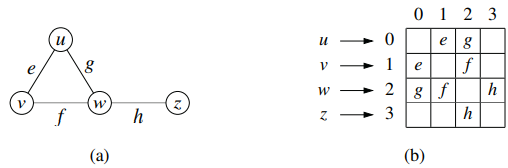


1. An undirected graph G; (b) a schematic representation of the adjacency map structure for G. Each vertex maintains a secondary map in which neighboring vertices serve as keys, with the connecting edges as associated values. As with the adjacency list, we presume that there is also an overall list E of all Edge instances.

Adjacency Matrix Structure

The **adjacency matrix** structure for a graph G augments the edge list structure with a matrix A (that is, a two-dimensional array, as in Section 3.1.5), which allows us to locate an edge between a given pair of vertices in worst-case constant time. In the adjacency matrix representation, we think of the vertices as being the integers in the set {0,1,... ,n−1} and the edges as being pairs of such integers. This allows us to store references to edges in the cells of a two-dimensional n × n array A. Specifically, the cell A[i][ j] holds a reference to the edge (u,v), if it exists, where u is the vertex with index i and v is the vertex with index j. If there is no such edge, then A[i][ j] = null. We note that array A is symmetric if graph G is undirected, as A[i][ j] = A[ j][i] for all pairs i and j.

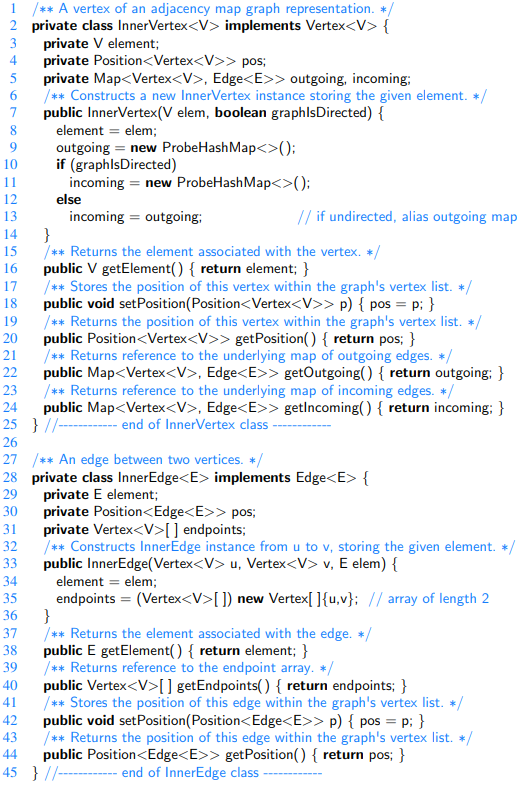
The most significant advantage of an adjacency matrix is that any edge (u,v) can be accessed in worst-case O(1) time; recall that the adjacency map supports that operation in O(1) expected time. However, several operation are less efficient with an adjacency matrix. For example, to find the edges incident to vertex v, we must presumably examine all n entries in the row associated with v; recall that an adjacency list or map can locate those edges in optimal O(deg(v)) time. Adding or removing vertices from a graph is problematic, as the matrix must be resized. Furthermore, the O(n2) space usage of an adjacency matrix is typically far worse than the O(n + m) space required of the other representations.



1. An undirected graph G; (b) a schematic representation of the auxiliary adjacency matrix structure for G, in which n vertices are mapped to indices 0 to n−1. Although not diagrammed as such, we presume that there is a unique Edge instance for each edge, and that it maintains references to its endpoint vertices. We also assume that there is a secondary edge list (not pictured), to allow the edges( ) method to run in O(m) time, for a graph with m edges.

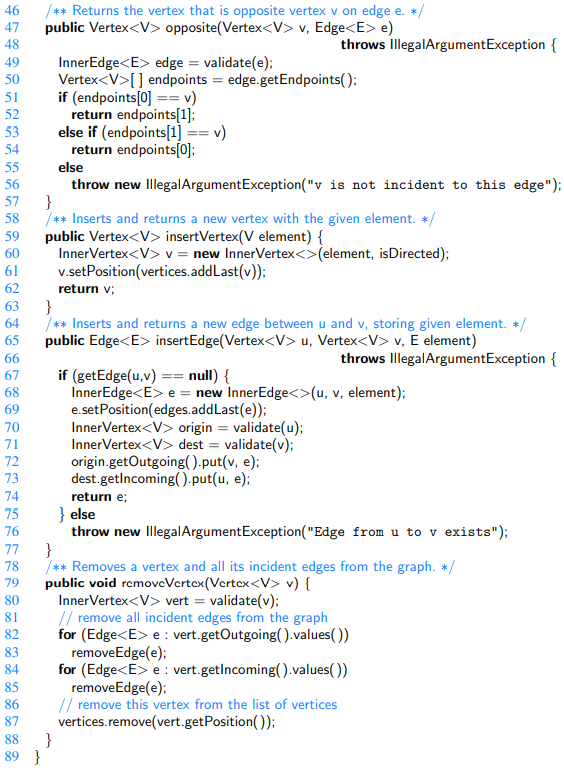
Java Implementation

In this section, we provide an implementation of the Graph ADT, based on the adjacency map representation, as described earlier. We use positional lists to represent each of the primary lists V and E, as originally described in the edge list representation. Additionally, for each vertex v, we use a hash-based map to represent the secondary incidence map I(v).



Above: InnerVertex and InnerEdge classes (to be nested within the AdjacencyMapGraph class). Interfaces Vertex and Edge are not shown.





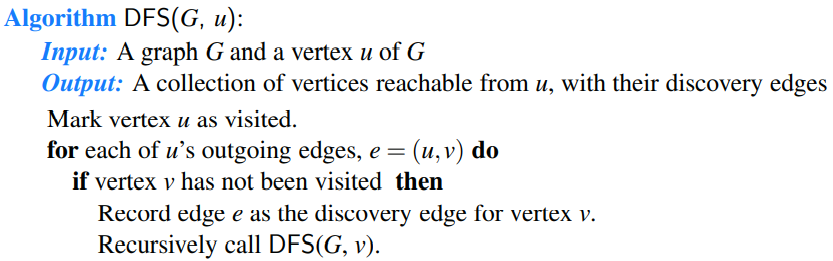
Graph Traversals

Formally, a **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges. A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time.

In the remainder of this section, we will present two efficient graph traversal algorithms, called **depth-first search** and **breadth-first search**.

