Algorithms and Data Structures
CS-CO-412

David Vernon
Professor of Informatics
University of Skövde
Sweden

david@vernon.eu
www.vernon.eu

Graphs

Lecture 12

Topic Overview

- Graphs
  - **Types of graph**
  - **Adjacency matrix representation**
  - **Adjacency list representation**
  - **Breadth-First Search traversal**
  - **Depth-First Search traversal**
  - **Topological Sorting**
  - Minimum Spanning Tree
    - Prim's Algorithm
    - Kruskal's algorithm
  - Shortest Path Algorithms
    - Dijkstra's algorithm
    - Floyd's algorithm

Graphs

- Important way of modelling and representing the organization of many systems and problems
  - Road networks
  - Electronic circuits
  - Telecommunication networks
  - Human interaction
  - Social networks
  - Eco-system networks
  - Robot navigation paths
  - Any relationship …

Graphs

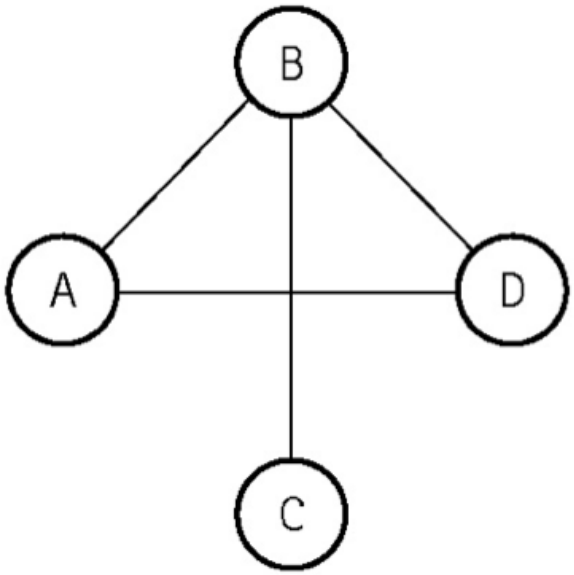
- A graph $G = (V, E)$ consists of
  - A set of *vertices* V
  - A set E of vertex pairs or *edges*
- Vertex: node in a graph
- Edge (arc): a pair of vertices representing a connection between two nodes in a graph
- Undirected graph: a graph in which the edges have no direction
- Directed graph (*digraph*): a graph in which each edge is directed from one vertex to another (or the same) vertex

Graphs

- The key to solving many algorithmic problems is to think of them in terms of graphs

- The key to using graphs algorithms effectively in applications is to model your problem correctly to take advantage of *existing* graph algorithms

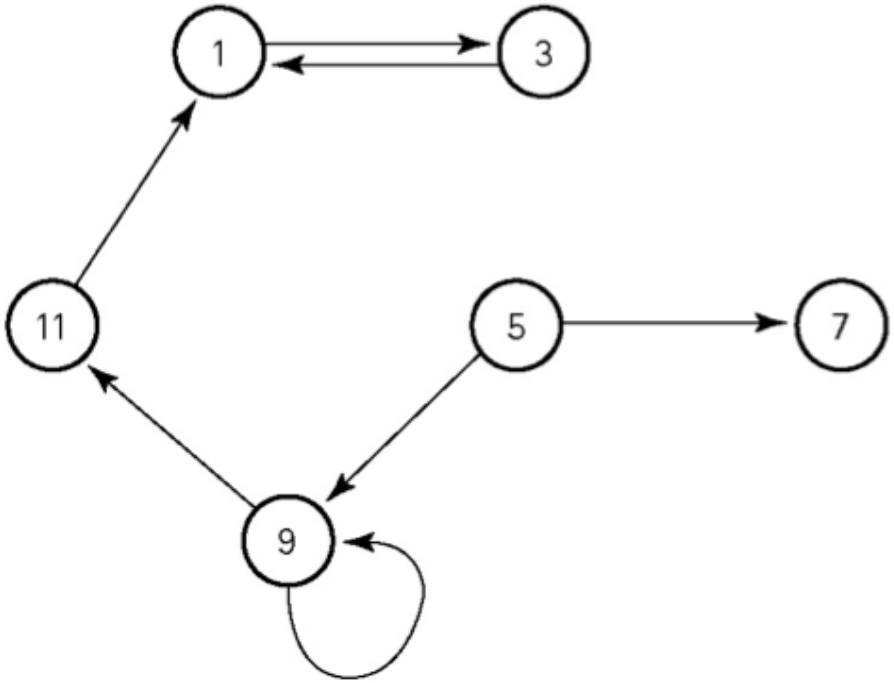
Graphs



Undirected graph G

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

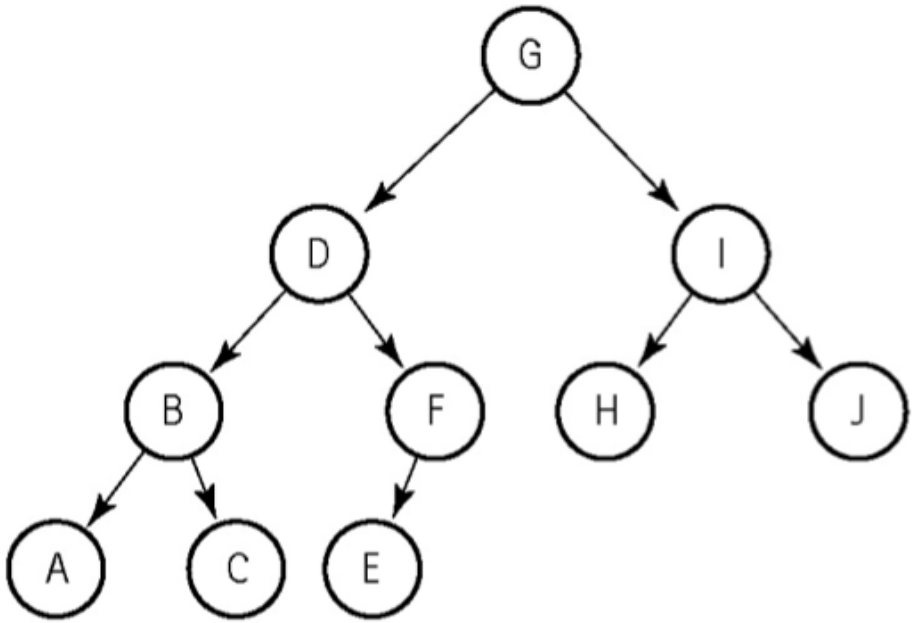
Graphs



Directed graph G

$V = \{1, 3, 5, 7, 9, 11\}$
$E = \{(1, 3), (3, 1), (5, 7), (5, 9), (9, 9), (9, 11), (11, 1)\}$

Graphs



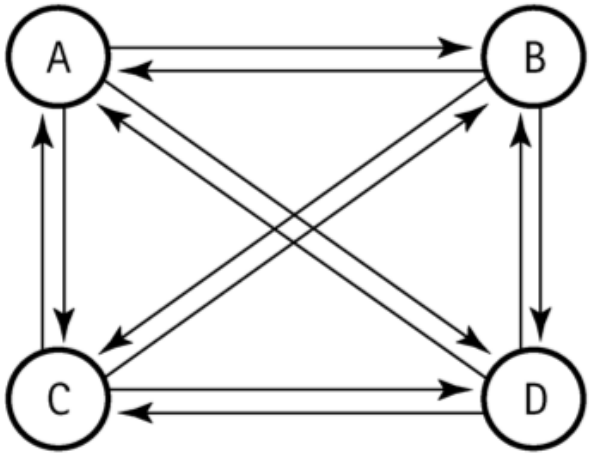
Directed graph G

$V = \{A, B, C, D, E, F, G, H, I, J\}$
$E = \{(G, D), (G, I), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E)\}$

Graphs

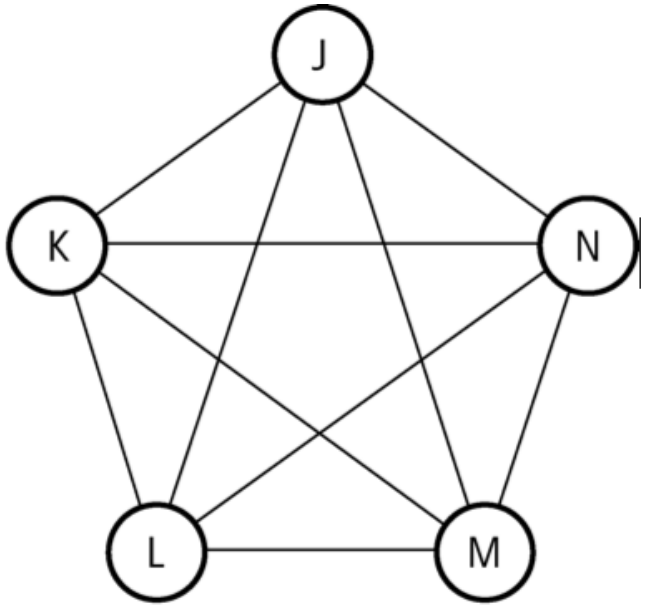
- Adjacent vertices
  - Two vertices in a graph that are connected by an edge
- Path
  - A sequence of vertices that connects two nodes in a graph
- Complete graph
  - A graph in which every vertex is directly connected to every other vertex
- Weighted graph
  - A graph in which each edge carries a value

