Advanced algorithms

asymptotic notation, graphs and their representation in computers

Jiří Vyskočil, Radek Mařík

2013

Introduction

- Subject WWW pages:

  https://cw.felk.cvut.cz/doku.php/courses/ae4m33pal/start

- Goals

  Individual implementation of variants of standard (basic and intermediate) problems from several selected IT domains with rich applicability. Algorithmic aspects and effectiveness of practical solutions is emphasized. The seminars are focused mainly on implementation elaboration and preparation, the lectures provide a necessary theoretical foundation.

- Prerequisites

  The course requires **programming skills** in at least one of programming languages C/C++/Java. There are also homework programming tasks. Understanding to basic data structures such as arrays, lists, and files and their usage for data processing is assumed.

Asymptotic notation

- Asymptotic upper bound:

$$f(n) \in O(g(n))$$

- Meaning:

  The value of the function f is on or below the value of the function g (within a constant factor)

- Definition:

$$(\exists c > 0)(\exists n_0)(\forall n > n_0) : \ |f(n)| \leq |c \cdot g(n)|$$

Asymptotic notation

- Asymptotic lower bound :

$$f(n) \in \Omega(g(n))$$

- Meaning:

  The value of the function f is on or above the value of the function g (within a constant factor)

- Definition:

$$(\exists c > 0)(\exists n_0)(\forall n > n_0) : \; |c \cdot g(n)| \leq |f(n)|$$

Asymptotic notation

- Asymptotic tight bound :

$$f(n) \in \Theta(g(n))$$

- Meaning:

The value of the function f is equal to the value of the function g (within a constant factor).

- Definition:

$$(\exists c_1, c_2 > 0)(\exists n_0)(\forall n > n_0): |c_1 \cdot g(n)| < |f(n)| < |c_2 \cdot g(n)|$$

Asymptotic notation

- Example: Consider two-dimensional array MxN of integers. What is asymptotic growth of searching for the maximum number in this array?

- upper:
  - $O((M+N)^2)$ ✓
  - $O(\max(M,N)^2)$ ✓
  - $O(N^2)$ ✗
  - $O(M \cdot N)$ ✓

- lower:
  - $\Omega(1)$ ✓
  - $\Omega(M)$ ✓
  - $\Omega(M \cdot N)$ ✓

- tight:
  - $\Theta(M \cdot N)$

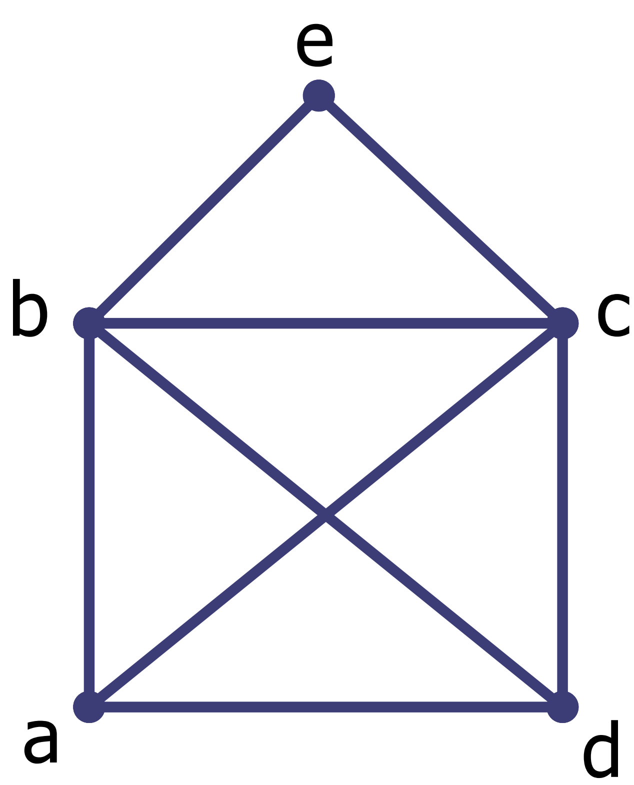
Graphs

- A graph is an ordered pair of a set of vertices (nodes) and a set of edges (arcs)
- $G = (V, E)$

