

04-630

Data Structures and Algorithms for Engineers

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 18

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

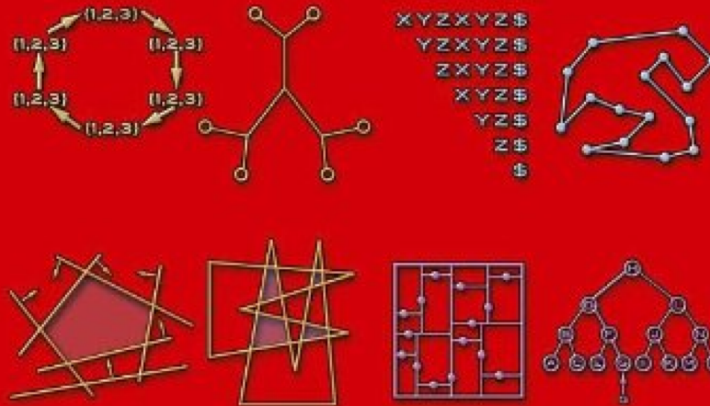
- A graph $G = (V, E)$ consists of
 - A set of *vertices* V
 - A set E of vertex pairs or *edges*
- Vertex: node in a graph
- Edge (arc): a pair of vertices representing a connection between two nodes in a graph
- Undirected graph: a graph in which the edges have no direction
- Directed graph [*digraph*]: 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**

Copyrighted Material.
Second Edition

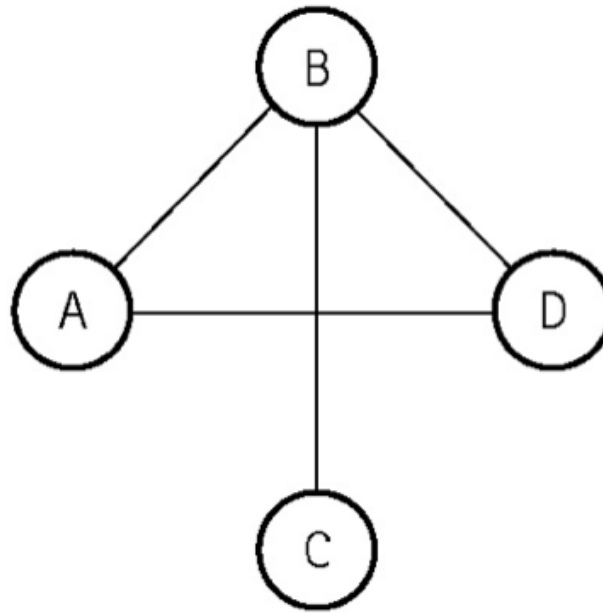
THE Algorithm Design MANUAL



Steven S. Skiena

 Springer
Copyrighted Material

Graphs

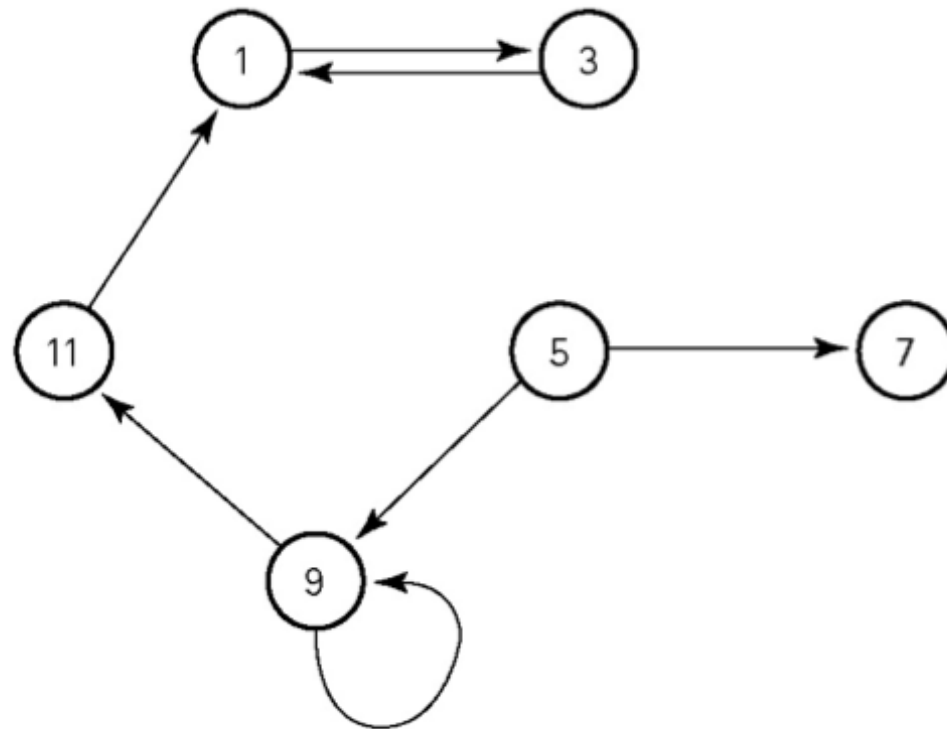


Undirected graph G

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

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

Graphs

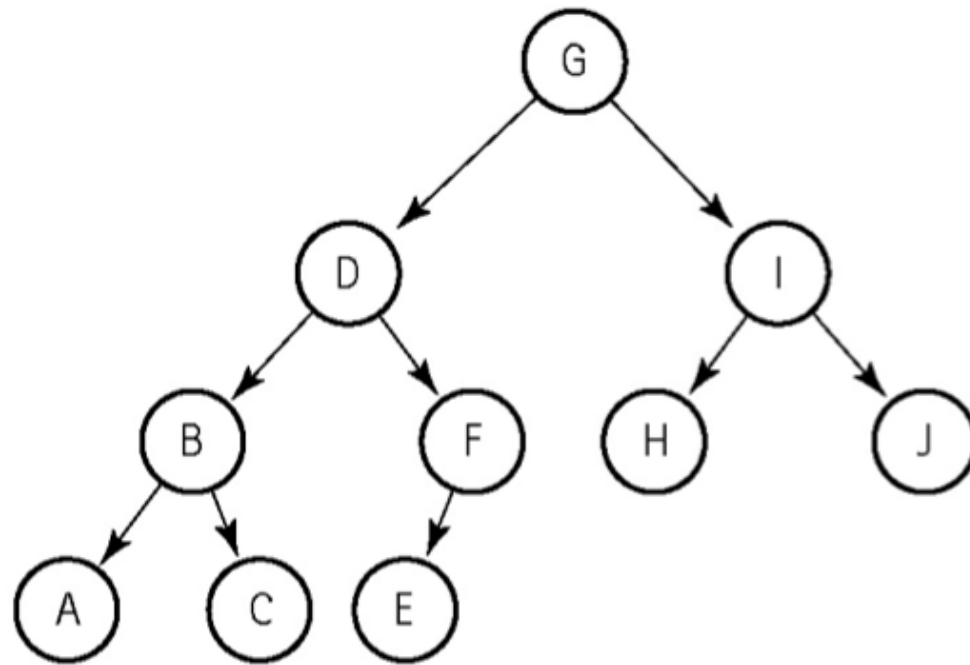


Directed graph G

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

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

Graphs



Directed graph G

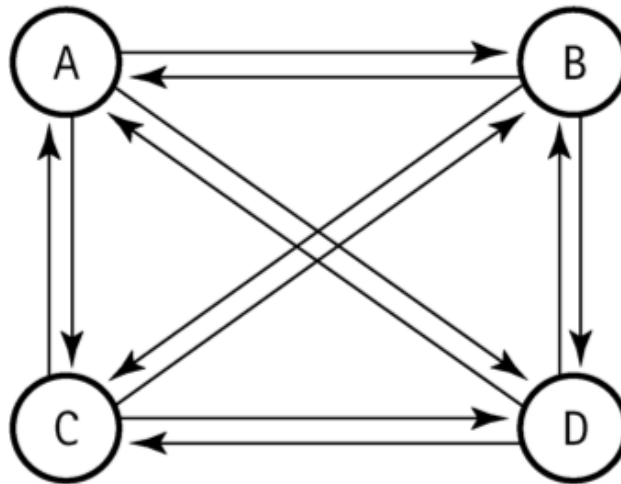
$V = \{A, B, C, D, E, F, G, H, I, J\}$

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

Graphs

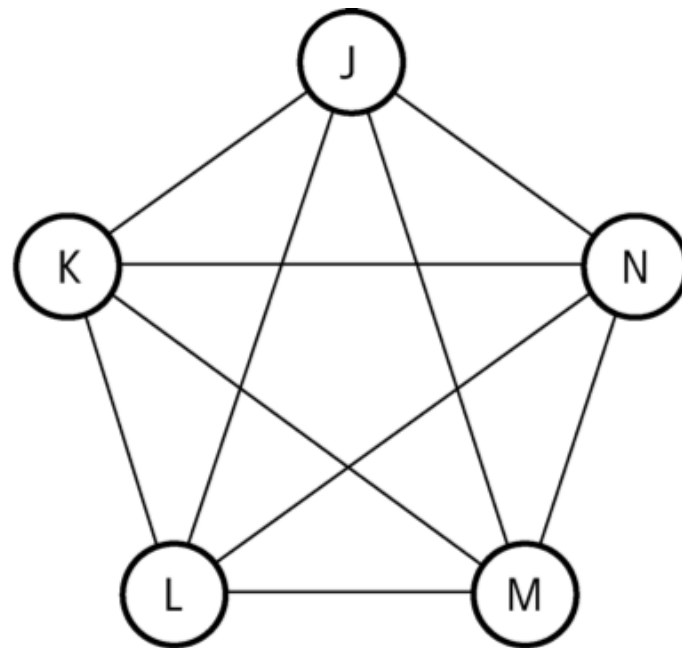
- Adjacent vertices
 - Two vertices in a graph that are connected by an edge
- Path
 - A sequence of vertices that connects two nodes in a graph
- Complete graph
 - A graph in which every vertex is directly connected to every other vertex
- Weighted graph
 - A graph in which each edge carries a value