Graphs



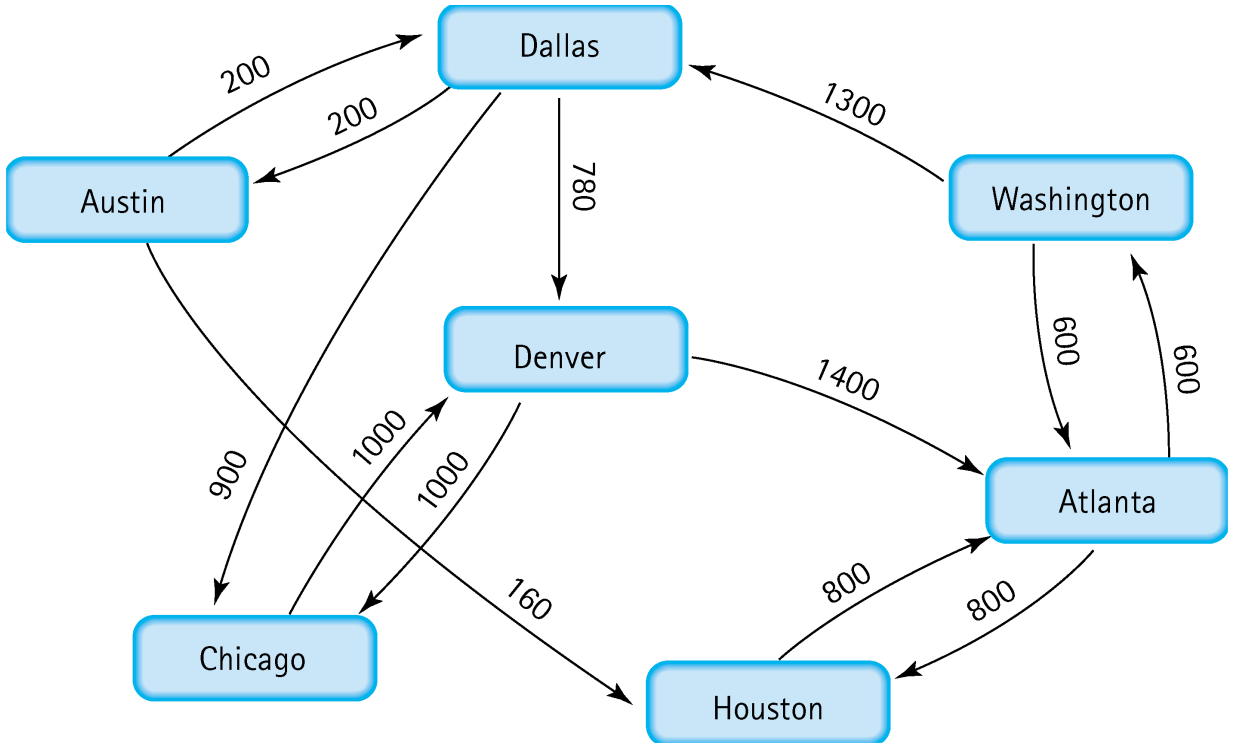
A complete directed graph G

Graphs



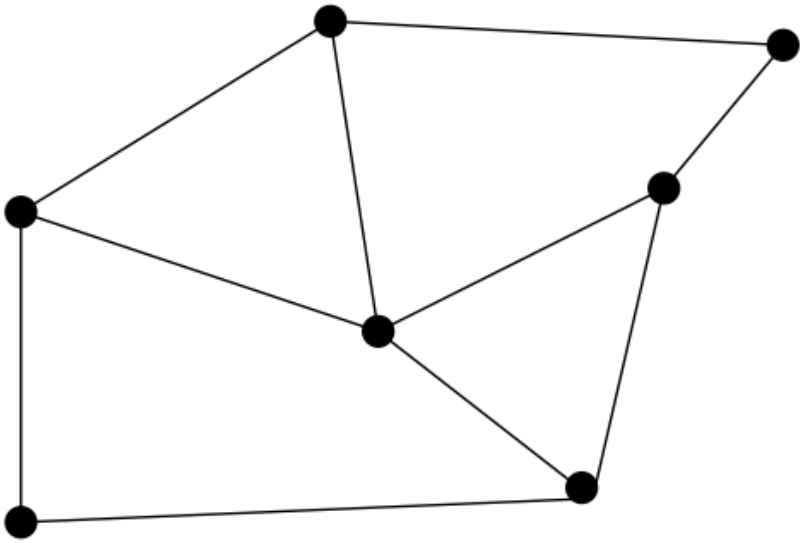
A complete undirected graph G

Graphs

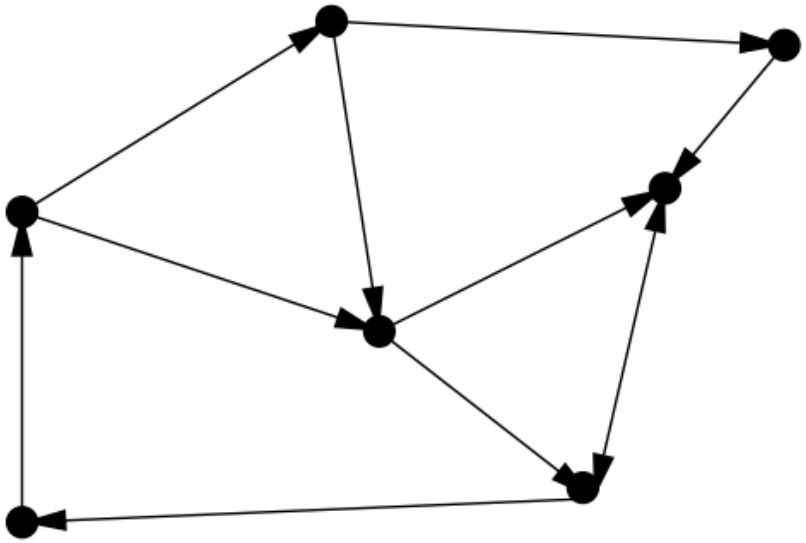


A weighted graph G

Graphs



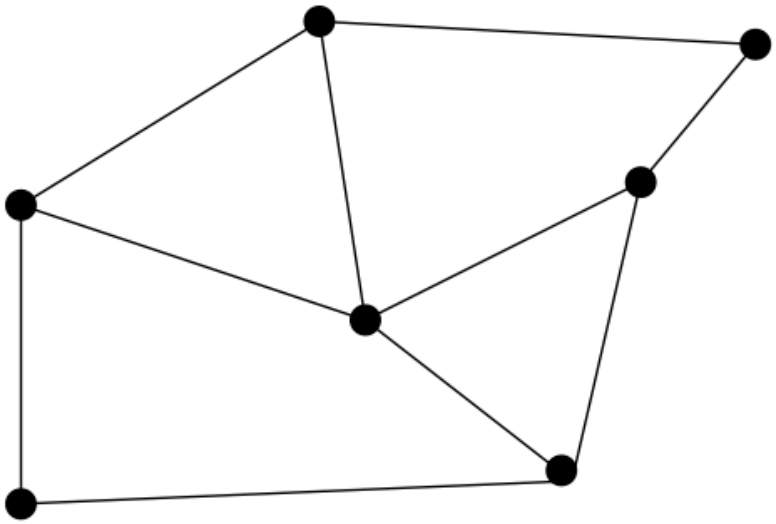
undirected



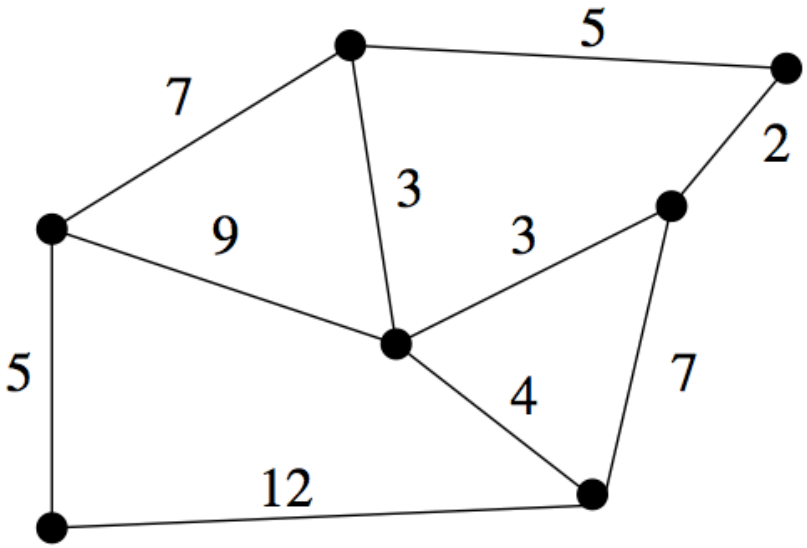
directed

A graph G is undirected if edge $(x, y) \in E$ implies $(y, x) \in E$

Graphs



unweighted

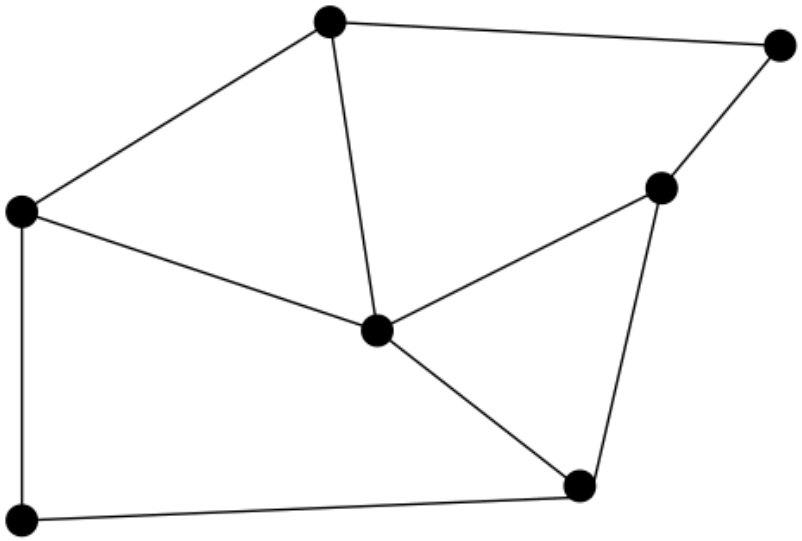


weighted

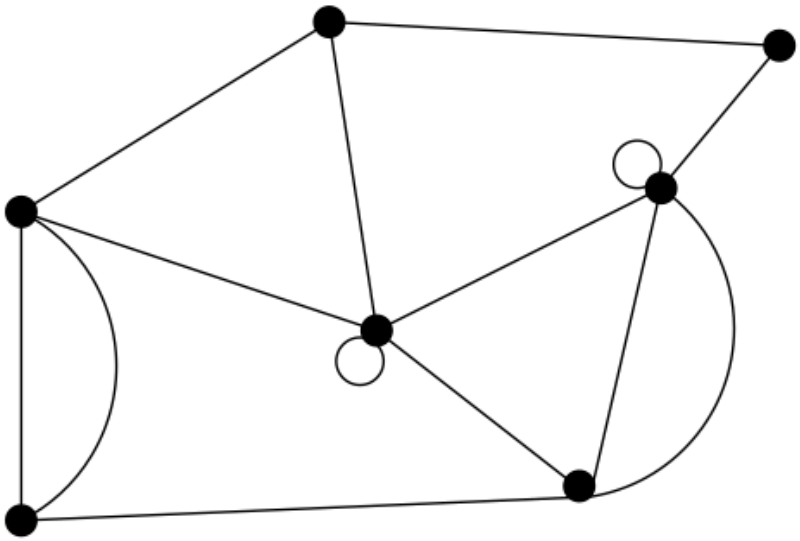
For unweighted graphs, the shortest path must have the fewest number of edges and can be found using breadth-first search (see later).

Shortest paths in weighted graphs requires more sophisticated algorithms (see later)

Graphs