  where V is a set of vertices and
  E is a set of edges

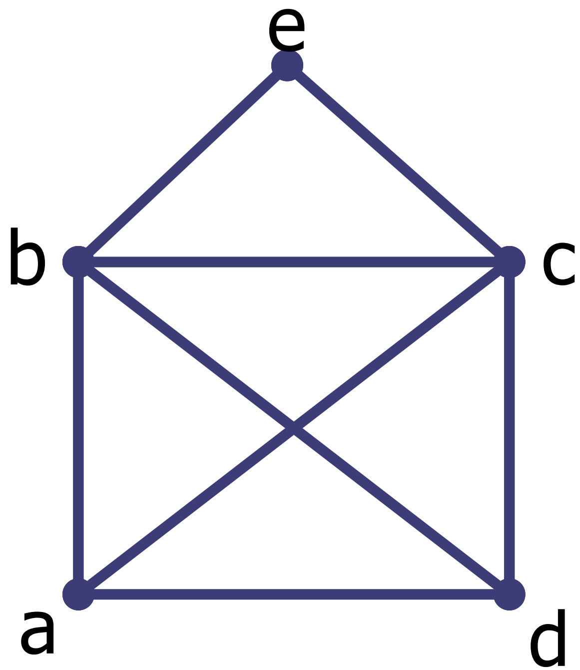
  such as: $E \subseteq \binom{V}{2}$

- Example:
  - V={a,b,c,d,e}
  - E={{a,b},{b,e},{e,c},{c,d}, {d,a},{a,c},{b,d},{b,c}}



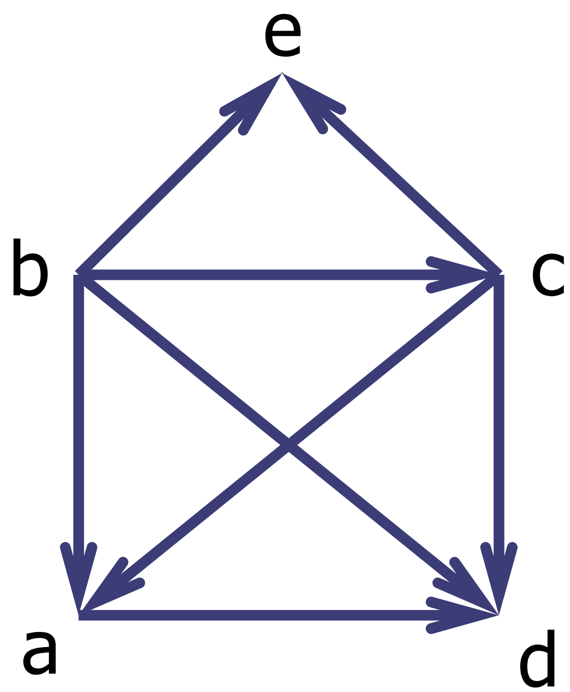
Graphs - orientation

- Undirected graph
  - Edge is **not ordered** pair of vertices
  - E={{a,b},{b,e},{e,c},{c,d}, {d,a},{a,c},{b,d},{b,c}}



- Directed graph (digraph)
  - Edge is an **ordered** pair of vertices

E={(b,a),(b,e),(c,e),(c,d), (a,d),(c,a),(b,d),(b,c)}



Graphs – weighted graph

- Weighted graph
  - A number (weight) is assigned to each edge
  - Often, the weight is formalized using a weight function: $w: E \to \mathbb{R}$

