Graph Algorithms

R. J. Renka

Department of Computer Science & Engineering
University of North Texas

11/05/2013
9.1 Definitions

- A **graph** $G = (V, E)$ is a set of **vertices** $V$ and **edges** $E$.
- An **edge** is a pair $(v, w)$ for $v, w \in V$.
- The pair is ordered in a **directed graph** (**digraph**).
- An edge $(v, w)$ of an **undirected graph** has no direction, and $(v, w) = (w, v)$.
- Vertex $w$ is **adjacent** to $v$ iff $(v, w) \in E$.
- An edge may have an associated **weight** or **cost**.
- A **path** is a sequence of vertices $w_1, w_2, \ldots, w_N$ such that $(w_i, w_{i+1}) \in E$ for $1 \leq i \leq N - 1$.
- The **path length** is the number of edges $N - 1$.
- Each vertex has a path to itself. If there is no edge from $v$ to itself, the path has length 0; otherwise the path $v, v$ is a **loop**.
A **simple path** is one in which all vertices are distinct, except that the first and last could be the same. A loop is a simple path.

A **cycle** in a digraph is a path of length at least 1 such that \( w_1 = w_N \); the cycle is simple if the path is simple.

A cycle in an undirected graph is defined the same way but with the requirement that the edges are distinct so that \( u, v, u \) is not a cycle in an undirected graph.

A directed graph is **acyclic** if it has no cycles. A **dag** is a directed acyclic graph.

The **indegree** of vertex \( v \) is the number of edges \( (u, v) \).
An undirected graph is **connected** if there is a path from every vertex to every other vertex. A directed graph with this property is **strongly connected**. If a directed graph is not strongly connected but the underlying graph (without edge directions) is connected, the graph is **weakly connected**.

A **complete graph** is one with an edge between every pair of vertices.

A **tree** is a connected undirected acyclic graph.

A **rooted tree** is a directed graph in which there is a vertex \( v \) (the root) such that every other vertex can be reached from \( v \) by a unique path.

A **dense graph** is one in which \( |E| \) is close to maximal, where \( |\cdot| \) denotes set cardinality. A **sparse graph** is one which is not dense: \( |E| = O(|V|^k) \) for \( 0 \leq k < 2 \).
Example 1: Vertices are airports, and edges are nonstop flights, possibly weighted by time, distance, or cost. The graph is directed, and we would expect it to be strongly connected, but not complete. It would be sparse but with high-degree vertices (hubs).

Example 2: Traffic flow in a city could be modeled by a graph in which vertices are intersections, and edges are portions of streets. A directed graph is necessary to allow for one-way streets. Edge weights might be capacity (number of lanes) or speed limit. The graph would be very sparse: $|E| \approx 4|V|$ for Manhattan-like traffic flow with no one-way streets.
We assume the vertices are numbered 1, 2, \ldots, |V|. If vertices are designated by strings, we can use a map with vertex names as keys, and vertex numbers or pointers to Vertex objects as values.

An **adjacency matrix** $A$ has $A[u][v] = \text{true}$ iff there is an edge from $u$ to $v$. Alternatively, we might store a weight in $A[u][v]$ with $A[u][v] = \infty$ if $(u, v)$ is not an edge. Since the storage cost is $\Theta(|V|^2)$, and the cost of traversing an adjacency list is $\Theta(|V|)$, this only makes sense for a dense graph. It has the advantage of constant cost to decide if $(v, w)$ is an edge, for arbitrary $v$ and $w$.

The alternative is to store an **adjacency list**, usually as a linked list of pointers and edge weights, for each vertex — either as a data member of the Vertex class or using a map from vertex designators to adjacency lists. The space required is linear in the size of the graph: $O(|E| + |V|)$. There are two adjacencies for each edge in the case of an undirected graph.
Figures 9.1-9.2

Figure 9.1 A directed graph

Figure 9.2 An adjacency list representation of a graph
Figure 9.3 An acyclic graph representing course prerequisite structure
A topological sort is an ordering of vertices in a dag such that, if there is an edge (or path) from $v_i$ to $v_j$, then $v_i$ precedes $v_j$ in the ordering.

**Example 1:** In a dag with vertices representing courses and edges representing a prerequisite structure, a topological sort is any course sequence that does not violate the prerequisite structure. See Figure 9.3.

**Example 2:** In a dag with vertices representing triangles that can be pairwise ordered by depth (distance from the eye position) and with an edge from $u$ to $v$ iff $u$ and $v$ overlap in the projection plane and $u$ exceeds $v$ in depth, a topological sort is any valid triangle drawing order for the painter’s algorithm.

Note that the graph must be directed and acyclic. Also, the topological ordering is generally not unique.
A simple algorithm is to find any vertex with indegree 0, remove it, along with its edges, from the graph, and then repeat the procedure on the remainder of the graph. Then the ordering is the order of removal.

```cpp
void Graph::topsort( )
{
    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        Vertex v = findNewVertexOfIndegreeZero( );
        if( v == NOT_A_VERTEX )
            throw CycleFoundException( );
        v.topNum = counter;
        for each Vertex w adjacent to v
            w.indegree--;    
    }
}
```

**Figure 9.5** Simple topological sort pseudocode
In the simple algorithm, each scan for a vertex with indegree 0 takes $O(|V|)$ time, and there are $|V|$ scans, resulting in run time $O(|V|^2)$. A good algorithm consists of the following steps.