Graphs



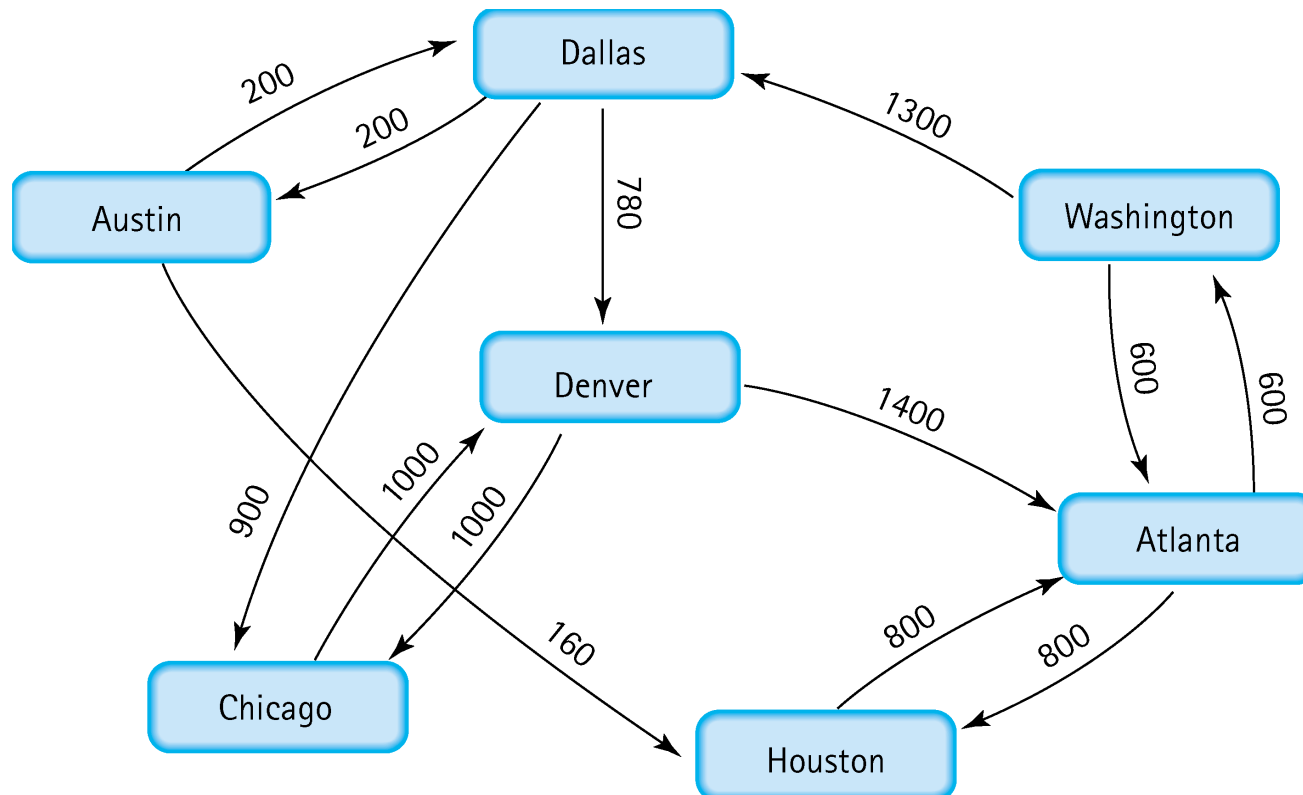
A complete directed graph G

Graphs



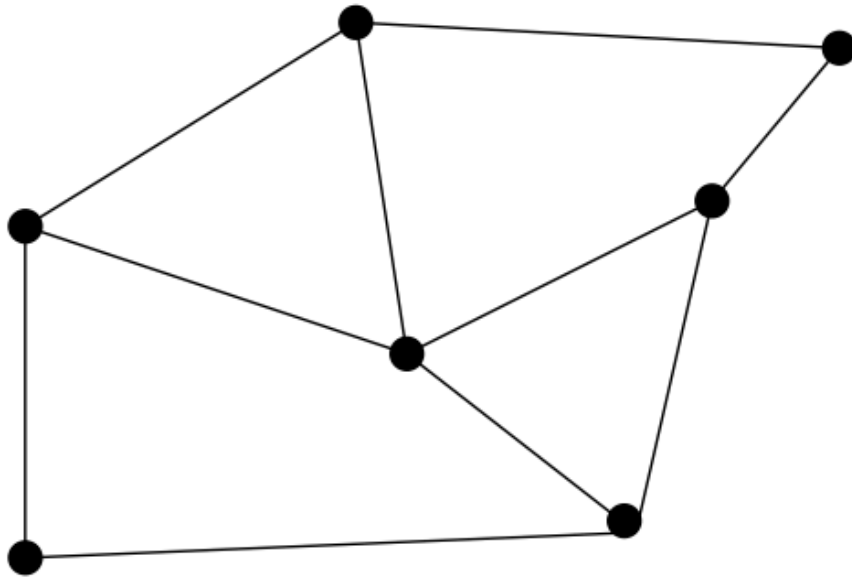
A complete undirected graph G

Graphs

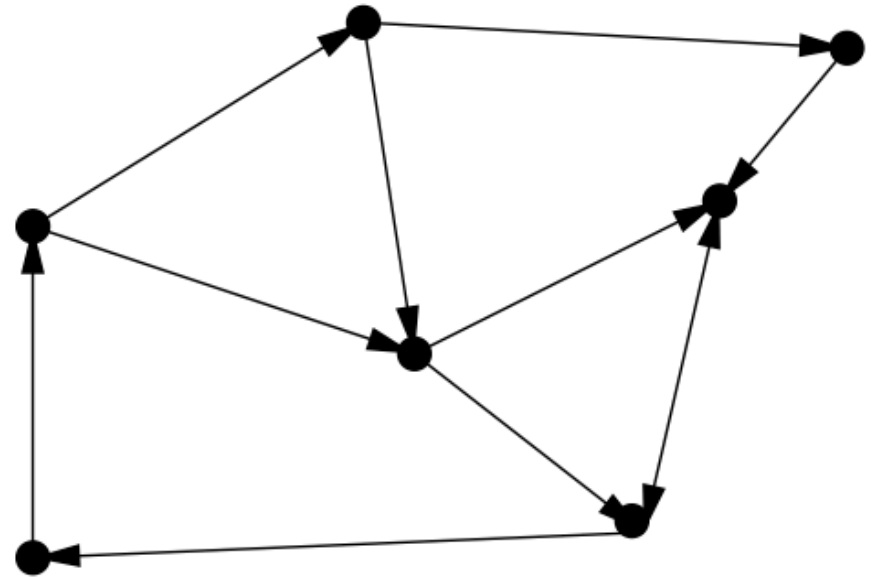


A weighted graph G

Graphs



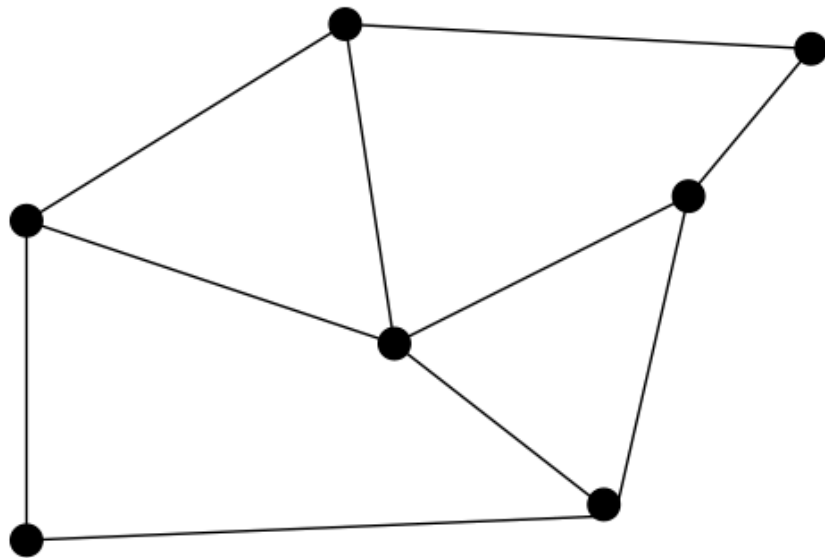
undirected



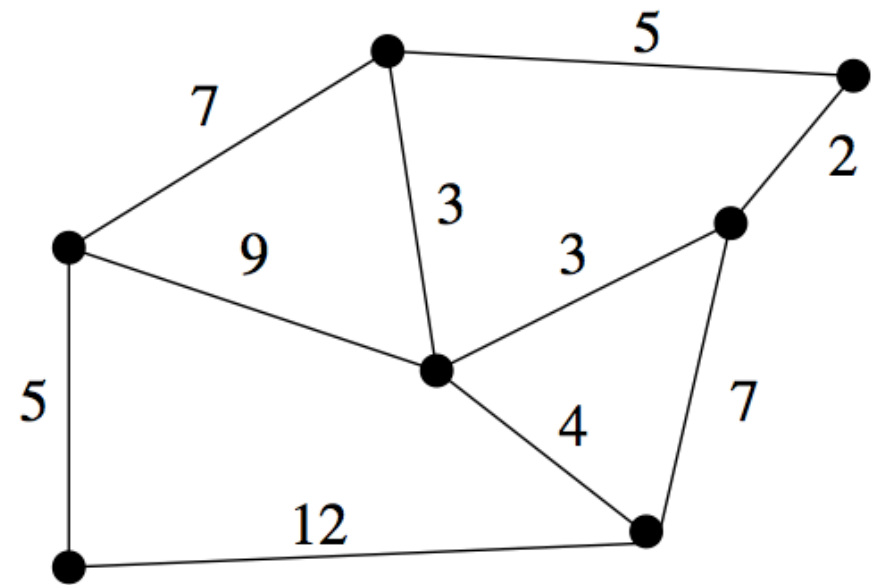
directed

A graph G is undirected if edge (x, y) is an element of E implies (y, x) is an element of E