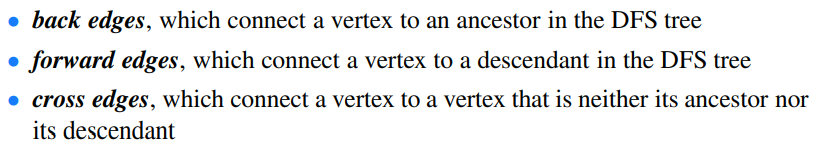
Depth-First Search(DFS)

Depth-first search is useful for testing a number of properties of graphs, including whether there is a path from one vertex to another and whether or not a graph is connected.

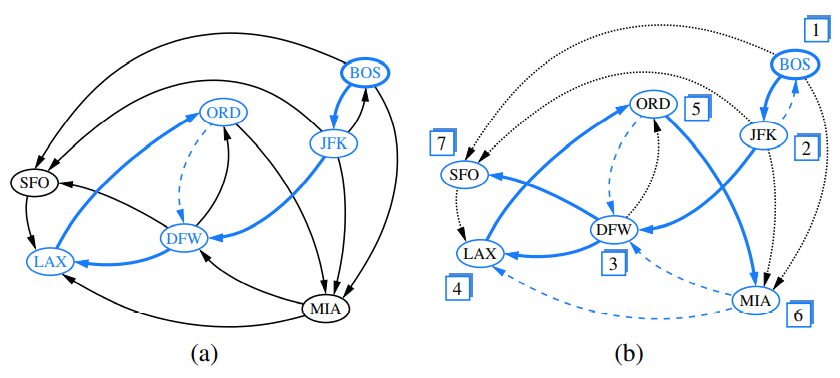


Classifying Graph Edges with DFS

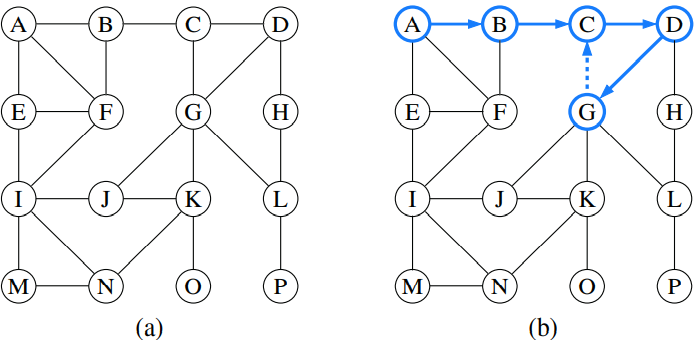
When performing a DFS on a directed graph, there are three possible kinds of nontree edges:

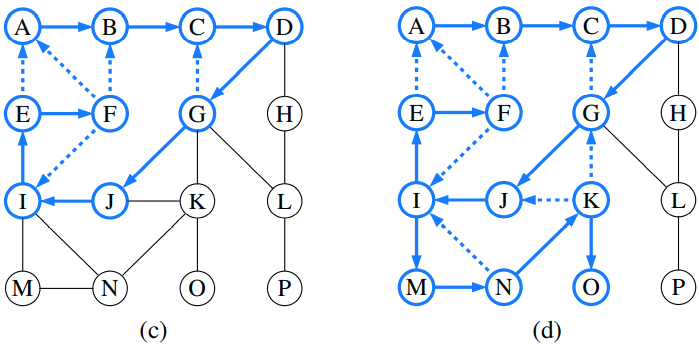


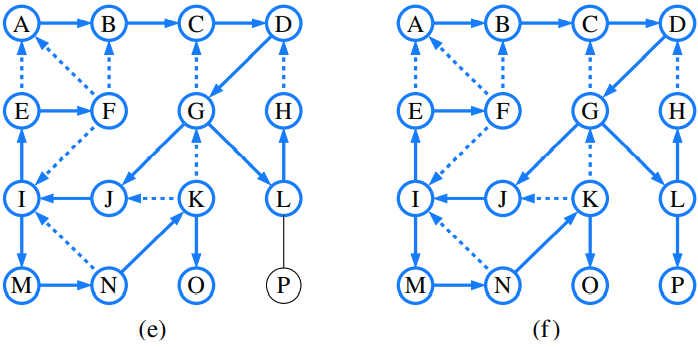
Below: An example of a DFS in a directed graph, starting at vertex (BOS): (a) intermediate step, where, for the first time, a considered edge leads to an already visited vertex (DFW); (b) the completed DFS. The tree edges are shown with thick blue lines, the back edges are shown with dashed blue lines, and the forward and cross edges are shown with dotted black lines. The order in which the vertices are visited is indicated by a label next to each vertex. The edge (ORD,DFW) is a back edge, but (DFW,ORD) is a forward edge. Edge (BOS,SFO) is a forward edge, and (SFO,LAX) is a cross edge.



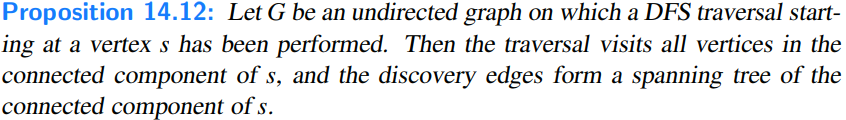
Below: Example of depth-first search traversal on an undirected graph starting at vertex A. We assume that a vertex’s adjacencies are considered in alphabetical order. Visited vertices and explored edges are highlighted, with discovery edges drawn as solid lines and nontree (back) edges as dashed lines: (a) input graph; (b) path of tree edges, traced from A until back edge (G,C) is examined; (c) reaching F, which is a dead end; (d) after backtracking to I, resuming with edge (I,M), and hitting another dead end at O; (e) after backtracking to G, continuing with edge (G,L), and hitting another dead end at H; (f) final result

