A graph consists of a set of vertices $V$ and a set of edges $E = \{(v_1, v_2) | v_1, v_2 \in V\}$.

If $(v_1, v_2)$ are ordered, we have a directed graph.

$$V = \{A, B, C, D, E\}$$
$$E = \{(A, C), (A, D), (A, E), (B, E), (C, A), (C, B), (C, D), (D, C), (E, D)\}$$
If \((v_1, v_2)\) are unordered, we have a undirected graph.

\[ V = \{A, B, C, D, E\} \]
\[ E = \{(A, C), (A, D), (A, E), (B, C), (B, E), (C, A), (C, B), (C, D), (D, C), (D, A), (D, E), (E, A), (E, B), (E, D)\} \]
Terminology

Vertices $v_i$ and $v_j$ are adjacent if either $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$.

A path in a graph is a sequence of vertices $v_1, v_2, \ldots, v_n$ where all $(v_i, v_{i+1}) \in E$, $\forall i = 1, \ldots, n - 1$. The length of this path is $n - 1$.

$V = \{A, B, C, D, E\}$

$E = \{(A, C), (A, D), (A, E), (B, E), (C, A), (C, B), (C, D), (D, C), (E, D)\}$

Some paths:

- $A, D, C, B$
- $A, D, C, A$
- $A, C, A, E, D, C, A$
More Terminology

- Cycle: a path of length \( l \) where \( v_1 = v_l \)
- Simple path: all vertices are distinct except possibly the \( v_1 \)
- Connected Components: a directed graph is strongly connected if there is a directed path from every vertex to every other vertex.
- Weighted graph: a graph with "costs" associated with each edge.

- Simple path: All vertices are distinct except possibly the \( v_1 \)
- Cycle: a path of length \( l \) where \( v_1 = v_l \)
- Connected Components: A directed graph is strongly connected if there is a directed path from every vertex to every other vertex.
- Weighted graph: a graph with "costs" associated with each edge.

- Cycle: a path of length \( l \) where \( v_1 = v_l \)
- Simple path: All vertices are distinct except possibly the \( v_1 \)
- Connected Components: A directed graph is strongly connected if there is a directed path from every vertex to every other vertex.
- Weighted graph: a graph with "costs" associated with each edge.

- Cycle: a path of length \( l \) where \( v_1 = v_l \)
- Simple path: All vertices are distinct except possibly the \( v_1 \)
- Connected Components: A directed graph is strongly connected if there is a directed path from every vertex to every other vertex.
- Weighted graph: a graph with "costs" associated with each edge.

- Cycle: a path of length \( l \) where \( v_1 = v_l \)
- Simple path: All vertices are distinct except possibly the \( v_1 \)
- Connected Components: A directed graph is strongly connected if there is a directed path from every vertex to every other vertex.
- Weighted graph: a graph with "costs" associated with each edge.

- Cycle: a path of length \( l \) where \( v_1 = v_l \)
- Simple path: All vertices are distinct except possibly the \( v_1 \)
- Connected Components: A directed graph is strongly connected if there is a directed path from every vertex to every other vertex.
- Weighted graph: a graph with "costs" associated with each edge.

- Cycle: a path of length \( l \) where \( v_1 = v_l \)
- Simple path: All vertices are distinct except possibly the \( v_1 \)
- Connected Components: A directed graph is strongly connected if there is a directed path from every vertex to every other vertex.
- Weighted graph: a graph with "costs" associated with each edge.
A directed graph containing no cycles

\[ V = \{A, B, C, D, E\} \]
\[ E = \{(A, C), (A, D), (A, E), (B, E), (C, B), (C, D), (E, D)\} \]

A tree is also an example of a directed acyclic graph.
Graph ADT

Constructor: $\text{Graph} \leftarrow \text{Graph}$

InsertVertex: $(\text{Graph}, \text{vertex}) \rightarrow \text{Graph}$
- Pre: vertex should not already be in the graph.
- Action: Add a new vertex to the graph.

InsertEdge: $(\text{Graph}, \text{vertex}_1, \text{vertex}_2) \rightarrow \text{Graph}$
- Pre: vertex\text{ }_1 and vertex\text{ }_2 should not already be in the graph.
- Action: Add an edge to the graph.

Action/: create a new graph.

Action/: Add an edge to the graph.

Pre: vertex\text{ }_1 and vertex\text{ }_2 must already be in the graph.
DeleteVertex:

Action: Find all vertices \( v_1 \) where \((v_1, v_2)\) is an edge

\[ \text{GetNeighbors: } (\text{Graph, vertex}) \rightarrow \text{vertex-list} \]

Pre: vertex must be in the graph.

in the graph.