Graphs



unweighted

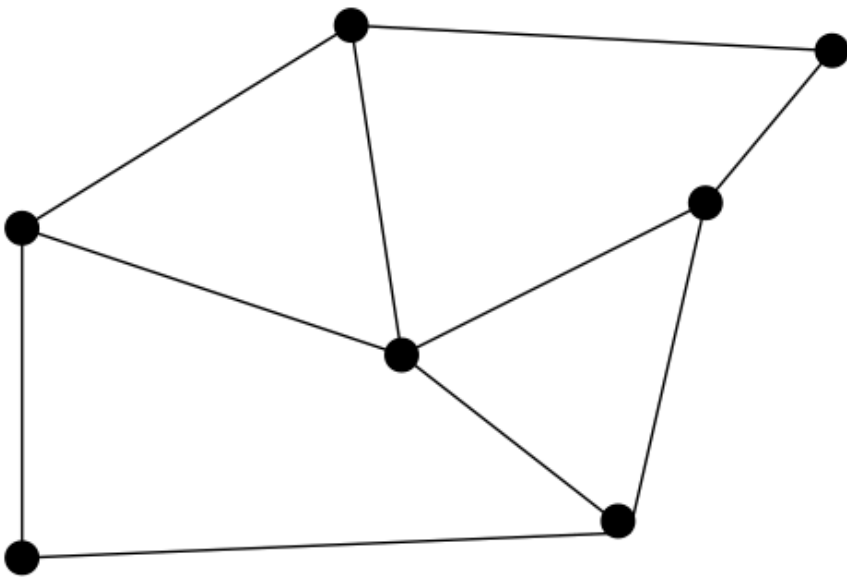


weighted

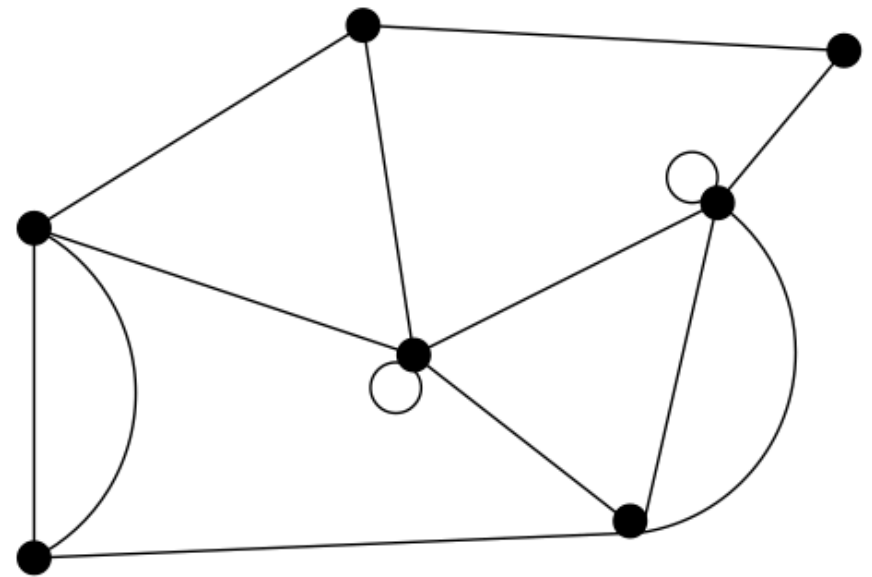
For unweighted graphs, the shortest path must have the fewest number of edges and can be found using **breadth-first search** (see later).

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

Graphs



simple



non-simple

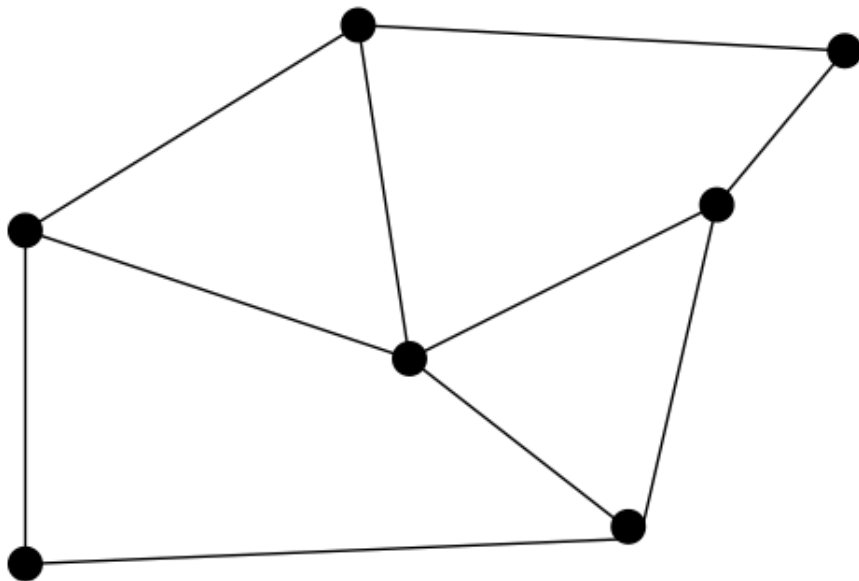
Certain types of edges complicate the task of working with graphs.

A *self-loop* is an edge (x, x) involving only one vertex.

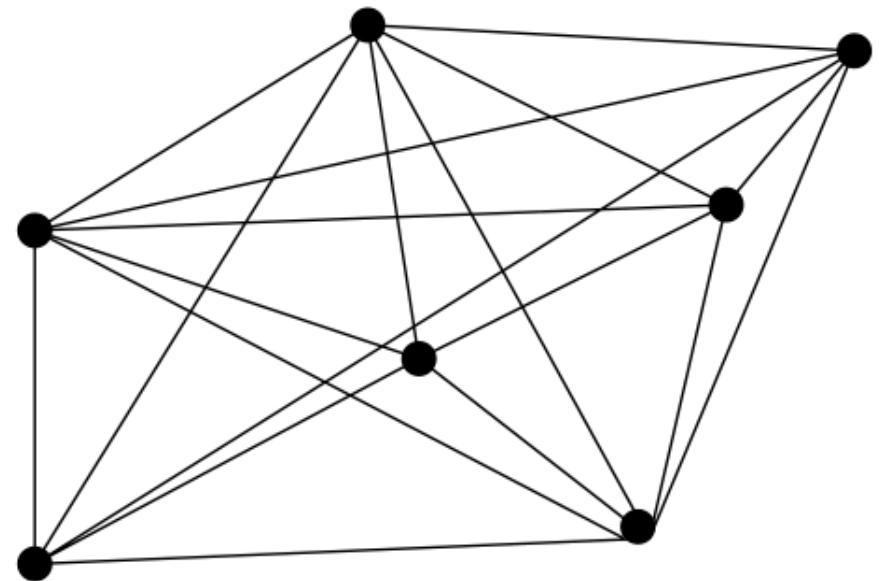
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph

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

Graphs



sparse



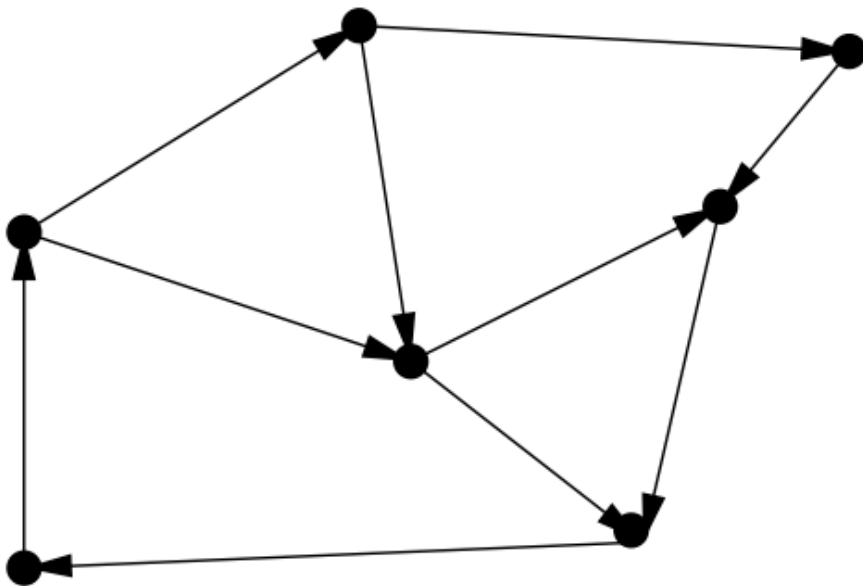
dense

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

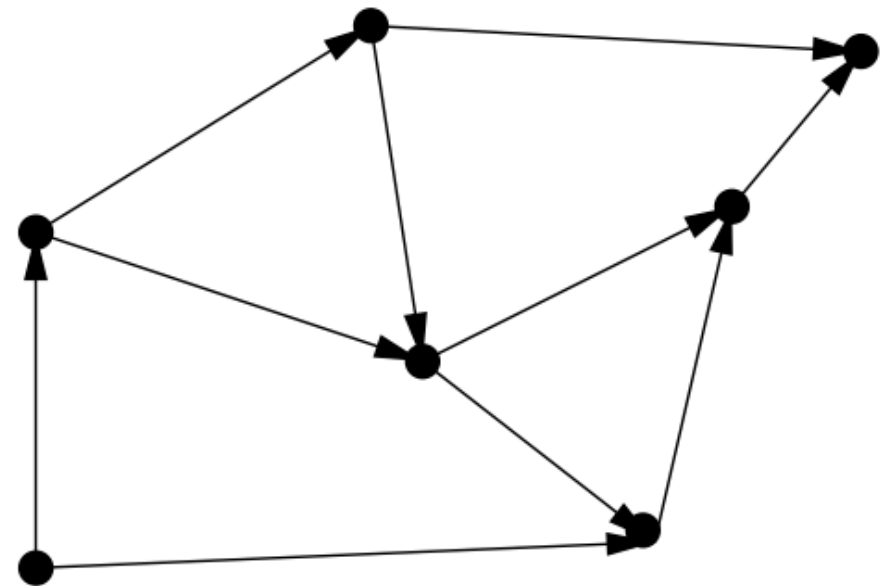
Graphs where a large fraction of the vertex pairs define edges are called dense