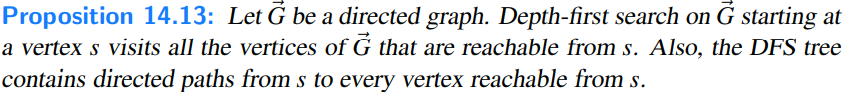




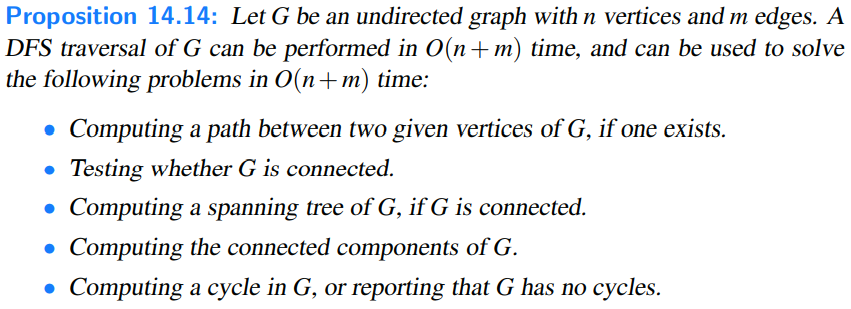


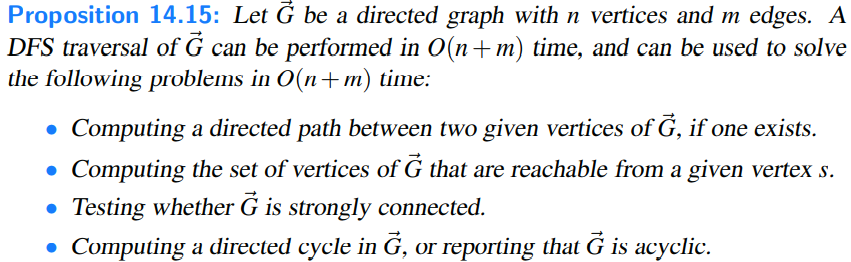
Properties of a Depth-First Search





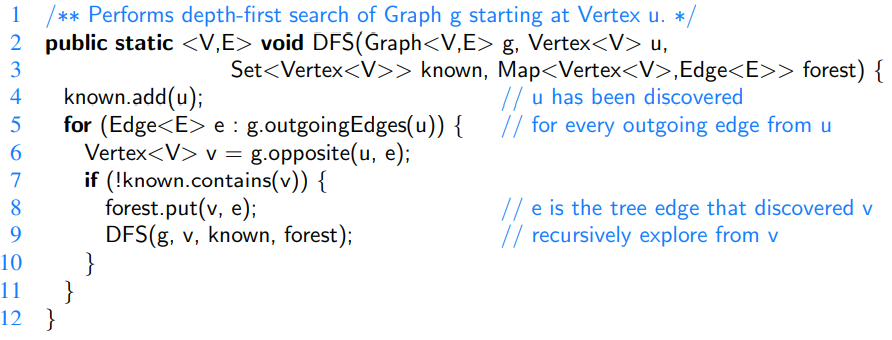
Running Time of Depth-First Search





DFS Implementation and Extensions

Below: Recursive implementation of depth-first search on a graph, starting at a designated vertex u. As an outcome of a call, visited vertices are added to the known set, and discovery edges are added to the forest.



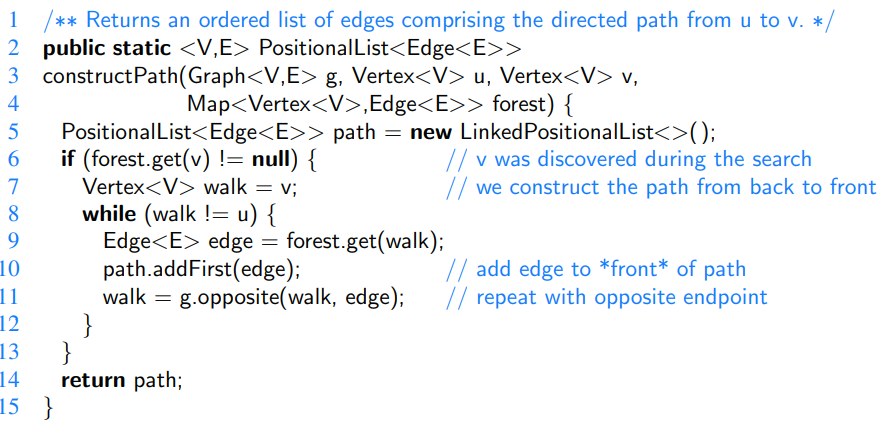
Our DFS method does not make any assumption about how the Set or Map instances are implemented; however, the O(n + m) running-time analysis of the previous section does presume that we can “mark” a vertex as explored or test the status of a vertex in O(1) time. If we use hash-based implementations of the set and map structure, then all of their operations run in O(1) expected time, and the overall algorithm runs in O(n+m) time with very high probability. In practice, this is a compromise we are willing to accept.

If vertices can be numbered from 0,...,n−1 (a common assumption for graph algorithms), then the set and map can be implemented more directly as a lookup table, with a vertex label used as an index into an array of size n. In that case, the necessary set and map operations run in worst-case O(1) time. Alternatively, we can “decorate” each vertex with the auxiliary information, either by leveraging the generic type of the element that is stored with each vertex, or by redesigning the Vertex type to store additional fields. That would allow marking operations to be performed in O(1)-time, without any assumption about vertices being numbered.

Reconstructing a Path from u to v

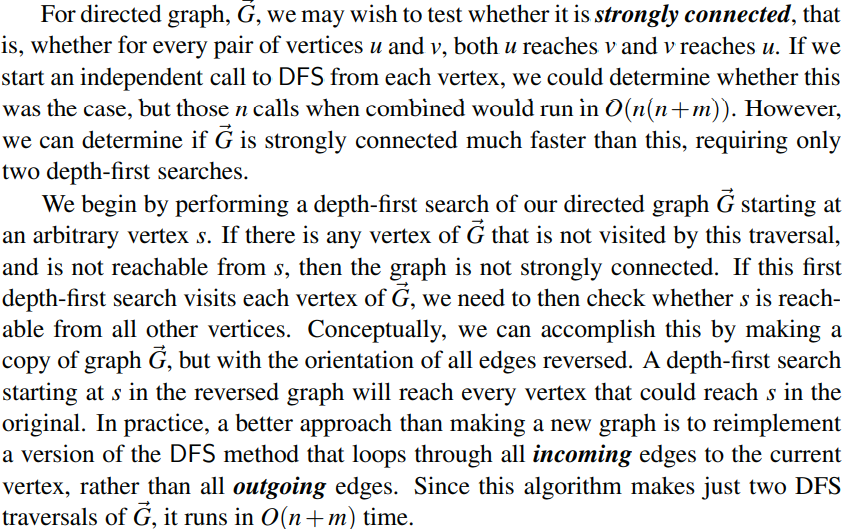
To reconstruct the path, we begin at the end of the path, examining the forest of discovery edges to determine what edge was used to reach vertex v. We then determine the opposite vertex of that edge and repeat the process to determine what edge was used to discover it. By continuing this process until reaching u, we can construct the entire path. Assuming constant-time lookup in the forest map, the path reconstruction takes time proportional to the length of the path, and therefore, it runs in O(n) time (in addition to the time originally spent calling DFS).

Below: Method to reconstruct a directed path from u to v, given the trace of discovery from a DFS started at u. The method returns an ordered list of vertices on the path.