1. Compute and store the indegrees: $O(|E|)$.
2. Place all vertices with indegree 0 in a queue.
3. While the queue is not empty, remove a vertex $v$, decrement the indegrees of all vertices adjacent to $v$, and place any with indegree 0 in the queue.

The ordering is the order in which vertices are dequeued. The time is $O(|E| + |V|)$: each edge $(v, w)$ is processed only once because there are no cycles. Since order is not important, a stack could be used in place of the queue.
```c
void Graph::topsort( )
{
    Queue<Vertex> q;
    int counter = 0;

    q.makeEmpty( );
    for each Vertex v
        if( v.indegree == 0 )
            q.enqueue( v );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );
        v.topNum = ++counter; // Assign next number

        for each Vertex w adjacent to v
            if( --w.indegree == 0 )
                q.enqueue( w );
    }

    if( counter != NUM_VERTICES )
        throw CycleFoundException( );
}
```

**Figure 9.7** Pseudocode to perform topological sort.
Figures 9.4 and 9.6

**Figure 9.4** An acyclic graph

![Acyclic Graph Diagram](image)

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Indegree Before Dequeue #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>$v_1$</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>1</td>
</tr>
<tr>
<td>$v_3$</td>
<td>2</td>
</tr>
<tr>
<td>$v_4$</td>
<td>3</td>
</tr>
<tr>
<td>$v_5$</td>
<td>1</td>
</tr>
<tr>
<td>$v_6$</td>
<td>3</td>
</tr>
<tr>
<td>$v_7$</td>
<td>2</td>
</tr>
</tbody>
</table>

**Figure 9.6** Result of applying topological sort to the graph in Figure 9.4
In a weighted graph with cost $c_{ij}$ for edge $(v_i, v_j)$, the cost of path $v_1, v_2, \ldots, v_N$ is the **weighted path length**: $\sum_{i=1}^{N-1} c_{i,i+1}$. The **unweighted path length** is $N - 1$.

**Single-source Shortest-path Problem**: Given a weighted directed graph $G = (V, E)$, and a distinguished vertex $s$, find the shortest weighted path from $s$ to every other vertex in $G$.

The cost of finding the shortest path from $s$ to any single vertex is not faster (by more than a constant factor) than finding the path from $s$ to all vertices — at least not for any currently known algorithms.

Applications include finding the cheapest way to broadcast a message from one computer to a set of computers, where costs might be communications costs (phone bill) or delay costs (time required), etc. Another application is finding the best route between two points in a mass transit system.
Figures 9.8-9.9

Figure 9.8 A directed graph $G$

Figure 9.9 A graph with a negative-cost cycle
The graph may have negative weights which makes the problem harder. If there is a **negative-cost cycle**, the shortest paths are not defined because the cost can be made arbitrarily small (negative and large in magnitude) by going around the cycle: \(v_2, v_5, v_4\) in Figure 9.9.

The following table displays the optimal operation counts for the various shortest path problems.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unweighted</td>
<td>(O(</td>
</tr>
<tr>
<td>Weighted, no negative edges</td>
<td>(O(</td>
</tr>
<tr>
<td>Weighted, negative edges</td>
<td>(O(</td>
</tr>
<tr>
<td>Weighted, no neg. edges, acyclic</td>
<td>(O(</td>
</tr>
</tbody>
</table>
An algorithm for the unweighted case is based on a **breadth-first search**, analogous to a level-order traversal of a tree: mark $s$ as length 0; for length $l = 0, 1, 2, \ldots$, mark all the vertices that are adjacent to a length-$l$ vertex and not already marked as having length $l + 1$.

An example with $s = v_3$ is depicted in Figures 9.10 to 9.14.