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

Graphs



cyclic



acyclic

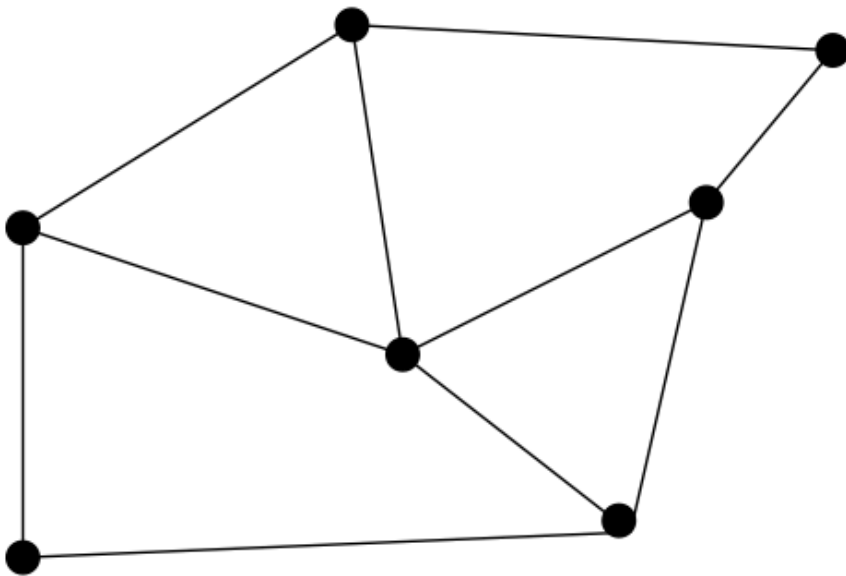
An **acyclic** graph does not contain any cycles

Trees are connected, acyclic undirected graphs

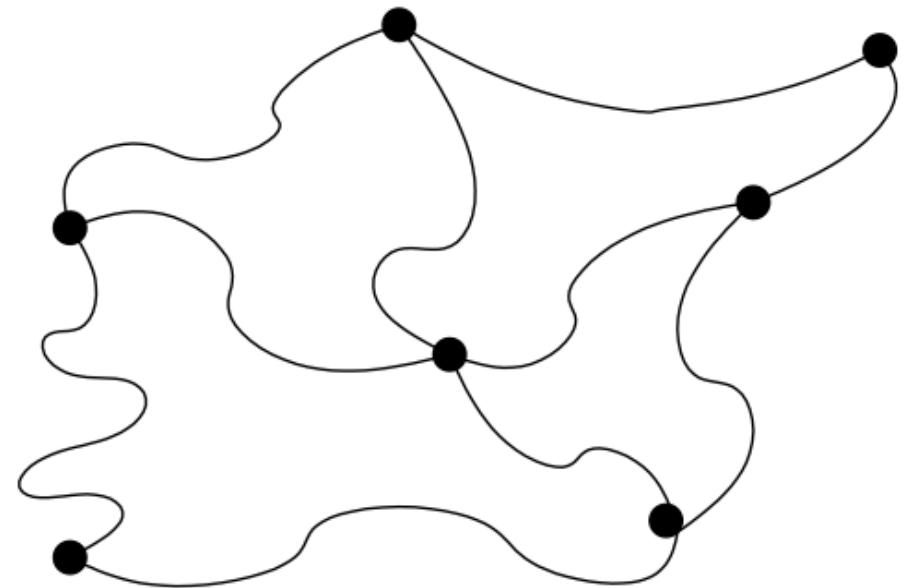
Directed acyclic graphs are called **DAGs**. They arise in scheduling problems where a directed edge $[x, y]$ indicates that activity x must occur before activity y

A topological sort orders the vertices of a DAG w.r.t. these precedence constraints

