Graph
Tree vs. Graph

T is the root node.
Vertices and Edges

• A graph $G$ consists of a *set of vertices* $V$ (also called nodes) and a *set of edges* $E$. In the graph below the vertices are $\{0, 1, 2, 3, 4\}$ and the edges are described using pairs of vertices, i.e. $\{0, 1\}, \{0, 2\}, \{1,3\}, \{1,4\}, \{2,4\}$.

$$G = (V, E)$$

$$V = \{0, 1, 2, 3, 4\}$$

$$E = \{ \{0,1\}, \{0,2\}, \{1,3\}, \{1,4\}, \{2,4\} \}$$
Different Types of Edges in Graphs

- **Directed edge**: there is only one path from A (the origin), to B (the destination).

- **Undirected edge**: the path between A and B is bidirectional. Origin and destination are not fixed.
Undirected Graphs

- An *undirected graph* is a graph such that if \((u, v)\) is an *edge* in the graph then \((v, u)\) is also an *edge* in the graph. Informally, in *undirected graphs* the edges do not have arrows indicating direction.

![Diagram of an undirected graph and a directed graph or digraph]
Graphs are everywhere

Who uses graphs in a fundamental way?

Google Maps, Google Plus, Google Search Engine, Facebook, LinkedIn, Twitter, Pinterest, Tinder, Amazon Web Services, Amazon Market Place, Amazon Distribution, Netflix, Cisco, AT&T, Bell, Rogers, FedEx, UPS, McDonalds...

Can you guess how?
Social networks (Facebook)

- undirected graph
PageRank (Google Search)

https://en.wikipedia.org/wiki/PageRank
Traveling Salesman Problem

Given a list of cities and the distances between them, find the shortest tour that visits all cities and returns to the start.
A *weighted graph* is a graph where each *edge* is assigned a numerical value, i.e. a *weight* (typically representing capacity or cost).

- The *weight* of the edge \{0, 1\} is 7.1
- The *weight* of the edge \{0, 2\} is 8.2
- The *weight* of the edge \{1, 4\} is 9.2
- The *weight* of the edge \{2, 4\} is 8.2
- The *weight* of the edge \{1, 3\} is 10.2

*Weight* is a property of *edges*. 
Terminology

Self Loops

• A self-loop is a an edge whose start and end points are the same vertex.

• There is a self-loop at vertex 1.

We will be investigating graphs without self-loops.
Terminology

Multi-edge (parallel edge)
• An edge is called multi-edge if it occurs more than once in a graph.

If a graph contains no self-loop or multi-edge, it’s called a simple graph.
Terminology

A Path

• A path from vertex $v_1$ to vertex $v_k$ is a sequence of vertices $(v_1, v_2, ..., v_k)$ such that each pair $\{v_i, v_{i+1}\}$ in the sequence is an edge in the graph.

• $(0, 2, 4, 5)$ is a path from vertex 0 to vertex 5.

• $\{0, 2\}$, $\{2,4\}$, and $\{4,5\}$ are all edges in the graph
Terminology

Connected Graph

- A graph is *connected* if there is a path between any two pair of *vertices* in the graph.

A *Connected* Graph  
A graph with 2 components
Terminology

Adjacent Vertices
• Two vertices, $u$ and $v$, are adjacent if $\{u, v\}$ is an edge in the graph (i.e. they are directly connected).

• Vertex 0 is adjacent to vertices 1 and 2 because both $\{0, 1\}$ and $\{0, 2\}$ are edges in the graph.

• Adjacency describes vertices.
• You may think of adjacent
• as being a path of length 1.
Terminology

Incident

• Describes the relationship between vertices and edges. The edge \{u, \nu\} is \textit{incident on} vertices \textit{u} and \textit{v} (i.e. it has \textit{u} and \textit{v} as endpoints).

• The edges \{0, 1\} and \{0, 2\} are both \textit{incident on} the vertex 0.
Terminology

Degree of a Vertex

• The *degree* of a *vertex* is the number of *edges* incident on it.
• *Vertex* 3 has *degree* 1.
  
  It is incident on \{1, 3\}.

• *Vertex* 0 has *degree* 2.
  
  It is incident on \{0, 1\} and \{0, 2\}.

• *Vertex* 1 has *degree* 3.
  
  It is incident on \{0, 1\}, \{1, 3\} and \{1, 4\}

• *Degree* is a property of *vertices*. 
Caution. There are many types of graphs. Be careful about what you read (e.g. on the web) because what may be true for one type of graph may not be true for another type.