w({a,b}) = 1.1

w({b,e}) = 2.0

w({e,c}) = 0.3

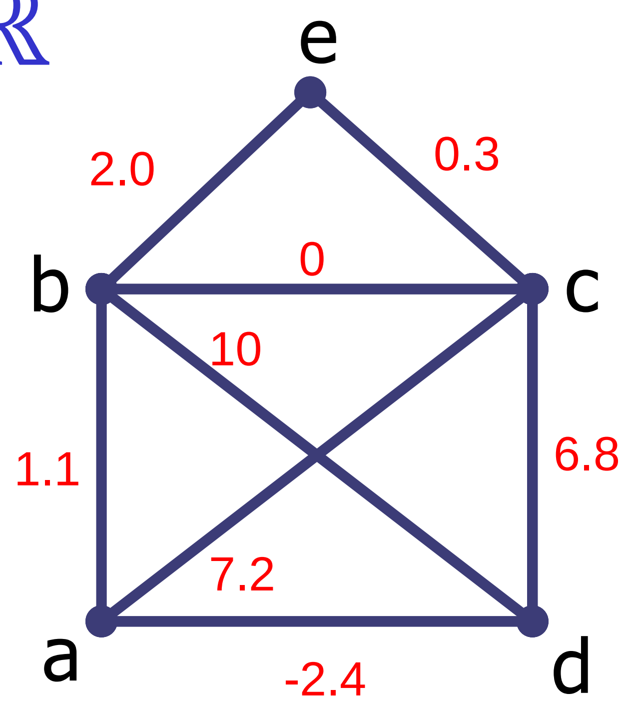
w({c,d}) = 6.8

w({d,a}) = -2.4

w({a,c})= 7.2

w({b,d})= 10

w({b,c})= 0

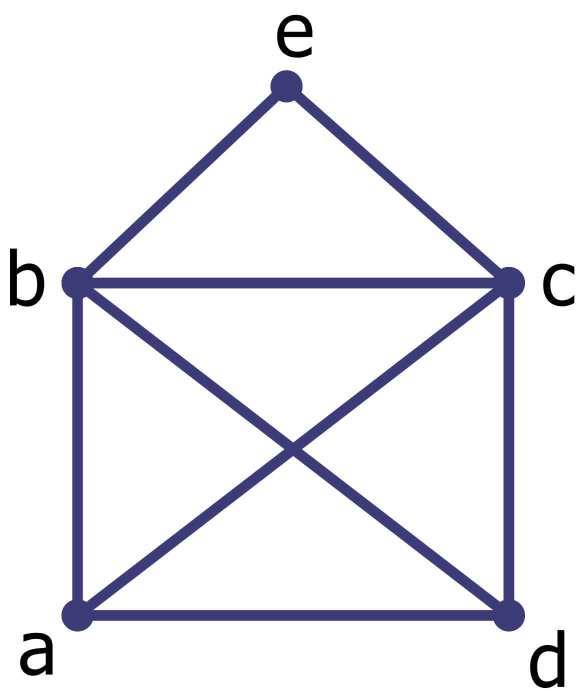


Graphs – node degree

- incidence
  - If two nodes *x*,*y* are linked by edge *e*, nodes *x*,*y* are said to be incident to edge *e* or, edge *e* is incident to nodes *x*,*y*.
- Node degree (for undirected graph)
  - A function that returns a number of edges incident to a given node.

$$\deg(u) = |\{e \in E | u \in e\}|$$

deg(a)=3
deg(b)=4
deg(c)=4
deg(d)=3
deg(e)=2



Graphs – node degree

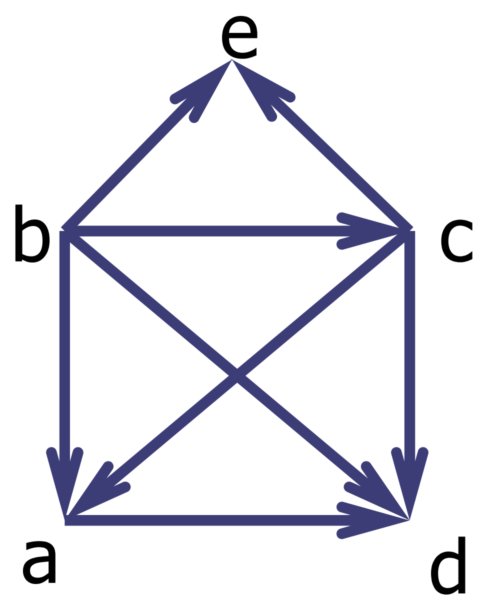
- Node degree (for directed graphs)
  - indegree

$$deg^{+}(u) = |\{e \in E \mid (\exists v \in V) : e = (v,u)\}|$$

  - outdegree

$$deg^{-}(u) = |\{e \in E \mid (\exists v \in V) : e = (u,v)\}|$$

$deg^{+}(a)=2$	$deg^{-}(a)=1$
$deg^{+}(b)=0$	$deg^{-}(b)=4$
$deg^{+}(c)=1$	$deg^{-}(c)=3$
$deg^{+}(d)=3$	$deg^{-}(d)=0$
$deg^{+}(e)=2$	$deg^{-}(e)=0$



Graphs – handshaking lemma

- **Handshaking lemma** (for undirected graphs)

$$\sum_{v \in V} deg(v) = 2|E|$$

- Explanation: Each edges is added twice – once for the source node, then once for target node.
- The variant for directed graphs

