GRAPHS – Definitions

- A graph $G = (V, E)$ consists of
  - a set of vertices, $V$, and
  - a set of edges, $E$, where each edge is a pair $(v, w)$ s.t. $v, w \in V$
- Vertices are sometimes called nodes, edges are sometimes called arcs.
- If the edge pair is ordered then the graph is called a directed graph (also called digraphs).
- We also call a normal graph (which is not a directed graph) an undirected graph.
  - When we say graph we mean that it is an undirected graph.
Graph – Definitions

• Two vertices of a graph are **adjacent** if they are joined by an edge.
• Vertex w is **adjacent to** v iff \((v,w) \in E\).
  – In an undirected graph with edge \((v, w)\) and hence \((w,v)\) w is adjacent to v and v is adjacent to w.
• A **path** between two vertices is a sequence of edges that begins at one vertex and ends at another vertex.
  – i.e. \(w_1, w_2, \ldots, w_N\) is a path if \((w_i, w_{i+1}) \in E\) for \(1 \leq i \leq N-1\)
• A **simple path** passes through a vertex only once.
• A **cycle** is a path that begins and ends at the same vertex.
• A **simple cycle** is a cycle that does not pass through other vertices more than once.
Graph – An Example

The graph $G = (V, E)$ has 5 vertices and 6 edges:

$V = \{1, 2, 3, 4, 5\}$

$E = \{ (1,2),(1,3),(1,4),(2,5),(3,4),(4,5),(2,1),(3,1),(4,1),(5,2),(4,3),(5,4) \}$

- **Adjacent:**
  - 1 and 2 are adjacent -- 1 is adjacent to 2 and 2 is adjacent to 1

- **Path:**
  - 1,2,5 (a simple path), 1,3,4,1,2,5 (a path but not a simple path)

- **Cycle:**
  - 1,3,4,1 (a simple cycle), 1,3,4,1,4,1 (cycle, but not simple cycle)
Graph -- Definitions

- A **connected graph** has a path between each pair of distinct vertices.
- A **complete graph** has an edge between each pair of distinct vertices.
  - A complete graph is also a connected graph. But a connected graph may not be a complete graph.
Directed Graphs

• If the edge pair is ordered then the graph is called a **directed graph** (also called *digraphs*).
• Each edge in a directed graph has a direction, and each edge is called a **directed edge**.
• Definitions given for undirected graphs apply also to directed graphs, with changes that account for direction.
• Vertex w is **adjacent to** v iff (v,w) \(\in\) E.
  – i.e. There is a direct edge from v to w
  – w is **successor** of v
  – v is **predecessor** of w
• A **directed path** between two vertices is a sequence of directed edges that begins at one vertex and ends at another vertex.
  – i.e. \(w_1, w_2, \ldots, w_N\) is a path if \((w_i, w_{i+1}) \in E\) for \(1 \leq i \leq N-1\)
Directed Graphs

• A **cycle** in a directed graph is a path of length at least 1 such that \( w_1 = w_N \).
  – This cycle is simple if the path is simple.
  – For undirected graphs, the edges must be distinct

• A **directed acyclic graph (DAG)** is a type of directed graph having no cycles.

• An undirected graph is **connected** if there is a path from every vertex to every other vertex.

• A directed graph with this property is called **strongly connected**.
  – If a directed graph is not strongly connected, but the underlying graph (without direction to arcs) is connected then the graph is **weakly connected**.
The graph $G = (V,E)$ has 5 vertices and 6 edges:

$V = \{1,2,3,4,5\}$
$E = \{(1,2),(1,4),(2,5),(4,5),(3,1),(4,3)\}$

- **Adjacent:**
  2 is adjacent to 1, but 1 is NOT adjacent to 2
- **Path:**
  1,2,5 (a directed path),
- **Cycle:**
  1,4,3,1 (a directed cycle),
Weighted Graph

• We can label the edges of a graph with numeric values, the graph is called a *weighted graph*.

![Weighted (Undirected) Graph](image1)

![Weighted Directed Graph](image2)
Graph Implementations

• The two most common implementations of a graph are:
  – *Adjacency Matrix*
    • A two dimensional array
  – *Adjacency List*
    • For each vertex we keep a list of adjacent vertices
Adjacency Matrix

• An **adjacency matrix** for a graph with \( n \) vertices numbered 0,1,...,n-1 is an \( n \) by \( n \) array matrix such that \( \text{matrix}[i][j] \) is 1 (true) if there is an edge from vertex \( i \) to vertex \( j \), and 0 (false) otherwise.

• When the graph is weighted, we can let \( \text{matrix}[i][j] \) be the weight that labels the edge from vertex \( i \) to vertex \( j \), instead of simply 1, and let \( \text{matrix}[i][j] \) equal to \( \infty \) instead of 0 when there is no edge from vertex \( i \) to vertex \( j \).

