Chapter 9
Graph Algorithms
Introduction

- graph theory
  - useful in practice
  - represent many real-life problems
  - can be ________ if not careful with data structures
an undirected graph $G = (V, E)$ is a finite set $V$ of ___________ together with a set $E$ of ___________

an edge is a pair $(v, w)$, where $v$ and $w$ are vertices

this definition allows

___________, or edges that connect vertices to themselves

parallel edges, or multiple edges that connect the same pair of vertices

a graph without self-loops is a simple graph

a graph with parallel edges is sometimes called a multigraph
Definitions

- two vertices are **adjacent** if there is an _____ between them
  - the edge is said to be **incident** to the two vertices

- if there are no parallel edges, the **degree** of a vertex is the number of edges ______________ to it
  - self-loops add only ____ to the degree

- a **subgraph** of a graph $G$ is a subset of $G$'s edges together with the incident vertices
Definitions

- A path in a graph is a sequence of vertices connected by edges.
  - A simple path is a path with no repeated vertices, except possibly the first and last.

- A cycle is a path of at least one edge whose first and last vertices are the same.
  - A simple cycle is a cycle with no repeated edges of vertices other than the first and last.

- The length of a path is the number of edges in the path.
Definitions

- A graph is **connected** if every vertex is connected to every other vertex by a ________ through the graph.

- A **connected component** $G'$ of a graph $G$ is a maximal connected subgraph of $G$: if $G'$ is a subset of $F$ and $F$ is a connected subgraph of $G$, then $F = G'$.

- A graph that is not connected consists of a set of connected ____________

- A graph without cycles is called **acyclic**.
- A tree is a connected, acyclic _______________ graph
- A forest is a _______________ set of trees
- A spanning tree of a connected graph is a subgraph that is a tree and also contains all of the graph's _____________
- A spanning forest of a graph is the union of spanning trees of its connected components
Definitions

– if $|V|$ is the number of vertices and $|E|$ is the number of edges, then, in a graph without self-loops and parallel edges, there are $|V| (|V| - 1)/2$ possible ____________

– a graph is **complete** if there is an edge between every _________ of vertices

– the **density** of a graph refers to the proportion of possible pairs of vertices that are connected

– a **sparse** graph is one for which $|E| \ll |V|(|V| - 1)/2$

– a **dense** graph is a graph that is not ____________

– a **bipartite** graph is one whose vertices can be divided into two sets so that every vertex in one set is connected to at least one vertex in the other set
Definitions

- in a **directed graph** or **digraph** the pairs \((v, w)\) indicating edges are \[ \text{__________} \]: the edge \((v, w)\) goes from \(v\) (the tail) to \(w\) (the head)
- since edges have a direction, we use the notation \(v \rightarrow w\) to denote an edge from \(v\) to \(w\)
- edges in digraphs are frequently called **arcs**
- the \[ \text{__________} \] of a vertex \(w\) is the number of arcs \(v \rightarrow w\) (i.e., the number of arcs coming into \(w\)), while the **outdegree** of \(w\) is the number of arcs \(w \rightarrow v\) (i.e., the number of arcs exiting \(w\))
- we will call \(w\) a **source** if its indegree is _____
- an **aborescence** is a directed graph with a distinguished vertex \(u\) (the root) such that for every other vertex \(v\) there is a unique directed path from \(u\) to \(v\)
in a directed graph, two vertices $v$ and $w$ are ___connected___ if there is a directed path from $v$ to $w$ and a directed path from $w$ to $v$

a digraph is strongly connected if all its vertices are strongly connected

if a digraph is not strongly connected but the underlying undirected graph is connected, then the digraph is called weakly connected

a weighted graph has weights or costs associated with each ___

weighted graphs can be directed or undirected

a road map with mileage is the prototypical example
Example

- airport connections

http://allthingsgraphed.com/2014/08/09/us-airlines-graph/
Graph Representation

- two concerns: ___________ and ___________
- we’ll consider directed graphs, though undirected graphs are similar
- the following graph has 7 vertices and 12 edges
Graph Representation