$$\sum_{v \in V} (deg^{+}(v) + deg^{-}(v)) = 2|E|$$

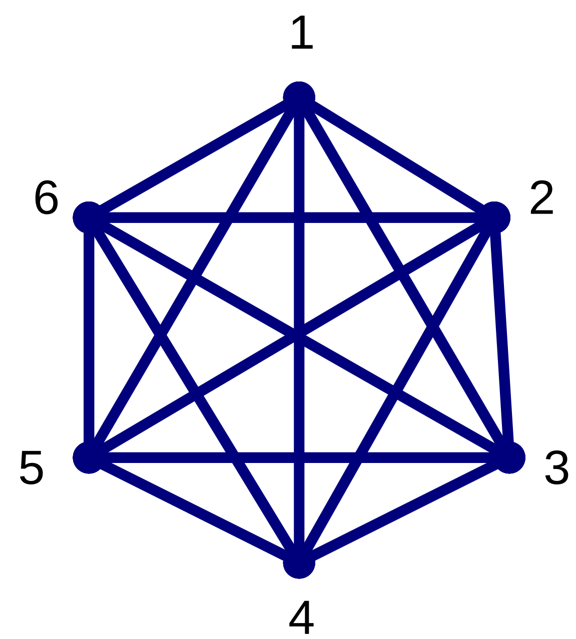
Graphs – complete graph

- complete graph
  - Every two nodes are linked by an edge

$$E = \binom{V}{2}$$

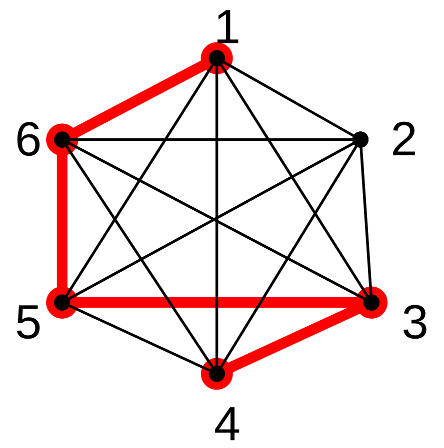
  - A consequence

$$(\forall v \in V) : \deg(v) = |V| - 1$$



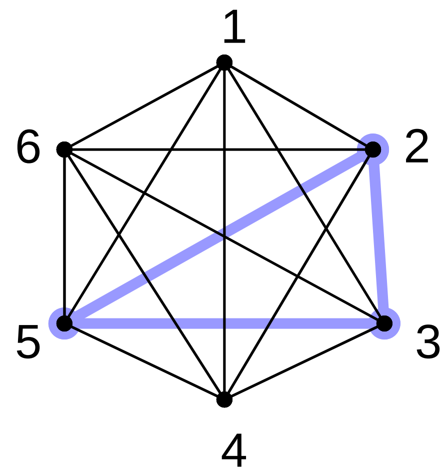
Graphs – path, circuit, cycle

- path
  - A **path** is a sequence of vertices and edges (v_0, e_1, v_1,..., e_t, v_t), where all vertices v_0,..., v_t *differ from each other* and for every i = 1,2,...,t, e_i = {v_{i-1}, v_i} $\in$ E(G). Edges are traversed in forward direction.



(1,{1,6},6,{6,5},5,{5,3},3,{3,4},4)

- circuit
  - A **circuit** is a closed path, i.e. a sequence (v_0, e_1, v_1,..., e_t, v_t = v_0),.

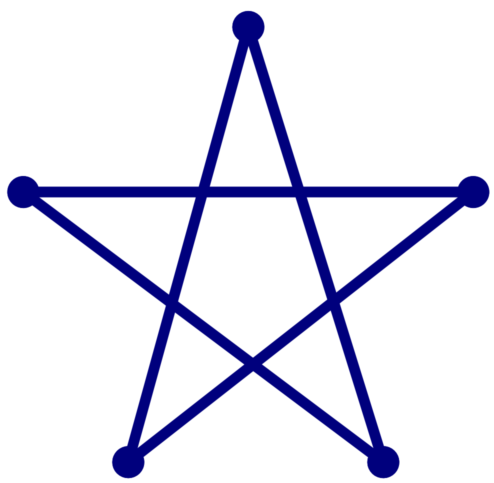


(2,{2,5},5,{5,3},3,{3,2},2)

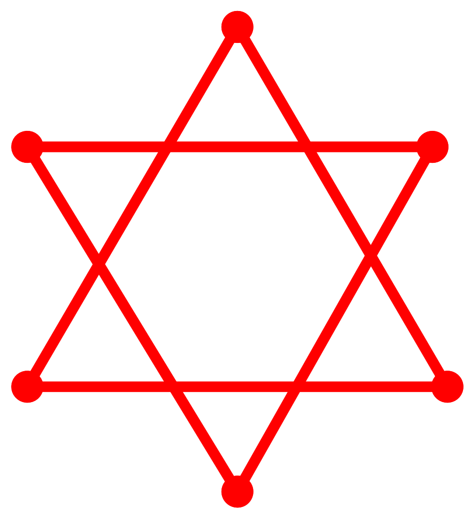
- cycle
  - A **cycle** is a closed simple chain. Edges can be traversed in both directions.

Graphs – connectivity

- connectivity
  - Graph G is **connected** if for every pair of vertices *x* and *y* in G, there is a path from *x* to y.