**Figure 9.10** An unweighted directed graph $G$
Figures 9.11-9.12

**Figure 9.11** Graph after marking the start node as reachable in zero edges

**Figure 9.12** Graph after finding all vertices whose path length from $s$ is 1
Figures 9.13-9.14

Figure 9.13  Graph after finding all vertices whose shortest path is 2

Figure 9.14  Final shortest paths
Figure 9.16 Pseudocode for unweighted shortest-path algorithm (simple algorithm with time $O(|V|^2)$)

```cpp
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
                
            v.known = true;
            for each Vertex w adjacent to v
                if( w.dist == INFINITY )
                    
                w.dist = currDist + 1;
                w.path = v;

}
void Graph::unweighted( Vertex s )
{
  Queue<Vertex> q;

  for each Vertex v
    v.dist = INFINITY;

  s.dist = 0;
  q.enqueue( s );

  while( !q.isEmpty() )
  {
    Vertex v = q.dequeue();

    for each Vertex w adjacent to v
      if( w.dist == INFINITY )
      {
        w.dist = v.dist + 1;
        w.path = v;
        q.enqueue( w );
      }
  }
}

**Figure 9.18** Pseudocode for unweighted shortest-path algorithm (good algorithm with time $O(|E| + |V|)$)
Figure 9.19 How the data change during the unweighted shortest-path algorithm

<table>
<thead>
<tr>
<th>v</th>
<th>Initial State</th>
<th>v3 Dequeued</th>
<th>v1 Dequeued</th>
<th>v6 Dequeued</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>known  d_v  p_v</td>
<td>known  d_v  p_v</td>
<td>known  d_v  p_v</td>
<td>known  d_v  p_v</td>
</tr>
<tr>
<td>v1</td>
<td>F   ∞   0</td>
<td>F   1   v3</td>
<td>T   1   v3</td>
<td>T   1   v3</td>
</tr>
<tr>
<td>v2</td>
<td>F   ∞   0</td>
<td>F   ∞   0</td>
<td>F   2   v1</td>
<td>F   2   v1</td>
</tr>
<tr>
<td>v3</td>
<td>F   0   0</td>
<td>T   0   0</td>
<td>T   0   0</td>
<td>T   0   0</td>
</tr>
<tr>
<td>v4</td>
<td>F   ∞   0</td>
<td>F   ∞   0</td>
<td>F   2   v1</td>
<td>F   2   v1</td>
</tr>
<tr>
<td>v5</td>
<td>F   ∞   0</td>
<td>F   ∞   0</td>
<td>F   ∞   0</td>
<td>F   ∞   0</td>
</tr>
<tr>
<td>v6</td>
<td>F   ∞   0</td>
<td>F   1   v3</td>
<td>F   1   v3</td>
<td>T   1   v3</td>
</tr>
<tr>
<td>v7</td>
<td>F   ∞   0</td>
<td>F   ∞   0</td>
<td>F   ∞   0</td>
<td>F   ∞   0</td>
</tr>
</tbody>
</table>

Q: v3, v1, v6

<table>
<thead>
<tr>
<th>v</th>
<th>v2 Dequeued</th>
<th>v4 Dequeued</th>
<th>v5 Dequeued</th>
<th>v7 Dequeued</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>known  d_v  p_v</td>
<td>known  d_v  p_v</td>
<td>known  d_v  p_v</td>
<td>known  d_v  p_v</td>
</tr>
<tr>
<td>v1</td>
<td>T   1   v3</td>
<td>T   1   v3</td>
<td>T   1   v3</td>
<td>T   1   v3</td>
</tr>
<tr>
<td>v2</td>
<td>T   2   v1</td>
<td>T   2   v1</td>
<td>T   2   v1</td>
<td>T   2   v1</td>
</tr>
<tr>
<td>v3</td>
<td>T   0   0</td>
<td>T   0   0</td>
<td>T   0   0</td>
<td>T   0   0</td>
</tr>
<tr>
<td>v4</td>
<td>F   2   v1</td>
<td>T   2   v1</td>
<td>T   2   v1</td>
<td>T   2   v1</td>
</tr>
<tr>
<td>v5</td>
<td>F   3   v2</td>
<td>F   3   v2</td>
<td>T   3   v2</td>
<td>T   3   v2</td>
</tr>
<tr>
<td>v6</td>
<td>T   1   v3</td>
<td>T   1   v3</td>
<td>T   1   v3</td>
<td>T   1   v3</td>
</tr>
<tr>
<td>v7</td>
<td>F   ∞   0</td>
<td>F   3   v4</td>
<td>F   3   v4</td>
<td>F   3   v4</td>
</tr>
</tbody>
</table>

Q: v4, v5, v5, v7

v7, empty
Dijkstra’s Algorithm (1959) is a prime example of a greedy algorithm — one that solves a problem in stages by doing what appears to be optimal at each stage.

Consider the problem of making change with the smallest number of coins. A greedy algorithm starts with the largest coin and proceeds in stages to the smallest, using the maximum number of coins at each step. This works with U.S. currency, but suppose there was a 12-cent coin, and you needed to return 15 cents? The greedy algorithm may fail to be optimal.
**Figure 9.20** The directed graph $G$ (again)

- $v_1$ to $v_2$: 2
- $v_1$ to $v_3$: 4
- $v_2$ to $v_3$: 3
- $v_2$ to $v_4$: 2
- $v_3$ to $v_4$: 1
- $v_3$ to $v_6$: 5
- $v_4$ to $v_5$: 10
- $v_4$ to $v_6$: 8
- $v_4$ to $v_7$: 4
- $v_5$ to $v_7$: 6
- $v_6$ to $v_7$: 1

<table>
<thead>
<tr>
<th>$v$</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_3$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_4$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_5$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 9.21** Initial configuration of table used in Dijkstra's algorithm
### Figures 9.22-9.23

<table>
<thead>
<tr>
<th>$v$</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>F</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_4$</td>
<td>F</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 9.22** After $v_1$ is declared *known*

<table>
<thead>
<tr>
<th>$v$</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>F</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>F</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>F</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>9</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>5</td>
<td>$v_4$</td>
</tr>
</tbody>
</table>

**Figure 9.23** After $v_4$ is declared *known*
### Figures 9.24-9.25

<table>
<thead>
<tr>
<th></th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>T</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>F</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>F</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>9</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>5</td>
<td>$v_4$</td>
</tr>
</tbody>
</table>

**Figure 9.24** After $v_2$ is declared *known*

<table>
<thead>
<tr>
<th></th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>T</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>T</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>T</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>8</td>