- adjacency matrix
- 2D matrix where an element is 1 if \((u, v) \in A\) and 0 otherwise
Graph Representation

- adjacency matrix
  - alternatively, could use costs $\infty$ or $-\infty$ for ____________
  - not efficient if the graph is ______ (number of edges small)
    - matrix $O(|V|^2)$
    - e.g., street map with 3,000 streets results in intersection matrix with 9,000,000 elements
- adjacency list
  - standard way to represent graphs
  - undirected graph edges appear ________ in list
  - more efficient if the graph is sparse (number of edges small)
    - matrix $O(|E| + |V|)$
Graph Representation

- adjacency list
- for each vertex, keep a list of vertices
Graph Representation

- adjacency list alternative
- for each vertex, keep a ___________ of adjacent vertices
Topological Sort

- a directed acyclic graph (DAG) is a digraph with no directed cycles
  - a DAG always has at least one __________

- topological sort
  - an ordering of the vertices in a directed graph such that if there is a path from v to w, then v appears __________ w in the ordering
  - not possible if graph has a __________
Topological Sort

-example directed acyclic graph

W&M Computer Science prerequisite chart

- Math 413 Numerical Analysis I
- Math 414 Numerical Analysis II
- 429 Special Topics Prerequisites vary by topic
- 495-496 Honors
- 411 Computational Problem Solving
- 241 Discrete Structures
- 243 Discrete Structures
- 131 Concepts in CS
- 135 Web Design*
- 498 Internship
- 312 Programming Languages
- 304 Computer Organization
- 301 Software Development
- 303 Algorithms
- 442 Compiler Construction*
- 435 Software Engineering
- 415 Systems Programming
- 454 Computer Architectures*
- 424 Computer & Network Security
- 426 Simulation I
- 427 Computer Graphics I
- 421 Database Systems
- 423 Finite Automata
- 432 Web Programming*

- 434 Network Systems & Design
- 444 Operating Systems

- † Math 112 prerequisite
- † Math 211 prerequisite
- †May substitute Math 214
- - - one of the other
- * not offered 2016-2017
- Revised 04-16-16
Topological Sort

- topological sort
  - determine the indegree for every \( v \in V \)
  - place all source vertices in a queue
  - while there remains a \( v \in V \)
    - find a \underline{___________} vertex
    - append the source vertex to the topological sort
    - delete the source and its adjacent \underline{__________} from \( G \)
    - update the \underline{______________} of the remaining vertices in \( G \)
    - place any new source vertices in the queue
  - when no vertices remain, we have our ordering, or, if we are missing vertices from the output list, the graph has no topological sort
- since finding vertex with 0 indegree must look at _______ vertices, and this is performed $|V|$ times, $O(|V|^2)$
Topological Sort

- instead, we can keep all the vertices with indegree 0 in a ______ and choose from there

\(-O(|E| + |V|)\)
Topological Sort

- adjacency list alternative
  - for each ________, keep a vector of adjacent vertices
Shortest-Path Algorithms

- shortest-path problems
  - input is a weighted graph with a ________ on each edge
  - weighted path length: $\sum_{i=1}^{N-1} c_{i,i+1}$

- single-source shortest-path problem
  - given as input a weighted graph, $G = (V, E)$ and a __________ vertex $s$, find the shortest weighted path from $s$ to every other vertex in $G$
Shortest-Path Algorithms

- example
  - shortest weighted path from $v_1$ to $v_6$ has cost of ______
  - no path from $v_6$ to $v_1$
Shortest-Path Algorithms

- ______________ edges can cause problems
- path from \( v_5 \) to \( v_4 \) has cost of 1, but a shorter path exists by following the negative loop, which has cost -5
- shortest paths thus ______________