Connected graph



Disconnected graph

Graphs - trees

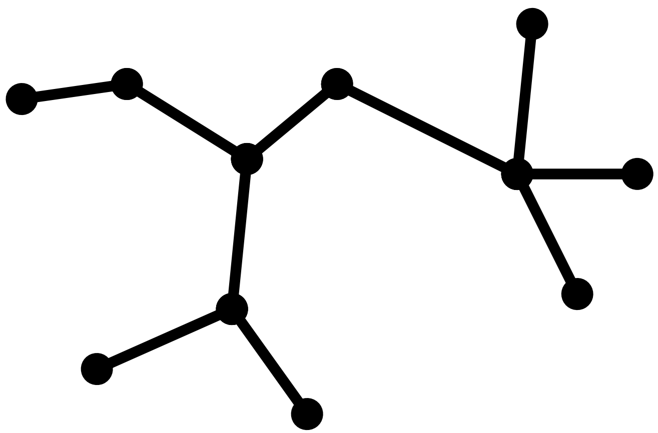
- tree

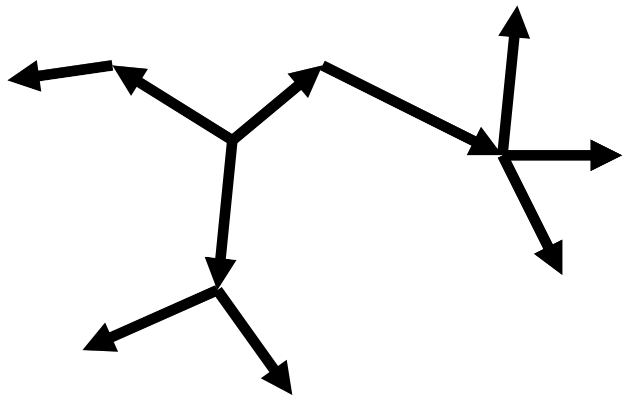
The following definitions of a tree (graph G) are equivalent:

  - G is a connected graph without cycles.
  - G is such a graph so that a cycle occurs if an arbitrary new edges is added.
  - G is such a connected graph so that it becomes disconnected if any edge is removed.
  - G is a connected graph with |V|-1 edges.
  - G is a graph in which every two vertices are connected by just one path.

Graphs - trees

- Undirected trees
  - A **leaf** is a node of degree 1.
- Directed trees (the orientation might be opposite sometimes!)
  - A **leaf** is a node with no outgoing edge.
  - A **root** is a node with no incoming edge.





Graphs – adjacency matrix

- Adjacency matrix
  - Let G=(V,E) be a graph with n vertices.

    Let's label vertices $v_1, \ldots, v_n$ (in some order).

    Adjacency matrix of graph G is a square matrix

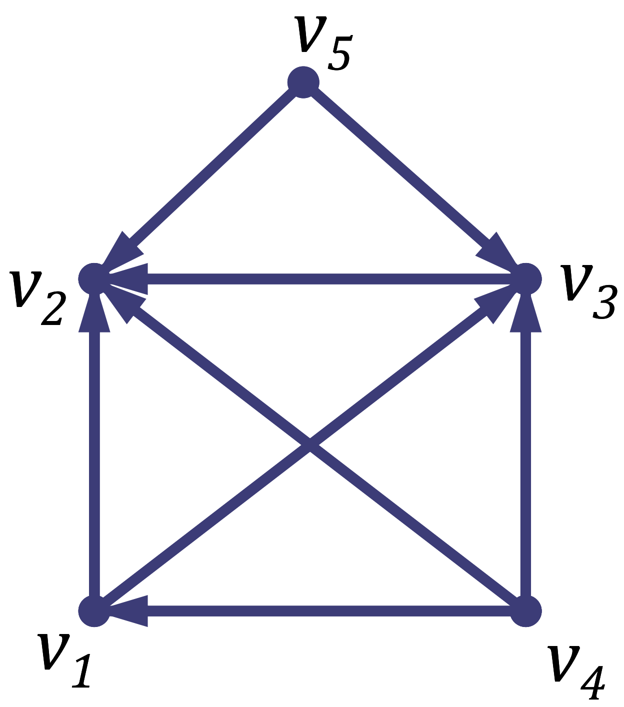
$$A_G = \left(a_{i,j}\right)_{i,j=1}^{n}$$

defined as follows

$$a_{i,j} = \begin{cases} 1 & \text{for } \{v_i, v_j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

Graphs – adjacency matrix (for directed graph)

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



Graphs – Laplacian matrix

- Laplacian matrix
  - Let G=(V,E) be a graph with n vertices

    Let's label vertices $v_1, \ldots, v_n$ (in an arbitrary order).