Graphs



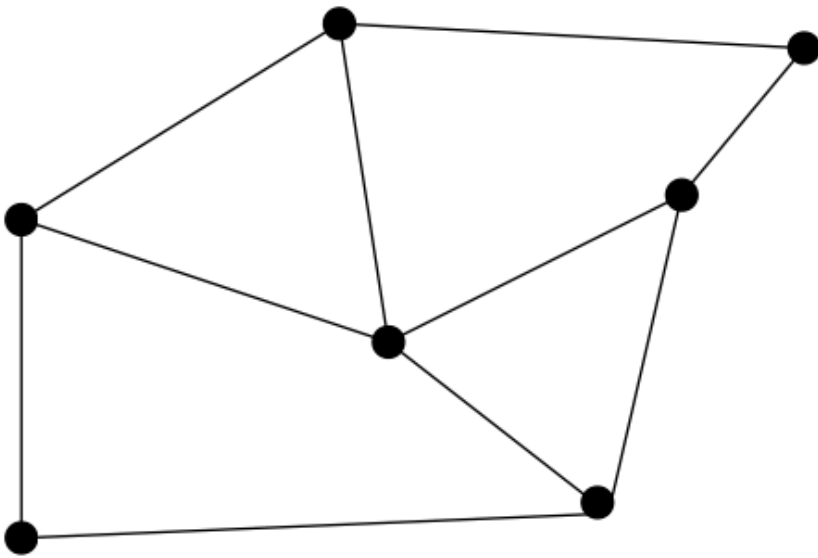
embedded



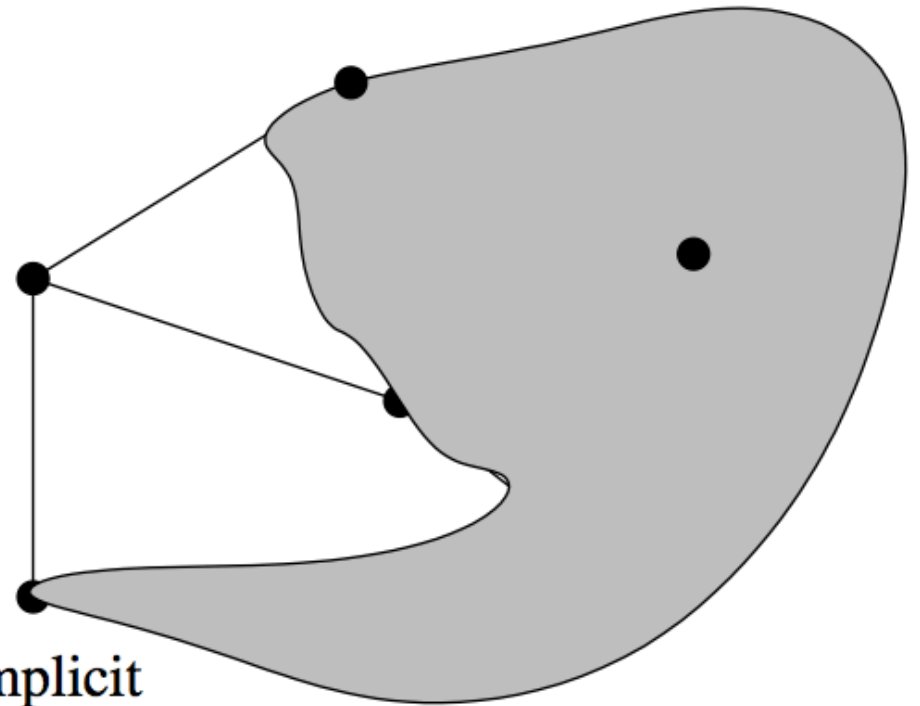
topological

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

Graphs



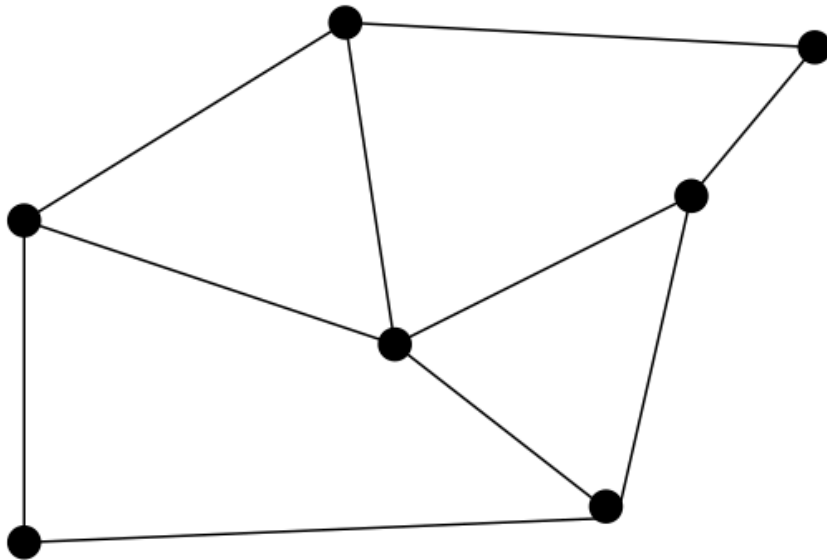
explicit



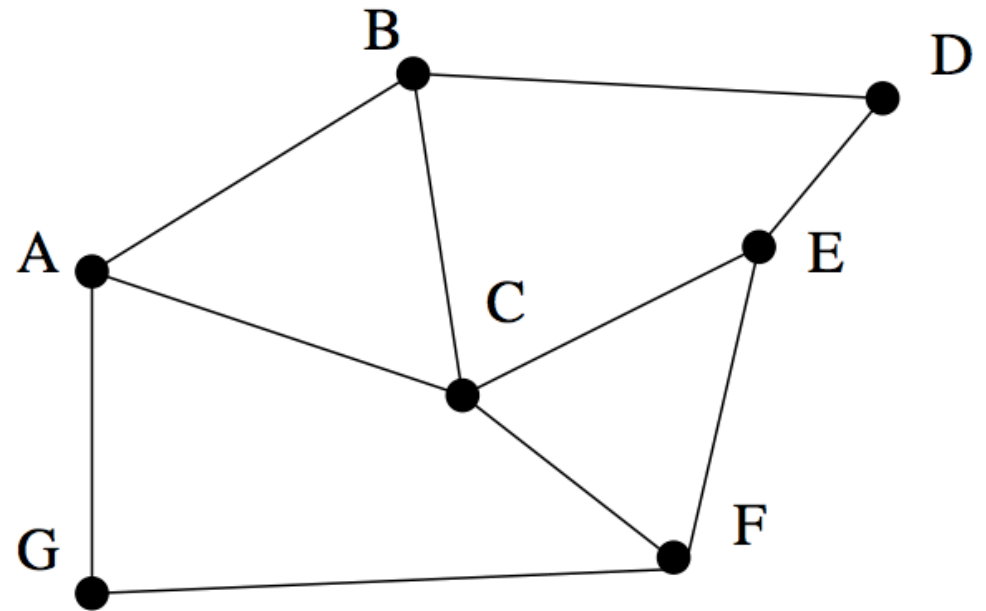
implicit

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

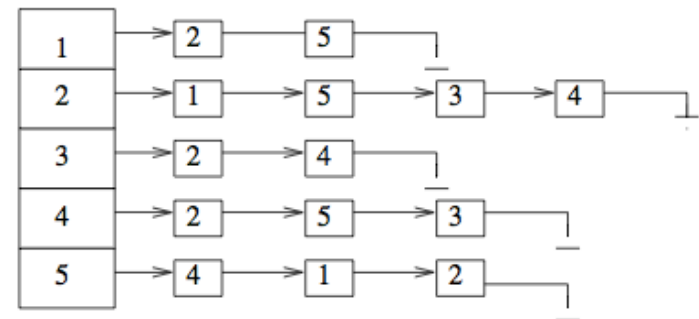
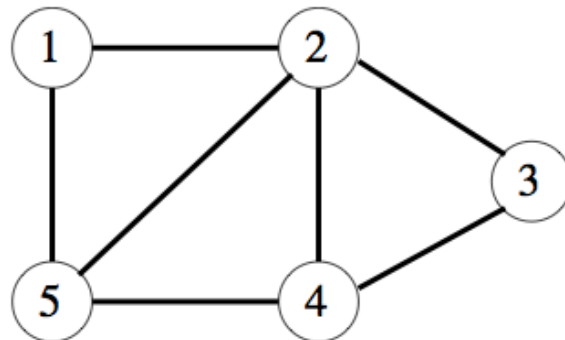
Each vertex is assigned a unique name in a **labelled graph** to distinguish it from other vertices. In unlabelled graphs, no such distinctions are made.

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 $M[i, j] = w$, 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

graph

.num Vertices

.vertices

[0]	"Atlanta"	"
[1]	"Austin"	"
[2]	"Chicago"	"
[3]	"Dallas"	"
[4]	"Denver"	"
[5]	"Houston"	"
[6]	"Washington"	"
[7]		
[8]		
[9]		

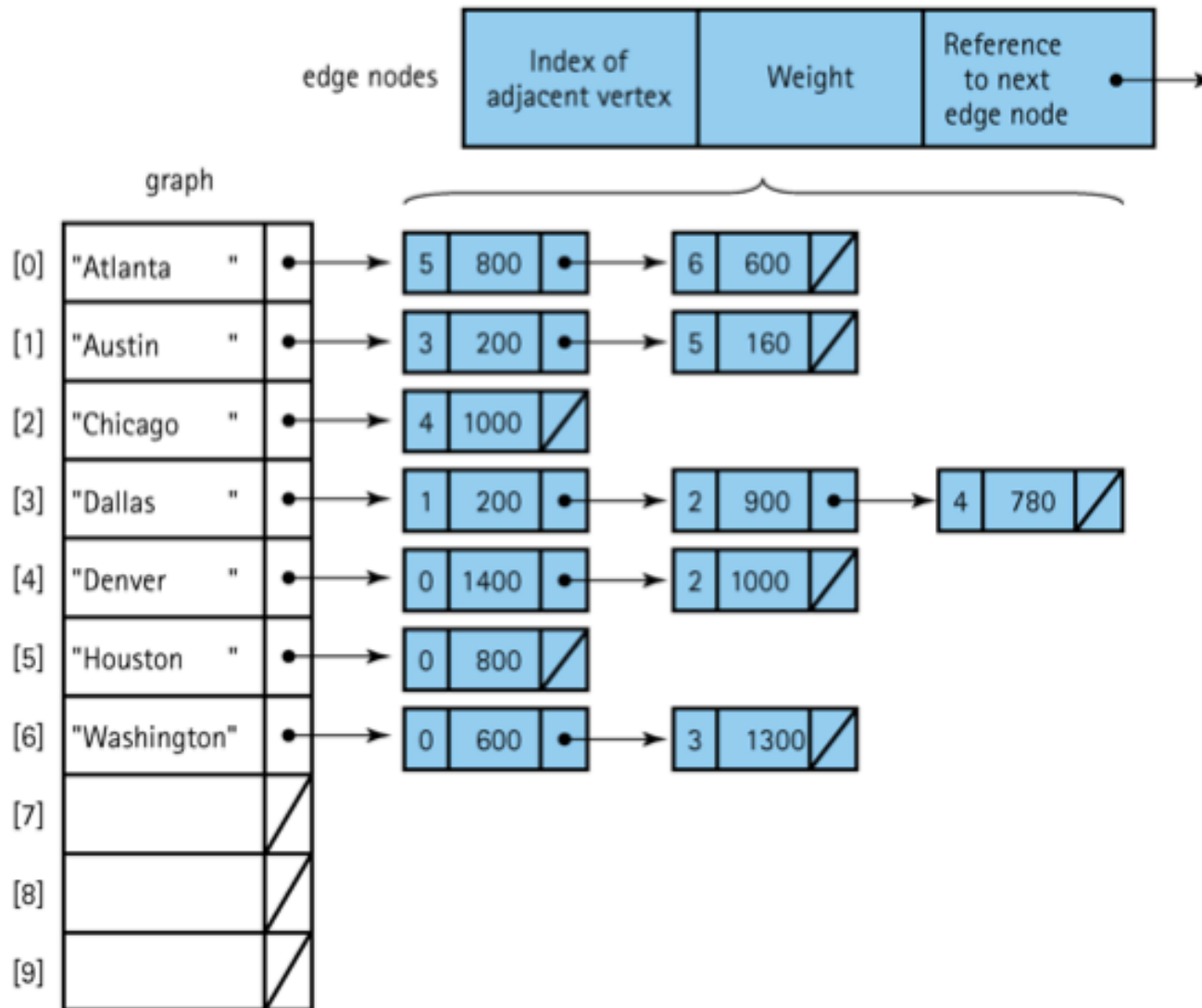
.edges

[0]	0	0	0	0	0	800	600	•	•	•
[1]	0	0	0	200	0	160	0	•	•	•
[2]	0	0	0	0	1000	0	0	•	•	•
[3]	0	200	900	0	780	0	0	•	•	•
[4]	1400	0	1000	0	0	0	0	•	•	•
[5]	800	0	0	0	0	0	0	•	•	•
[6]	600	0	0	1300	0	0	0	•	•	•
[7]	•	•	•	•	•	•	•	•	•	•
[8]	•	•	•	•	•	•	•	•	•	•
[9]	•	•	•	•	•	•	•	•	•	•
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

(Array positions marked '•' are undefined)

