Chapter 20

Graphs
• Set $V$ of vertices and $E$ of edges
Subgraph

- Subset of vertices and edges

(a) A campus map as a graph; (b) a subgraph
Adjacent vertices

• Joined by an edge
  – Library–Dormitory
  – Library–Student Union
Path

• A sequence of edges that begins at one vertex and ends at another vertex

• May pass through the same vertex more than once
Simple path

- A path that passes through a vertex only once
Cycle

• A path that begins and ends at the same vertex
  – Library–Dormitory–Student Union–Gymnasium–Student Union–Library
Simple cycle

- A cycle that does not pass through a vertex more than once
  - Library–Dormitory–Student Union–Library
Connected graph

- A graph that has a path between each pair of distinct vertices
Disconnected graph

- A graph that has at least one pair of vertices without a path between them
Complete graph

- A graph that has an edge between each pair of distinct vertices
• A graph whose edges have numeric labels
• Edges do not indicate a direction
• Each edge is a directed edge
  – Vertex $y$ is adjacent to vertex $x$ if there is a directed edge from $x$ to $y$
Graphs as ADTs

• Vertices may or may not contain values

• A graph may have directed or undirected edges

• Insertion/deletion operations apply to vertices and edges

• Graphs may have traversal operations
• n vertices numbered 0, 1, ..., n – 1

• An n by n array matrix such that 
  matrix[i][j] is 
  • 1 (or true) if there is an edge from vertex i to vertex j 
  • 0 (or false) if there is no edge from vertex i to vertex j
Adjacency Matrix
Unweighted/Directed

Directed Graph

```
Directed Graph

Adjacency Matrix

0 1 2 3 4 5 6 7 8
P 0 0 1 0 0 1 0 0 0
Q 0 0 0 0 0 0 1 0 0
R 0 0 0 0 0 0 1 0 0
S 0 0 0 0 0 0 1 0 0
T 0 0 0 0 0 0 1 0 0
W 0 0 0 0 0 0 1 0 0
X 0 0 0 0 0 0 0 0 0
Y 0 0 0 0 0 0 0 0 0
Z 0 0 0 0 0 0 0 0 0
```
• n vertices numbered 0, 1, ..., n – 1

• An n by n array matrix such that matrix[i][j] is
  – The weight that labels the edge from vertex i to vertex j if there is an edge from i to j
  – $\infty$ if there is no edge from vertex i to vertex j
Adjacency Matrix
Weighted/Undirected

(a)  

A
8
B
1
6
D
9
C
3
2

(b)

\[
\begin{array}{c|cccc}
& A & B & C & D \\
\hline
0 & \infty & 8 & \infty & 6 \\
1 & 8 & \infty & 9 & \infty \\
2 & \infty & 9 & \infty & \infty \\
3 & 6 & \infty & \infty & \infty \\
\end{array}
\]
Adjacency List – Directed

• n vertices numbered 0, 1, ..., n – 1
  – Consists of n linked lists

  – The $i^{th}$ linked list has a node for vertex $j$ if and only if the graph contains an edge from vertex $i$ to vertex $j$

  – The node can contain either
    • Vertex $j$’s value, if any
    • An indication of vertex $j$’s identity
Adjacency List
Unweighted/Directed

(a) Graph

(b) Adjacency List

0: P
1: Q
2: R
3: S
4: T
5: W
6: X
7: Y
8: Z

R → W
X → Y
X → T
T → W
S → Y
R → Z
Implementing Graphs

• Two common operations on graphs
  – Determine whether there is an edge from vertex i to vertex j
  – Find all vertices adjacent to a given vertex i

• Adjacency matrix
  – Supports operation 1 more efficiently

• Adjacency list
  – Supports operation 2 more efficiently
  – Often requires less space than an adjacency matrix
Graph – Using the STL

• An adjacency list representation of a graph may be implemented with a vector of maps

• For a weighted graph
  – The vector elements represent the vertices of a graph
  – The map for each vertex contains element pairs
    • Each pair consists of an adjacent vertex and an edge weight
class Edge {
    public:
    int v, w, weight;
    Edge(int firstVertex, int secondVertex, int edgeWeight) {
        v = firstVertex;
        w = secondVertex;
        weight = edgeWeight;
    }
};

class Graph {
    public:
    int numVertices;
    int numEdges;
    vector< map<int, int> > adjList;
    Graph(int n);
    int GetNumVertices() const;
    int GetNumEdges() const;
    int GetWeight(Edge e) const;
    void Add(Edge e);
    void Remove(Edge e);
    map<int, int>::iterator FindEdge(int v, int w);
};
Graph Traversals