    Laplacian matrix of graph G is a square matrix

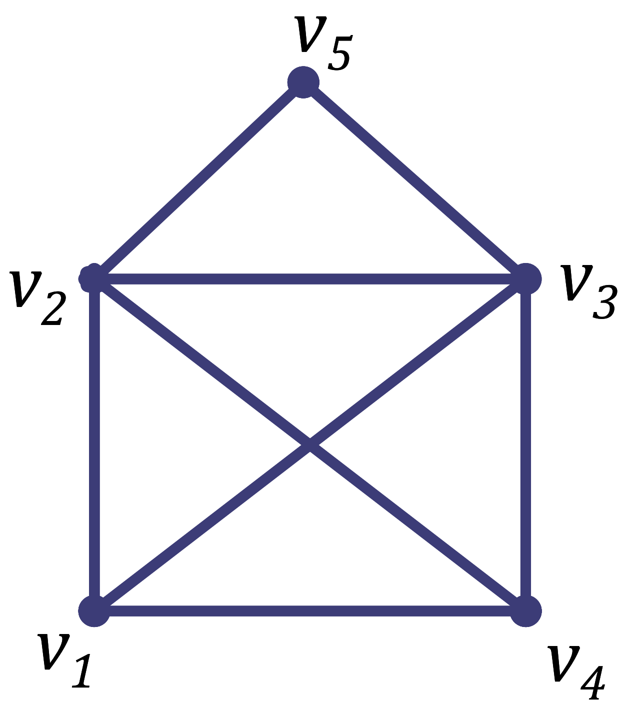
$$L_{\mathrm{G}} = \left(l_{i,j}\right)_{i,j=1}^{n}$$

defined as follows

$$l_{i,j} = \begin{cases} \deg(v_i) & \text{for } i = j \\ -1 & \text{for } \{v_i, v_j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

Graphs – Laplacian matrix

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



Graphs – distance matrix

- Distance matrix
  - Let G=(V,E) is a graph with n vertices and a weight function w.

    Let's label vertices $v_1, \ldots, v_n$ (in an arbitrary order).

    Distance matrix of graph G is a square matrix

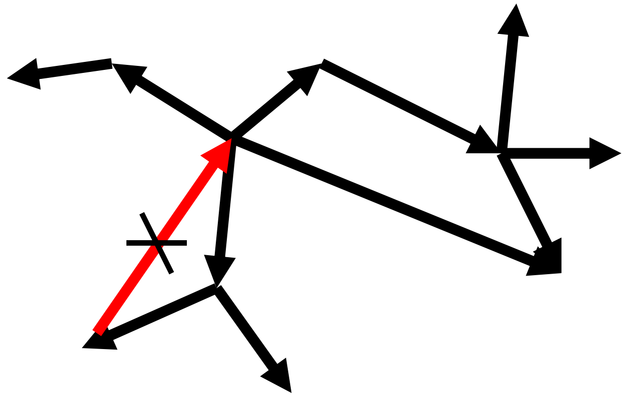
$$A_G = \left(a_{i,j}\right)_{i,j=1}^{n}$$

defined by the formula

$$a_{i,j} = \begin{cases} w(\{v_i, v_j\}) & \text{for } \{v_i, v_j\} \in E \\ 0 & \text{otherwise} \end{cases}$$

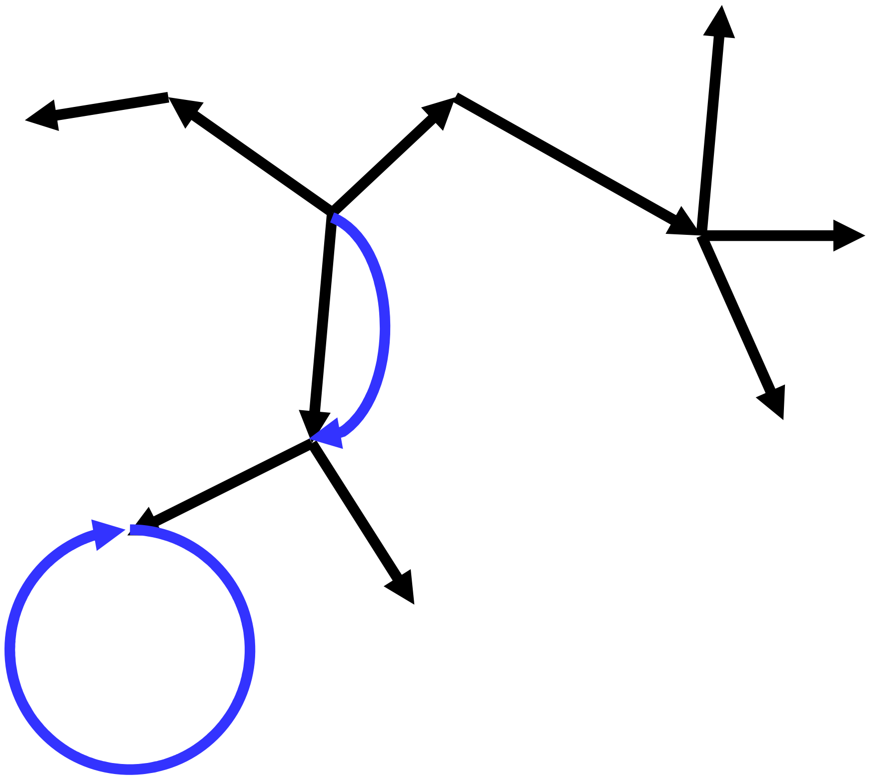
Graphs – DAG

- DAG (Directed Acyclic Graph)
  - DAG is a directed graph without cycles (=acyclic)



Graphs – multigraph

- Multigraph (pseudograph)
  - It is a graph where multiple edges and/or edges incident to a single node are allowed.



