# Graph Algorithms

# Graph Theory

## Leonard Euler (1707 – 1783)

“Solutio problematis ad geometriam situs pertinentis”, *Commentarii academiae scientiarum Petropolitanae* 8, 1741, pp. 128-140

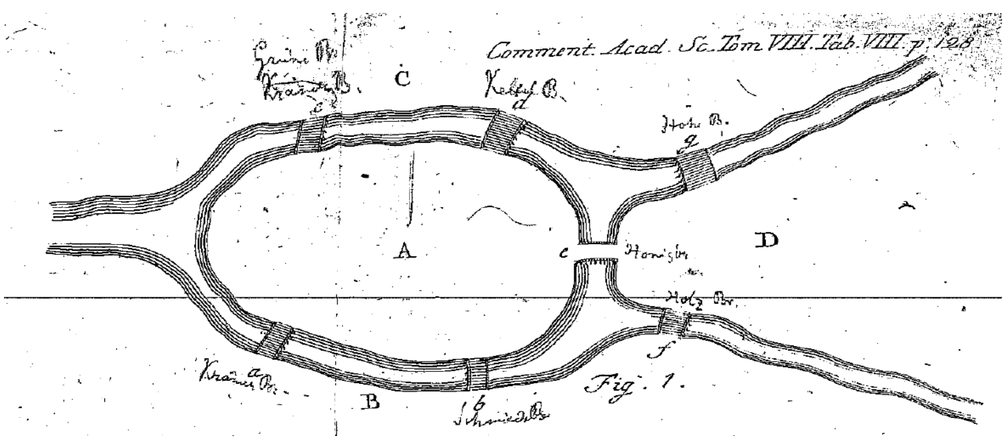
(The solution of a problem relating to the geometry of position)

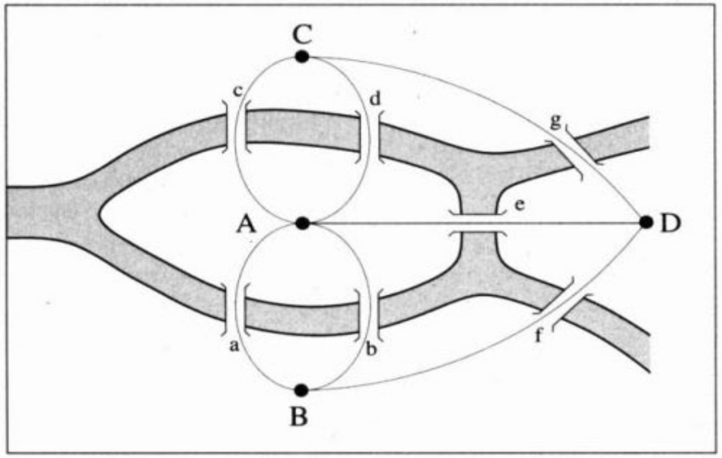
- Seven-bridges of Königsberg problem:

crossing the seven bridges only once and ending where you started

=> no solution

- The first topological result in geometry



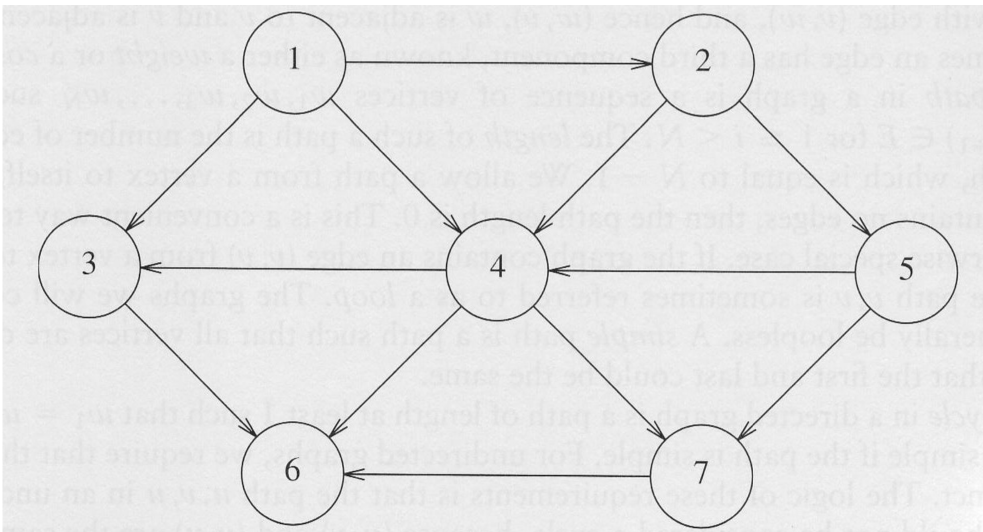
# Graph Algorithms

- A graph $G=(V, E)$ consists of a set of vertices, V, and a set of edges E.
- An edge is a pair (v, w) where v, w are from V.
- If pairs are ordered => directed graph
- w is adjacent to v ⇔(v, w) belongs to E
- An edge can have a component known as cost or weight.
- Path: sequence of vertices $w_1, w_2, \ldots, w_N$, such that $(w_i, w_{i+1})$ in E for $1 \le i < N$
- Path length = number of edges on the path
- Simple path: all vertices are distinct except that the first and the last could be the same.
- A cycle in a directed graph is a path of length $\geq 1$ such that $w_1=w_N$
- Directed Acyclic Graph (DAG) => no cycles
- An undirected graph is connected if there is a path from every vertex to every other vertex.
- A connected directed graph is called strongly connected.
- Complete graph: there is an edge between every pair of vertices.

# Applications of Graphs

- Modeling the airport system:
  - Airports = vertices, two vertices are connected by an edge if there is a direct flight between the two vertices.
  - Weight of an edge = cost of the flight
  - Problem: find the cheapest flight between two airports.
- Modeling the road traffic:
  - Intersections = vertices
  - Streets = edges
  - Weight = speed limit or capacity (# of lanes)
  - Problem: find the shortest route
- Modeling computer networks

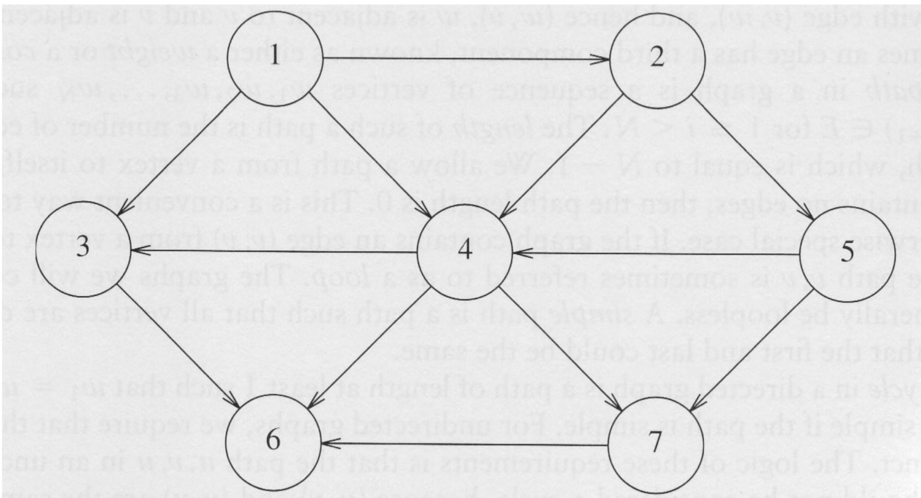
# Representation of Graphs: Adjacency Matrix