GetNeighbors:

Pre: called to InsertVertex for \( v_1 \), \( v_2 \) should have occurred previously.

Action: Remove the edge \((v_1, v_2)\) from the graph.

Graph \( \leftarrow (\text{Graph}, \text{vertex}) \)

DeleteEdge:

Pre: graph must have occurred previously.

Action: Remove vertex \( v_1 \) from the graph.

Graph \( \leftarrow (\text{Graph}, \text{vertex}) \)

DeleteVertex:

Pre: graph must have occurred previously.

Action: Remove vertex \( v \). Also remove any edges of the form \((v_1, v_2)\), where either \( v_1 \) or \( v_2 \) is vertex \( v \).
Traversing a Graph: DFS

Depth First Search (DFS)

```plaintext
main()
{
  for (v is node in graph)
  {
    if (v has not been visited)
    {
      dfs(v);
      mark v as having been visited
      list = neighbors of v
      while there is an unvisited neighbor w of v
      {
        dfs(w)
      }
    }
  }
}
```

Cross arcs
DF tree arcs
Back arcs
Forward arcs
Breadth First

\text{main} = \text{true for } \forall \text{queue}(x)\
for all neighbors } \text{ with queue } = \text{false,}
mark } \text{ as having been visited

() = \text{queue}(\forall)\\
\text{while not queue empty do}
\text{queue}(\forall)\\
\text{bfs}(\text{vertex } \forall)\\
\{\\
\text{if } \forall \text{ has not been visited } (\text{bs} (\forall))\\
\text{for } \forall \text{ is node in graph}
initializes all nodes - visited = false, queue = false
\} \text{ main}
\text{queue } \forall
Cross arcs
DF tree arcs
Back arcs
Forward arcs
Representation of Graphs

V = \{A,B,C,D,E\}
E = \{(A,C),(A,D),(A,E),
      (B,E),(C,A),(C,B),
      (C,D),(D,C),(E,D) \} \}

- Adjacency matrix – Space $O(|V|^2)$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
• Adjacency list

A=0  →  2  →  3  →  4

B=1  →  4

C=2  →  0  →  3

D=3  →  1  →  2

E=4  →  3
Adjacency table

```
template <int max_size>
class Digraph {
  int count; // number of vertices, at most max_size
  bool adjacency[max_size][max_size];
}
```
Adjacency Lists

typedef int Vertex;
template <int max_size>
class Digraph {
  int count;
  List <Vertex> neighbors[max_size];
};
class Edge;                       // forward declaration
class Vertex {
    Edge *first_edge;             // start of adjacency list
    Vertex *next_vertex;        // next vertex on linked list of vertices
}

class Edge {
    Vertex *end_point;           // vertex to which the edge points
    Edge *next_edge;            // next edge on adjacency list
};
class Digraph {
    Vertex *first_vertex;        // header for list of vertices
};
Suppose we have 10 tasks and a graph represents a partial order on these tasks where $T_i \rightarrow T_j$ means that task $T_i$ must be complete before task $T_j$ can start.

What order should the tasks be done?

Some possibilities:

1. $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 10$
2. $1 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 9 \rightarrow 8 \rightarrow 10$
3. $1 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 6 \rightarrow 5 \rightarrow 9 \rightarrow 8 \rightarrow 7 \rightarrow 10$
Topological sort

- **Depth-first**
  - find node with no successor and place it last in order
  - use depth-first traversal

- **Breadth-first**
  - find node with no predecessor and place it first in order
  - use breadth-first traversal
typedef int Vertex;
template <int graph_size>
class Digraph {
    public:
        Digraph();
        void read();
        void write();
        void depth_sort(List<Vertex> &topological_order);
        void breadth_sort(List<Vertex> &topological_order);
    private:
        int count; List<Vertex> neighbors[graph_size];
        void recursive_depth_sort(Vertex v, bool visited[],
                                List<Vertex> &topological_order);
};
template <int graph_size>
void Digraph<graph_size>::depth_sort(List<Vertex> &topological_order)
  /*Post: The vertices of the Digraph are placed into List topological_order with a depth-first traversal of those vertices that do not belong to a cycle.
Uses: Methods of class List, and function recursive_depth_sort to perform depth-first traversal.*/
{
  bool visited[graph_size];
  Vertex v;
  for (v = 0; v < count; v++)  visited[v] = false;
  topological_order.clear();
  for (v = 0; v < count; v++)
  {
    if (!visited[v])  // Add v and its successors into topological order.
      recursive_depth_sort(v, visited, topological_order);
  }
}
template <int graph_size>
void Digraph<graph_size>::recursive_depth_sort(Vertex v, bool *visited,
                                             List<Vertex> &topological_order)
{
    visited[v] = true;
    int degree = neighbors[v].size();
    for (int i = 0; i < degree; i++) {
        Vertex w; // A (neighboring) successor of v
        neighbors[v].retrieve(i, w);
        if (!visited[w]) // Order the successors of w.
            recursive_depth_sort(w, visited, topological_order);
    }
    topological_order.insert(0, v); // Put v into topological_order.
}
Topological Sort