Graphs – incidence matrix

- Incidence matrix
  - Let G=(V,E) be a graph where |V|=n and |E|=m.

    Let's label vertices $v_1, \ldots, v_n$ (in some arbitrary order) and edges $e_1, \ldots, e_m$ (in some arbitrary order). **Incidence matrix of graph G** is a matrix of type

$$\{-1, 0, +1\}^{n \times m}$$

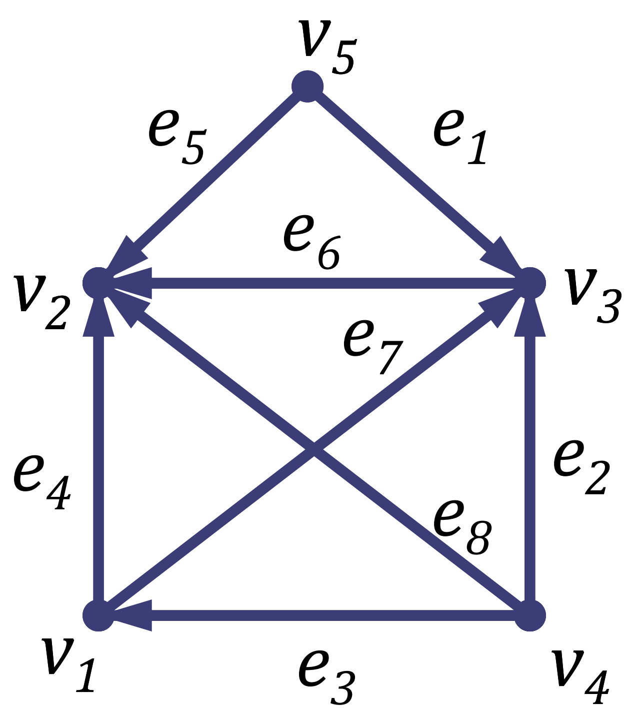
defined by the formula

$$(I)_{i,j} = \begin{cases} -1 & \text{for } e_j = (v_i, *) \\ +1 & \text{for } e_j = (*, v_i) \\ 0 & \text{otherwise} \end{cases}$$

In other words, every edge has -1 at the source vertex and +1 at the target vertex. There is +1 at both vertices for undirected graphs.

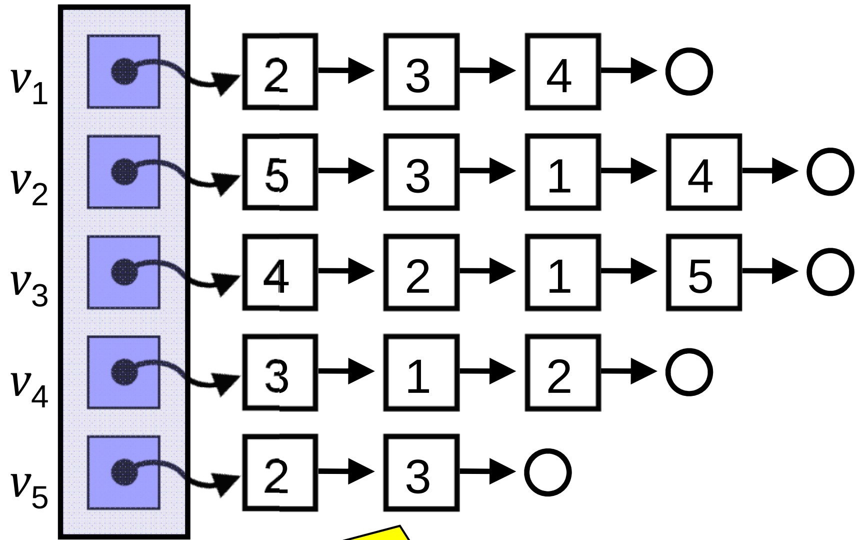
Graphs – incidence matrix

	1	2	3	4	5	6	7	8
1	0	0	1	-1	0	0	-1	0
2	0	0	0	1	1	1	0	1
3	1	1	0	0	0	-1	1	0
4	0	-1	-1	0	0	0	0	-1
5	-1	0	0	0	-1	0	0	0