![Graph with nodes and edges labeled with weights](image)
many examples where we want to find shortest paths

if vertices represent computers and edges connections, the cost represents _______________ costs, delay costs, or combination of costs

if vertices represent airports and edges costs to travel between them, shortest path is _______________ route

we find paths from one vertex to _______ others since no algorithm exists that finds shortest path from one vertex to one other faster
Shortest-Path Algorithms

- four problems
  - unweighted shortest-path
  - weighted shortest-path with no negative edges
  - weighted shortest-path with negative edges
  - weighted shortest-path in acyclic graphs
Unweighted Shortest Paths

- unweighted shortest-path
- find shortest paths from $s$ to all other vertices
- only concerned with number of ___________ in path
- we will not actually record the path ________________
Unweighted Shortest Paths

- example
- start with $v_3$
- example
  - mark 0 length to $v_3$
Unweighted Shortest Paths

- example
  - mark 1 length for $v_1$ and $v_6$
Unweighted Shortest Paths

- example
  - mark 2 length for $v_2$ and $v_4$
Unweighted Shortest Paths

- example
  - final path assignments
Unweighted Shortest Paths

- searching an unweighted shortest-path uses a __________________ search
  - processes vertices in layers, according to _______________
  - begins with initializing path lengths

<table>
<thead>
<tr>
<th></th>
<th>known</th>
<th>d_v</th>
<th>p_v</th>
</tr>
</thead>
<tbody>
<tr>
<td>v₁</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>v₂</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>v₃</td>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>v₄</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>v₅</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>v₆</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>v₇</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>

- a vertex will be marked __________ when the shortest path to it is found
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTEXES; 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;
                            }
                }
Unweighted Shortest Paths

- with this algorithm
  - path can be printed
  - running time: $O(|V|^2)$
- bad case

  ![Graph diagram]

- can reduce time by keeping vertices that are unknown \___________ from those known
- new running time: $O(|E| + |V|)$
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 );
            }
    }
}
## Unweighted Shortest Paths

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

Q: $v_3$, $v_1$, $v_6$, $v_6$, $v_2$, $v_4$, $v_2$, $v_4$

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

Q: $v_4$, $v_5$, $v_5$, $v_7$, $v_7$, empty
Dijkstra’s Algorithm

- weighted shortest-path – Dijkstra’s algorithm
- more difficult, but ideas from ____________ algorithm can be used
- keep information as before for each vertex
  - known
  - set ____________ \( d_w = d_v + c_{v,w} \) if \( d_w = \infty \) using only known vertices
  - \( p_v \) the last vertex to cause a change to \( d_v \)
- ____________ algorithm
  - does what appears to be best thing at each stage
  - e.g., counting money: count quarters first, then dimes, nickels, pennies
  - gives change with least number of coins
Dijkstra’s Algorithm

- pseudocode
- assumption: no weights
- origin $s$ is given

Initialization: $S \leftarrow \{s\}$ and $D \leftarrow V - \{s\}$.  
Set $\text{dist}[s] \leftarrow 0$ and $\text{dist}[v] \leftarrow \infty$ for all other $v$.

While there remains a $v \in D$:

1. Select a vertex $v \in D$ which has the shortest path length from $s$ to $v$ using only vertices in $S$ (e.g., known vertices).
2. $S \leftarrow S \cup \{v\}$ and $D \leftarrow D - \{v\}$. 
Dijkstra's Algorithm

- pseudocode (cont.)

```plaintext
def Dijkstra(G, s):
    # Initialize distances and known status
    dist = {v: float('inf') for v in G}
    known = {v: False for v in G}
    dist[s] = 0

    while True:
        v = argmin(v for v in G if not known[v])
        known[v] = True

        for w in G[v]:
            if not known[w]:
                # Edge relaxation
                new_dist = min(dist[w], dist[v] + G[v][w]
                if new_dist < dist[w]:
                    dist[w] = new_dist
                    from[w] = v
```