<td>$v_3$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>5</td>
<td>$v_4$</td>
</tr>
</tbody>
</table>

**Figure 9.25** After $v_5$ and then $v_3$ are declared *known*
### Figure 9.26 After $v_7$ is declared *known*

<table>
<thead>
<tr>
<th>$v$</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>T</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>T</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>T</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>6</td>
<td>$v_7$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>T</td>
<td>5</td>
<td>$v_4$</td>
</tr>
</tbody>
</table>

### Figure 9.27 After $v_6$ is declared *known* and algorithm terminates

<table>
<thead>
<tr>
<th>$v$</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>T</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>T</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>T</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>T</td>
<td>6</td>
<td>$v_7$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>T</td>
<td>5</td>
<td>$v_4$</td>
</tr>
</tbody>
</table>
Figure 9.28 Stages of Dijkstra’s algorithm
Figure 29

/**
 * PSEUDOCODE sketch of the Vertex structure.
 * In real C++, path would be of type Vertex *,
 * and many of the code fragments that we describe
 * require either a dereferencing * or use the
 * -> operator instead of the . operator.
 * Needless to say, this obscures the basic algorithmic ideas.
 */

struct Vertex
{
    List adj;       // Adjacency list
    bool known;
    DistType dist;  // DistType is probably int
    Vertex path;    // Probably Vertex *, as mentioned above
                     // Other data and member functions as needed
};

Figure 9.29 Vertex class for Dijkstra's algorithm (pseudocode)
/**
 * Print shortest path to v after dijkstra has run.
 * Assume that the path exists.
 */

void Graph::printPath( Vertex v )
{
    if( v.path != NOT_A_VERTEX )
    {
        printPath( v.path );
        cout << " to ";
    }
    cout << v;
}

Figure 9.30 Routine to print the actual shortest path
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( ; ; )
    {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A_VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
                if( v.dist + cvw < w.dist )
                    {
                        // Update w
                        decrease( w.dist to v.dist + cvw );
                        w.path = v;
                    }
    }
}

**Figure 9.31** Pseudocode for Dijkstra’s algorithm
Theorem: Dijkstra’s algorithm is correct. Denote by $S$ the set of known vertices. We will prove the following by induction on the number of elements in $S$:

1. $v \in S \Rightarrow d(v)$ is the length of the shortest path from the source vertex to $v$.

2. $w \notin S \Rightarrow d(w)$ is the length of the shortest special path (one with all intermediate nodes in $S$) from the source to $w$.

proof: For the basis, when $S$ consists of only the source $s$, the neighbors of $s$ have their $d$ values set to the edge lengths which coincide with special path lengths, and the other vertices have $d = \infty$ — the ‘lengths’ of the nonexistent edges connecting them to $s$.

For the inductive hypothesis, assume that the two conditions hold just before a new vertex $v$ is added to $S$. 
(1) By the inductive hypothesis this condition holds for all elements of $S$ other than $v$ (those already in $S$ before the addition of $v$), and $d(v)$ is the length of the shortest special path to $v$. Suppose that the shortest path to $v$ passes through a vertex not in $S$, and denote by $x$ the first such node encountered on the path to $v$. The initial part of the path, as far as $x$, is a special path with length at least $d(x)$ by the induction hypothesis, part (2). Also, $d(x) \geq d(v)$ because the algorithm chose $v$ before $x$. Therefore, by the assumption that edge weights are nonnegative, the shortest path to $v$ cannot (or at least need not) go through $x$, and $d(v)$ is the length of the shortest path.
(2) When $v$ is added to $S$, for each node $w \notin S$ other than $v$, there are only two possibilities: either its shortest special path is unaltered or it passes through $v$. The algorithm treats the case that $v$ is the last node of $S$ in the special path to $w$. What about a special path to $w$ that includes $v$ but with some other vertex $x$ as the last node of $S$ before $w$? The algorithm correctly ignores such a path because it cannot be shorter than the path of length $d(x) + c_{x,w}$ that was already considered when $x$ was added to $S$: $d(v) + d(v,x) + c_{x,w} \geq d(x) + c_{x,w}$ because $d(v) \geq d(x)$.
The run time depends on how we find the minimum distance. Scanning takes $O(|V|)$, giving $O(|V|^2)$ for the algorithm. Updating $d_w$ is constant for each edge — $O(|E|)$. If the graph is dense, we get $O(|E|) = O(|V|^2)$ — optimal.

However, graphs are usually sparse, requiring a priority queue with selection of $v$ by $\text{deleteMin}$ (because it must be removed from the queue). We must also update $w$’s distance by one of the following methods.

- $\text{decreaseKey} — O(\log |V|)$ for both $\text{decreaseKey}$ and $\text{deleteMin}$, resulting in run time $O(|E| \log |V| + |V| \log |V|) = O(|E| \log |V|)$. This is messy because priority queues do not efficiently support find, requiring another data structure to access elements of the priority queue.
Run Time for Dijkstra’s Algorithm continued