simple



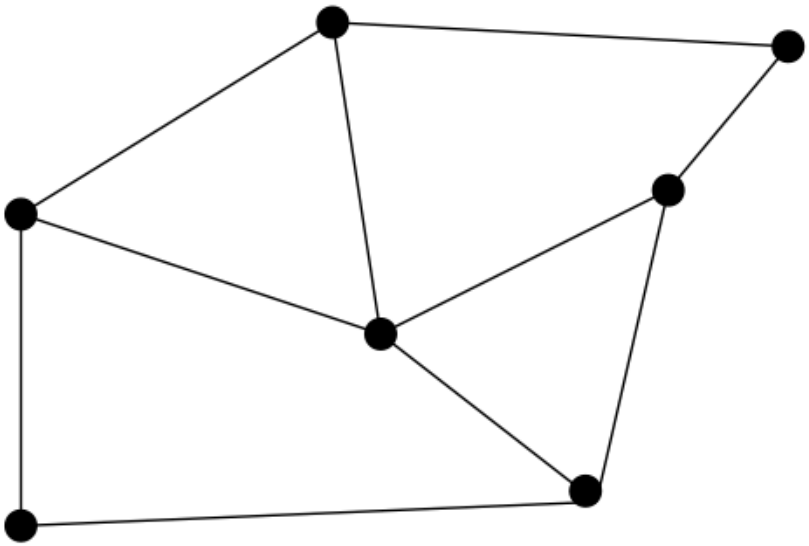
non–simple

Certain types of edges complicate the task of working with graphs.

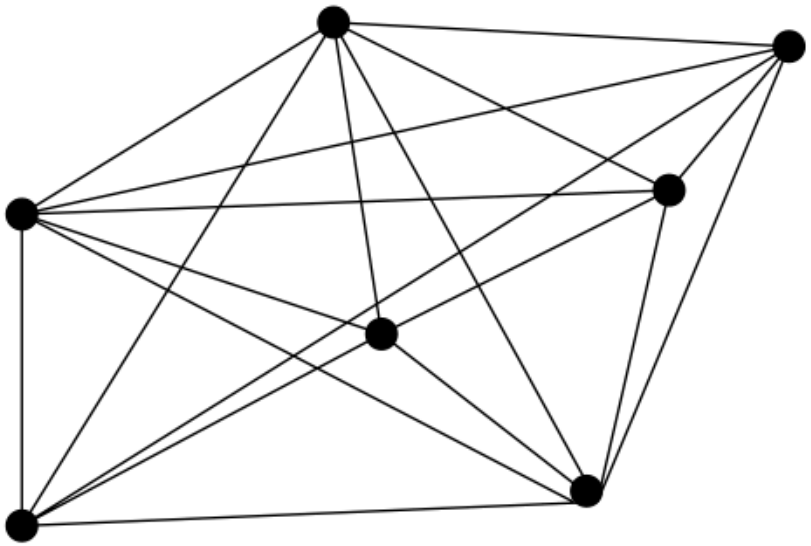
A *self-loop* is an edge (x, x) involving only one vertex.
An edge (x, y) is a *multiedge* if it occurs more than once in the graph

Graphs that do not have these types of edges are called *simple*

Graphs



sparse



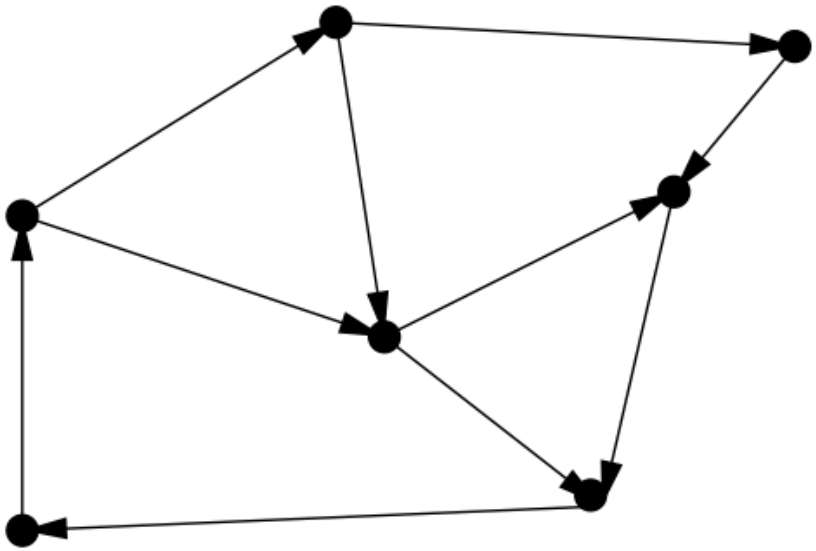
dense

There are $\binom{n}{2} = \frac{n!}{(n-2)!\,2!}$ possible vertex pairs in a simple undirected graph with n vertices.

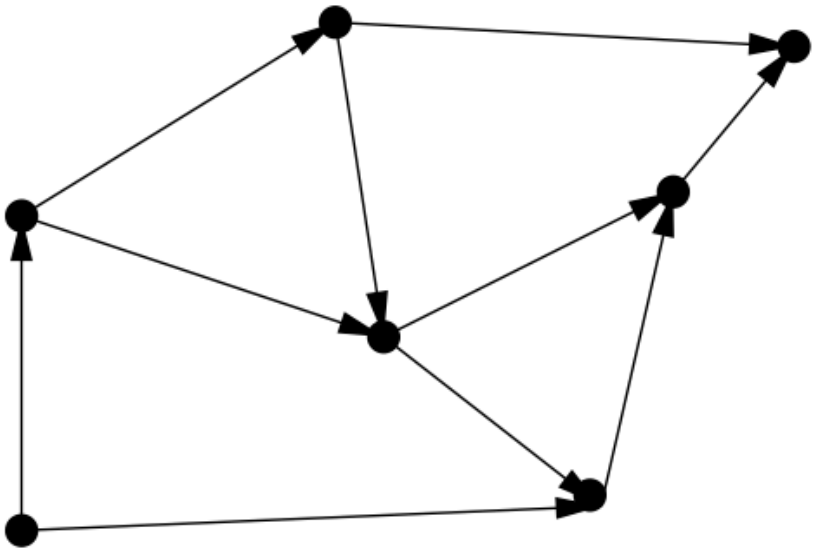
Graphs where a large fraction of the vertex pairs define edges are called dense

Typically dense graphs has a quadratic number of edges, sparse graphs are linear in size

