Graph Algorithms

Skip Sections 9.4 and 9.7.

In this chapter, we will study:

- graph terminology.
- graph representations: adjacency matrix and adjacency lists.
- shortest path and minimum spanning tree algorithms.

Real life examples: airport system, highway system, course prerequisite structure (Fig. 9.3), map coloring.

Definitions

- A *graph* $G = (V, E)$, where V is a set of *vertices* and E is a set of *edges*.
- An *edge* is a pair (v, w), where $v, w \in V$. We could have an edge of the form (v, v), which we refer to as a *loop*. However, most graphs we will see are loopless.
- There are two kinds of graphs: *directed* graphs, where an edge (v, w) is ordered, and *undirected* graphs, where the edge (v, w) is unordered. We will focus on directed graphs in this chapter.
- Vertex w is *adjacent* to v if and only if $(v, w) \in E$.
- Often, an edge has a third component, known as its *weight* or *cost*.
- A *path* is a sequence of vertices $w_1, w_2, w_3, \ldots, w_n$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i < n$.

- The *length* of a path is a number of edges on the path.
- A *simple* path is a path where all vertices are distinct, except the first and the last, which *could* be the same. For example, $w_1, w_2, w_3, \ldots, w_n$. What do we know?
- A *cycle* is a path of length at least 1 such that $w_1 = w_n$. A cycle is simple if the path is simple.
- A directed graph is *acyclic* if it has no cycles. A directed acyclic graph is known as a *DAG*.

Are all Trees also DAGs? Conversely?

Graph ADT

- Edge(v, w): returns true if there's an edge between v and w, and false otherwise.
- Adjacent(v): returns the set of vertices adjacent to v.
- Cost(v, w): returns the cost associated with the edge from v to w. The cost is infinite if there is no such edge.

How would you represent a graph to support these operations?

Representation of Graphs

Adjacency matrix: we represent a graph as a two-dimensional array.

For each edge (u, v), we set $a[u, v] = 1$. Otherwise, the entry in the array is 0.

What is a compact way of representing edge costs?

How much space does this representation require?

What is the minimal number of edges in a graph?

What is the maximal number of edges as a function of the number of vertices?

We say that a graph is *dense* if $|E| = O(|V|^2)$.

Spare Graphs

At the other extreme, we say that the graph is *sparse*.

What are some examples of sparse and dense graphs?

For sparse graphs, a better representation is an adjacency list: for each vertex, we keep a list of all adjacent vertices. (See Fig. 9.2, pg. 280).

Where are the space requirements for an adjacency list?

	ADJACENCY LIST	**ADJACENCY MATRIX**
Edge(v,w)		
Adjacent(v)		
Cost(v,w)		

Topological Sort

A topological sort is an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears *after* v_i in the ordering.

What does a topological sort of a graph representing course prerequisite structure give us?

Why is topological sort only defined for acyclic graphs?

Is the ordering produced by a topological sort unique?

What is a simple algorithm for performing a topological sort on a DAG?

Details of Topological Sort

Indegree(v) = the number of edges in E of the form (u, v).
In a graph represented as an adjacency list, what data structure would you use to store the indegree of each vertex?

How would you compute the initial value of indegree for each vertex? Given an indegree array, we can apply the algorithm in Fig. 9.5.

Find each new vertex of indegree 0 by scanning the indegree array (which takes $O(|V|)$ time and there are $|V|$ such calls, so the running time of the algorithm is $O(|V|^2)$. How would you improve the running time of this algorithm for sparse graphs?

How many times is the for-loop executed?
How many queue operations are performed?

Shortest-Path Algorithms

The cost of a path $v_1\ v_2\ \ldots\ v_n$ is $\Sigma_{i=1}^{n-1} c_{i,i+1}$.

This is the *weighted path length.*

The *unweighted path length* is simply the number of edges on the path. (How many in this case?)

Generally when it is not specified whether we refer to weighted or unweighted path length, then we are talking about **weighted** path length.

Single-Source Shortest-Path Problem: Given a weighted graph D and a distinguished vertex S, find the shorted weighted path from S to every other vertex in G.

Sample Shortest-Path Problem: Suppose we have a subset of the Internet and we want to transmit the news from a particular server to all sites on the Net, what is the cheapest way to do this?