- Insert $w$ with its new value of $d_w$ into the priority queue when $d_w$ changes, thus allowing more than one entry per vertex. Then deleteMin must check that $v$ is not already known (in which case it is ignored, and another deleteMin is performed). This is simpler and easier, but requires more space in the priority queue — as large as $|E|$. The algorithm is still $O(|E| \log |V|)$ but slower:

$$|E| \leq |V|^2 \Rightarrow \log |E| \leq 2 \log |V|.$$
Negative edge costs are rare in practice. With a negative weight, it is easy to construct a graph for which Dijkstra’s algorithm fails because, while vertex $u$ is known, there is an unknown vertex $v$ with a negative path back to $u$ so that the path from $s$ to $v$ to $u$ is better than $s$ to $u$; e.g., $c_{s,u} = 2$, $c_{s,v} = 3$, $c_{v,u} = -2$.

If we add a constant to each edge weight then negative edge costs are eliminated, but the algorithm doesn’t work because paths with more edges are unfairly penalized.

Figure 9.32 combines the unweighted edge algorithm with the weighted algorithm and works correctly, assuming there are no negative-cost cycles, but the run time is $O(|E| \cdot |V|)$ because each vertex can dequeue at most $|V|$ times. An infinite loop with an infinite-cost cycle can be avoided by terminating the algorithm if a vertex is dequeued $|V| + 1$ times.
void Graph::weightedNegative( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty() )
    {
        Vertex v = q.dequeue();

        for each Vertex w adjacent to v
            if( v.dist + cvw < w.dist )
            {
                // Update w
                w.dist = v.dist + cvw;
                w.path = v;
                if( w is not already in q )
                    q.enqueue( w );
            }
    }
}
For an acyclic graph with no negative edge weights Dijkstra’s algorithm can be improved by changing the order in which vertices are declared known (the vertex selection rule) from the minimum distance vertex, requiring a priority queue, to topological order.

The algorithm requires only one pass with selection and update taking place as the topological sort is performed.

The algorithm fails when there is a cycle because a vertex can only be selected when it has indegree zero.

Variable known is not needed. The pseudocode algorithm on the following page is similar to that of Figure 9.32 except for enqueueing \( w \) whenever \( w \) has indegree 0. It assumes that indegrees have been stored — \( O(|V|) \). Since there is no priority queue, the run time is \( O(|E| + |V|) \).
void Graph::acyclicDijkstra( Vertex s ) {
    Queue<Vertex> q;
    q.makeEmpty();
    for each Vertex v {
        if (v.indegree == 0) q.enqueue(v);
        v.dist = INFINITY;
        v.path = NOT_A_VERTEX; }
    s.dist = 0;
    while (!q.isEmpty()) {
        vertex v = q.dequeue();
        for each Vertex w adjacent to v {
            if ( --w.indegree == 0 ) q.enqueue(w);
            if (v.dist + Cv,w < w.dist) {
                decrease w.dist to v.dist + Cv,w;
                w.path = v; } } }
}
1. Downhill skiing problem: if a path can only go downhill, no cycles are possible.

2. Nonreversible chemical reaction: vertices are states, edges are state transitions, and edge weights are energy releases. The graph is acyclic if only high energy to low energy transitions are possible.

3. Critical path analysis: nodes are activities with time requirements (activity-node graph), edges represent precedence relationships: \((v, w)\) means \(v\) must be completed before \(w\) begins. Hence an acyclic graph. Activities that do not depend on each other may be done in parallel by different servers. An example is a construction project. Important questions are the earliest completion time and which activities can be delayed without affecting the earliest completion time.
The activity-node graph is converted into an *event-node graph* in which each node is an event representing completion of an activity and all its dependent activities. Edges are activities with time requirements as weights. Dummy edges and nodes are added so that the indegree of ordinary nodes is 1 (except for the first one). Refer to Figures 9.33 and 9.34. Define

- \( EC_i = \) Earliest completion time for node \( i \)
- \( LC_i = \) Latest completion time for node \( i \) such that the final completion time is not affected
- \( \text{Slack}_{(v,w)} = \) Maximum delay time for activity \( (v, w) \) such that the final completion time is not affected

For vertices (nodes) 1, 2, ..., \( n \)

\[
EC_1 = 0, \quad EC_w = \max_{(v,w) \in E} (EC_v + c_{v,w})
\]

\[
LC_n = EC_n, \quad LC_v = \min_{(v,w) \in E} (LC_w - c_{v,w})
\]

\[
\text{Slack}_{(v,w)} = LC_w - EC_v - c_{v,w}
\]
Figures 9.33-9.34

**Figure 9.33** Activity-node graph

**Figure 9.34** Event-node graph
Figures 9.35-9.36

**Figure 9.35** Earliest completion times

**Figure 9.36** Latest completion times
Earliest completion time, latest completion time, and slack

- $EC_w$ is computed from a list of reverse adjacencies of $w$ using topological order.
- $LC_v$ is computed from $v$’s adjacency list using reverse topological order.
- Critical activities are those with zero slack time.
- A critical path consists entirely of zero-slack edges. There is necessarily at least one such path.
We assume an undirected graph. The directed graph problem appears to be more difficult.

**Defn:** A *minimum spanning tree* in an undirected graph is a tree formed from graph edges that connects all the vertices at lowest total cost.

- A minimum spanning tree exists if and only if the graph is connected but is not necessarily or usually unique.
- The number of edges is $|V| - 1$. It’s a tree because it is acyclic. It is spanning because it includes every vertex.