• Adjacency matrix for an undirected graph is symmetrical.
  – i.e. \( \text{matrix}[i][j] \) is equal to \( \text{matrix}[j][i] \)

• Space requirement \( O(|V|^2) \)

• Acceptable if the graph is dense.
Adjacency Matrix – Example 1

A directed graph

Its adjacency matrix

\[
\begin{array}{cccccccccc}
 & P & Q & R & S & T & W & X & Y & Z \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Adjacency Matrix – Example 2

An Undirected Weighted Graph

Its Adjacency Matrix

```
+---+---+---+---+---+
|   | 0 | 1 | 2 | 3 |
+---+---+---+---+---+
| A | ∞ | 8 | ∞ | 6 |
| B | 8 | ∞ | 9 | ∞ |
| C | ∞ | 9 | ∞ | ∞ |
| D | 6 | ∞ | ∞ | ∞ |
+---+---+---+---+---+
```
Adjacency List

• An *adjacency list* for a graph with *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*.

• Adjacency list is a better solution if the graph is sparse.

• Space requirement is O(|E| + |V|), which is linear in the size of the graph.

• In an undirected graph each edge (v,w) appears in two lists.
  – Space requirement is doubled.
Adjacency List – Example 1

A directed graph

Its Adjacency List

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

- 0: R
- 1: X
- 2: R
- 3: X
- 4: T
- 5: W
- 6: S
- 7: Y
- 8: Z
Adjacency List – Example2

An Undirected Weighted Graph

Its Adjacency List
Adjacency Matrix vs Adjacency List

- Two common graph operations:
  1. Determine whether there is an edge from vertex i to vertex j.
  2. Find all vertices adjacent to a given vertex i.

- An adjacency matrix supports operation 1 more efficiently.
- An adjacency list supports operation 2 more efficiently.

- An adjacency list often requires less space than an adjacency matrix.
  - Adjacency Matrix: Space requirement is $O(|V|^2)$
  - Adjacency List: Space requirement is $O(|E| + |V|)$, which is linear in the size of the graph.
  - Adjacency matrix is better if the graph is dense (too many edges)
  - Adjacency list is better if the graph is sparse (few edges)
Graph Traversals

- A graph-traversal algorithm starts from a vertex v, visits all of the vertices that can be reachable from the vertex v.
- A graph-traversal algorithm visits all vertices if and only if the graph is connected.
- A connected component is the subset of vertices visited during a traversal algorithm that begins at a given vertex.
- A graph-traversal algorithm must mark each vertex during a visit and must never visit a vertex more than once.
  - Thus, if a graph contains a cycle, the graph-traversal algorithm can avoid infinite loop.
- We look at two graph-traversal algorithms:
  - Depth-First Traversal
  - Breadth-First Traversal
Depth-First Traversal

• For a given vertex v, the depth-first traversal algorithm proceeds along a path from v as deeply into the graph as possible before backing up.

• That is, after visiting a vertex v, the depth-first traversal algorithm visits (if possible) an unvisited adjacent vertex to vertex v.

• The depth-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to v.
  – We may visit the vertices adjacent to v in sorted order.
Depth-First Traversal – Example

- A depth-first traversal of the graph starting from vertex v.

- Visit a vertex, then visit a vertex adjacent to that vertex.

- If there is no unvisited vertex adjacent to visited vertex, back up to the previous step.
Recursive Depth-First Traversal Algorithm

dft(in v:Vertex) {
    // Traverses a graph beginning at vertex v
    // by using depth-first strategy
    // Recursive Version
    Mark v as visited;
    for (each unvisited vertex u adjacent to v)
        dft(u)
}
Iterative Depth-First Traversal Algorithm

dft(in v:Vertex) {
    // Traverses a graph beginning at vertex v
    // by using depth-first strategy: Iterative Version
    s.createStack();
    // push v into the stack and mark it
    s.push(v);
    Mark v as visited;
    while (!s.isEmpty()) {
        if (no unvisited vertices are adjacent to the vertex on
            the top of stack)
            s.pop();  // backtrack
        else {
            Select an unvisited vertex u adjacent to the vertex
            on the top of the stack;
            s.push(u);
            Mark u as visited;
        }
    }
}
Trace of Iterative DFT – starting from vertex a

Node visited: a, b, c, d, g, e, (backtrack), f, (backtrack), (backtrack), h, (backtrack), (backtrack), (backtrack), a, b, c, (backtrack), a, i, (backtrack), a, (backtrack), (empty)
Breath-First Traversal

• After visiting a given vertex \( v \), the breadth-first traversal algorithm visits every vertex adjacent to \( v \) that it can before visiting any other vertex.