Another Sample Shortest-Path Problem: A robot trying to find its way from one location in Sieg Hall to another.

In Fig. 9.8, what is the shortest path from v_1 to v_6?

What is the shortest unweighted path between v_1 and v_6 in Fig. 9.8?

What is the shortest path from v_5 to v_4 in Fig 9.9?

What is the shortest path from v_4 to v_4?

In general, in the absence of a *negative cost* cycle, the shortest path from vertex v to itself is 0.

Shortest-Path Problems

Unweighted shortest-path problem: $O(|E| + |V|)$

Weighted shortest-path problem, no negative edges: $O(|E| \ \log |V|)$.

General weighted shortest-path problem: $O(|E| * |V|)$.

Weighted shortest-path for acyclic graphs: $O(|E| + |V|)$.

Is it possible to solve this problem in less than $O(|E| + |V|)$ time?

Unweighted Shortest Path

We focus on the length of the shortest path.

It turns out that actually keeping track of the path is a matter of simple bookkeeping.

We use a strategy called *breadth-first search*. It operates by processing vertices in layers from the vertices closest to the start to the most distant vertices. This is a generalization of level-order traversal for trees.

Update rule: Lines 5-7.

Invariant: Shortest path to vertices adjacent to known vertices.

P_v is the last vertex to cause a change to the value of d_v.

Analysis

What does the algorithm do?

What is the role of P_v?

What is the running time of the algorithm in Fig. 9.16?

How can we reduce the number of iterations in the outer loop?

What is the impact on the worst-case running time? (See Fig. 9.17.)

How can we speed up the loop in lines 2 and 3?

What data structure would you use as the basis of a better algorithm, $O(|E| + |V|)$?

Greedy Algorithms

Dijkstra's algorithm is a greedy algorithm. Greedy algorithms always do what appears to be best locally.

Example: Let's suppose you want to make change for an arbitrary sum of money using quarters, dimes, nickels, and pennies, and you want to use the minimal number of coins. What is the solution for $1? Or $1.35?

What is a greedy algorithm for this problem? Does this algorithm always work?

How about if we have a 12 cent piece?

What is the optimal solution when the sum if 15 cents?

Dijkstra's Algorithm

Problem: Finding the shortest path in a weighted, directed graph.

- We use a similar algorithm to the unweighted case, keeping track of known, D[v], and p_v.
- A tentative distance $D[v]$ is kept for each vertex. $D[v]$ is the shortest path length from s to v using only known vertices.
- At each stage, the algorithm selects a vertex v, which has the smallest $D[v]$ among all the unknown vertices, and declares that the shortest path from s to v is known. Then, the algorithm updates the values of $D[w]$.
- In the unweighted case, we set $D[w] = D_[v] + 1$ if $D[w] = \infty$. Here we set $D[w] = D[v] + C[v, w]$ and we do that if this is an improved value for $D[w]$. **The algorithm decides whether or not to use v on the path to w.**

Complexity Analysis

The time for Dijkstra's algorithm depends on the following factors:

- Is the graph dense or sparse?
- Is the graph represented as an adjacency matrix or adjacency list?
- How are lines 4, 5 and 7, 8 implemented?

If we search the D array for a minimum, the running time is?

If the graph is dense, the running time is $O(V^2)$. Why?

If the graph is sparse, what data structure would you use for line 5?   How about lines 7,8?

Is $O(E \log(V))$ better $O(V^2)$?

Minimum Spanning Trees

We assume that G is undirected and connected.

A minimal spanning tree (MST) is: a tree, reaches every vertex (contrary to the book which says covers every edge), and has minimal cost. Let's look at Fig. 9.48 for an example.

What is the number of edges in an MST?

There are two algorithms for finding MST's. Both are greedy: Prim's algorithm and Kruskal's algorithm.

Both algorithms grow the tree by repeatedly selecting an edge from the graph.

Prim's Algorithm

Selection rule: at each stage, grow the tree by choosing the edge (u, v) such that the cost of (u, v) is the *smallest* among all edges where u is in the tree and v is not.

What if adding an edge creates a cycle?

Analysis: same as for Dijkstra's algorithm.