- For any spanning tree $T$, adding an edge to $T$ creates a cycle, and removing an edge from the cycle reinstates the spanning tree property.

An example application is the problem of wiring a house with a minimum of cable and no other (electrical) constraints.
Figure 9.48 A graph $G$ and its minimum spanning tree
Prim’s algorithm is a greedy algorithm. Choose a starting vertex and then, at each stage choose the minimum edge \((u, v)\) such that \(u \in T\) and \(v \notin T\).

The algorithm is the same as Dijkstra’s algorithm (Figure 9.31) except that there are two adjacencies per edge, distance \(v.\text{dist} = d_v\) is the weight of the shortest edge connecting \(v\) to a known vertex, and the update rule is \(d_w = \min(d_w, c_{v,w})\) for each unknown \(w\) adjacent to selected vertex \(v\).

The analysis is the same as Dijkstra’s algorithm: \(O(|E| \log |V|)\) for a sparse graph using a binary heap, or \(O(|V|^2)\) for a dense graph.
Figure 9.49 Prim’s algorithm after each stage
Figures 9.50-9.51

<table>
<thead>
<tr>
<th>v</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_3$</td>
<td>F</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>$v_4$</td>
<td>F</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>$v_5$</td>
<td>F</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 9.50** Initial configuration of table used in Prim’s algorithm

<table>
<thead>
<tr>
<th>v</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>F</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>F</td>
<td>4</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>F</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 9.51** The table after $v_1$ is declared known
### Figures 9.52-9.53

<table>
<thead>
<tr>
<th>$v$</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>F</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>F</td>
<td>2</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>F</td>
<td>7</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>8</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>4</td>
<td>$v_4$</td>
</tr>
</tbody>
</table>

**Figure 9.52** The table after $v_4$ is declared *known*

<table>
<thead>
<tr>
<th>$v$</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>T</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>T</td>
<td>2</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>F</td>
<td>7</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>5</td>
<td>$v_3$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>4</td>
<td>$v_4$</td>
</tr>
</tbody>
</table>

**Figure 9.53** The table after $v_2$ and then $v_3$ are declared *known*
### Figures 9.54-9.55

<table>
<thead>
<tr>
<th>$v$</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>T</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>T</td>
<td>2</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>F</td>
<td>6</td>
<td>$v_7$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>1</td>
<td>$v_7$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>T</td>
<td>4</td>
<td>$v_4$</td>
</tr>
</tbody>
</table>

**Figure 9.54** The table after $v_7$ is declared *known*

<table>
<thead>
<tr>
<th>$v$</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>T</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>T</td>
<td>2</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>T</td>
<td>6</td>
<td>$v_7$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>T</td>
<td>1</td>
<td>$v_7$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>T</td>
<td>4</td>
<td>$v_4$</td>
</tr>
</tbody>
</table>

**Figure 9.55** The table after $v_6$ and $v_5$ are selected (Prim’s algorithm terminates)
Kruskal’s algorithm is another greedy algorithm. At each stage select the smallest edge that does not create a cycle.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Weight</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(v_1, v_4)$</td>
<td>1</td>
<td>Accepted</td>
</tr>
<tr>
<td>$(v_6, v_7)$</td>
<td>1</td>
<td>Accepted</td>
</tr>
<tr>
<td>$(v_1, v_2)$</td>
<td>2</td>
<td>Accepted</td>
</tr>
<tr>
<td>$(v_3, v_4)$</td>
<td>2</td>
<td>Accepted</td>
</tr>
<tr>
<td>$(v_2, v_4)$</td>
<td>3</td>
<td>Rejected</td>
</tr>
<tr>
<td>$(v_1, v_3)$</td>
<td>4</td>
<td>Rejected</td>
</tr>
<tr>
<td>$(v_4, v_7)$</td>
<td>4</td>
<td>Accepted</td>
</tr>
<tr>
<td>$(v_3, v_6)$</td>
<td>5</td>
<td>Rejected</td>
</tr>
<tr>
<td>$(v_5, v_7)$</td>
<td>6</td>
<td>Accepted</td>
</tr>
</tbody>
</table>

**Figure 9.56** Action of Kruskal’s algorithm on $G$

$|V| = 7 \Rightarrow |E| = 6$. Hence, the algorithm is terminated when $|V| - 1 = 6$ edges have been accepted. The worst-case run time is $O(|E| \log |V|)$ — the same as Prim but, in practice, it is faster.
Figure 9.57 Kruskal’s algorithm after each stage
void Graph::kruskal()
{
    int edgesAccepted = 0;
    DisjSet ds( NUM_VERTICES );
    PriorityQueue<Edge> pq( getEdges() );
    Edge e;
    Vertex u, v;

    while( edgesAccepted < NUM_VERTICES - 1 )
    {
        pq.deleteMin( e );    // Edge e = (u, v)
        SetTypeuset = ds.find( u );
        SetTypevset = ds.find( v );
        if(uset != vset)
        {
            // Accept the edge
            edgesAccepted++;
            ds.unionSets(uset, vset);
        }
    }
}
A **depth-first search** is a generalization of preorder traversal from trees to graphs: starting at some vertex \( v \), process \( v \) and then recursively traverse all vertices adjacent to \( v \).

For a tree, all vertices are visited from the root in \( \Theta(|E|) = \Theta(|V|) \) time. For an arbitrary graph we avoid cycles by marking each visited vertex \( v \), and recursively calling \( \text{dfs} \) on all adjacent unmarked vertices. If the graph is unconnected we need an outer loop on vertices: \( O(|E| + |V|) \).