• Visits all vertices of the graph if and only if the graph is connected
  – A connected component – The subset of vertices visited during a traversal that begins at a given vertex

• To prevent indefinite loops
  • Mark each vertex during a visit, and
  • Never visit a vertex more than once
• Depth-First Search (DFS) Traversal
  – Proceeds along a path from a vertex v as deeply into the graph as possible before backing up
  – A last visited, first explored strategy
  – Has a simple recursive form
  – Has an iterative form that uses a stack
DFS :: Recursive

dfs(v: Vertex) {
    Mark v as visited
    for (each unvisited vertex u adjacent to v) {
        dfs(u)
    } // end for
} // end dfs()
DFS :: Iterative

dfs(v: Vertex) {
    s = a new empty stack

    // Push v onto the stack and mark it
    s.push(v)
    Mark v as visited
    while (!s.isEmpty()) {
        if (all adjacent nodes of top vertex have been visited)
            pop()
        else {
            Select an unvisited vertex u adjacent to the vertex on the top of the stack
            push(u)
            Mark u as visited
        } // end if
    } // end while
} // end dfs()
DFS: A

A

B
C
E

D

G
H

F
I
DFS: ABE
DFS: ABE

A - B - C - D - E

G - D - H - F - I
DFS: ABEC DH

A B C D E

G D H I F
DFS: ABECDHG
DFS:ABECDHGI
DFS:ABECGHIF
BFS Traversal

- Breadth-First Search (BFS) Traversal
  - Visits every vertex adjacent to a vertex \( v \) that it can before visiting any other vertex
  - A first visited, first explored strategy
  - An iterative form uses a queue
  - A recursive form is possible, but not simple
BFS :: Iterative

```java
bfs(v: Vertex) {
    q = a new empty queue

    // Add v to the queue and mark it
    q.add(v)
    Mark v as visited

    while (!q.isEmpty()) {
        w = q.remove()
        for (each unvisited vertex u adjacent to w) {
            Mark u as visited
            q.add(u)
        }
    }
}
```
BFS: A

A -- B -- C
   |     |   
   |     |   
   G -- D -- H

             
             |   
             |   
             E -- C -- F
             |   |
             |   |
             I -- H

40
BFS: ABCE
BFS: ABCDEI

A — B — E
A — C — D — I
A — C — H — G
H — F
BFS: ABCEDI
BFS: ABCDEI H
BFS: ABCEDIHFG
BFS – Using the STL

• A BFS class can be implemented with the STL \textit{vector} and \textit{queue} containers

• Two vectors of integers
  – \textit{mark} stores vertices that have been visited
  – \textit{parents} stores the parent of each vertex for use by other graph algorithms.

• A queue of \textit{Edges}
  – BFS processes the edges from each vertex’s adjacency list in the order that they were pushed onto the queue
• **Topological order**
  – A list of vertices in a directed graph without cycles such that vertex x precedes vertex y if there is a directed edge from x to y in the graph
  – There may be several topological orders in a given graph

• **Topological sorting**
  – Arranging the vertices into a topological order
Topological Sorting

A directed graph without cycles

The graph above arranged according to the topological orders:

(a) $a, g, d, b, e, c, f$

(b) $a, b, g, d, e, f, c$
'backward' topo sort

- Find a vertex that has no successor
- Remove that vertex and all edges that lead to it, and add the vertex to the beginning of a list of vertices
- Add each subsequent vertex that has no successor to the beginning of the list
'backward' topo sort

topSort1(in theGraph: Graph, out aList: List)

n = number of vertices in theGraph
for (step = 1 through n) {
    Select a vertex v that has no successors
    aList.insert(1,v)
    Delete from theGraph vertex v and its edges
}
'backward' topo sort
'backward' topo sort

L: c

Diagram:
'backward' topo sort

L: f c

Diagram:

- a connected to b
- d connected to e
- g connected to d
'backward' topo sort

L: f c

\[
\begin{align*}
\text{a} & \rightarrow \text{b} \\
\text{d} & \rightarrow \text{e} \\
\text{g} & \\
\end{align*}
\]
'backward' topo sort

L: e f c

Diagram of a directed graph with nodes a, b, d, g, and edges from g to d, d to a, a to b.
'backward' topo sort

L: e f c

Diagram:

- Vertex e
- Vertex f
- Vertex c
- Vertex a
- Vertex b
- Vertex d
- Vertex g

Edges:
- e to a
- f to a
- c to a
- a to b
- b to d
- d to g
'backward' topo sort

L: b e f c

Diagram:
- a
- d
- g
'backward' topo sort

L: d b e f c

\[ \text{a} \]

\[ \text{g} \]
'backward' topo sort

L: a d b e f c
'backward' topo sort

L: a d b e f c
'backward' topo sort

L: g a d b e f c
'forward' topo sort

• A modification of the iterative DFS algorithm

• Push all vertices that have no predecessor onto a stack

• Each time you pop a vertex from the stack, add it to the beginning of a list of vertices

• When the traversal ends, the list of vertices will be in topological order
'forward' topo sort

topSort2(in theGraph:Graph, out aList:List)

    s.createStack()
    for (all vertices in the graph){
        if (v has no predecessors) {
            s.push(v)
            Mark v as visited
        }
    }
    while (!s.isEmpty()) {
        if (v has unvisited adjacent nodes) {
            Select an unvisited vertex u adjacent to v
            s.push(u)
            Mark u as visited
        } else {
            s.pop(v)
            aList.insert(1, v)
        }
    }
'forward' topo sort

S:

Diagram of a directed graph with nodes labeled a, b, c, d, e, f, and g.
'forward' topo sort

S: a
'forward' topo sort

S: a g

Diagram:

- Vertex a
- Vertex b connected to a
- Vertex c connected to b
- Vertex d connected to a, b, c
- Vertex e connected to d
- Vertex f connected to e
- Vertex g connected to a
'forward' topo sort

S: a g
'forward' topo sort

S: a g d
'forward' topo sort

S: a g d
'forward' topo sort

S: a g d e

Diagram:

- Nodes: a, b, c, d, e, f, g
- Edges: a -> b, a -> d, a -> g, b -> c, d -> e, e -> f, e -> g
'forward' topo sort

S: a g d e

Diagram of the directed graph with vertices labeled a, b, c, d, e, f, and g, and edges connecting them as shown.
'forward' topo sort

S: a g d e c
'forward' topo sort

S: a g d e c f
'forward' topo sort

S: a g d e c

L: f
'forward' topo sort

S: a g d e

L: c f
'forward' topo sort

S: a g d

L: e c f
'forward' topo sort

S: a g

L: d e c f
'forward' topo sort

S: a

L: g d e c f
'forward' topo sort

S: a

L: g d e c f
'forward' topo sort

$S$: a b

$L$: g d e c f
'forward' topo sort

S: a

L: b g d e c f
'forward' topo sort

S:

L: a b g d e c f
• A connected undirected subgraph of $G$ that contains all of $G$’s vertices and enough of its edges to form a tree
Spanning Trees

Remove edges until there are no cycles

A connected graph with cycles

A spanning tree for the graph
Spanning Trees

• Detecting a cycle in an undirected connected graph
  – A connected undirected graph that has \( n \) vertices must have at least \( n - 1 \) edges
  
  – A connected undirected graph that has \( n \) vertices and exactly \( n - 1 \) edges cannot contain a cycle

  – A connected undirected graph that has \( n \) vertices and more than \( n - 1 \) edges must contain at least one cycle
Spanning Trees

• Connected graphs that each have four vertices and three edges
To create a depth-first search (DFS) spanning tree:

- Traverse the graph using a depth-first search and mark the edges that you follow.

- After the traversal is complete, the graph’s marked vertices and edges form the spanning tree.
The DFS Spanning Tree

dfsTree(in v:Vertex)

Mark v as visited
for (each unvisited vertex u adjacent to v) {
    Mark the edge from u to v
    dfsTree(u)
}
DFS: A

Diagram of a depth-first search (DFS) starting at node A.
DFS: AB
DFS: ABE

A

G

H

B

C

D

E

F

I
DFS: ABEC
DFS:ABECD
DFS: ABEC DH
DFS: ABECDHG
DFS: ABEC DHGI
DFS:ABECDHGIF
The BFS Spanning Tree

• To create a breath–first search (BFS) spanning tree
  – Traverse the graph using a bread–first search and mark the edges that you follow
  – When the traversal is complete, the graph’s marked vertices and edges form the spanning tree
The BFS Spanning Tree

bfsTree(in v:Vertex)

q.createQueue()
q.enqueue(v)
Mark v as visited

while(!q.isEmpty()) {

d.dequeue(w)

for (each unvisited vertex u adjacent to w) {
    Mark u as visited
    Mark edge between w and u
    q.enqueue(u)
}
}
BFS: A
BFS: ABC
BFS: A B C E
BFS: ABCE DIH
BFS: ABCEDIHFG
BFST: ABCEDIHF