Dijkstra’s Algorithm

- example: start at $v_1$
Dijkstra’s Algorithm

- example

\[
\begin{array}{c}
v_1 \\
v_2 \\
v_3 \\
v_4 \\
v_5 \\
v_6 \\
v_7
\end{array}
\]

\[
\begin{array}{c}
\text{known} \\
\text{d}_v \\
p_v
\end{array}
\]

\[
\begin{array}{cccc}
v_1 & T & 0 & 0 \\
v_2 & F & 2 & v_1 \\
v_3 & F & \infty & 0 \\
v_4 & F & 1 & v_1 \\
v_5 & F & \infty & 0 \\
v_6 & F & \infty & 0 \\
v_7 & F & \infty & 0
\end{array}
\]

\[
\begin{array}{cccc}
v_1 & T & 0 & 0 \\
v_2 & F & 2 & v_1 \\
v_3 & F & 3 & v_4 \\
v_4 & T & 1 & v_1 \\
v_5 & F & 3 & v_4 \\
v_6 & F & 9 & v_4 \\
v_7 & F & 5 & v_4
\end{array}
\]

\[
\begin{array}{cccc}
v_1 & T & 0 & 0 \\
v_2 & T & 2 & v_1 \\
v_3 & F & 3 & v_4 \\
v_4 & T & 1 & v_1 \\
v_5 & F & 3 & v_4 \\
v_6 & F & 9 & v_4 \\
v_7 & F & 5 & v_4
\end{array}
\]
Dijkstra’s Algorithm

– example

\[
v_1 \quad v_2 \\
\downarrow \quad \downarrow \\
v_3 \quad v_4 \quad v_5 \\
\downarrow \quad \downarrow \quad \downarrow \\
v_6 \quad v_7 \\
\]

<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>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>

\( v_5 \), \( v_3 \)

<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>

\( v_7 \)

<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>

\( v_6 \)
Dijkstra’s Algorithm

- example – stages shown on the graph
Dijkstra’s Algorithm

– example – stages shown on the graph (cont.)
Dijkstra’s Algorithm: Correctness

**Proposition.** Dijkstra’s algorithm solves the single-origin shortest-paths problem in a weight digraph with nonnegative weights.

**Proof.** If $v$ is reachable from $s$, then every edge $v ightarrow w$ is relaxed exactly once, when $v$ is relaxed, resulting in

$$\text{dist}[w] \leq \text{dist}[v] + \text{weight}[v \rightarrow w].$$

This inequality holds until the algorithm terminates, since

1. dist[w] can only decrease, because relaxations can only decrease a dist[] value, and
2. dist[v] never changes, because edge weights are nonnegative and we choose the lowest dist[] value at each step, so no later relaxation can reduce dist[v].

Thus, after all vertices reachable from $s$ have been added to the shortest paths tree, the shortest paths optimality conditions hold.
Dijkstra’s Algorithm

- complexity
  - sequentially scanning vertices to find minimum $d_v$ takes $O(|V|)$, which results in $O(|V|^2)$ overall
  - at most one update per edge, for a total of $O(|E| + |V|^2) = O(|V|^2)$
    - if graph is dense, with $|E| = \Theta(|V|^2)$, algorithm is close to ________________
    - if graph is sparse, with $|E| = \Theta(|V|)$, algorithm is too ____________
  - distances could be kept in a ________________ queue that reduces running time to $O(|E| + |V| \lg |V|)$
Dijkstra’s Algorithm

- implementation
  - information for each vertex

```c
/**
 * 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
};
```
- implementation (cont.)
- path can be printed recursively

```cpp
/**
 * 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;
}
```
Dijkstra’s Algorithm

-implementation (cont.)