Graphs



cyclic

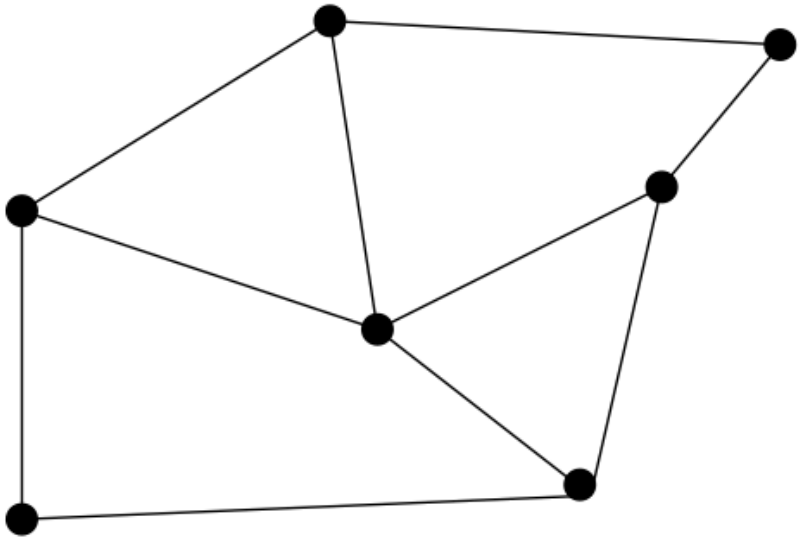


acyclic

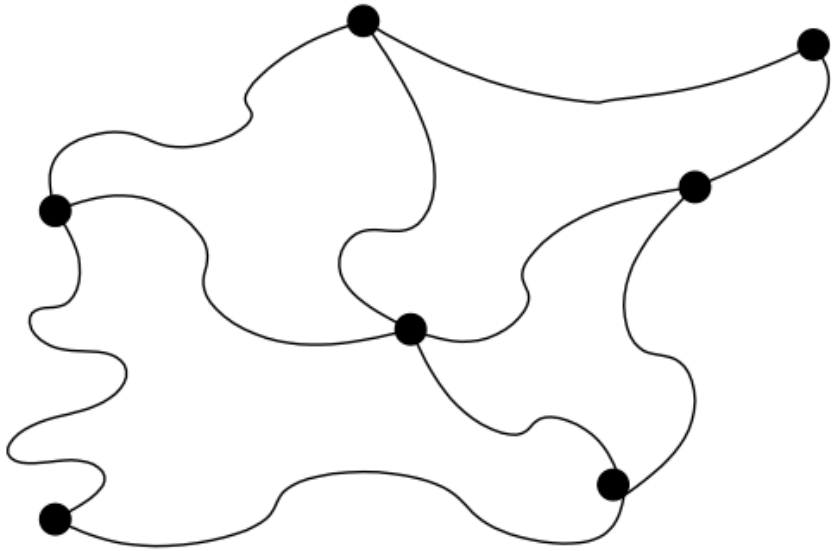
An acyclic graph does not contain any cycles
Trees are connected, acyclic undirected graphs

Directed acyclic graphs are called *DAGs*. They arise in scheduling problems where a directed edge (x, y) indicates that activity x must occur before activity y
A topological sort orders the vertices of a DAG w.r.t. these precedence constraints

Graphs



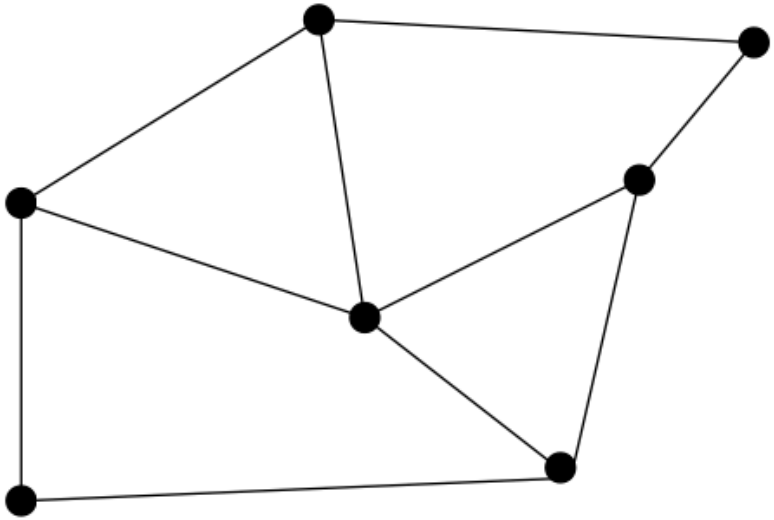
embedded



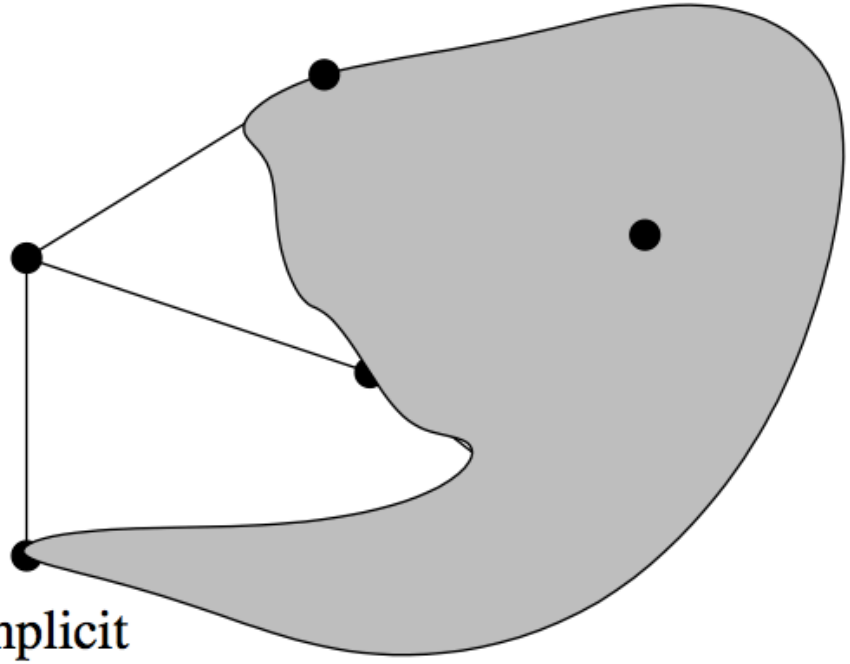
topological

A graph is embedded if the vertices and edges are assigned geometric positions

Graphs



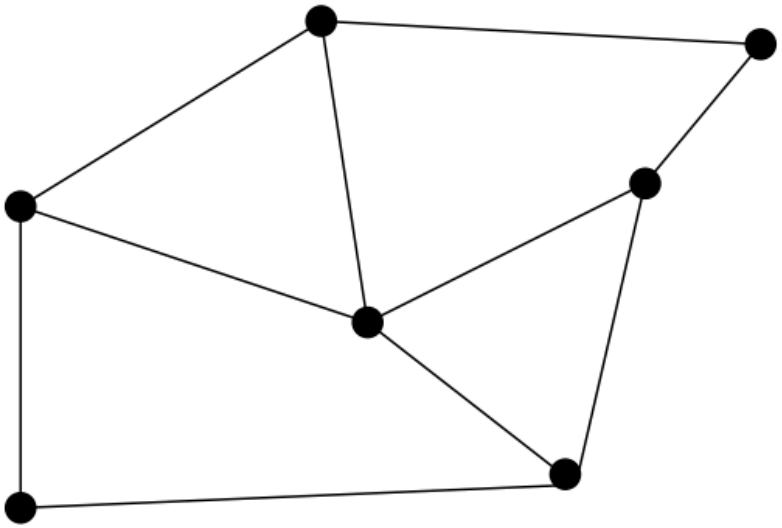
explicit



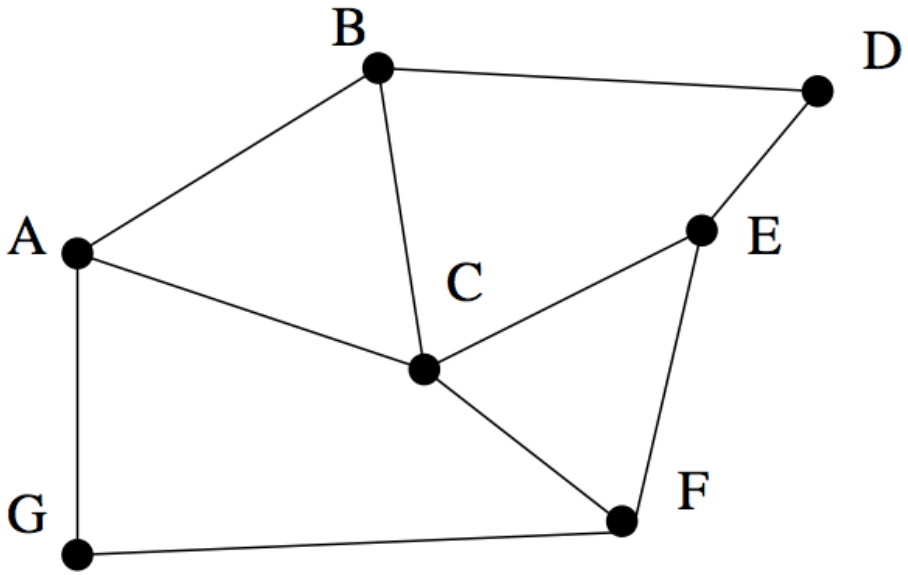
implicit

Certain graphs are not explicitly constructed and then traversed, but built as we use them (e.g. in a backtrack search; see later)

Graphs



unlabeled



labeled

Each vertex is assigned a unique name in a labelled graph to distinguish it from other vertices. In unlabelled graphs, no such distinctions are made.