Testing for Connectivity

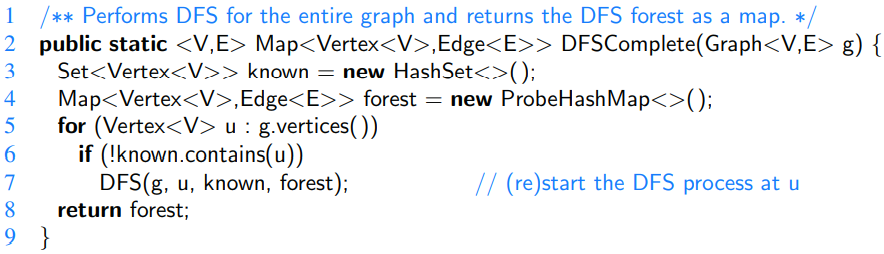
In the case of an undirected graph, we simply start a depth-first search at an arbitrary vertex and then test whether known.size( ) equals n at the conclusion. If the graph is connected, then by Proposition 14.12, all vertices will have been discovered; conversely, if the graph is not connected, there must be at least one vertex v that is not reachable from u, and that will not be discovered.



Computing All Connected Components

When a graph is not connected, the next goal we may have is to identify all of the connected components of an undirected graph, or the strongly connected components of a directed graph. We will begin by discussing the undirected case.

If an initial call to DFS fails to reach all vertices of a graph, we can restart a new call to DFS at one of those unvisited vertices.



It returns a map that represents a DFS forest for the entire graph. We say this is a forest rather than a tree, because the graph may not be connected.

Detecting Cycles with DFS

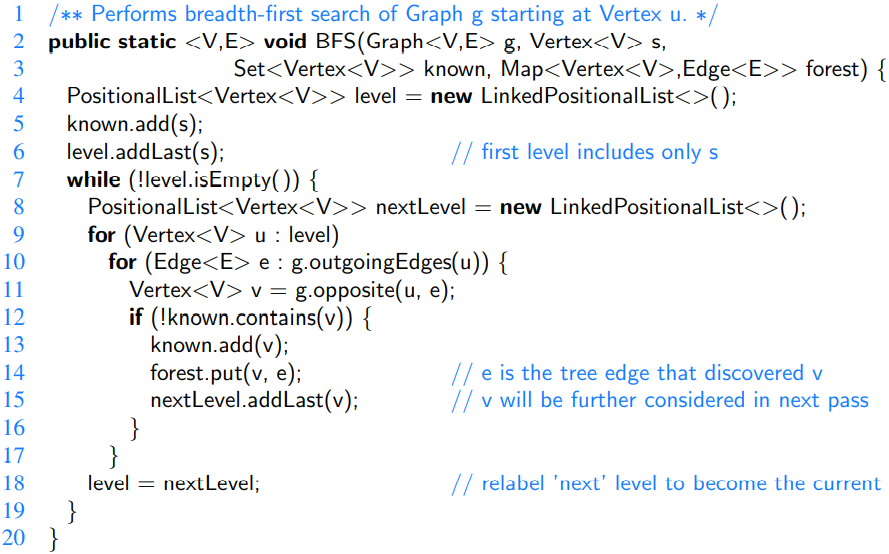
For both undirected and directed graphs, a cycle exists if and only if a back edge exists relative to the DFS traversal of that graph. It is easy to see that if a back edge exists, a cycle exists by taking the back edge from the descendant to its ancestor and then following the tree edges back to the descendant. Conversely, if a cycle exists in the graph, there must be a back edge relative to a DFS.

Algorithmically, detecting a back edge in the undirected case is easy, because all edges are either tree edges or back edges. In the case of a directed graph, additional modifications to the core DFS implementation are needed to properly categorize a nontree edge as a back edge. When a directed edge is explored leading to a previously visited vertex, we must recognize whether that vertex is an ancestor of the current vertex. This can be accomplished, for example, by maintaining another set, with all vertices upon which a recursive call to DFS is currently active. We leave details as an exercise.

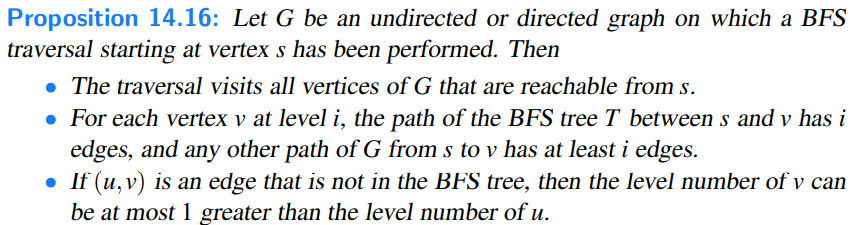
Breadth-First Search (BFS)

The advancing and backtracking of a depth-first search, as described in the previous section, defines a traversal that could be physically traced by a single person exploring a graph. In this section, we will consider another algorithm for traversing a connected component of a graph, known as a **breadth-first search** (BFS). The BFS algorithm is more akin to sending out, in all directions, many explorers who collectively traverse a graph in coordinated fashion.

A BFS proceeds in rounds and subdivides the vertices into **levels**.



When discussing DFS, we described a classification of nontree edges being either **back edges**, which connect a vertex to one of its ancestors, **forward edges**, which connect a vertex to one of its descendants, or **cross edges**, which connect a vertex to another vertex that is neither its ancestor nor its descendant. For BFS on an undirected graph, *all* nontree edges are *cross edges* (see Exercise C-14.46), and for BFS on a directed graph, *all* nontree edges are either *back edges or cross edges.*





Although our implementation of BFS in Code Fragment 14.8 progresses level by level, the BFS algorithm can also be implemented using a single FIFO queue to represent the current fringe of the search. Starting with the source vertex in the queue, we repeatedly remove the vertex from the front of the queue and insert any of its unvisited neighbors to the back of the queue.

In comparing the capabilities of DFS and BFS, both can be used to efficiently find the set of vertices that are reachable from a given source, and to determine paths to those vertices. However, BFS guarantees that those paths use as few edges as possible. For an undirected graph, both algorithms can be used to test connectivity, to identify connected components, or to locate a cycle. For directed graphs, the DFS algorithm is better suited for certain tasks, such as finding a directed cycle in the graph, or in identifying the strongly connected components.

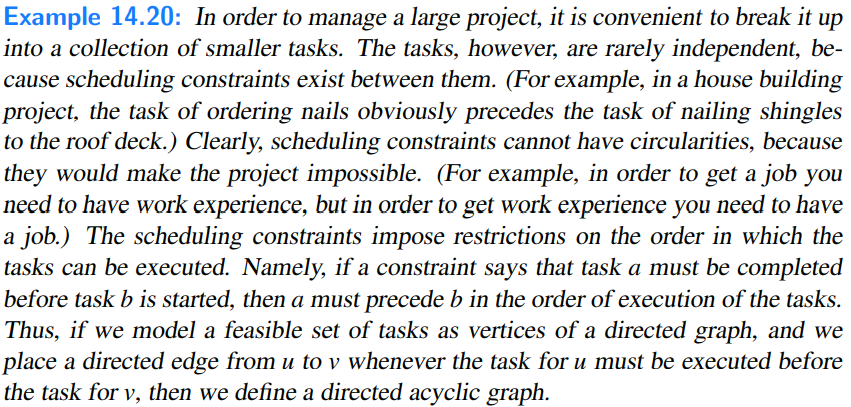
Directed Acyclic Graphs

Directed graphs without directed cycles are encountered in many applications. Such a directed graph is often referred to as a directed acyclic graph, or DAG, for short. Applications of such graphs include the following:

• Prerequisites between courses of an academic program.