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

    s.dist = 0;

    while( there is an unknown distance vertex )
    {
        Vertex v = smallest unknown distance vertex;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
            {
                DistType cvw = cost of edge from v to w;

                if( v.dist + cvw < w.dist )
                {
                    // Update w
                    decrease( w.dist to v.dist + cvw );
                    w.path = v;
                }
            }
    }
}
```
Graphs with Negative Edges

- try to apply Dijkstra’s algorithm to graph with _________ edges

Label each node with best known distance from origin $a$. Relax the edges adjacent to $a$. Select $b$, the closest node to $S$, and add it to $S$. Relax the outgoing edges adjacent to $b$. Select $c$, the closest node to $S$, and add it to $S$. We’ve now assimilated all nodes into $S$, so we’re done.
Graphs with Negative Edges

- possible solution: ________ a delta value to all weights such that none are negative
  - calculate _______________ path on new graph
  - apply path to original graph
  - does not work: longer paths become weightier
- combination of algorithms for weighted graphs and unweighted graphs can work
  - drastic _______________ in running time: $O(|E| \cdot |V|)$
All-Pairs Shortest Paths

- given a weighted digraph, find the shortest paths between _____ vertices in the graph
- one approach: apply Dijkstra's algorithm _______________________
  - results in $O(|V|^3)$
- another approach: apply Floyd-Warshall algorithm
  - uses ________________ programming
  - also results in $O(|V|^3)$
Minimum Spanning Tree

- assumptions
  - graph is ________________
  - edge weights are not necessarily Euclidean distances
  - edge weights need not be all the same
  - edge weights may be zero or ________________

- minimum spanning tree (MST)
  - also called minimum-weight spanning tree of a weighted graph
  - spanning tree whose weight (the sum of the weights of the edges in the tree) is the ________________ among all spanning trees
Minimum Spanning Tree

example
Minimum Spanning Tree

- two algorithms to find the minimum spanning tree
  - Prim’s Algorithm
  - Kruskal’s Algorithm
Minimum Spanning Tree

- Prim’s algorithm
- grows tree in successive stages

Initialization: \( S \leftarrow \{s\} \) and \( D \leftarrow V - \{s\} \).

While there remains a \( v \in D \):

1. Find an edge with minimum weight \( (u, v) \) such that \( u \in S \) and \( v \in D \).
2. \( S \leftarrow S \cup \{v\} \) and \( D \leftarrow D - \{v\} \).
Minimum Spanning Tree

- Prim’s algorithm: example
Minimum Spanning Tree

- Prim’s algorithm: example 2
- Prim’s algorithm: example 2 (cont.)
Minimum Spanning Tree

- Prim’s algorithm: example 2 (cont.)
  - $v_1$, $v_4$, $v_2$ & $v_3$, $v_7$, $v_6$ & $v_5$

![Graph diagram]

<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>

<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>
Minimum Spanning Tree

- Prim’s algorithm
  - runs on _________________ graphs
  - running time: $O(|V|^2)$ without heaps, which is optimal for ____________ graphs
  - running time: $O(|E| \lg |V|)$ using binary heaps, which is good for ______________ graphs
Minimum Spanning Tree

- Kruskal’s algorithm
  - continually select the edges in order of ________________ weight
  - accept the edge if it does not cause a ________________ with already accepted edges
Minimum Spanning Tree

- Kruskal’s algorithm: example
Minimum Spanning Tree

- Kruskal’s algorithm: example 2
Minimum Spanning Tree

- Kruskal’s algorithm: example 2 (cont.)
Minimum Spanning Tree

- Kruskal’s algorithm
  - time complexity: $O(|E| \lg|E|)$ with proper choice and use of data structures
  - in the worst case, $|E| = \Theta(|V|^2)$, so the worst-case time complexity is $O(|E| \lg|V|)$
What problems can we solve algorithmically? Which problems are easy? Which problems are hard?

- Eulerian circuit: Given a vertex $s$, start at $s$ and find a cycle that visits every ___________ exactly once