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

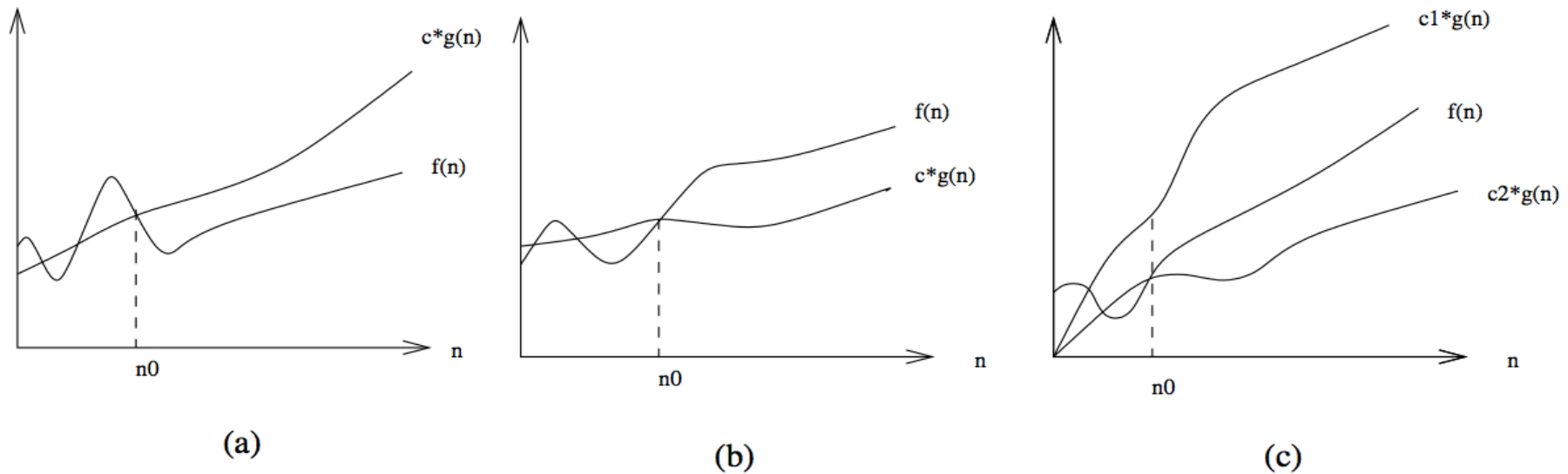
Worst-case and average-case complexity

$f(n) = O(g(n))$ means $c \cdot g(n)$ is an *upper bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\leq c \cdot g(n)$, for large enough n (i.e. , $n \geq n_0$ for some constant n_0).

$f(n) = \Omega(g(n))$ means $c \cdot g(n)$ is a *lower bound* on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\geq c \cdot g(n)$, for all $n \geq n_0$.

$f(n) = \Theta(g(n))$ means $c_1 \cdot g(n)$ is an upper bound on $f(n)$ and $c_2 \cdot g(n)$ is a lower bound on $f(n)$, for all $n \geq n_0$. Thus there exist constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ and $f(n) \geq c_2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.

Worst-case and average-case complexity



Illustrating the big (a) O , (b) Ω , and (c) Θ notations

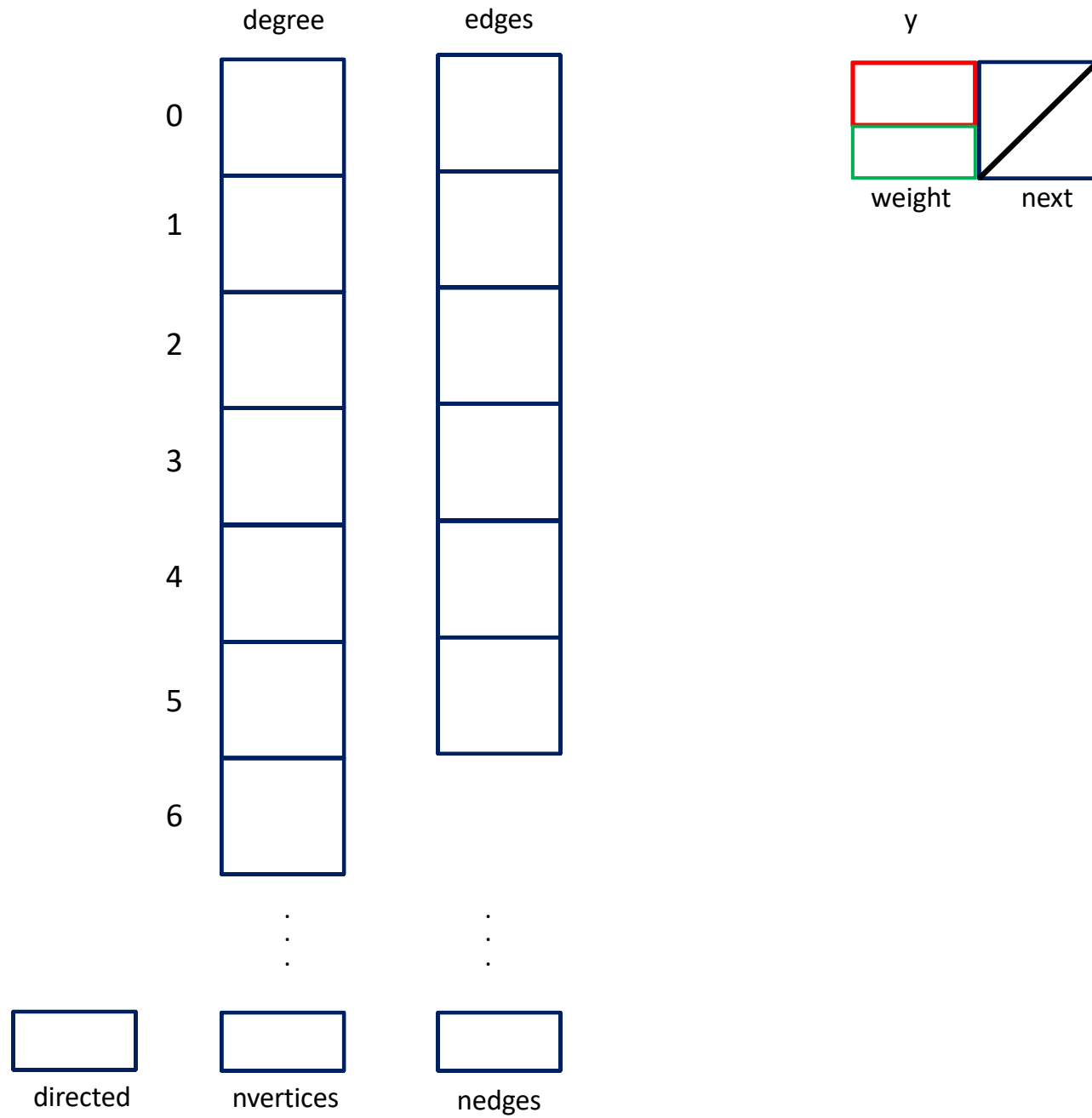
Graphs

```
/* 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 */

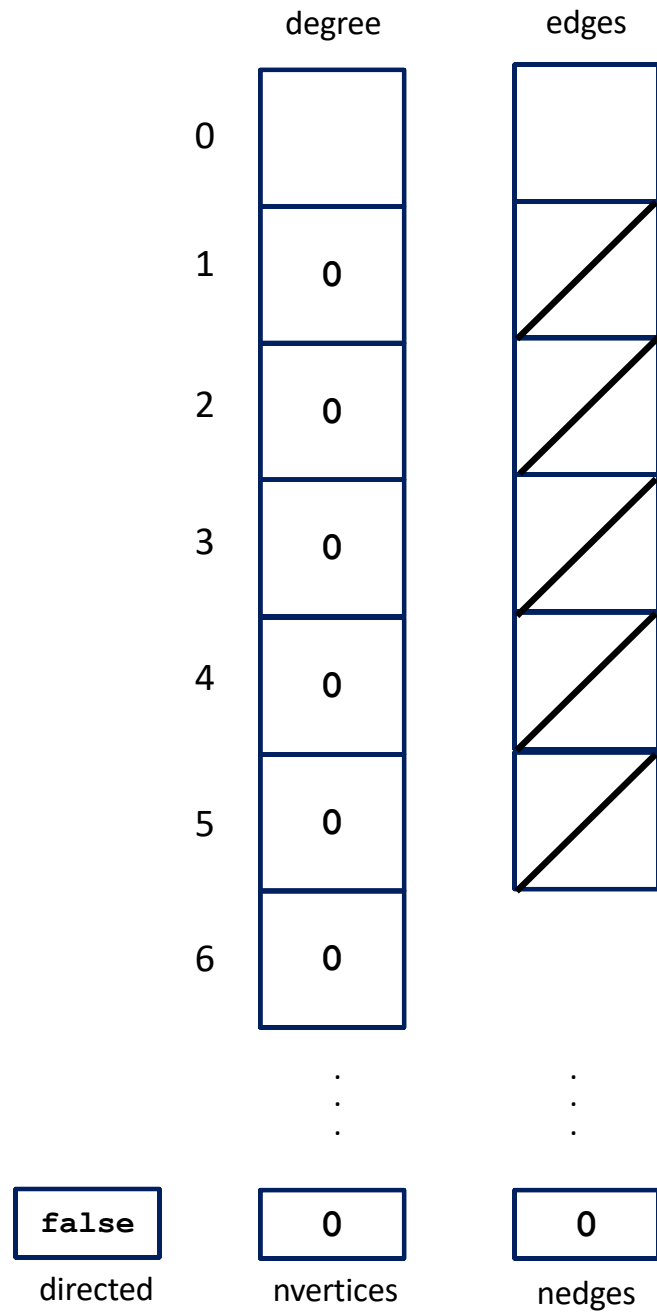
initialize_graph(graph *g, bool directed){

    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