Graphs can be:

• directed vs. undirected
• connected vs. disconnected
• weighted vs. unweighted
• simple vs. with parallel edges and loops

For Example
What is the maximum number of edges in a graph with $n$ vertices?
Graph Example: The Paris Metro

• This subway map of Paris is:
  • Undirected
  • Connected
  • Cyclic (not a tree!)
  • Vertex-labeled
The Graph ADT

Basic Operations on Graphs

• Graph(V, E): create a new graph
• isEmpty(): does G have any vertices
• numEdges(): how many edges in G
• numVertices(): how many vertices in G
• edges(): return a list of all the edges in G
• vertices(): return a list of all the vertices in G
• isAdjacent(v1, v2): is \{v1, v2\} an edge in G
• neighbours(v): return a list of all vertices adjacent to \( v \)
The Graph ADT

Basic Operations on Graphs

• **addVertex(v):** add a vertex to G
• **removeVertex(v):** remove v and the edges that contain it
• **addEdge(v1,v2):** add the edge \{v1, v2\} to G
• **removeEdge(v1,v2):** remove edge but leave vertices

More Elaborate Operations (not always implemented)

• **traverse(v):** return a list of vertices connected to v
• **isConnected():** is G a connected graph
• **path(v1,v2):** is there a path from v1 to v2
• **shortestPath(v1,v2):** find a shortest path from v1 to v2
• **minimumSpanningTree():** find a minimum spanning tree

... and many more. Graphs are a rich data type (*many* possible operations)
Graph Implementations

Implementation 1: Edge List

Represent $G$ as a list of edges $[e_1, e_2, e_3, ..., e_m]$, where each edge $e_j$ is a pair of vertices $=(v_{j1}, v_{j2})$

Example

Edge list = [{4, 6}, {4,5}, {3, 4}, {3,2}, {2,5}, {1,2}, {1, 5}]
Graph Implementations

**Problem with Edge List:**
How would you represent vertices with no edges?
You still need some collection for vertices only.
Graph Implementations

Implementation 2: Adjacency Matrix

- Create an $n \times n$ matrix where
  - if $\{v_i, v_j\}$ is an edge then both $m[i-1, j-1]$ and $m[j-1, i-1]$ are 1, else both are 0.

<table>
<thead>
<tr>
<th>vertex</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph Implementations

**Problem with Adjacency Matrix**

- Would you be able to use adjacency matrix to hold the entire graph of Facebook?
- Not suitable for large sparse graphs

<table>
<thead>
<tr>
<th>vertex</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph Implementations

**Complexity of Adjacency Matrix**

- Testing adjacency takes $O(1)$ time *(good!)*
- Listing all neighbours of a vertex takes $O(n)$ time *(bad!)*

<table>
<thead>
<tr>
<th>vertex</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
# Graph Implementations

## Implementation 3: Adjacency Lists

For each vertex, list the vertices adjacent to it.

### Example

<table>
<thead>
<tr>
<th>Vertex</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[2, 5]</td>
</tr>
<tr>
<td>2</td>
<td>[1, 3, 5]</td>
</tr>
<tr>
<td>3</td>
<td>[2, 4]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 5, 6]</td>
</tr>
<tr>
<td>5</td>
<td>[1, 2, 4]</td>
</tr>
<tr>
<td>6</td>
<td>[4]</td>
</tr>
</tbody>
</table>

[Diagram of a graph with vertices and their adjacency lists]
Graph Implementations

Complexity of Adjacency Lists

• Testing adjacency takes $O(n)$ time (bad!)
• Getting list of neighbours can take $O(1)$ time (good!)

Example

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[2, 5]</td>
</tr>
<tr>
<td>2</td>
<td>[1, 3, 5]</td>
</tr>
<tr>
<td>3</td>
<td>[2, 4]</td>
</tr>
<tr>
<td>4</td>
<td>[3, 5, 6]</td>
</tr>
<tr>
<td>5</td>
<td>[1, 2, 4]</td>
</tr>
<tr>
<td>6</td>
<td>[4]</td>
</tr>
</tbody>
</table>
## Performance

<table>
<thead>
<tr>
<th>§ $n$ vertices, $m$ edges</th>
<th>§ no parallel edges</th>
<th>§ no self-loops</th>
<th>Space</th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$n + m$</td>
<td>$n + m$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>incidentEdges($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
<td></td>
<td>$n$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>areAdjacent ($v$, $w$)</td>
<td>$m$</td>
<td>$\min(\text{deg}(v), \text{deg}(w))$</td>
<td></td>
<td>$1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insertVertex($o$)</td>
<td>1</td>
<td>1</td>
<td></td>
<td>$n^2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insertEdge($v$, $o$)</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
<td></td>
<td>$n^2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>removeEdge($e$)</td>
<td>1</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Graph Traversals

Goal
Determine which vertices in G can be reached from a starting vertex v.