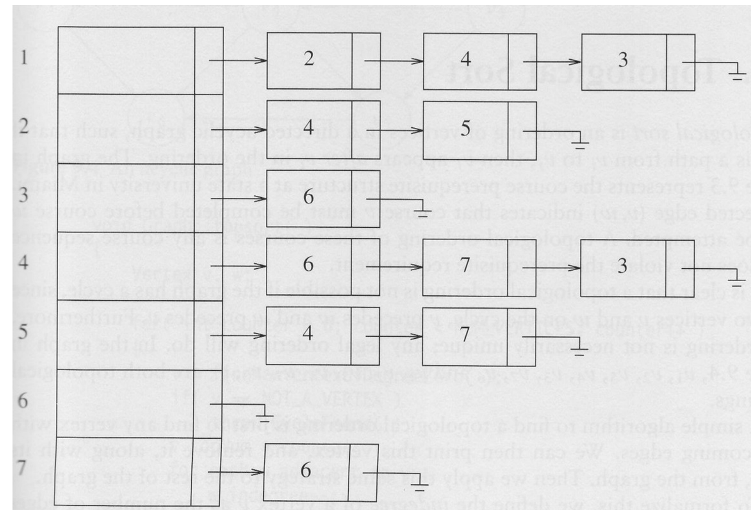



$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

- For each edge (u, v) set a[u][v] = true, otherwise set to false.
- Good for dense graphs i.e. $|E| = \Theta(|V|^2)$ => has many edges
- Space requirement: $O(|V|^2)$ => quadratic
- If weighted graph => store the weight on A[u][v]; -∞ or +∞ if not an edge.
- Lower bound for an algorithm (using adjacency matrix representation) needing to traverse all the edges => $\Omega(|V|^2)$

# Representation of Graphs: Adjacency List




- For each vertex keep a list of adjacent vertices
- Good for sparse graphs i.e. $|E| = O(|V|^2)$ => has few edges
- Space requirement: $O(|V| + |E|)$ => linear
- If weighted graph => store weight in each cell
- Undirected graphs: an edge appears in two lists.
- Lower bound for an algorithm (using adjacency list representation) needing to traverse all the edges => $\Omega(|V|+|E|)$

# Topological Sort

- Topological sort:

  an ordering of vertices in a directed acyclic graph, such that if there is a path from $v_i$ to $v_j$ then $v_j$ appears after $v_i$ in the ordering.

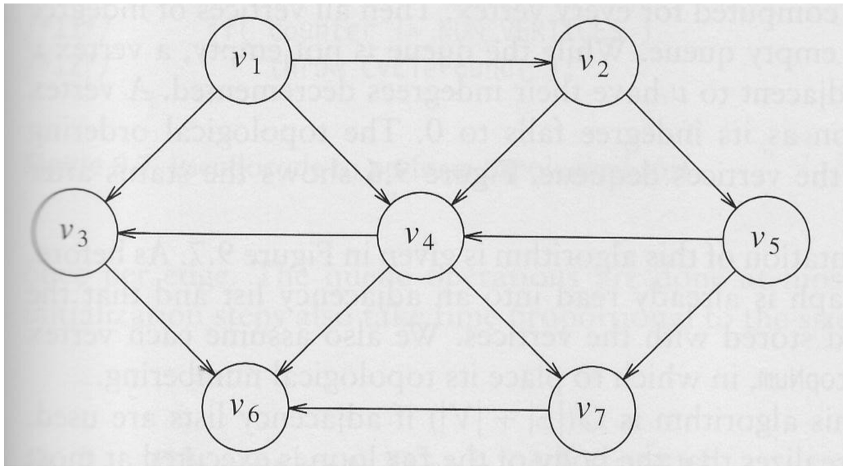
- If the graph has cycles => topological sort is not possible.
- Example:

  courses prerequisite structure

  => a topological ordering of the courses is a valid sequence of courses.