• The breath-first traversal algorithm does not completely specify the order in which it should visit the vertices adjacent to \( v \).
  – We may visit the vertices adjacent to \( v \) in sorted order.
Breath-First Traversal – Example

- A breath-first traversal of the graph starting from vertex v.
- Visit a vertex, then visit all vertices adjacent to that vertex.
Iterative Breath-First Traversal Algorithm

bft(in v:Vertex) {
    // Traverses a graph beginning at vertex v
    // by using breath-first strategy: Iterative Version
    q.createQueue();
    // add v to the queue and mark it
    q.enqueue(v);
    Mark v as visited;
    while (!q.isEmpty()) {
        q.dequeue(w);
        for (each unvisited vertex u adjacent to w) {
            Mark u as visited;
            q.enqueue(u);
        }
    }
}
Trace of Iterative BFT – starting from vertex a

<table>
<thead>
<tr>
<th>Node visited</th>
<th>Queue (front to back)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td>(empty)</td>
</tr>
<tr>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>f</td>
<td>bf</td>
</tr>
<tr>
<td>i</td>
<td>b fi</td>
</tr>
<tr>
<td></td>
<td>fi</td>
</tr>
<tr>
<td>c</td>
<td>fic</td>
</tr>
<tr>
<td>e</td>
<td>fice</td>
</tr>
<tr>
<td></td>
<td>ice</td>
</tr>
<tr>
<td>g</td>
<td>iceg</td>
</tr>
<tr>
<td></td>
<td>c eg</td>
</tr>
<tr>
<td>d</td>
<td>egd</td>
</tr>
<tr>
<td></td>
<td>gd</td>
</tr>
<tr>
<td>h</td>
<td>h</td>
</tr>
<tr>
<td></td>
<td>(empty)</td>
</tr>
</tbody>
</table>
Some Graph Algorithms

• Shortest Path Algorithms
  – Unweighted shortest paths
  – Weighted shortest paths (Dijkstra’s Algorithm)

• Topological sorting

• Network Flow Problems

• Minimum Spanning Tree

• Depth-first search Applications
Unweighted Shortest-Path problem

- Find the shortest path (measured by number of edges) from a designated vertex \( S \) to every vertex.
Algorithm

1. Start with an initial node $s$.
   - Mark the distance of $s$ to $s$, $D_s$ as 0.
   - Initially $D_i = \infty$ for all $i \neq s$.

2. Traverse all nodes starting from $s$ as follows:
   1. If the node we are currently visiting is $v$, for all $w$ that are adjacent to $v$:
      - Set $D_w = D_v + 1$ if $D_w = \infty$.
   2. Repeat step 2.1 with another vertex $u$ that has not been visited yet, such that $D_u = D_v$ (if any).
   3. Repeat step 2.1 with another unvisited vertex $u$ that satisfies $D_u = D_v + 1$ (if any)
Figure 14.21A
Searching the graph in the unweighted shortest-path computation. The darkest-shaded vertices have already been completely processed, the lightest-shaded vertices have not yet been used as $v$, and the medium-shaded vertex is the current vertex, $v$. The stages proceed left to right, top to bottom, as numbered (continued).
**Figure 14.21B**
Searching the graph in the unweighted shortest-path computation. The darkest-shaded vertices have already been completely processed, the lightest-shaded vertices have not yet been used as \( v \), and the medium-shaded vertex is the current vertex, \( v \). The stages proceed left to right, top to bottom, as numbered.
Unweighted shortest path algorithm

void Graph::unweighted_shortest_paths(vertex s)
{
    Queue q(NUM_VERTICES);
    Vertex v, w;

    q.enqueue(s);
    s.dist = 0;
    while (!q.isEmpty())
    {
        v = q.dequeue();
        v.known = true; // not needed anymore
        for each w adjacent to v
            if (w.dist == INFINITY)
            {
                w.dist = v.dist + 1;
                w.path = v;
                q.enqueue(w);
            }
    }
}
Weighted Shortest-path Problem

- *Find the shortest path (measured by total cost) from a designated vertex S to every vertex. All edge costs are nonnegative.*
Weighted Shortest-path Problem

• The method used to solve this problem is known as Dijkstra’s algorithm.
  – An example of a greedy algorithm
  – Use the local optimum at each step

• Solution is similar to the solution of unweighted shortest path problem.

• The following issues must be examined:
  – How do we adjust $D_w$?
  – How do we find the vertex $v$ to visit next?
Figure 14.23
The eyeball is at $v$ and $w$ is adjacent, so $D_w$ should be lowered to 6.
Dijkstra’s algorithm

• The algorithm proceeds in stages.
• At each stage, the algorithm
  – selects a vertex \( v \), which has the smallest distance \( D_v \) among all the \textit{unknown} vertices, and
  – declares that the shortest path from \( s \) to \( v \) is \textit{known}.
  – then for the adjacent nodes of \( v \) (which are denoted as \( w \)) \( D_w \) is updated with new distance information
• How do we change \( D_w \)?
  – If its current value is larger than \( D_v + c_{v,w} \) we change it.
Figure 14.25A
Stages of Dijkstra’s algorithm. The conventions are the same as those in Figure 14.21 (continued).
Figure 14.25B
Stages of Dijkstra’s algorithm. The conventions are the same as those in Figure 14.21.
Implementation

• A queue is no longer appropriate for storing vertices to be visited.
• The priority queue is an appropriate data structure.
• Add a new entry consisting of a vertex and a distance, to the priority queue every time a vertex has its distance lowered.