```
/* build graph from data */

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 */

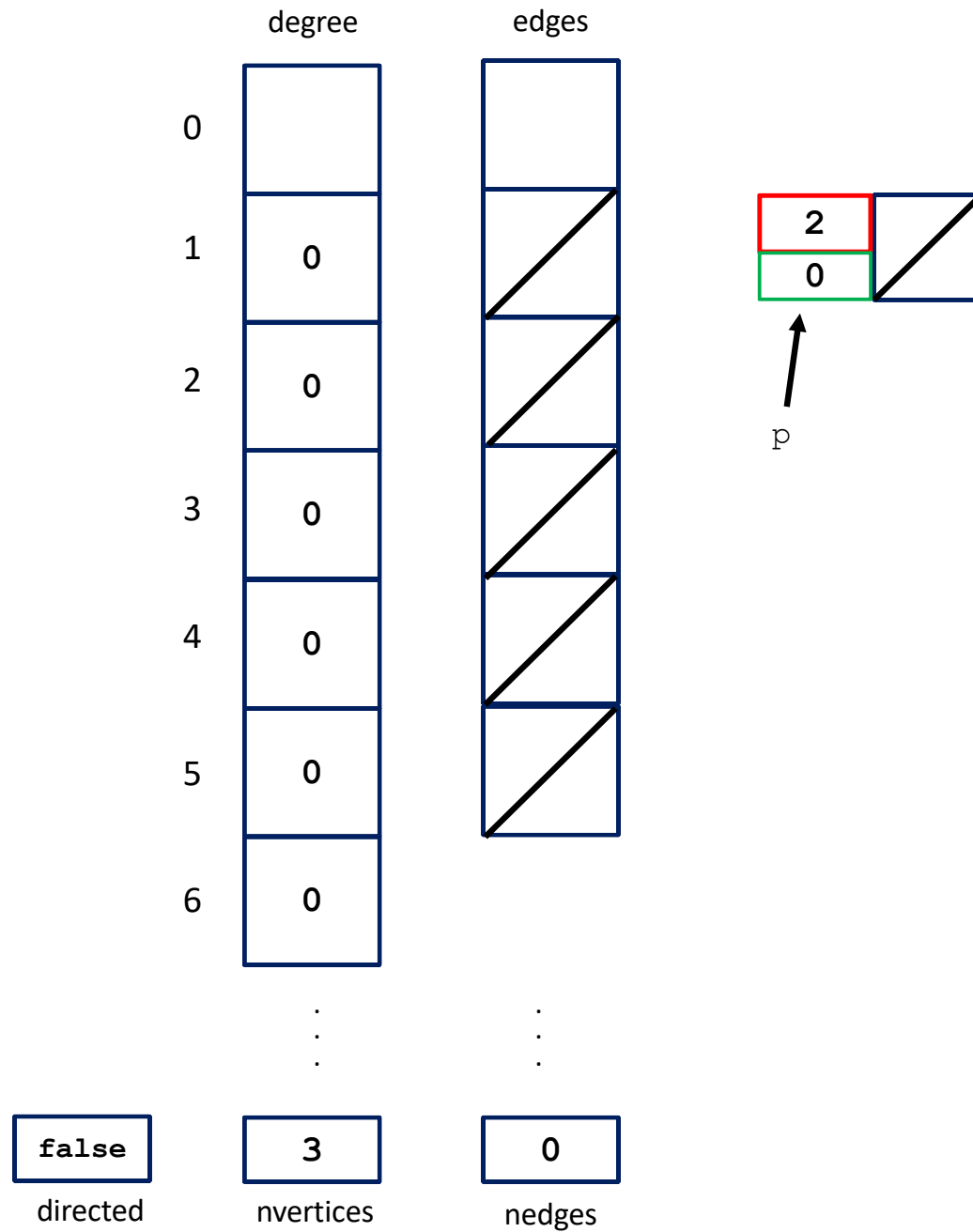
    p->weight = 0;
    p->y = y;
    p->next = g->edges[x];

    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 */
}
```

`insert_edge(g, 1, 2, false)`



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 */

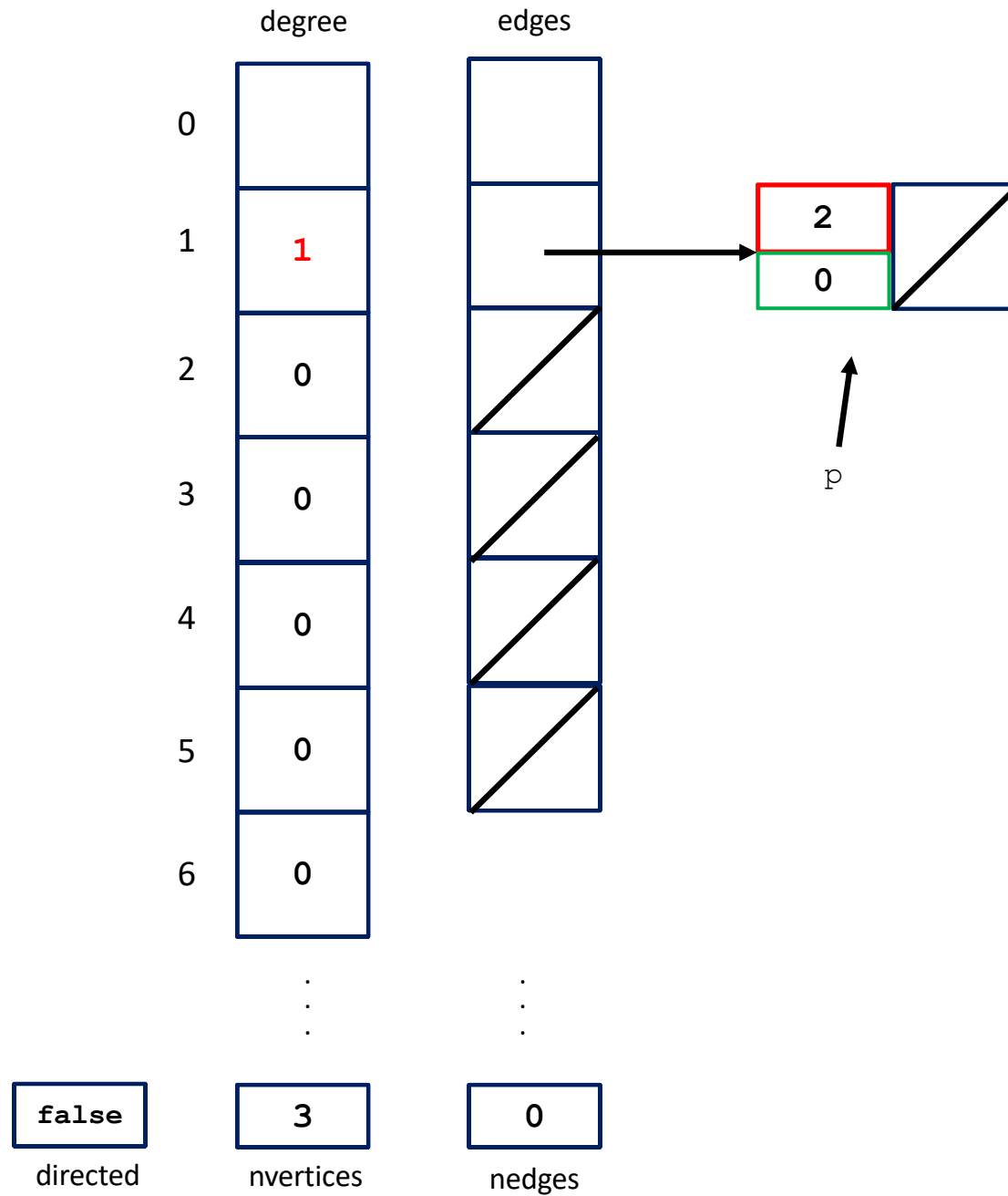
    p->weight = 0;
    p->y = y;
    p->next = g->edges[x];

    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 */
}
```

`insert_edge(g, 1, 2, false)`



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 */

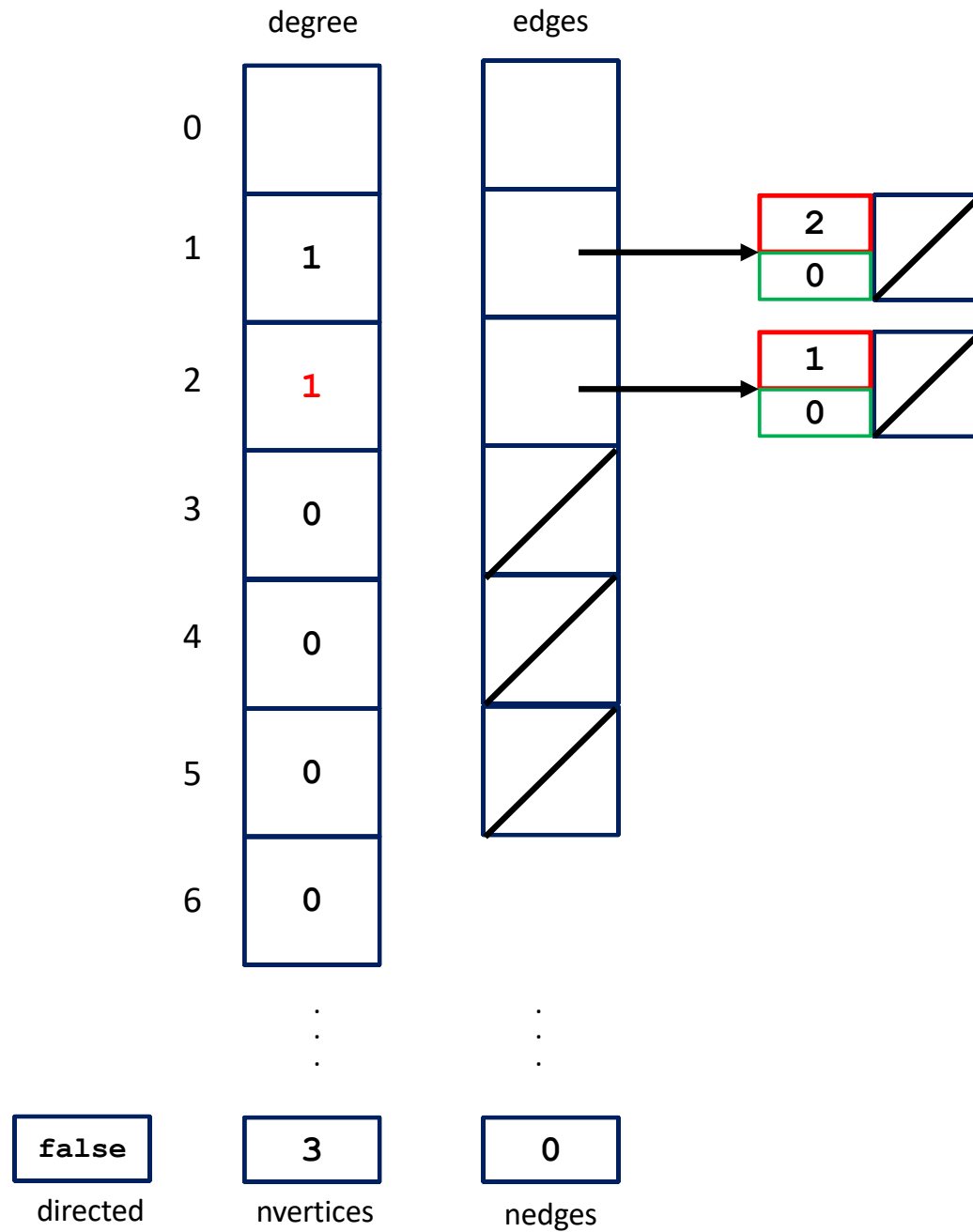
    p->weight = 0;
    p->y = y;
    p->next = g->edges[x];