Sub-graph isomorphism testing: determine whether the topological structure of two (sub-)graphs are identical if we ignore any labels (typically solved using backtracking, by trying to assign each vertex in each graph a label such that the structures are identical)

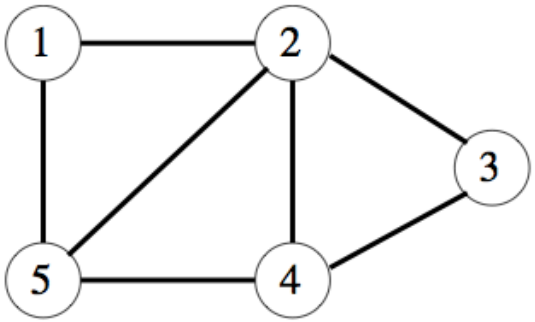
Graphs

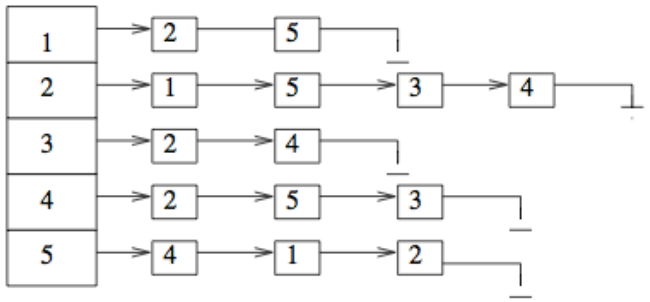
- Assuming a graph $G = (V, E)$ with n vertices and m edges, there are two basic choices for data structures

  – Adjacency Matrix: an $n \times n$ matrix M, where element $M[i, j] = 1$ if (i, j) is an edge of G, and 0 if it isn't (or, alternatively $M[i, j] = \mathrm{w}$, the weight of the edge)

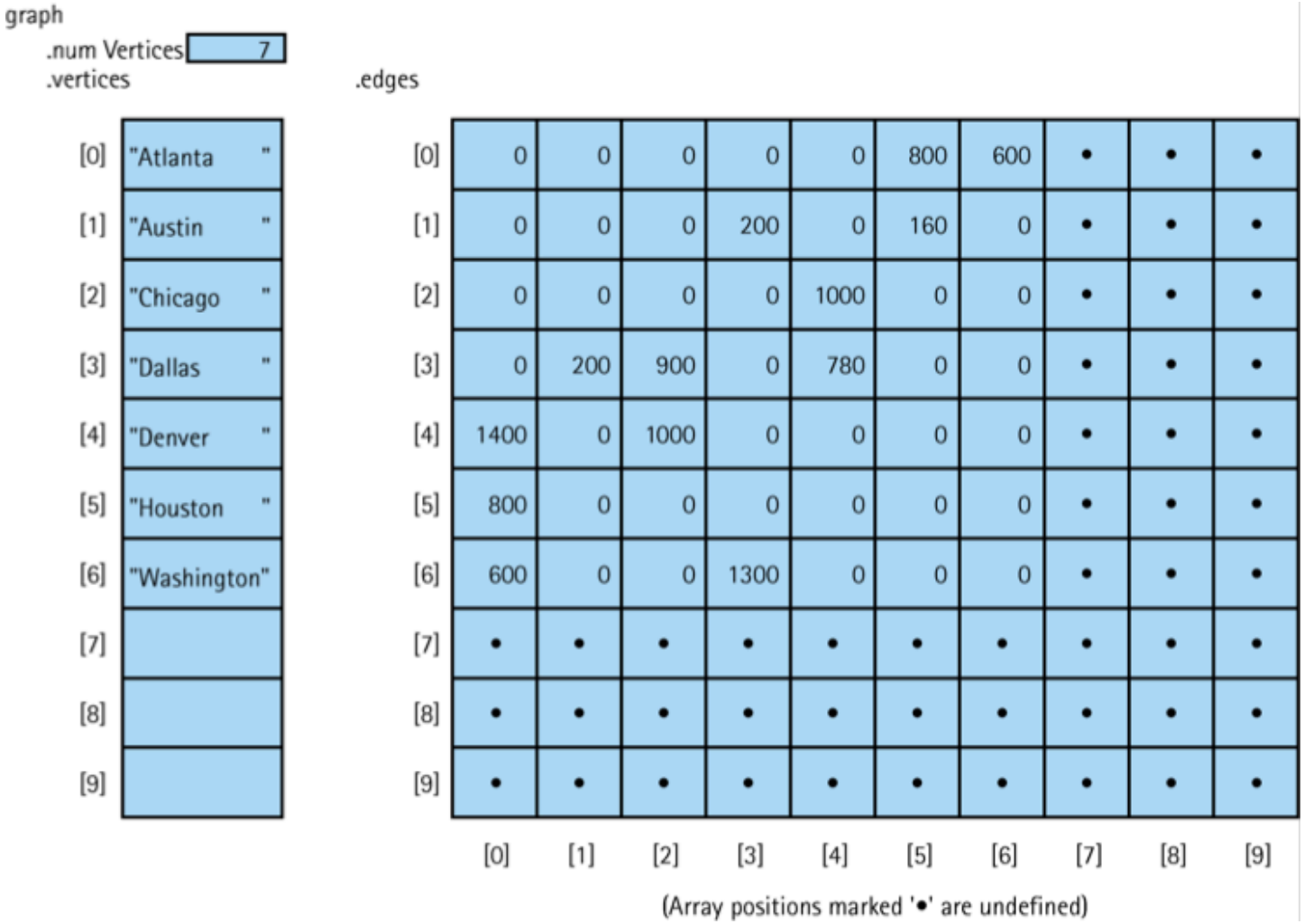
  – Adjacency List: a linked list that stores the neighbours that are adjacent to each vertex

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



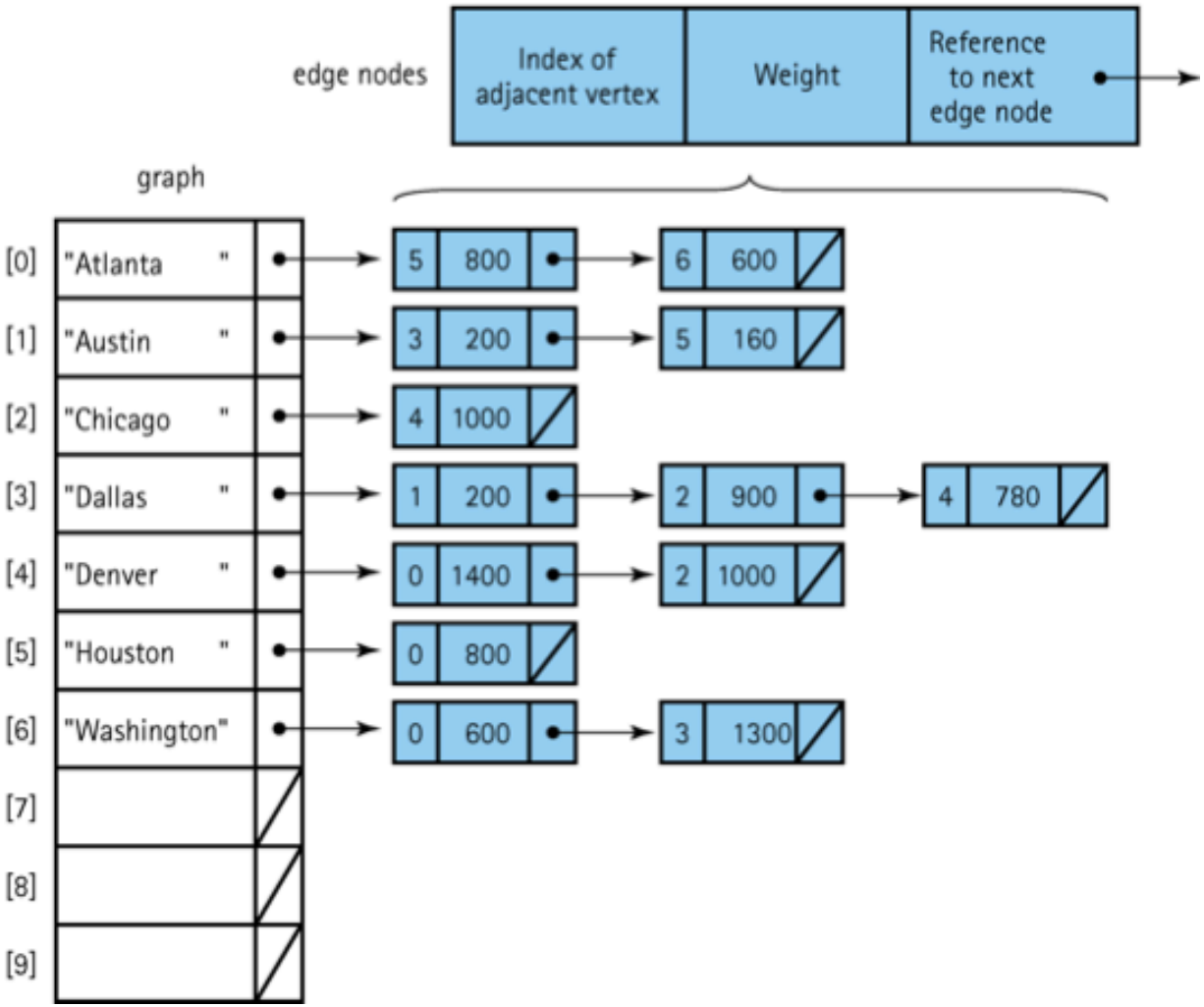


Graphs



Adjacency Matrix for Flight Connections

Graphs



Adjacency List for Flight Connections

Graphs

While Adjacency Matrices are simpler, Adjacency Lists are the right data structure for most applications of graphs

Comparison	Winner
Faster to test if (x, y) is in graph?	adjacency matrices
Faster to find the degree of a vertex?	adjacency lists
Less memory on small graphs?	adjacency lists $(m + n)$ vs. (n^2)
Less memory on big graphs?	adjacency matrices (a small win)
Edge insertion or deletion?	adjacency matrices $O(1)$ vs. $O(d)$
Faster to traverse the graph?	adjacency lists $\Theta(m + n)$ vs. $\Theta(n^2)$
Better for most problems?	adjacency lists