• Inheritance between classes of an object-oriented program.

• Scheduling constraints between the tasks of a project. We will explore this latter application further in the following example:



Topological Ordering

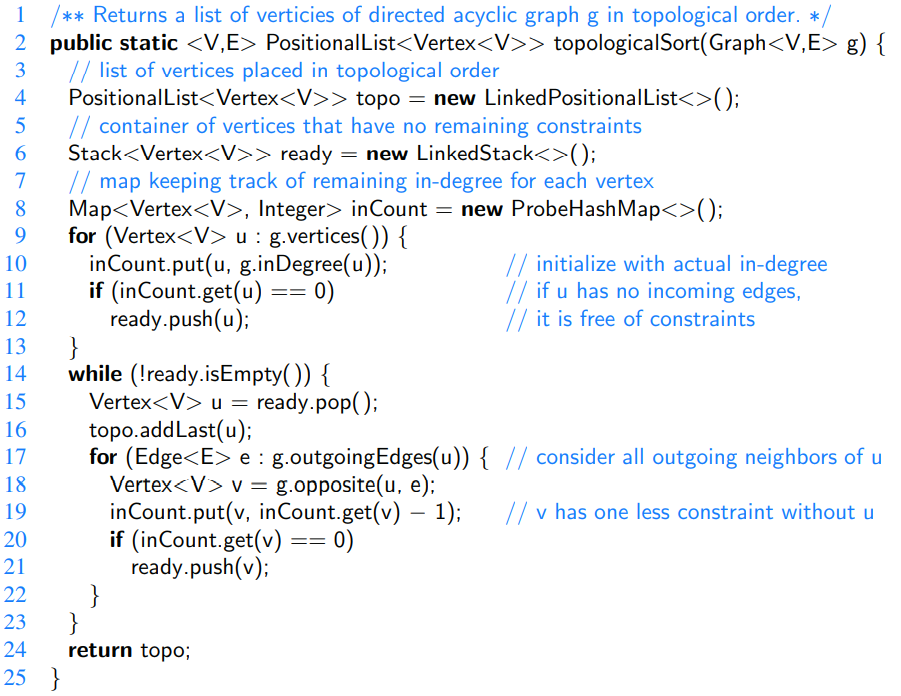
Let G~ be a directed graph with n vertices. A **topological ordering** of G~ is an ordering v1,...,vn of the vertices of G~ such that for every edge (vi,vj) of G~ , it is the case that i < j. That is, a topological ordering is an ordering such that any directed path in G~ traverses vertices in increasing order. [Note that a directed graph may have more than one topological ordering.]



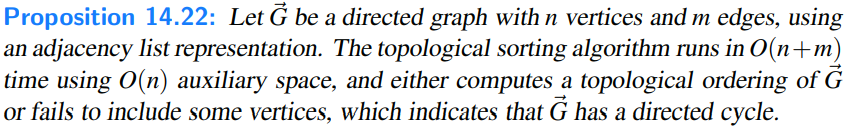
Below: Two topological orderings of the same acyclic directed graph.



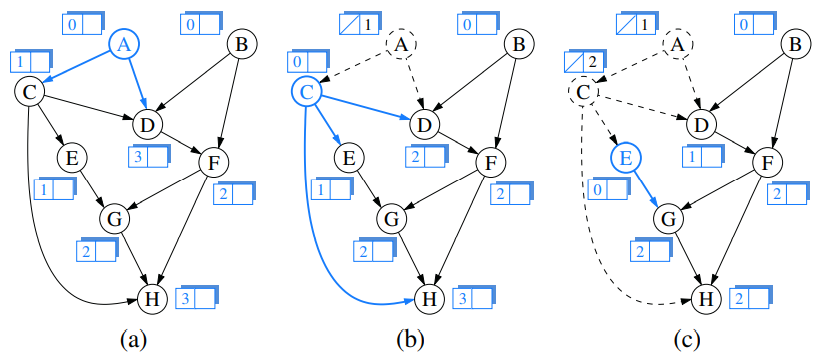
Java implementation for the topological sorting algorithm:

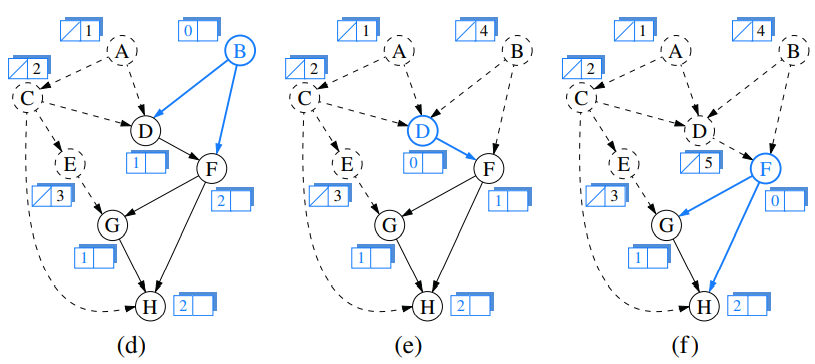


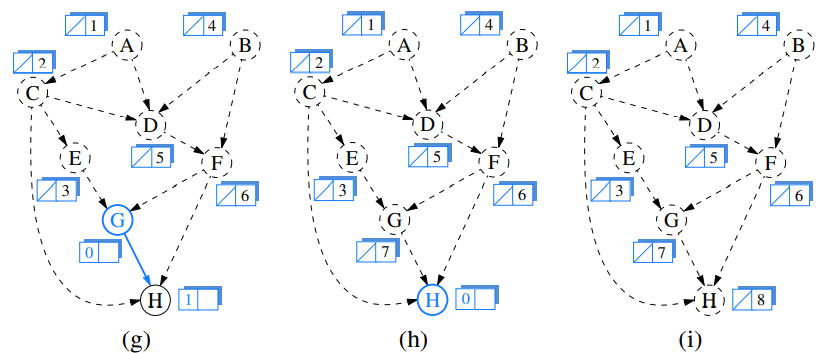
Time and Space Complexity



Below: Example of a run of algorithm topologicalSort [Belongs to the previous java Code]. The label near a vertex shows its current inCount value, and its eventual rank in the resulting topological order. The highlighted vertex is one with inCount equal to zero that will become the next vertex in the topological order. Dashed lines denote edges that have already been examined, which are no longer reflected in the inCount values.