Note
• Tree traversals are special kind of graph traversals
• Multiple traversals possible
• Loops are possible: must mark vertices to ensure we do not loop indefinitely
Depth-first Search/Traversal (DFS)

Basic Structure
• Need a starting vertex v (mark it)
• Proceed along a path from v as far as possible
• Then, backup to previous vertex, and visit its unvisited neighbours
• Repeat while unvisited, reachable vertices remain
• Could be implemented recursively or with a stack.
• If graph is a tree, DFS performs pre-order traversal
Depth-first Search/Traversal (DFS)

**Recursive Pseudocode**
- Recursively explore the graph, back track as necessary.
- Careful not to repeat vertices.

**Pseudocode**

```python
class Graph():
    def dfs(u):
        u.visit()
        u.visited = True
        for (each vertex v such that (u,v) ∈ E):
            if ! v.visited:
                dfs(v)
```

![Graph Diagram]

0 —- 1 —- 2
    
3 —- 4 —- 5
Depth-first Search/Traversal (DFS)

**Stack Pseudocode**

```
discovered = {}  # who's been discovered
s = Stack()     # who is waiting to processed

def dfs(u):
    s.push(u)
    discovered[u] = True

    while not s.empty():
        v = s.pop()
        process(v)  # i.e. do what you want
        for w in neighbours(v):
            if w not in discovered:
                s.push(w)
                discovered[w] = True
```
Depth-first Search/Traversal (DFS)

Pseudocode

dfs(0):
  s.push(u)
  discovered[u] = True
while not s.empty():
  v = s.pop()
  process(v) # i.e. do what you want
  for w in neighbours(v):
    if w not in discovered:
      s.push(w)
      discovered[w] = True

s = [0]
discovered.keys() = [0]
Depth-first Search/Traversal (DFS)

**Pseudocode**

```python
dft(0):
    s.push(u)
    discovered[u] = True
    while not s.empty():
        v = s.pop()
        process(v)  # i.e. do what you want
        for w in neighbours(v):
            if w not in discovered:
                s.push(w)
                discovered[w] = True
```

processed: 0
s = [1, 2 ]
discovered.keys() = [0, 1, 2]
Depth-first Search/Traversal (DFS)

**Pseudocode**

dft(0):
    s.push(u)
    discovered[u] = True
while not s.empty():
    v = s.pop()
    process(v) # i.e. do what you want
    for w in neighbours(v):
        if w not in discovered:
            s.push(w)
            discovered[w] = True

processed: 2
s = [1, 4]
discovered.keys() = [0, 1, 2, 4]
Depth-first Search/Traversal (DFS)

**Pseudocode**

dft(0):
    s.push(u)
    discovered[u] = True
while not s.empty():
    v = s.pop()
    process(v)  # i.e. do what you want
    for w in neighbours(v):
        if w not in discovered:
            s.push(w)
            discovered[w] = True

processed: 4
s = [1, 5]
discovered.keys() = [0, 1, 2, 4, 5]
Depth-first Search/Traversal (DFS)

Pseudocode

dft(0):
  s.push(u)
  discovered[u] = True
while not s.empty():
  v = s.pop()
  process(v)  # i.e. do what you want
  for w in neighbours(v):
    if w not in discovered:
      s.push(w)
      discovered[w] = True

processed: 5
s = [1]
discovered.keys() = [0, 1, 2, 4, 5]
Depth-first Search/Traversal (DFS)

Pseudocode

dft(0):
  s.push(u)
  discovered[u] = True
while not s.empty():
  v = s.pop()
  process(v) # i.e. do what you want
  for w in neighbours(v):
    if w not in discovered:
      s.push(w)
      discovered[w] = True

processed: 1
s = [3]
discovered.keys() = [0, 1, 2, 4, 5, 3]
Depth-first Search/Traversal (DFS)

**Pseudocode**

dft(0):
  s.push(u)
  discovered[u] = True
while not s.empty():
  v = s.pop()
  process(v)  # i.e. do what you want
  for w in neighbours(v):
    if w not in discovered:
      s.push(w)
      discovered[w] = True

processed: 3
s = []
discovered.keys() = [0, 1, 2, 4, 5, 3]
Depth-first Search/Traversal (DFS)

Pseudocode

dft(0):
  s.push(u)
  discovered[u] = True
while not s.empty():
  v = s.pop()
  process(v) # i.e. do what you want
  for w in neighbours(v):
    if w not in discovered:
      s.push(w)
      discovered[w] = True

s = []  # done!
discovered.keys() = [0, 1, 2, 4, 5, 3]
Depth-first Search/Traversal (DFS)

Applications
• Find all the vertices connected to v
• Determine if G is a connected graph
• Determine if there is a path between vertices u and v

With modifications:
• Finding all connected components of G
• Checking if G contains a cycle
Depth-first Search/Traversal (DFS)