A

B

C

D

E

F

G

H

I

A -> B -> C -> D
C -> E
C -> F

G -> H
Minimum Spanning Trees

• Sum of all edge costs is minimum

• There may be several minimum spanning trees for a particular graph
Prim's Algorithm

- Finds a minimal spanning tree that begins at any vertex
  - Find the least-cost edge \((v, u)\) from a visited vertex \(v\) to some unvisited vertex \(u\)
  - Mark \(u\) as visited
  - Add the vertex \(u\) and the edge \((v, u)\) to the minimum spanning tree
  - Repeat the above steps until there are no more unvisited vertices
MST: A

Diagram of a minimum spanning tree (MST) with vertices labeled A, B, C, D, E, F, G, H, I. Edges and their weights are indicated as follows:

- A to B: 2
- A to C: 3
- B to C: 1
- C to F: 3
- C to I: 2
- D to C: 1
- D to H: 4
- H to G: 5
- E to I: 3
- F to I: 3

Weights are shown for each edge connecting the vertices.
MST: AB

A

B

C

D

E

F

G

H

I

1

2

3

4

5
MST: ABE
MST:ABECD
MST: ABECDI
MST: ABECDF
MST: ABECIDF HG

Graph with edges and weights:

- A to B: 2
- B to C: 1
- C to D: 1
- D to H: 4
- H to G: 5
- C to E: 3
- E to I: 3
- I to F: 3
- F to D: 2
- C to H: 3
Dijkstra’s

• Finds the shortest path between a given origin and all other vertices

• `vertexSet` stores vertices along the path

• `weight[v]` is the weight of the shortest (cheapest) path from vertex 0 to vertex `v` that passes through vertices in `vertexSet`
Dijkstra

--- | --- | --- | --- | --- | --- | --- | --- |
1   | 0 | -   | 0   | 0   | 8   | -   | 9   |

First row of adjacency matrix
Dijkstra

\[
\begin{array}{cccccc}
1 & - & 0 & 0 & 8 & - & 9 & 4 \\
\end{array}
\]

\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{dijkstra.png}
\end{figure}
Dijkstra

-----|---|-----|------|------|------|------|------|
1    |   |     | 0    | 0    | 8    | -    | 9    | 4    |
2    | 4 | 0,4 | 0    | 8    | 5    | 9    | 4    |
Dijkstra

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>-</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0,4</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>4</td>
</tr>
</tbody>
</table>
Dijkstra

<table>
<thead>
<tr>
<th>Step</th>
<th>(v)</th>
<th>set</th>
<th>(w[0])</th>
<th>(w[1])</th>
<th>(w[2])</th>
<th>(w[3])</th>
<th>(w[4])</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>-</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0,4</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0,4,2</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
Dijkstra

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>8</td>
<td>-</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0,4</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0,4,2</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
Dijkstra

\[
\begin{array}{c|c|c|c|c|c|c}
\hline
1 & - & 0 & 0 & 8 & - & 9 & 4 \\
2 & 4 & 0,4 & 0 & 8 & 5 & 9 & 4 \\
3 & 2 & 0,4,2 & 0 & 7 & 5 & 8 & 4 \\
4 & 1 & 0,4,2,1 & 0 & 7 & 5 & 8 & 4 \\
\end{array}
\]
Dijkstra

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>-</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0,4</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0,4,2</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0,4,2,1</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
### Dijkstra

![Diagram of a graph with nodes and edges labeled with weights.]

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>8</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0,4</td>
<td>0</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0,4,2</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0,4,2,1</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>0,4,2,1,3</td>
<td>0</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>
Summary

• The two most common implementations of a graph are the adjacency matrix and the adjacency list

• Graph searching
  – Depth-first search goes as deep into the graph as it can before backtracking
  – Bread-first search visits all possible adjacent vertices before traversing further into the graph
Summary

• Topological sorting produces a linear order of the vertices in a directed graph without cycles.

• Trees are connected undirected graphs without cycles.

• A spanning tree of a connected undirected graph
  – A subgraph that contains all the graph’s vertices and enough of its edges to form a tree.
Summary

• A minimum spanning tree for a weighted undirected graph
  – A spanning tree whose edge-weight sum is minimal

• The shortest path between two vertices in a weighted directed graph
  – The path that has the smallest sum of its edge weights