  - easy: solvable in $O(|E| + |V|)$ using depth-first search

- Hamiltonian circuit: Given a vertex $s$, start at $s$ and find a cycle that visits each remaining ___________ exactly once
  - really, really hard!
NP-Complete Problems

- halting problem
  - in 1936, A. Church and A. Turing independently proved the non-solvability of the halting problem:
    - is there an algorithm $\text{terminates}(p,x)$ that takes an arbitrary program $p$ and input $x$ and returns True if $p$ terminates when given input $x$ and False otherwise?

- difficult: try to run it on _______________
NP-Complete Problems

- halting problem
  - suppose we had such an algorithm $\text{terminates}(p,x)$
  - create a new program:

    ```
    program evil (z) {
        1: if terminates(z,z) goto 1
    }
    ```

  - program $\text{evil}()$ terminates if and only if the program $z$ does not terminate when given its own code as input
  - no such algorithm can exist
NP-Complete Problems

- decision problem
  - has a yes or no answer
  - undecidable if it is ________________ to construct a single algorithm that will solve all instances of the problem
- the halting problem is ________________
NP-Complete Problems

- the class $P$
  - set of problems for which there exists a _________________ time algorithm for their solution
  - the runtime is bounded by a polynomial function of the __________ of the problem

- the class $NP$
  - set of decision problems for which the certification of a candidate __________ as being correct can be performed in polynomial time
  - non-deterministic polynomial time
NP-Complete Problems

- the class $NP$
  - for problems in $NP$, certifying a solution may not be difficult, but ___________ a solution may be very difficult
  - example: Hamiltonian circuit
    - given a graph $G$, is there a simple cycle in $G$ that includes every ___________
    - given a candidate solution, we can check whether it is a simple cycle in time $\propto |V|$, simply by ___________ the path
    - however, finding a Hamiltonian circuit is hard!
NP-Complete Problems

- reductions
  - problem $A$ reduces to problem $B$ if the solvability of $B$ implies the solvability of $A$
    - if $A$ is reducible to $B$, then $B$ is at least as hard to solve as $A$
  - in the context of algorithms, reducibility means an algorithm that solves $B$ can be reduced into an algorithm to solve $A$
    - example: if we can sort a set of numbers, we can find the median, so finding the median reduces to sorting
- reductions
  - problem $A$ can be __________ reduced to $B$ if we can solve problem $A$ using an algorithm for problem $B$ such that the cost of solving $A$ is
    
    \[
    \text{cost of solving } B + \text{a polynomial function of the problem size}
    \]

- example: once we have sorted an array $a[]$ of $N$ numbers, we can find the median in __________ time by computing $\frac{N}{2}$ and accessing $a[\frac{N}{2}]$
NP-Complete Problems

- reductions
- decision version of traveling salesperson problem (TSP):
- given a complete weighted graph and an integer $K$, does there exist a simple cycle that ___________ all vertices (tour) with total weight $\leq K$?
- clearly, this is in $NP$
- Hamiltonian circuit: given a graph $G = (V, E)$, find a simple cycle that visits all the vertices
- construct a new graph $G'$ with the same vertices as $G$ but which is ___________; if an edge in $G'$ is in $G$, give it weight 1; otherwise, give it weight 2
- construction requires $O(|E| + |V|)$ work
- apply ________ to see if there exists a tour with total weight $|V|$
NP-Complete Problems

- reductions
NP-Complete Problems

- \( NP \)-complete
  - a problem \( A \) is \( NP \)-complete if it is in \( NP \) and all other problems in \( NP \) can be \underline{polynomially} to \( A \) in polynomial time

- Boolean satisfiability (SAT): given a set of \( N \) boolean variables and \( M \) logical statements built from the variables using \textit{and} and \textit{not}, can you choose values for the variables so that all the statements are \underline{true}?