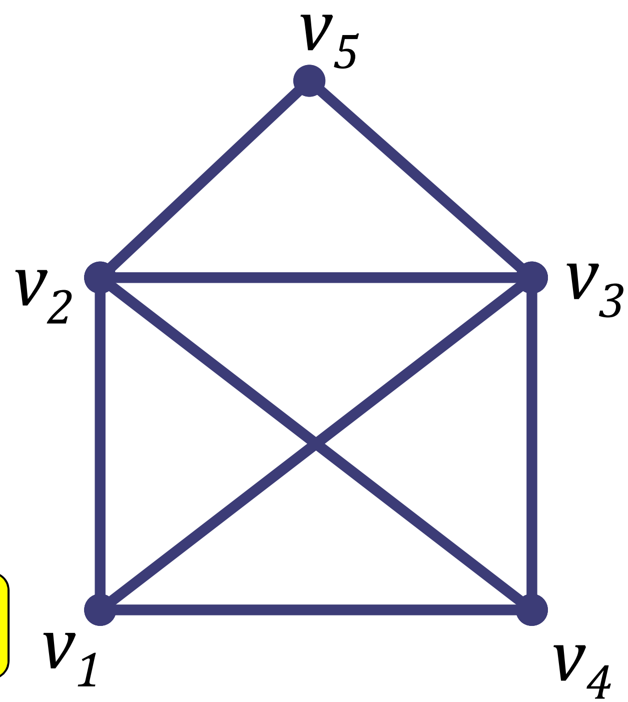


Graphs – adjacency list

- adjacency list (list of neighbours)
  - In an **adjacency list** representation, we keep, for each vertex in the graph, a list of all other vertices which it has an edge to (that vertex's "adjacency list").
  - For instance, the **adjacency list** of graph G could be an array P of pointers of size n, where P[i] points to a linked list of all node indices to which node v_i is linked by an edge (similarly defined for the case of directed graph).



A hash list or a hash table (instead of a linked list) can improve access times to vertices.



Comparison of graph representations

	Adjacency Matrix	Laplacian Matrix	Adjacency List	Incidence Matrix
Storage	$\lvert V\rvert \cdot \lvert V\rvert \in O(\lvert V\rvert^2)$		$O(\lvert V\rvert+\lvert E\rvert)$	$\lvert V\rvert \cdot \lvert E\rvert \in O(\lvert V\rvert \cdot \lvert E\rvert)$
Add vertex	$O(\lvert V\rvert^2)$		$O(\lvert V\rvert)$	$O(\lvert V\rvert \cdot \lvert E\rvert)$
Add edge	$O(1)$			$O(\lvert V\rvert \cdot \lvert E\rvert)$
Remove vertex	$O(\lvert V\rvert^2)$		$O(\lvert E\rvert)$	$O(\lvert V\rvert \cdot \lvert E\rvert)$
Remove edge	$O(1)$		$O(\lvert V\rvert)$	$O(\lvert V\rvert \cdot \lvert E\rvert)$
Check: are u, v adjacent?	$O(1)$		$deg(v) \in O(\lvert V\rvert)$	$O(\lvert E\rvert)$
Process vertex neighbours	$O(\lvert V\rvert)$		$deg(v) \in O(\lvert V\rvert)$	$O(\lvert E\rvert)$
Query: get vertex v degree deg(v)	$O(\lvert V\rvert)$	$O(1)$	$deg(v) \in O(\lvert V\rvert)$	$\lvert E\rvert \in O(\lvert E\rvert)$
Remarks	Slow to add or remove vertices, because matrix must be resized/copied		When removing edges or vertices, need to find all vertices or edges	Slow to add or remove vertices and edges, because matrix must be resized/copied

important

- DFS - Depth First Search

```
procedure dfs(start_vertex : Vertex)
var     to_visit : Stack = empty;
        visited : Vertices = empty;
{
        to_visit.push(start_vertex);
        while (size(to_visit) != 0) {
                v = to_visit.pop();
                if v not in visited then {
                        visited.add(v);
                        for all x in neighbors of v {
                                to_visit.push(x);
                        }
                }
        }
}
```

Graphs - BFS

- BFS - Breadth First Search

```
procedure bfs(start_vertex : Vertex)
var     to_visit : Queue = empty;
        visited : Vertices = empty;
{
        to_visit.push(start_vertex);
        while (size(to_visit) != 0) {
                v = to_visit.pop();
                if v not in visited then {
                        visited.add(v);
                        for all x in neighbors of v {
                                to_visit.push(x);
                        }
                }
        }
}
```

Graphs – priority queue

- priority queue
  - Is a queue with operation **insert to the queue with a priority.**
  - In case the priority is the lowest, the queue behaves as push into a normal queue.
  - In case the priority is the highest, the queue behaves as push into a stack.
  - Both DFS and BFS might be realized using a priority queue with an appropriate value of priority during inserting of elements.

References

- Matoušek, J.; Nešetřil, J. *Kapitoly z diskrétní matematiky*. Karolinum. Praha 2002. ISBN 978-80-246-1411-3.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). *Introduction to Algorithms (2nd ed.)*. MIT Press and McGraw-Hill. ISBN 0-262-53196-8.