Asymptotic Time Complexity

- **Assume**: there are $n$ vertices and $m$ edges
- At most $n$ visits (each vertex visited once)
- Each visit examines neighbours of a vertex
  - At most $2m$ neighbours examined over all recursive calls (each edge has two endpoints)
  - To examine each edge, $O(1)$ steps
- *Total is $O(n+m)$ on adjacency list*
- *$O(n^2)$ on adjacency matrix*
Breadth-first Search/Traversal (BFS)

**Basic Structure**

- Needs a starting point \( v \) (mark it as discovered)
- Visit all the neighbours of \( v \), marking them as you go
- Then, visit all of the neighbours of the neighbours of \( v \), marking those in turn, etc.
- Repeat until all reachable vertices are marked
- Is implemented using *queue*
- If graph is a tree, BFS performs Level order traversal.
- Level-order provides the shortest paths from \( v \) to every reachable vertex from \( v \) (by definition!)
Breadth-first Search/Traversal (BFS)

Pseudocode

```python
def bfs(v):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True

while not Q.isEmpty():
    v = Q.dequeue()
    process(v)  # i.e. do what you want
    for w in neighbours(v):
        if w not in discovered:
            discovered[w] = True
            Q.enqueue(w)
```

```python
def process(origin):
    parent = origin;
    if (origin is None):
        depth = 0
    else:
        depth = origin.depth + 1
```

```
0

1 2

3 4

5
```

Breadth-first Search/Traversal (BFS)

**Pseudocode**

```python
def bft(v):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```

```
K = [0]
discovered.keys: [0]
```
Breadth-first Search/Traversal (BFS)

**Pseudocode**

```python
def bft(v):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Breadth-first Search/Traversal (BFS)

**Pseudocode**

```python
def bft(v):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```

Q = [2, 3, 4]
discovered.keys: [0, 1, 2, 3, 4]
Breadth-first Search/Traversal (BFS)

**Pseudocode**

```python
def bft(v):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```

```
0 1 2
3 4 5
```

Q = [3, 4]
discovered.keys: [0, 1, 2, 3, 4]
Breadth-first Search/Traversal (BFS)

Pseudocode

```python
def bft(0):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```

Q = [4]
discovered.keys: [0, 1, 2, 3, 4]
Breadth-first Search/Traversal (BFS)

**Pseudocode**

```python
def bft(0):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```

```
Q = [5]
discovered.keys: [0, 1, 2, 3, 4, 5]
```
Breadth-first Search/Traversal (BFS)

**Pseudocode**

```python
def bft(v):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```

Q = []

discovered.keys: [0, 1, 2, 3, 4, 5]
Breadth-first Search/Traversal (BFS)

**Pseudocode**

```python
def bft(v):
    Q = Queue()
    discovered = {}
    Q.enqueue(v)
    discovered[v] = True
    while not Q.isEmpty():
        v = Q.dequeue()
        process(v)
        for w in neighbours(v):
            if w not in discovered:
                discovered[w] = True
                Q.enqueue(w)
```

0 1 2 4 3 5

Q = []
discovered.keys: [0, 1, 2, 3, 4, 5]
Breadth-first Search/Traversal (BFS)

- Finding the distance of the vertex from starting vertex
- Finding the shortest path between them.
Breadth-first Search/Traversal (BFS)

Asymptotic Time Complexity

• **Assume:** there are $n$ vertices and $m$ edges
• Examine all $n$ vertices
  • - Add to queue and mark as discovered both $O(1)$
  • - $O(n)$ steps
• Examine all $m$ edges
  • - check if the neighbours have been discovered, $O(1)$
  • - $O(m)$ steps
• **Total $O(n+m)$**, same as Depth-first Traversal
• $O(n^2)$ on adjacency matrix
Breadth-first Search/Traversal (BFS)

Applications
• Determine if there is a path between vertices \( u \) and \( v \)
• Find the shortest path from \( v \) to \( w \) (in unweighted graphs only)
BFS and DFS remarks

• Using collection of discovered vertices in the implementation is not a good choice. It increases the complexity. It is better to use a flag in the vertex class itself.

• Both BFS and DFS are equal in terms of complexity. Choice always depends on application.
Weighted Graph Implementations

Weighted Graphs (and any other extensions)

*Edge List:*  
- store weight with edge

*Adjacency List:*  
- store weight with neighbour in list (vertex, weight)

*Adjacency Matrix:*  
- store weight in matrix at location for \( v_i, v_j \)  
- need special value, *None*, to distinguish zero weight from no edge
Common problems involving weighted graphs

1. **Construction a Minimum spanning tree problem** (Prim's and Kruskal's algorithms):
   - Application: Network design (telephone, electrical, hydraulic, TV cable, computer, road,..)

2. **Shortest path problem** (Dijkstra's algorithm).
   - Application: Find the fastest way to drive between two points in google maps.