  \[(x_1 \text{ AND } !x_2 \text{ AND } x_3), (!x_1 \text{ AND } x_7), (x_1 \text{ AND } x_{42}), \ldots\]

- SAT is \( NP \)-complete
– *NP*-complete
  – if we restrict attention to sets of boolean statements involving _____ variables, the problem is known as 3-SAT
  – 3-SAT is *NP*-complete
  – so, if you can solve 3-SAT in polynomial time, you can solve ________ problems in *NP* in polynomial time
  – meanwhile, 2-SAT is solvable in ______________ time!
NP-Complete Problems

- $NP$-complete problems
  - traveling salesperson
  - bin packing
  - knapsack
  - graph coloring
  - longest-path
NP-Complete Problems

- *NP*-hard problems
  - a problem $A$ is *NP*-hard if there exists a polynomial-time reduction from an *NP*-complete problem to $A$
  - an *NP*-hard problem is at __________ as hard as an *NP*-complete problem
- ________________ versions of *NP*-complete problems are typically *NP*-hard
  - optimization version of TSP: given a weighted graph, find a ________________ cost Hamiltonian circuit
  - if we can solve TSP, we can solve Hamiltonian circuit
Bin Packing

We are given \( n \) items of lengths \( \ell_1, \ldots, \ell_n \), where \( 0 < \ell_i \leq 1 \) for all \( i \).

The items must be packed in bins of length 1, and they must be placed end-to-end. Once placed in a bin, items cannot be moved.

How do we pack them in a way that uses the fewest number of bins? This is an NP-hard problem!

Input: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8.

Optimal packing:
- \( B_1: 0.2, 0.8 \)
- \( B_2: 0.7, 0.3 \)
- \( B_3: 0.4, 0.1, 0.5 \)

Online: item must be put in a bin before the next item is considered.

Offline: all the items are available for consideration at one time.
**Theorem.** No algorithm for online bin packing can always give an optimal solution.

**Proof.** Let $\varepsilon > 0$ be small (say, $\varepsilon = 1/64$), and consider an input sequence of $m$ items of length $\frac{1}{2} - \varepsilon$ followed by $m$ items of length $\frac{1}{2} + \varepsilon$.

Clearly, the optimal packing requires $m$ bins.

Suppose online algorithm $A$ yields this optimal solution. Since it is online, $A$ must place each of the first $m$ items in a separate bin.

Now give $A$ an input sequence of just $m$ items of length $\frac{1}{2} - \varepsilon$. $A$ will behave as before, placing each in a separate bin, thus using $m$ bins.

However, the optimal packing in this case requires only $\lceil m/2 \rceil$ bins.
Since an online algorithm never knows when the input might end, any performance guarantee for the algorithm must hold whenever an item is binned.

**Theorem.** There are inputs that cause any online bin packing algorithm to use at least $\frac{4}{3}$ the optimal number of bins.

**Proof.** Suppose not. Then there is an online algorithm $A$ that always uses less than $\frac{4}{3}$ the optimal number of bins.

Let $m$ be even. Apply $A$ to an input sequence of $m$ items of length $\frac{1}{2} - \varepsilon$ followed by $m$ items of length $\frac{1}{2} + \varepsilon$.

Consider the situation after $A$ has processed item $m$ (the last of the smaller items). Suppose $A$ has used $b$ bins at this point.
We know that the optimal number of bins for the first $m$ items is $m/2$, so our performance assumption means $b < \frac{4}{3} \frac{m}{2}$, or $2b/m < \frac{4}{3}$.

Now consider the situation when $A$ is finished packing all $2m$ items. All the bins used after bin $b$ can only contain one item since the inputs are the longer items.

The first $b$ bins can have at most 2 items each, and the remaining bins have one item each, so packing all $2m$ items requires at least $2m - b$ bins.

Since the optimal packing requires $m$ bins, the performance assumption means $2m - b < \frac{4}{3}m$, or $(2m - b)/m < \frac{4}{3}$. 
Thus we have two inequalities that hold:

\[ \frac{2b}{m} < \frac{4}{3} \]  

after the first \( m \) items,

\[ \frac{2m - b}{m} < \frac{4}{3} \]  