    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 */
}
```

`insert_edge(g, 2, 1, true)`



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 */

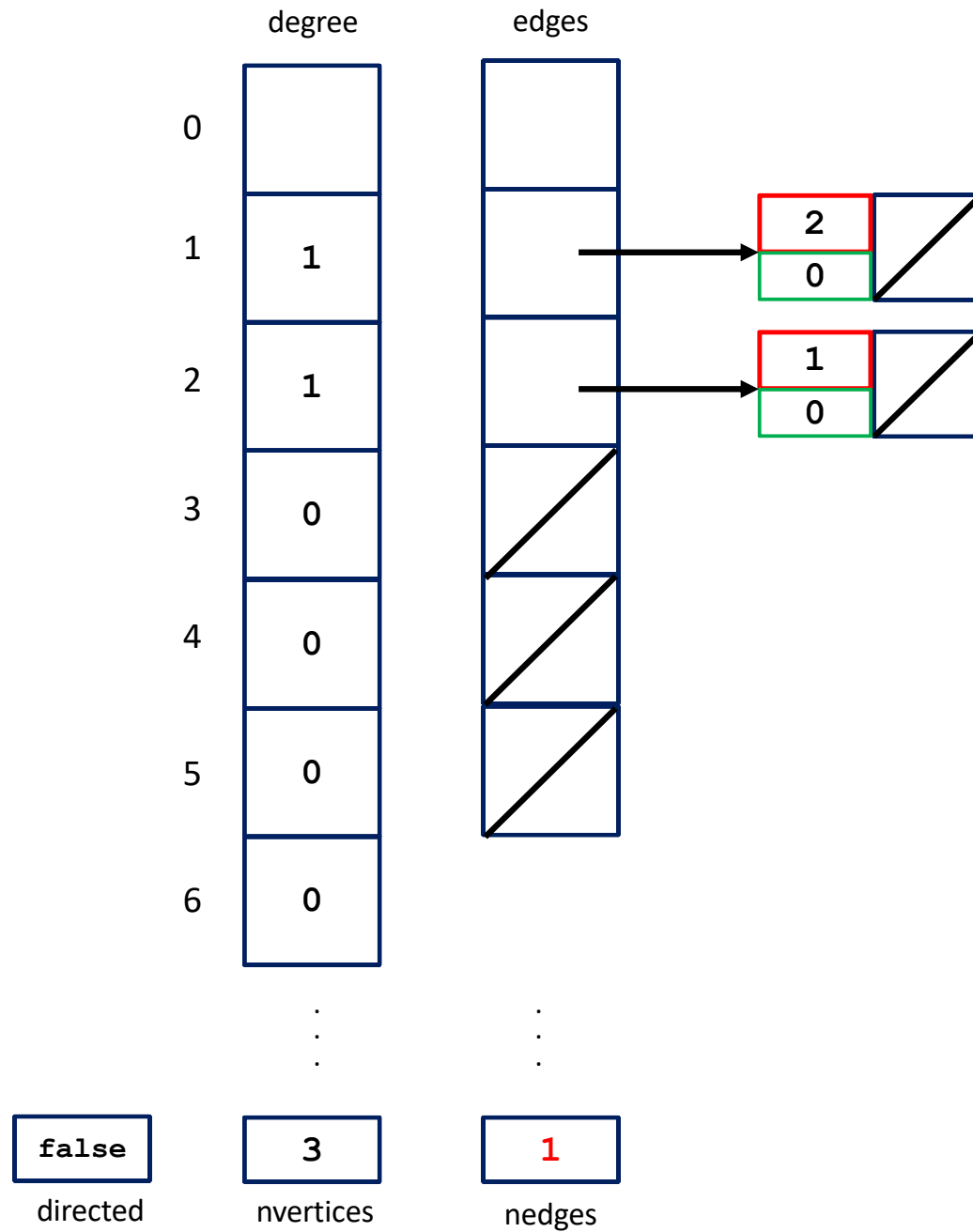
    p->weight = 0;
    p->y = y;
    p->next = g->edges[x];

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

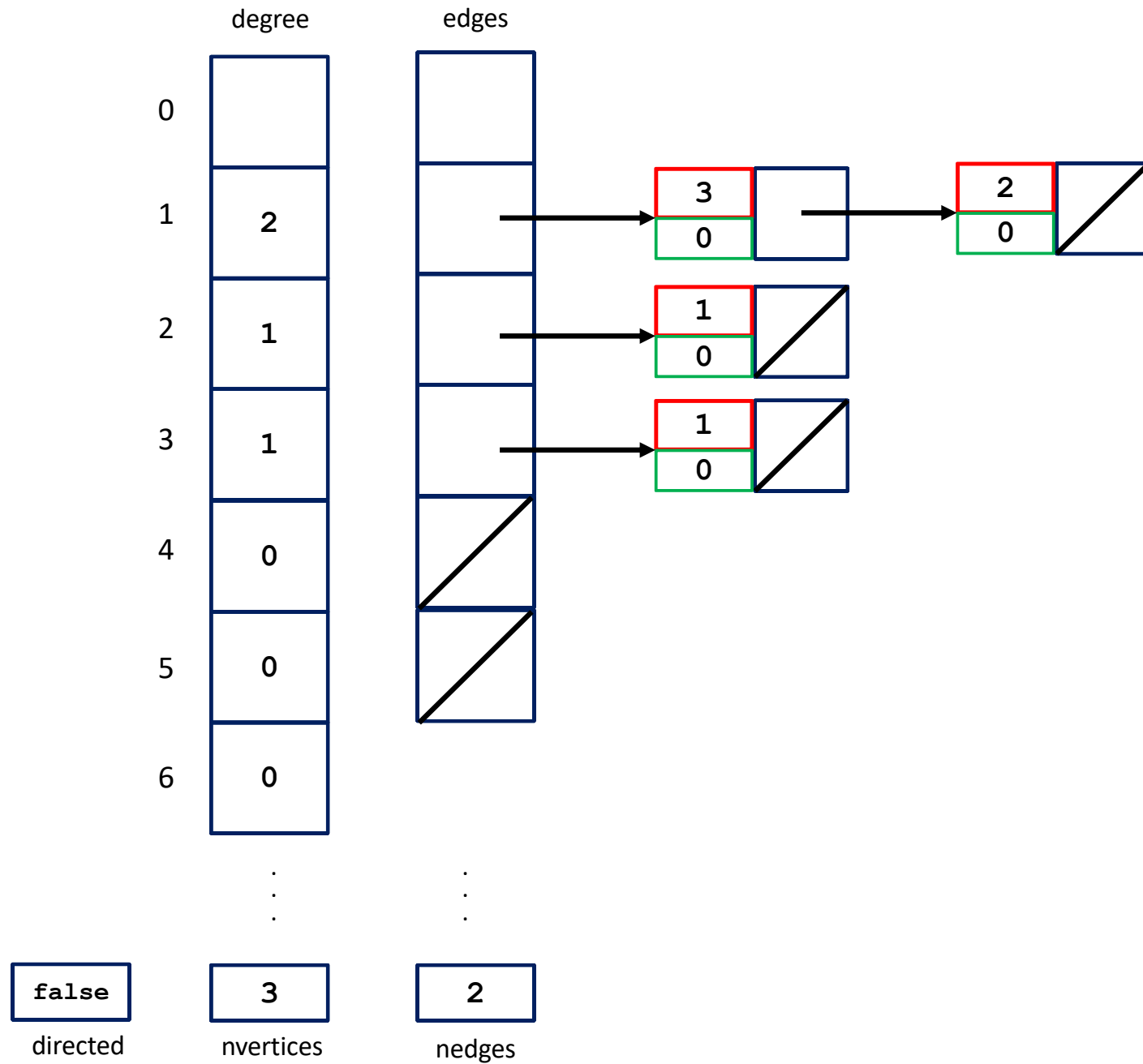
    g->degree[x] ++;

    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 */
}
```

`insert_edge(g, 2, 1, true)`



`insert_edge(g, 1, 3, false)`



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