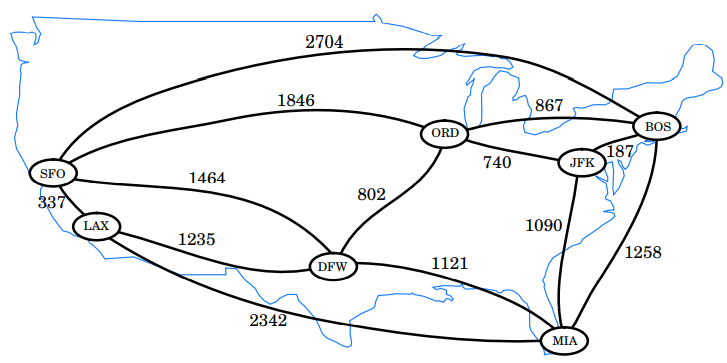
Shortest Paths

As we saw earlier, the breadth-first search strategy can be used to find a path with as few edges as possible from some starting vertex to every other vertex in a connected graph. This approach makes sense in cases where each edge is as good as any other, but there are many situations where this approach is not appropriate.

For example, we might want to use a graph to represent the roads between cities, and we might be interested in finding the fastest way to travel cross-country. In this case, it is probably not appropriate for all the edges to be equal to each other, for some inter-city distances will likely be much larger than others. Likewise, we might be using a graph to represent a computer network (such as the Internet), and we might be interested in finding the fastest way to route a data packet between two computers. In this case, it again may not be appropriate for all the edges to be equal to each other, for some connections in a computer network are typically much faster than others (for example, some edges might represent low-bandwidth connections, while others might represent high-speed, fiber-optic connections). It is natural, therefore, to consider graphs whose edges are not weighted equally.

Weighted Graphs

A **weighted graph** is a graph that has a numeric (for example, integer) label w(e) associated with each edge e, called the weight of edge e. For e = (u,v), we let notation w(u,v) = w(e). Example:



Above: A weighted graph whose vertices represent major U.S. airports and whose edge weights represent distances in miles. This graph has a path from JFK to LAX of total weight 2,777 (going through ORD and DFW). This is the minimumweight path in the graph from JFK to LAX.

Defining Shortest Paths in a Weighted Graph

The **length** (or weight) of a path is the sum of the weights of the edges of P.

The **distance** from a vertex u to a vertex v in G, denoted d(u,v), is the length of a minimum-length path (also called **shortest path**) from u to v, if such a path exists.

Suppose we are given a weighted graph G, and we are asked to find a shortest path from some vertex s to each other vertex in G, viewing the weights on the edges as distances. In this section, we explore efficient ways of finding all such shortest paths, if they exist. The first algorithm we discuss is for the simple, yet common, case when all the edge weights in G are nonnegative (that is, w(e) ≥ 0 for each edge e of G); hence, we know in advance that there are no negative-weight cycles in G. Recall that the special case of computing a shortest path when all weights are equal to one was solved with the BFS traversal algorithm presented earlier.

There is an interesting approach for solving this **single-source** problem based on the **greedy-method** design pattern. Recall that in this pattern we solve the problem at hand by repeatedly selecting the best choice from among those available in each iteration.

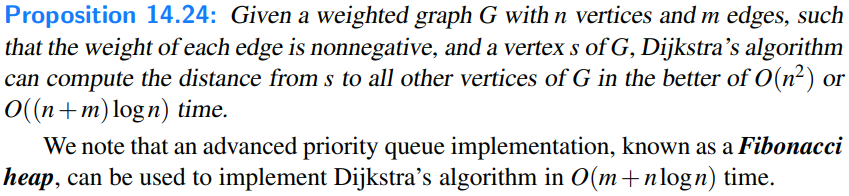
Dijkstra’s Algorithm

[Look it up online, that is easier. After that, read the book, because that is more detailed.]

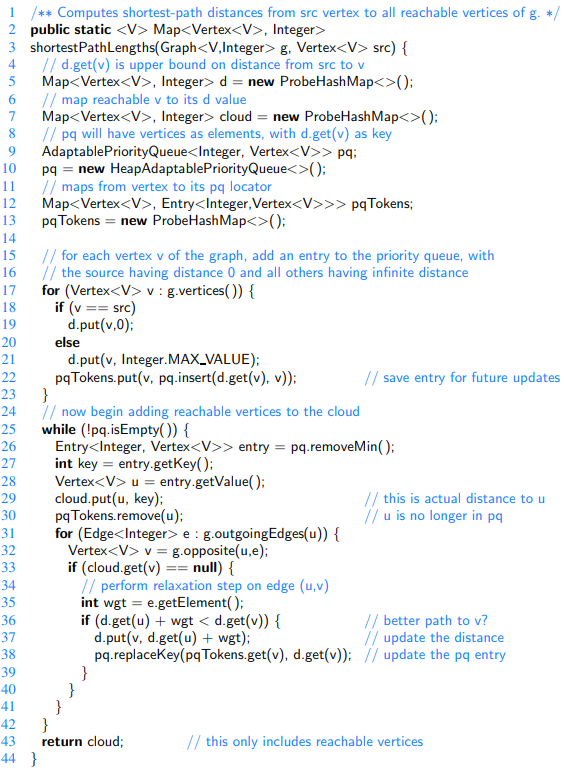
The Running Time of Dijkstra’s Algorithm

[Read more in the book 657(675)]

Depends on the implementation!