I. \( V \) vertices \( v \), compute \( \text{pred}^v \), the number of tasks that vertex \( v \) depends on (directly).

2. Find a vertex \( w \) where \( \text{pred}^w = 0 \). Output \( w \).

3. A vertices \( x \) where \( w \not\in x \), decrement \( \text{pred}^x \).

4. Repeat steps 2 and 3 until either:
   a) No more vertices in the graph — SUCCESS
   b) More tasks but none with \( \text{pred} = 0 \) — FAILURE.

Not start with a DAG (p) More tasks but none with \( \text{pred} = 0 \) — FAILURE — we did (q) More tasks but none with \( \text{pred} = 0 \) — FAILURE — we did.
template <int graph_size>
void Digraph<graph_size>::breadth_sort(List<Vertex> &topological_order)
{
    topological_order.clear();
    Vertex v, w;
    int predecessor_count[graph_size];
    for (v = 0; v < count; v++) predecessor_count[v] = 0;
    for (v = 0; v < count; v++)
        for (int i = 0; i < neighbors[v].size(); i++) {
            // Loop over all edges v -- w.
            neighbors[v].retrieve(i, w);
            predecessor_count[w]++;
        }
Queue ready_to_process;
for (v = 0; v < count; v++)
    if (predecessor_count[v] == 0)
        ready_to_process.append(v);
while (!ready_to_process.empty( )) {
    ready_to_process.retrieve(v);
    topological_order.insert(topological_order.size(), v);
    for (int j = 0; j < neighbors[v].size(); j++) {  // Traverse successors of v.
        neighbors[v].retrieve(j, w);
        predecessor_count[w]--;
        if (predecessor_count[w] == 0)
            ready_to_process.append(w);
    }
    ready_to_process.serve();
}
Spanning Trees

If the graph is undirected, DFS and BFS are good ways to see if the graph is connected (i.e., possible path between all nodes).

These traversals define spanning trees.
Spanning Trees

Initial Graph

Spanning Trees
A graph consists of a set of vertices $V$, a set of edges $E = \{(v_1, v_2) | v_1, v_2 \in V\}$, and a weighting function: $E \to \mathbb{I}$. 
Algorithm:

Idea: consider the heaviest edges first

Greedy

1. First, we need to view the graph of n nodes as n separate components.

2. Consider the edges in increasing order of their weights.

3. If an edge has ends in 2 different components, select that edge for the spanning tree and merge the components.

Otherwise, the edge will not be part of the MST.
Shortest Paths & Weighted Graphs

- Assumes no negative edges
- Dijkstra's Algorithm - Single source, all destinations
Graph G
Algorithm

1. Initialize
   \[ \text{settled} = \{ \text{initial} \} \]
   \[ \text{successor}(i) = \text{NULL} \]

2. For each node \( i \) where \( i \notin \text{settled} \)
   \[ \text{distance}(i) = \min \{ \text{distance}(i') + \text{cost}(i', i) \} \]
   where \( i' \in \text{settled} \) and \( i' \neq i \)

3. Choose \( i \leftarrow k \) where \( k \notin \text{settled} \) and \( k \in \text{settled} \) has the smallest of the \( \text{distance}(k) \) computed in the previous step.

4. If \( \text{settled} \neq \text{Null} \)
   \[ i = \text{successor}(\text{settled}) \]
   \[ \text{settled} = \text{settled} \cup \{ i \} \]
   \[ \text{successor}(i) = i \]

5. If \( \text{settled} = \text{Null} \)
   Go to step 2
Initially

Distance(B) = 4
Distance(D) = 8
Distance(C) = 12

Distance(B) = 4
Distance(D) = 8
Distance(C) = 12
Distance(E) = 13
Distance(F) = 17