Graphs

```
/* Adjacency list representation of a graph of degree MAXV          */
/*                                                                  */
/* Directed edge (x, y) is represented by edgenode y in x's         */
/* adjacency list. Vertices are numbered 1 .. MAXV                  */

#define MAXV 1000  /* maximum number of vertices */

typedef struct {
   int y;                          /* adjacent vertex number        */
   int weight;                     /* edge weight, if any           */
   struct edgenode *next;          /* next edge in list             */
} edgenode;

typedef struct {
        edgenode *edges[MAXV+1]; /* adjacency info: list of edges   */
        int degree[MAXV+1];      /* number of edges for each vertex */
        int nvertices;           /* number of vertices in graph     */
        int nedges;              /* number of edges in graph        */
        bool directed;           /* is the graph directed?          */
} graph;
```

Graphs

```
/* Initialize graph from data in a file                                  */

initialize_graph(graph *g, bool directed){

   int i;                              /* counter */

   g -> nvertices = 0;
   g -> nedges = 0;
   g -> directed = directed;

   for (i=1; i<=MAXV; i++)
      g->degree[i] = 0;

   for (i=1; i<=MAXV; i++)
      g->edges[i] = NULL;
}
```

Graphs

```
/* Initialize graph from data in a file                              */

read_graph(graph *g, bool directed) {

   int i;    /* counter                  */
   int m;    /* number of edges          */
   int x, y; /* vertices in edge (x,y) */

   initialize_graph(g, directed);

   scanf("%d %d",&(g->nvertices),&m);

   for (i=1; i<=m; i++) {
      scanf("%d %d",&x,&y);
      insert_edge(g,x,y,directed);
   }
}
```

Graphs

```
/* Initialize graph from data in a file                                  */

insert_edge(graph *g, int x, int y, bool directed) {

   edgenode *p;                    /* temporary pointer */

   p = malloc(sizeof(edgenode)); /* allocate edgenode storage */

   p->weight = NULL;
   p->y = y;
   p->next = g->edges[x];

   g->edges[x] = p;                /* insert at head of list        */

   g->degree[x] ++;

   if (directed == FALSE)          /* NB: if undirected add         */
      insert_edge(g,y,x,TRUE);     /* the reverse edge recursively  */
   else                            /* but directed TRUE so we do it */
      g->nedges ++;                /* only once                     */
}
```

Graphs

```
/* Print a graph                                                    */

print_graph(graph *g) {

   int i;                              /* counter           */
   edgenode *p;                        /* temporary pointer */

   for (i=1; i<=g->nvertices; i++) {
      printf("%d: ",i);
      p = g->edges[i];
      while (p != NULL) {
         printf(" %d",p->y);
         p = p->next;
      }
      printf("\n");
   }
}
```

Graphs

Consider using a well-established graph library for implementing graph-based applications

- LEDA Library of Efficient Data types and Algorithms

  www.algorithmic-solutions.com

- Boost Graph Library

  www.boost.org

  www.boost.org/libs/graph/doc

Traversing a Graph

- Visit every edge and every vertex in a systematic way
- Key idea: mark each vertex when we first visit it & keep track of what we have not yet completely explored
- Each vertex will exist in one of three states

1. Undiscovered – the vertex is in its initial untouched state
2. Discovered – the vertex has been found, but we have not yet checked out all its incident edges
3. Processed – the vertex after we have visited all its incident edges

Traversing a Graph

- Keep a record of all the vertices discovered but not yet completely processed
- Begin with a starting vertex
- Explore each vertex
  - Evaluate each edge leaving it
  - If the edge goes to an undiscovered vertex
    - Mark it discovered
    - Add it to the list of work to do
  - If the edge goes to a processed vertex, ignore it
  - If the edge goes to a discovered unprocessed vertex, ignore it

Traversing a Graph

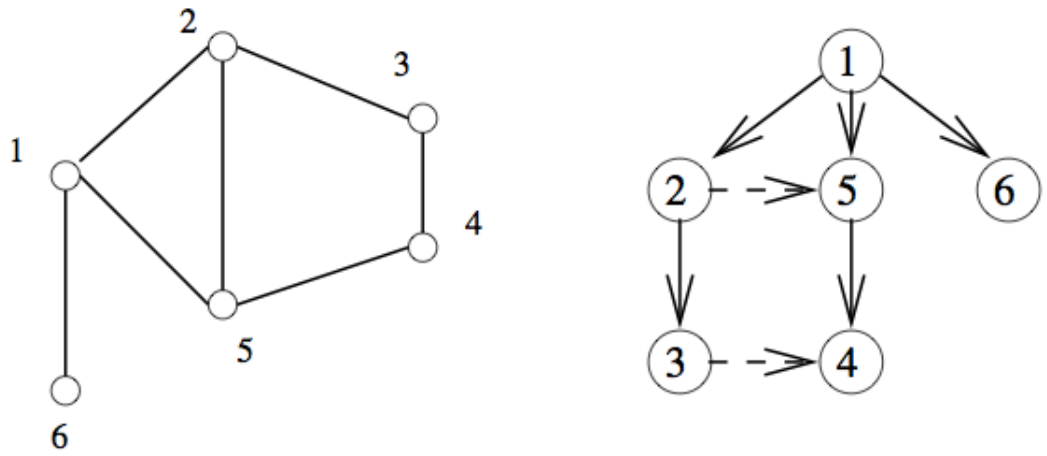
- There are two primary graph traversal algorithms
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
- The difference is the order in which they explore vertices

Traversing a Graph

- The order depends completely on the *container* data structure used to store the *discovered* but *not processed* vertices
  - BFS uses a queue
    - By storing the vertices in a FIFO queue, we explore the oldest unexplored vertices first
    - Thus explorations radiate out slowly from the starting vertex
  - DFS uses a stack
    - By storing the vertices in a LIFO stack, we explore the vertices by diving down a path, visiting a new neighbour if one is available, and backing up only when we are surrounded by previously discovered vertices
    - Thus explorations quickly wander away from out starting vertex

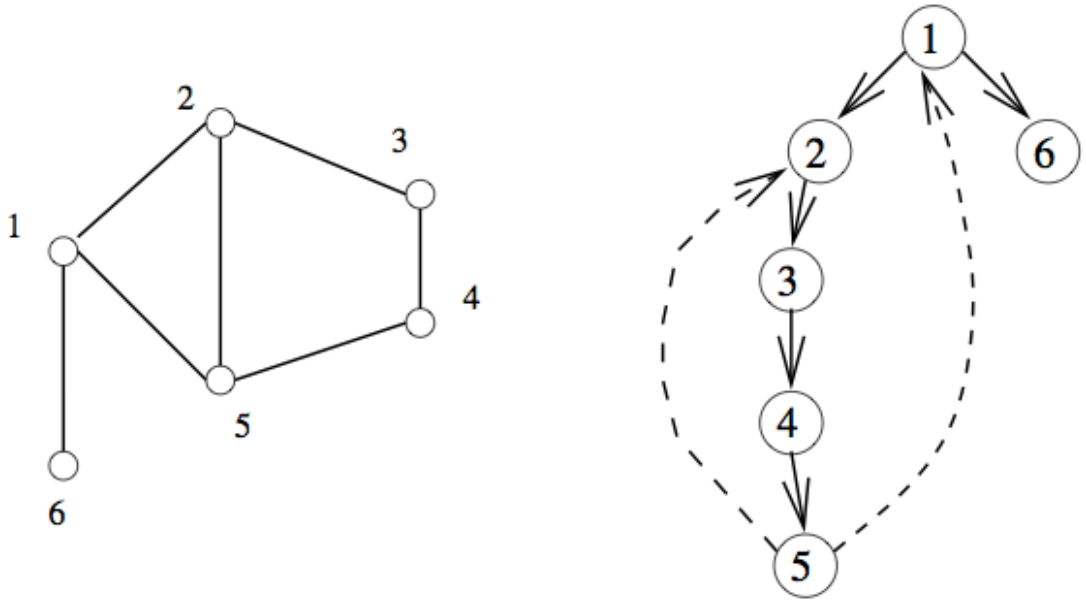
Traversing a Graph

Breadth-first search (BFS)



Traversing a Graph

Depth-first search (DFS)



Breadth-First Search

- Assign a direction to each edge, from discoverer vertex u to discovered vertex v
- Since each node has exactly one parent, except for the root (i.e. start vertex), this defines a tree on the vertices of the graph
- This tree defines the shortest path from the root to every other node in the tree
- This makes the BFS very useful for in shortest path problems (in unweighted graphs)

Breadth-First Search

```
BFS(G, s)
    for each vertex u ∈ V[G] − {s} do
        state[u] = "undiscovered"
        p[u] = nil, i.e. no parent is in the BFS tree
    state[s] = "discovered"
    p[s] = nil
    Q = {s}
    while Q ≠ ∅ do
        u = dequeue[Q]
        process vertex u as desired
        for each v ∈ Adj[u] do
            process edge (u, v) as desired
            if state[v] = "undiscovered" then
                state[v] = "discovered"
                p[v] = u
                enqueue[Q, v]
        state[u] = "processed"
```