Programming Dijkstra’s Algorithm in Java



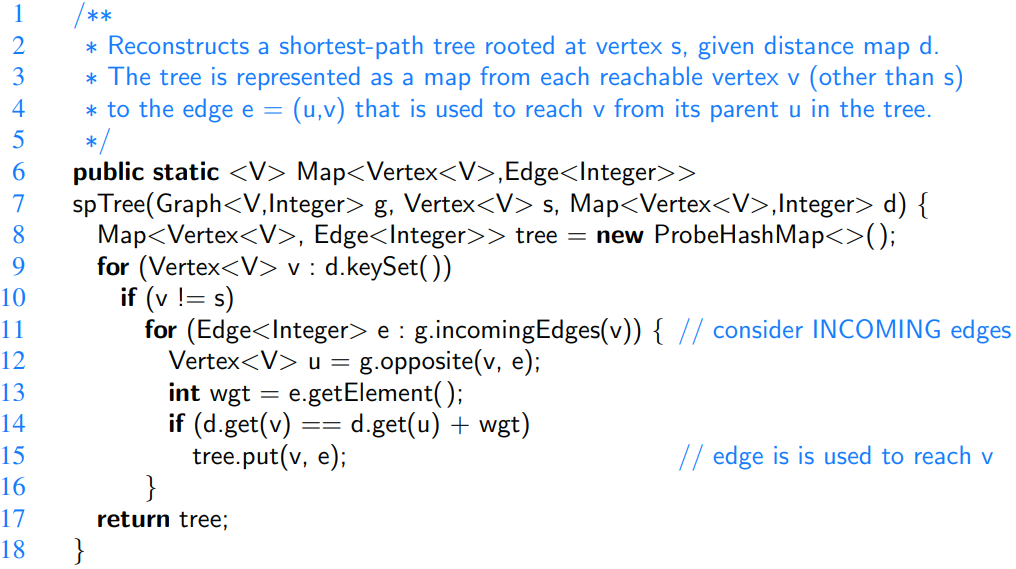
Reconstructing a Shortest-Path Tree

Our implementation computes the value D[v], for each vertex v, that is the length of a shortest path from the source vertex s to v. However, those forms of the algorithm do not explicitly compute the actual paths that achieve those distances.

Fortunately, it is possible to represent shortest paths from source s to every reachable vertex in a graph using a compact data structure known as a **shortest-path tree**. This is possible because if a shortest path from s to v passes through an intermediate vertex u, it must begin with a shortest path from s to u.

We next demonstrate that a shortest-path tree rooted at source s can be reconstructed in O(n+ m) time, given the D[v] values produced by Dijkstra’s algorithm using s as the source.

Java method that reconstructs a single-source shortest-path tree, based on knowledge of the shortest-path distances:

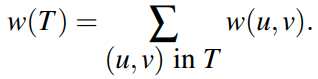


Minimum Spanning Trees

Suppose we wish to connect all the computers in a new office building using the least amount of cable. We can model this problem using an undirected, weighted graph G whose vertices represent the computers, and whose edges represent all the possible pairs (u,v) of computers, where the weight w(u,v) of edge (u,v) is equal to the amount of cable needed to connect computer u to computer v. Rather than computing a shortest-path tree from some particular vertex v, we are interested instead in finding a tree T that contains all the vertices of G and has the minimum total weight over all such trees. Algorithms for finding such a tree are the focus of this section.

Problem Definition

Given an undirected, weighted graph G, we are interested in finding a tree T that contains all the vertices in G and minimizes the sum

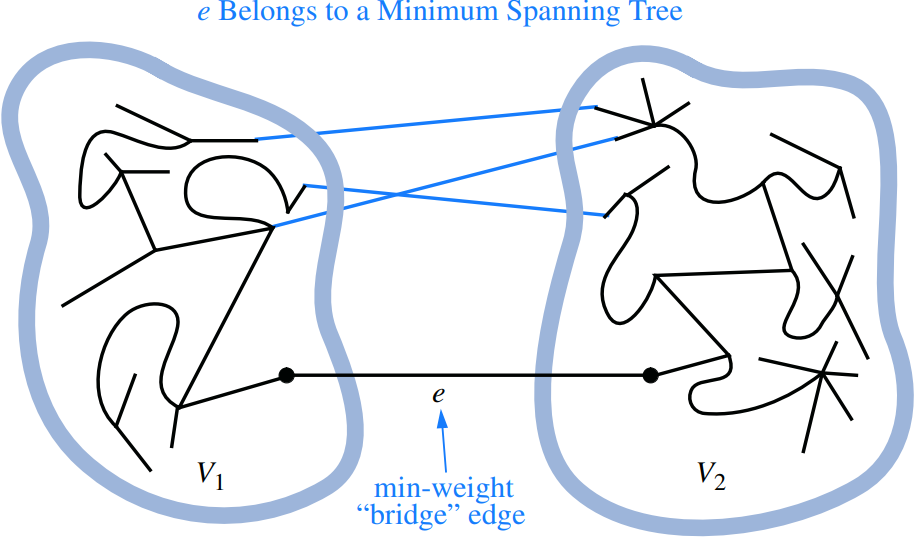


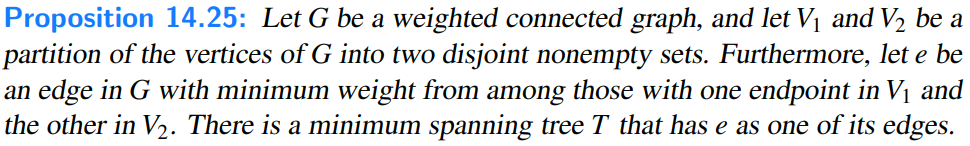
A tree, such as this, that contains every vertex of a connected graph G is said to be a **spanning tree**, and the problem of computing a spanning tree T with smallest total weight is known as the **minimum spanning tree** (or **MST**) problem.

The first algorithm we discuss is the **Prim-Jarnik algorithm**, which grows the MST from a single root vertex, much in the same way as Dijkstra’s shortest-path algorithm. The second algorithm we discuss is **Kruskal’s algorithm**, which “grows” the MST in clusters by considering edges in nondecreasing order of their weights.

In order to simplify the description of the algorithms, we assume, in the following, that the input graph G is undirected (that is, all its edges are undirected) and simple (that is, it has no self-loops and no parallel edges). Hence, we denote the edges of G as unordered vertex pairs (u,v).

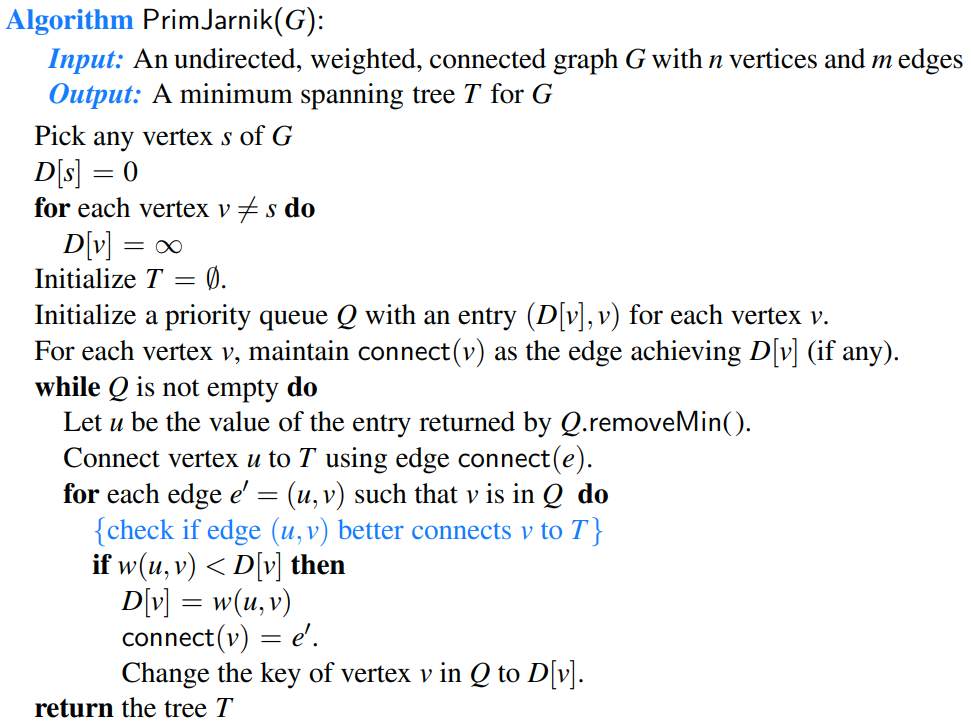
A Crucial Fact about Minimum Spanning Trees