```java
for each Vertex v
    v.visited = false; // Unmark v
for each Vertex v
    if (!v.visited) dfs(v);
```
Figure 9.59

void Graph::dfs( Vertex v )
{
    v.visited = true;
    for each Vertex w adjacent to v
        if( !w.visited )
            dfs( w );
}

Figure 9.59 Template for depth-first search (pseudocode)
9.6.2 Biconnectivity

A depth-first search in a connected undirected graph produces a *depth-first spanning tree*.

We can add a preorder numbering associated with the order in which vertices are marked: \( \text{Num}(v) \)

**Defn:** A connected undirected graph is *biconnected* if there are no vertices whose removal disconnects the rest of the graph.

If a graph is not biconnected, the vertices whose removal would disconnect the graph are *articulation points*. These are critical. If one of these is disrupted, there is no alternate path for network traffic (mail or transportation).
Figures 9.60-9.61

Figure 9.60 An undirected graph

Figure 9.61 Depth-first search of previous graph
Depth-first search provides a linear-time algorithm to find all articulation points in a connected graph. First compute $Num(v)$ in a preorder traversal; then compute $Low(v)$ as the lowest-numbered vertex that is reachable from $v$ by taking 0 or more tree edges followed by 0 or 1 back edges (graph edges not in the tree) for each vertex $v$.

$Low(v)$ is the minimum of

1. $Num(v)$ (0 tree edges, 0 back edges)
2. $\min_{(v,w)\in\{\text{back edges}\}} Num(w)$ (0 tree edges, 1 back edge)
3. $\min_{(v,w)\in\{\text{tree edges}\}} Low(w)$ (1 or more tree edges, 0 or 1 back edges)

The third option is computed recursively by a postorder traversal.
Figure 9.62 A graph with articulation points C and D
Figure 9.63 Depth-first tree for previous graph, with \textit{Num} and \textit{Low}

Figure 9.64 Depth-first tree that results if depth-first search starts at C
Theorem:

- The root is an articulation point iff it has more than one child.
- Any other vertex \( v \) is an articulation point iff \( v \) has a child \( w \) such that \( \text{Low}(w) \geq \text{Num}(v) \).

Note that the latter condition is always satisfied by the root which must be treated as a special case.

proof: Suppose \( v \) has a child \( w \) such that \( \text{Low}(w) \geq \text{Num}(v) \). Then the only way to reach any node lower than \( v \) from \( w \) is by going through \( v \). Since \( \text{Num}(v) > 1 \), \( v \) is an articulation point.

Conversely, suppose all children of \( v \) have \( \text{Low} \) values smaller than \( \text{Num}(v) \). Then either \( v \) has no children or its children are connected to some node lower than \( v \), and \( v \) is not an articulation point. □
/**
 * Assign num and compute parents.
 */
void Graph::assignNum( Vertex v )
{
    v.num = counter++;
    v.visited = true;
    for each Vertex w adjacent to v
    
    if( !w.visited )
    {
        w.parent = v;
        assignNum( w );
    }
}
Pseudocode to compute Low and to test for articulation points (test for the root is omitted)