Breadth-First Search

```
/* Breadth-First Search                                                  */

bool processed[MAXV+1];   /* which vertices have been processed */
bool discovered[MAXV+1];  /* which vertices have been found */
int parent[MAXV+1];       /* discovery relation */


/* Each vertex is initialized as undiscovered:                          */

initialize_search(graph *g){

   int i;                           /* counter */

   for (i=1; i<=g->nvertices; i++) {
      processed[i] = discovered[i] = FALSE;
      parent[i] = -1;
   }
}
```

Breadth-First Search

```
/* Once a vertex is discovered, it is placed on a queue.           */
/* Since we process these vertices in first-in, first-out order,   */
/* the oldest vertices are expanded first, which are exactly those */
/* closest to the root                                             */

bfs(graph *g, int start)
{
   queue q;                    /* queue of vertices to visit */
   int v;                      /* current vertex             */
   int y;                      /* successor vertex           */
   edgenode *p;                /* temporary pointer          */

   init_queue(&q);
   enqueue(&q,start);
   discovered[start] = TRUE;
```

Breadth-First Search

```
    while (empty_queue(&q) == FALSE) {
       v = dequeue(&q);
       process_vertex_early(v);
       processed[v] = TRUE;
       p = g->edges[v];

       while (p != NULL) {
          y = p->y;
          if ((processed[y] == FALSE) || g->directed)
             process_edge(v,y);
          if (discovered[y] == FALSE) {
             enqueue(&q,y);
             discovered[y] = TRUE;
             parent[y] = v;
          }
          p = p->next;
       }
       process_vertex_late(v);
    }
}
```

Breadth-First Search

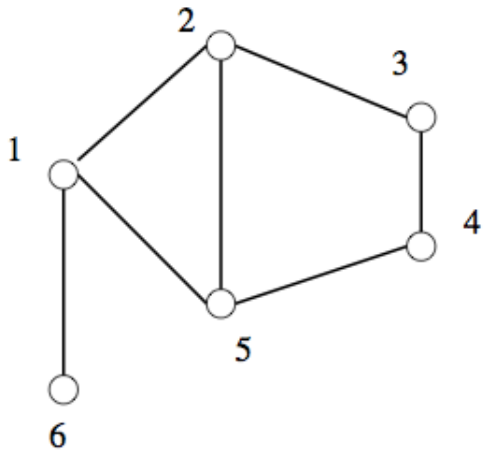
```
/* The exact behaviour of bfs depends on the functions              */
/*    process vertex early()                                        */
/*    process vertex late()                                         */
/*    process edge()                                                */
/* These functions allow us to customize what the traversal does    */
/* as it makes its official visit to each edge and each vertex.     */
/* Here, e.g., we will do all of vertex processing on entry         */
/* (to print each vertex and edge exactly once)                     */
/* so process vertex late() returns without action                  */

process_vertex_late(int v) {
}

process_vertex_early(int v){
   printf("processed vertex %d\n",v);
}

process_edge(int x, int y) {
   printf("processed edge (%d,%d)\n",x,y);
}
```

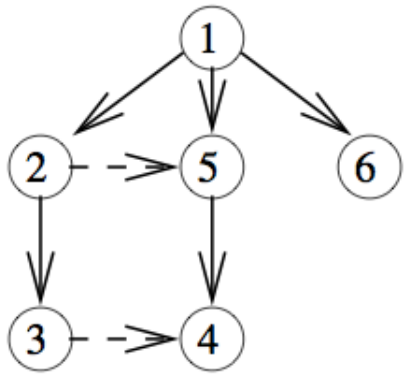
Breadth-First Search

```
/* this version just counts the number of edges                    */

process_edge(int x, int y) {
   nedges = nedges + 1;
}
```

Breadth-First Search

- Finding Paths
  - The `parent` array in `bfs()` is very useful for finding interesting paths through a graph
  - The vertex that discovered vertex `i` is defined as `parent[i]`





vertex	1	2	3	4	5	6
parent	-1	1	2	5	1	1

Breadth-First Search

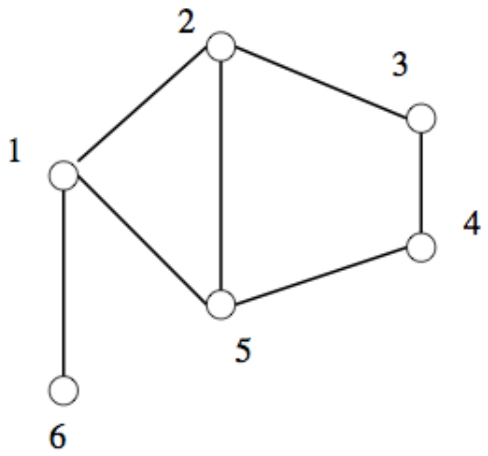
- Finding Paths
  - Every vertex is discovered during the course of a traversal so every node has a parent (except the root)
  - The parent relation defines a tree of discovery with the initial search node as the root of the tree
  - Because vertices are discovered in order of increasing distance from the root, this tree has a very important property
    - The unique tree path from the root to each node uses the smallest number of edges (and intermediate nodes) possible on any path from the root to that vertex
    - Thus BFS can be used to find shortest paths in an **unweighted** graph

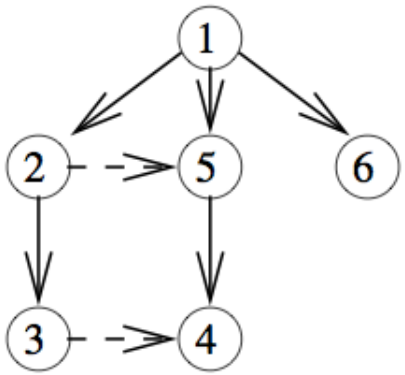
Breadth-First Search

- Finding Paths
  - To reconstruct a path we follow the chain of ancestors from the destination node x to the root
  - Note we have to work backwards (we only know the parents)
  - We find the path from to the root and
    - Either store it and explicitly reverse it using a stack
    - Or construct the path recursively (in which case the stack is implicit)

Breadth-First Search

```
find_path(int start, int end, int parents[]) {
   if ((start == end) || (end == -1))
      printf("\n%d",start);
   else {
      find_path(start,parents[end],parents);
      printf(" %d",end);
   }
}
```





vertex	1	2	3	4	5	6
parent	-1	1	2	5	1	1

Breadth-First Search

- Applications of Breadth-First Search
  - Identifying connected components
    - A graph is *connected* if there is a path between any two vertices
    - A *connected component* of an undirected graph is a maximal set of vertices such that there is a path between every pair of vertices
    - The components are separate “pieces” of the graph such that there is no connection between the pieces
    - Many complicated problems reduce to finding or counting connected components

Breadth-First Search

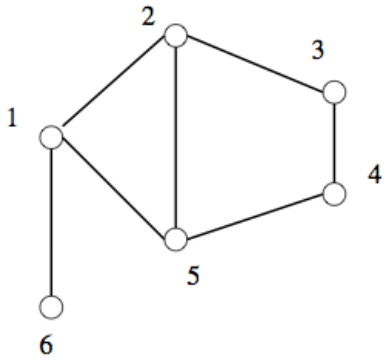
- Applications of Breadth-First Search
  - Two-Colouring Graphs
    - The *vertex-colouring* problem seeks to assign a label (or colour) to each vertex of a graph such that no edge links any two vertices of the same colour
    - The goal is use as few colours as possible
    - A graph is bipartite if it can be coloured without conflicts using only two colours

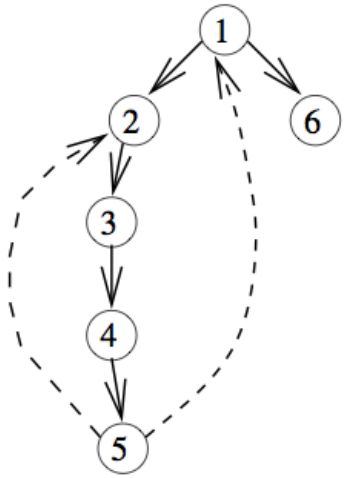
Depth-First Search

- This implementation of DFS uses the idea of a traversal time for each vertex
- The clock ticks each time we enter or exit any vertex
- We keep track of the entry and exit time for each vertex
- These entry and exit times are useful in many applications of DFS (e.g. topological sort; see later)
  - `process_vertex_early()` … take action on entry
  - `process_vertex_late()` … take action on exit
- DFS uses a stack but we can avoid using an explicit stack if we use recursion

Depth-First Search

- DFS partitions edges of an undirected graph into exactly two classes
  - Tree edges
  - Back edges





- Tree edges discover new vertices
  - Encoded in the `parent` relation
- Back edges link a vertex to an ancestor of the vertex being expanded

Depth-First Search

```
DFS(G, u)
        state[u] = "discovered"
        process vertex u if desired
        entry[u] = time
        time = time + 1
        for each v ∈ Adj[u] do
                process edge (u, v) if desired
                if state[v] = "undiscovered" then
                        p[v] = u
                        DFS(G, v)
        state[u] = "processed"
        exit[u] = time
        time = time + 1
```

Depth-First Search

```
/* Depth-First Search                                                    */

dfs(graph *g, int v){

   edgenode *p;                /* temporary pointer */
   int y;                      /* successor vertex  */

   if (finished) return;       /* allow for search termination */

   discovered[v] = TRUE;
   time = time + 1;
   entry_time[v] = time;

   process_vertex_early(v);
```