Prim-Jarnik Algorithm

[Check online for clearer understanding]



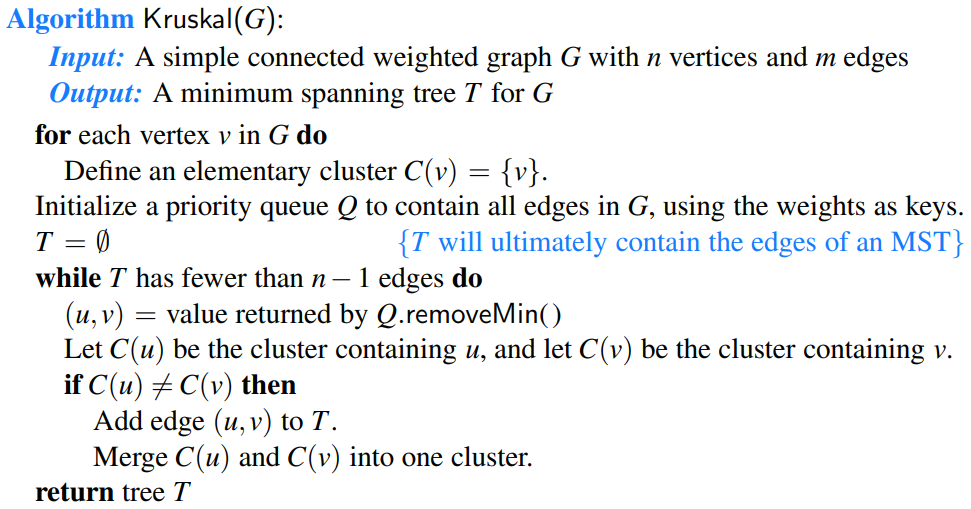
Running Time of the Prim-Jarnik Algorithm

With a heap-based priority queue, each operation runs in O(logn) time, and the overall time for the algorithm is O((n + m)logn), which is O(mlogn) for a connected graph. Alternatively, we can achieve O(n2) running time by using an unsorted list as a priority queue.

Kruskal’s Algorithm

[Check online, maybe even more important than the previous one.]

In this section, we will introduce Kruskal’s algorithm for constructing a minimum spanning tree. While the Prim-Jarnik algorithm builds the MST by growing a single tree until it spans the graph, Kruskal’s algorithm maintains many smaller trees in a forest, repeatedly merging pairs of trees until a single tree spans the graph.

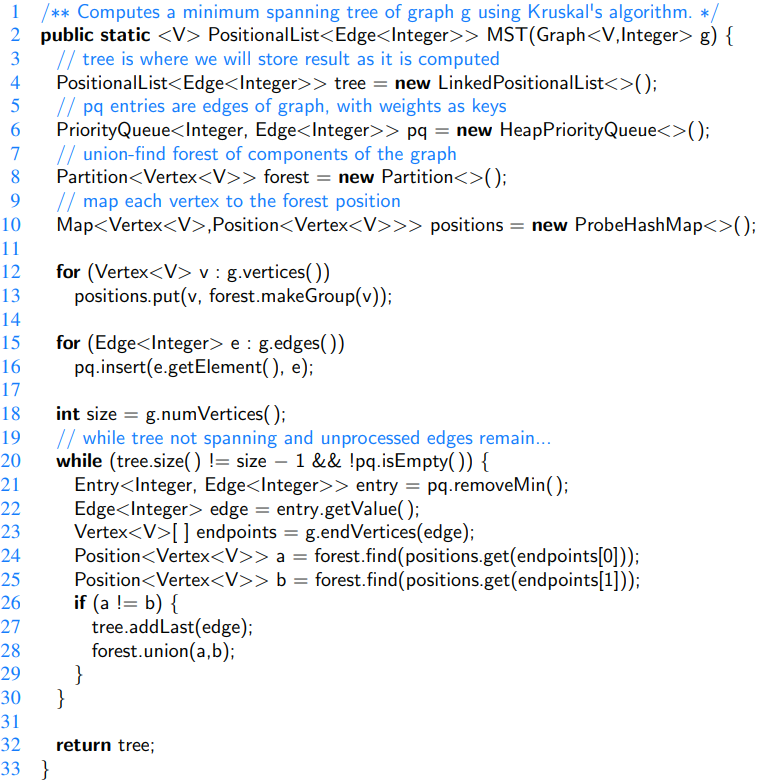


The Running Time of Kruskal’s Algorithm

In the context of Kruskal’s algorithm, we perform at most 2m “find” operations and n − 1 “union” operations. We will see that a simple union-find structure can perform that combination of operations in O(m + n log n) time.

For a connected graph, m ≥ n− 1; therefore, the bound of O(mlogn) time for ordering the edges dominates the time for managing the clusters. We conclude that the running time of Kruskal’s algorithm is O(mlogn).

Java Implementation



Disjoint Partitions and Union-Find Structures

In this section, we consider a data structure for managing a **partition** of elements into a collection of disjoint sets. Our initial motivation is in support of Kruskal’s minimum spanning tree algorithm, in which a forest of disjoint trees is maintained, with occasional merging of neighboring trees. More generally, the disjoint partition problem can be applied to various models of discrete growth.

[Ez a fejezet (5 oldal) most nem vilagos. De erdemes megnezni, mert olyanok vannak benne, hogy partition, cluster, Union-by-Size, Path Compression, stb, ja es egy implementacio is. A tavalyi slide-ok kozott is volt valami ide vonatkozo asszem, azert gondolom, h ez nem elhanyagolhato.]

Java implementation of a Partition class using union-by-size and path compression. We omit the validate method due to space limitation.