```cpp
/**
 * Assign low; also check for articulation points.
 */
void Graph::assignLow( Vertex v )
{
    v.low = v.num; // Rule 1
    for each Vertex w adjacent to v
    {
        if( w.num > v.num ) // Forward edge
        {
            assignLow( w );
            if( w.low >= v.num )
                cout << v << " is an articulation point" << endl;
            v.low = min( v.low, w.low ); // Rule 3
        }
        else
            if( v.parent != w ) // Back edge
                v.low = min( v.low, w.num ); // Rule 2
    }
}
```

**Figure 9.66** Pseudocode to compute Low and to test for articulation points (test for the root is omitted)
void Graph::findArt( Vertex v )
{
    v.visited = true;
    v.low = v.num = counter++;  // Rule 1
    for each Vertex w adjacent to v
    {
        if( !w.visited )  // Forward edge
        {
            w.parent = v;
            findArt( w );
            if( w.low >= v.num )
                cout << v << " is an articulation point" << endl;
            v.low = min( v.low, w.low );  // Rule 3
        }
        else
        {
            if( v.parent != w )  // Back edge
                v.low = min( v.low, w.num );  // Rule 2
        }
    }
}

**Figure 9.67** Testing for articulation points in one depth-first search (test for the root is omitted) (pseudocode)
**Puzzle**: Given a connected undirected graph, find a path or circuit that visits every edge exactly once. (Draw all edges exactly once without lifting the pen.)

In 1736 Euler invented graph theory and solved this problem — the **Euler path** (tour) or **Euler circuit** problem.

**Theorem**: An Euler circuit exists iff every vertex has even degree. An Euler path exists iff exactly two vertices have odd degree, and the path must start at one of them. The circuit or path, when it exists can be found in linear time by a depth-first search.

![Three drawings](image)

**Figure 9.68** Three drawings
Algorithm: Generate a path by a depth-first search, terminating only when there is no unmarked edge incident on the final vertex (which must be the initial vertex). Then find the first vertex on this path that has an unmarked edge, and perform another search; splice the new circuit into the old one.

**Figure 9.70** Graph for Euler circuit problem

First circuit: 5, 4, 10, 5 (Figure 9.71)
Second circuit: 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4
Splice: 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5 (Figure 9.72)
Third circuit: 3, 2, 8, 9, 6, 3
Euler Circuits continued

Figure 9.71 Graph remaining after 5, 4, 10, 5

Figure 9.72 Graph after the path 5, 4, 1, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5
Euler Circuits continued

The path is stored as a linked list so that splicing is cheap.

Along with each adjacency list, we store a pointer to the last edge scanned to avoid repetitious scanning.

When a path is spliced in, the search for the first vertex with an unmarked edge (start of next search) must begin at the splice point so that the total search time is $O(|E|)$. 

**Figure 9.73** Graph remaining after the path 5, 4, 1, 3, 2, 8, 9, 6, 3, 7, 4, 11, 10, 7, 9, 3, 4, 10, 5
The Euler circuit problem has a linear-time solution, but a similar problem is much harder. The Hamiltonian cycle problem is to find a simple cycle in an undirected graph that contains every vertex. The best known algorithm for this problem is not only not linear; it’s not even polynomial; it could take exponential time on some inputs.

The single-source unweighted shortest-path problem for directed graphs is also solvable in linear time, but the corresponding longest simple path problem is hard — exponential time.
9.7.1 Easy vs. Hard

The easiest problems require linear time. We’ve seen $O(\log N)$ operation counts, but only after the data has been read in $O(N)$ time, or in the case of arithmetic problems such as $\gcd(M,N)$, in which the time is $O(\log N)$, but $N$ has log $N$ bits, so that the time is linear in the number of bits.

At the other extreme are unsolvable problems — called **undecidable**; e.g., the **halting problem** is to decide if a program will terminate (as opposed to having an infinite loop). This is **recursively undecidable**. How would the checking program check itself? Suppose the halting problem was decidable. Then we could write a program \texttt{LOOP} that takes as input a program $P$ and runs $P$ on itself, then prints \texttt{YES} if $P$ loops (does not halt) or goes into an infinite loop otherwise. Running \texttt{LOOP} on itself results in a logical contradiction. Therefore \texttt{LOOP} cannot exist.
Any algorithm whose time complexity cannot be bounded by a polynomial is an \textbf{exponential time algorithm} even though this includes functions such as $n^{\log n}$, which is not normally regarded as exponential.

This simple distinction is remarkably effective for distinguishing between tractable and \textit{intractable problems}.

- It makes little difference how the problem is encoded or what computer model is used (unless perhaps the model is a quantum computer).
- Most exponential time algorithms are merely variations of exhaustive search, whereas polynomial time algorithms reflect a deeper understanding of the problem.
- For small problems the exponential time may be better.
The complexities are worst-case measures. There are some well-known algorithms (simplex for LP and branch-and-bound for knapsack) that are exponential on some problem instances but very fast for most problems that arise in practice. Such algorithms are rare.

The polynomially solvable problems that arise naturally tend to have time complexities of degree 2 or 3 and reasonably small constants.

Some problems (e.g., find all tours in a TSP having total length less than some bound) have solution of size exponential in the problem size, but these are unrealistic problems.

Intractable problems can be treated by approximation methods for finding near-optimal solutions.
9.7.2 The Class NP

*NP* stands for nondeterministic polynomial-time — a class of problems that can be solved in polynomial time on a hypothetical *nondeterministic machine* which is free to choose any sequence of instructions, and will always make the optimal choice at every step. This has not been proven to be as big an advantage as one might expect.

A problem with a yes/no answer (a decision problem) is in class *NP* if every instance for which the correct solution is “yes” can be decided in polynomial time. The Hamiltonian cycle problem, for example, is the question of whether an undirected graph has a simple circuit that includes all vertices. This is in *NP* because a nondeterministic machine, given a graph with a Hamiltonian cycle, could generate such a cycle and test that it is simple and includes every vertex in polynomial time; i.e., since the problem of checking a path for correctness is in *P* (deterministic polynomial time), the decision problem is in *NP*.
NP Problems

NP is the set of decision problems solvable in polynomial time on a non-deterministic Turing machine. Equivalently, it is the set of problems whose solutions can be verified by a deterministic Turing machine in polynomial time.

Examples

- Find a nontrivial divisor of a given integer (integer factorization).
- Find a simple path of length $k$ that goes through all nodes in an undirected graph (traveling salesman).
- Given a set of integers, find a subset that sums to zero (subset sum problem).

Since polynomial-time solution implies polynomial-time verification (check), we have $P \subset NP$.

It has not been proven that $P \neq NP$ (or that $P = NP$). Proving that a problem has an exponential lower bound is extremely difficult. (There's a $1,000,000$ Clay prize.)
A problem $P_2$ in $NP$ is **NP-complete** if any problem $P_1$ in $NP$ can be *polynomially reduced* to it: any instance of $P_1$ can be transformed to an instance of $P_2$ in polynomial time. Thus, if an $NP$-complete problem has a polynomial-time solution, then so does every problem in $NP$; i.e., $NP$-complete problems are the hardest problems in $NP$.

Suppose problem $P_1 \in NP$ and some known $NP$-complete problem $P_2$ is polynomially reducible to $P_1$. Then $P_1$ is $NP$-complete. This is the usual, sometimes easy, way to prove that a problem is $NP$-complete.
Given a complete graph $G = (V, E)$ with edge costs, and an integer $k$, is there a simple cycle that visits all vertices and has total cost at most $k$?

Assume we know that the Hamiltonian cycle problem is $NP$-complete. Clearly, TSP $\in NP$. We can show that TSP is $NP$-complete by showing that we can polynomially reduce the Hamiltonian cycle problem to it. Given a graph $G$ for HCP, construct $G'$ with the same vertices but added edges to make it complete. Assign weight 1 to edges that were in $G$, weight 2 to new edges. Let $k = |V|$. Then $G$ has a Hamiltonian cycle iff $G'$ has a Traveling Salesman tour of total weight $k = |V|$. 
Figure 9.78 Hamiltonian cycle problem transformed to traveling salesman problem
The *satisfiability problem* is that of deciding if a Boolean expression is satisfied (true) for some set of values of its variables.

This is clearly in $NP$. In 1971, Cook showed that it was $NP$-complete by proving that all problems in $NP$ could be polynomially reduced to it. He used the one known fact about $NP$ — non-deterministic polynomial time. He showed that a Turing machine could be simulated by an extremely complicated and long (but polynomial) Boolean formula that would be true iff the program being run produced a “yes” answer. This was the first $NP$-complete problem.

In 1972 Richard Karp presented a collection of $NP$-complete problems.
Figure 9.79 Graph used in Exercises 9.1 and 9.11

Figure 9.80 Graph used in Exercise 9.5
Figures 9.82-9.83

**Figure 9.82** Graph used in Exercise 9.15

**Figure 9.83** Graph used in Exercise 9.21