after the last \( m \) items.

From the first inequality we obtain \( b/m < \frac{2}{3} \), while from the second we obtain \( b/m > \frac{2}{3} \), which is a contradiction.
The next fit heuristic: when binning an item, check to see if it fits in the bin with the last item binned. If it does, place the new item there; otherwise, start a new bin.

Input: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8.

$B_1$: 0.2, 0.5
$B_2$: 0.4
$B_3$: 0.7, 0.1
$B_4$: 0.3
$B_5$: 0.8
Bin Packing

**Theorem**

Let $m$ be the optimal number of bins required to pack a list of items. Then next fit never uses more than $2m$ bins. There exist sequences for which next fit uses $2m - 2$ bins.

This is an example of an approximation result.

Next fit is an approximation algorithm; it is not guaranteed to produce optimal solutions, but there exists a bound on how poorly it can do, even though we do not know the optimal solution.
Bin Packing

Proof: Suppose next fit uses \( b \) bins. Let \( L_j \) be the total length of items in bin \( j \); then

\[
L_1 + L_2 + \cdots + L_b = \text{total length of items} \leq m.
\]

Consider any adjacent bins \( B_j \) and \( B_{j+1} \). The combined lengths of the items in these two bins must be larger than 1; otherwise all of these items would have been placed in \( B_j \):

\[
L_j + L_{j+1} > 1.
\]

For simplicity, assume \( b \) is even. Then

\[
m \geq (L_1 + L_2) + (L_3 + L_4) + \cdots + (L_{b-1} + L_{b-2})
\]

\[
> 1 + 1 + \cdots + 1 = \frac{b}{2},
\]

\( b/2 \) times

so \( b < 2m \).
Bin Packing

For the second part of the theorem, suppose \( n \), the number of items, is divisible by 4, and choose

\[
\ell_i = \begin{cases} 
1/2 & \text{if } i \text{ is odd}, \\
2/n & \text{if } i \text{ is even}.
\end{cases}
\]

The optimal packing requires \( n/4 \) bins containing 2 items of length \( 1/2 \) and one bin containing the \( n/2 \) items of length \( 2/n \).

Next fit, on the other hand, uses \( n/2 \) bins, with each bin containing one long and one short item.
The first fit heuristic: when binning an item, scan over all the bins in order and place the new item in the first bin that has room for it. If no such bin exists, open a new one.

Input: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8.

\[ B_1: 0.2, 0.5, 0.1 \]
\[ B_2: 0.4, 0.3 \]
\[ B_3: 0.7 \]
\[ B_4: 0.8 \]
Bin Packing

**Theorem**

Let $m$ be the optimal number of bins required to pack a list of items. Then first fit never uses more than $\frac{17}{10}m + \frac{7}{10}$ bins. There exist sequences for which next fit uses $\frac{17}{10}(m - 1)$ bins.

Consider a sequence of $6N$ items of size $\frac{1}{7} + \varepsilon$, followed by $6N$ items of size $\frac{1}{3} + \varepsilon$, followed by $6N$ items of size $\frac{1}{2} + \varepsilon$.

First fit will require $10N$ bins. However, if we apply next fit, we obtain a packing that requires only $6N$ bins.
The **best fit** heuristic: when binning an item, scan over all the bins in order and place the new item in the tightest spot.

Input: 0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8.

\[ B_1 : 0.2, 0.5, 0.1 \]
\[ B_2 : 0.4 \]
\[ B_3 : 0.7, 0.3 \]
\[ B_4 : 0.8 \]

In general, the worst-case behavior is the same as first fit.
Bin Packing

In offline bin packing, we can first sort the items, longest to shortest, and then pack them according to first fit or best fit.

The corresponding heuristics are called first fit decreasing and best fit decreasing (more properly they should be first fit nonincreasing and best fit nonincreasing).

First fit decreasing for: 0.8, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1.

\[B_1: 0.8, 0.2\]
\[B_2: 0.7, 0.3\]
\[B_3: 0.5, 0.4, 0.1\]

In this case we get the optimal solution.
Bin Packing

Theorem

Let \( m \) be the optimal number of bins required to pack a list of items. Then first fit decreasing never uses more than \( \frac{11}{9} m + \frac{6}{9} \) bins. There exist sequences for which first fit decreasing uses \( \frac{11}{9} m + \frac{6}{9} \) bins.