Depth-First Search

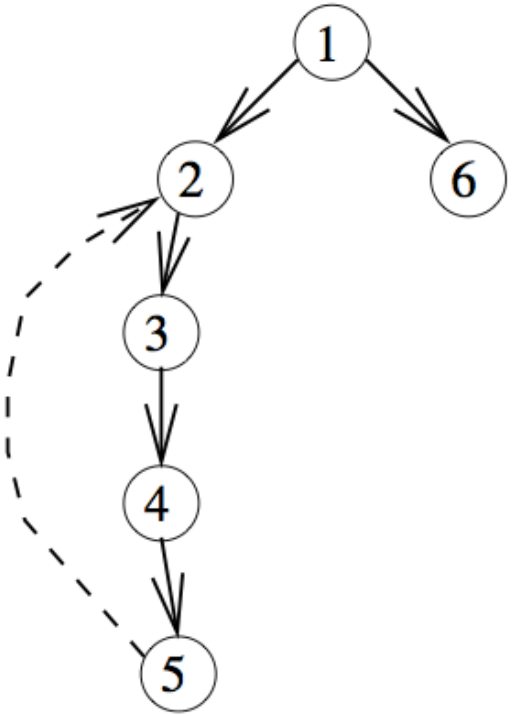
```
    p = g->edges[v];
    while (p != NULL) {
       y = p->y;
       if (discovered[y] == FALSE) {
          parent[y] = v;
          process_edge(v,y);
          dfs(g,y);
       }
       else if ((!processed[y]) || (g->directed))
          process_edge(v,y);

       if (finished) return;
       p = p->next;
    }

    process_vertex_late(v);

    time = time + 1;
    exit_time[v] = time;

    processed[v] = TRUE;
}
```

Depth-First Search

- Depth-First Search uses essentially the same idea as backtracking
  - Exhaustively searching all possibilities by advancing it is is possible
  - And backing up as soon as there is no unexplored possibility for further advancement
  - Both are most easily understood as recursive algorithms
- DFS organizes vertices by entry/exit times
- DFS classifies edges as either tree edges or back edges

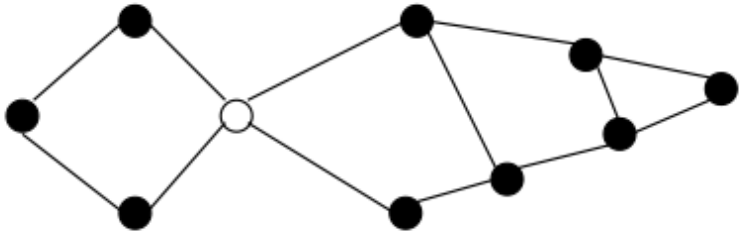
Depth-First Search

- Applications of Depth-First Search
  - Finding *Cycles*
    - If there are no back edges, then all edges are tree edges and no cycles exist
    - Finding a back edge identifies a cycle



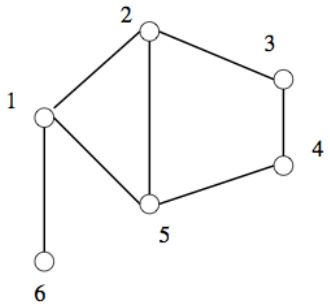
Depth-First Search

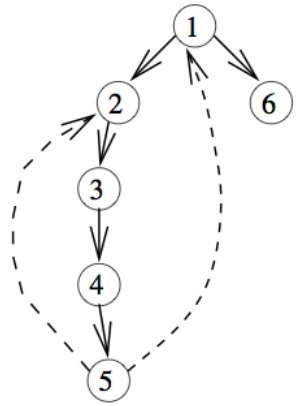
- Applications of Depth-First Search
  - Finding *Articulation Vertices* (also known as a *cut node*): weakest points in a graph/network



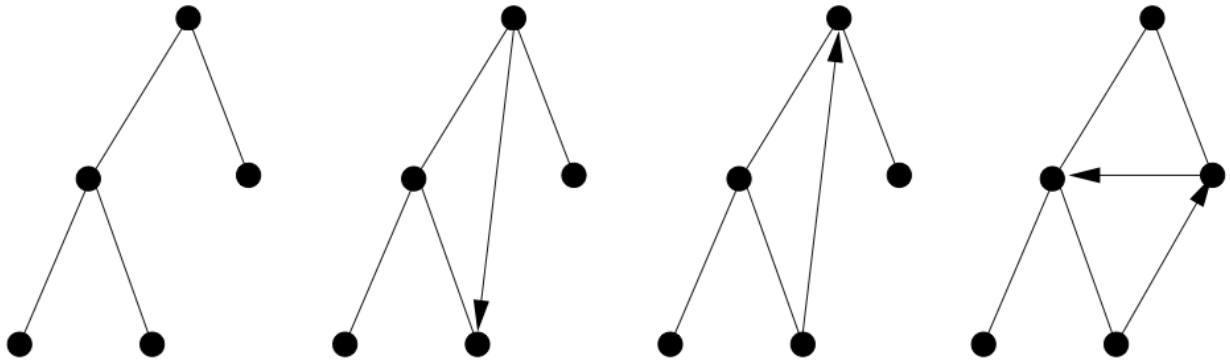
Depth-First Search

- Depth-First Search on Directed Graphs
  - When traversing undirected graphs, every edge is either in the depth-first search tree or it is a back edge to an ancestor in the tree





  - For directed graphs, there are 4 depth-first search labellings



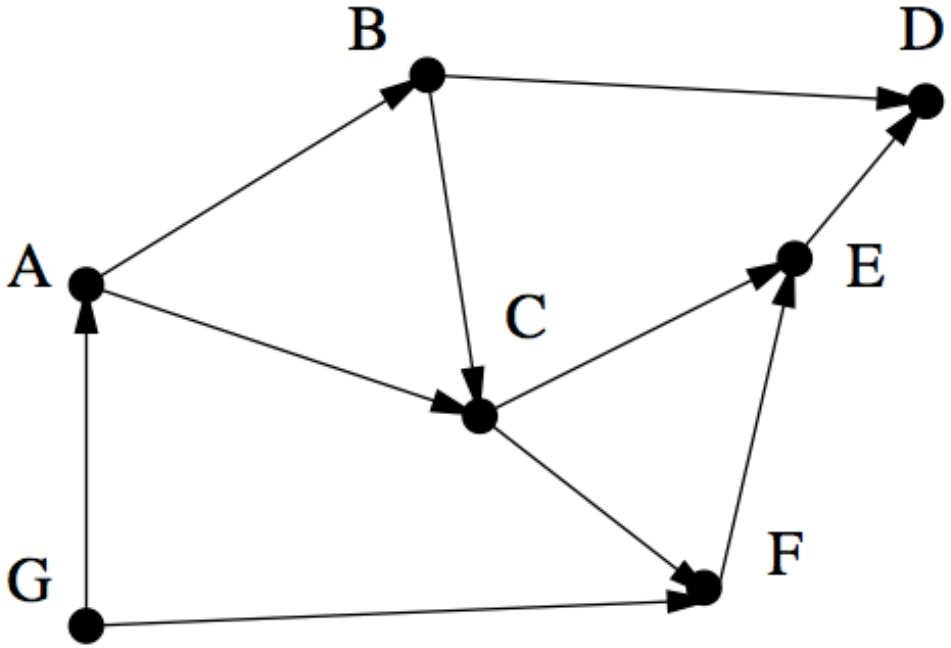
Depth-First Search

```
int edge_classification(int x, int y){
   if (parent[y] == x) return(TREE);
   if (discovered[y] && !processed[y]) return(BACK);
   if (processed[y] && (entry_time[y]>entry_time[x])) return(FORWARD);
   if (processed[y] && (entry_time[y]<entry_time[x])) return(CROSS);
   printf("Warning: unclassified edge (%d,%d)\n",x,y);
}
```

Topological Sorting

- The most important operation on directed acyclic graphs (DAGs)
- It orders the vertices on a line such that all directed edges go from left to right
  - Not possible if the graph contains a directed cycle
  - It provides a ordering to process each vertex before any of its successors
  - E.g. is edges represent precedence constraints, such that the edge (x, y) means job x must be done before job y
  - Any topological sort defines a valid schedule
- Each DAG has at least one topological sort

Topological Sorting



A DAG with only one topological sort (G, A, B, C, F, E, D)

Topological Sorting

- Topological sorting can be performed using DFS
- A directed graph is a DAG iff there are no back edges
- Labelling the vertices in the reverse order in which they are *processed* (completed) finds the topological sort of a DAG

Topological Sorting

- Why? Consider what happens to each directed edge (x, y) as we encounter it exploring vertex x

  – If y is currently undiscovered, then we start a DFS of y before we can continue with x. Thus y is marked processed/completed before x is, and x appears before y in the topological order

  – If y is discovered but not processed/completed, then (x, y) is a back edge, which is forbidden in a DAG

  – If y is processed/completed, then it will have been so labeled before x. Therefore, x appears before y in the topological order

Topological Sorting

```
process_vertex_late(int v){
        push(&sorted,v);
}

process_edge(int x, int y){

   int class;                    /* edge class */

   class = edge_classification(x,y);

   if (class == BACK)
      printf("Warning: directed cycle found, not a DAG\n");
}
```

Topological Sorting

```
/* Perform topological sort by doing a DFS on the graph,              */
/* pushing each vertex on a stack as soon as we have evaluated        */
/* all outgoing edges.                                                */
/* The top vertex on the stack always as no incoming edges from any */
/* vertex on the stack.                                               */
/* After the DFS, repeatedly popping the vertices from the stack      */
/* yields a topological ordering                                      */

topsort(graph *g) {

   int i;

   init_stack(&sorted);

   for (i=1; i<=g->nvertices; i++)
      if (discovered[i] == FALSE)
         dfs(g,i);

   print_stack(&sorted);        /* report topological order */
}
```