- Indegree of a vertex v = number of edges (u,v)

# Topological Sort




Topological orderings:

$v_1, v_2, v_5, v_4, v_3, v_7, v_6$

$v_1, v_2, v_5, v_4, v_7, v_3, v_6$

Algorithm idea:

- find v with indegree = 0
- print v
- remove v along with its edges

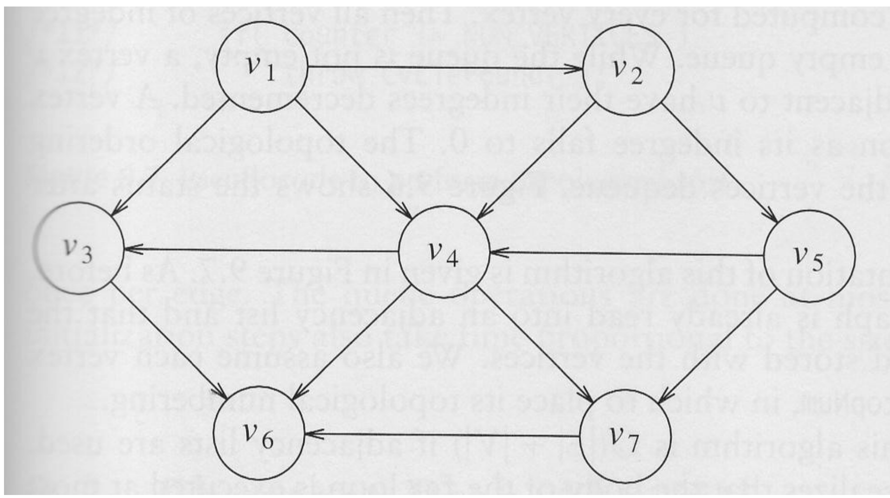
```
void Graph::topsort( )
{
    Vertex v, w;

    for( int counter = 0; counter < NUM_VERTICES; counter++ )
    {
        v = findNewVertexOfDegreeZero( );
        if( v == NOT_A_VERTEX )
            throw CycleFound( );
        v.topNum = counter;
        for each w adjacent to v
            w.indegree--;
    }
}
```

=> $O(|V|^2)$

# Topological Sort

- Better algorithm: avoid scanning the array of vertices of indegree 0.
- Idea: when decrement the indegree of a vertex, place it in a queue if indegree becomes 0.




| | Indegree Before Dequeue # | | | | | | |
|---|---|---|---|---|---|---|---|
| Vertex | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| $v_1$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $v_2$ | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| $v_3$ | 2 | 1 | 1 | 1 | 0 | 0 | 0 |
| $v_4$ | 3 | 2 | 1 | 0 | 0 | 0 | 0 |
| $v_5$ | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| $v_6$ | 3 | 3 | 3 | 3 | 2 | 1 | 0 |
| $v_7$ | 2 | 2 | 2 | 1 | 0 | 0 | 0 |
| *Enqueue* | $v_1$ | $v_2$ | $v_5$ | $v_4$ | $v_3, v_7$ | | $v_6$ |
| *Dequeue* | $v_1$ | $v_2$ | $v_5$ | $v_4$ | $v_3$ | $v_7$ | $v_6$ |

# Topological Sort

```
void Graph::topsort( )
{
        Queue<Vertex> q;
        int counter = 0;

        q.makeEmpty( );
        for each vertex v
          if( v.indegree == 0 )
                q.enqueue( v );
        while( !q.isEmpty( ) )
        {
          Vertex v = q.dequeue( );
          v.topNum = ++counter;

          for each w adjacent to v
                if( --w.indegree == 0 )
                        q.enqueue( w );
        }

        if( counter != NUM_VERTICES )
          throw CycleFound( );
}
```

=> $T = O(|E| + |V|)$

# Shortest Path Algorithms

- Weighted graph:

  each edge $(v_i, v_j)$ has an associated cost $c_{i,j}$ to traverse the edge.

- Cost of path $v_1 v_2 \ldots v_N = c_{1,2} + c_{2,3} + \ldots + c_{N-1,N}$

  => weighted path length

- Unweighted path length = number of edges on the path.
- Single-Source Shortest Path Problem (SS-SP):

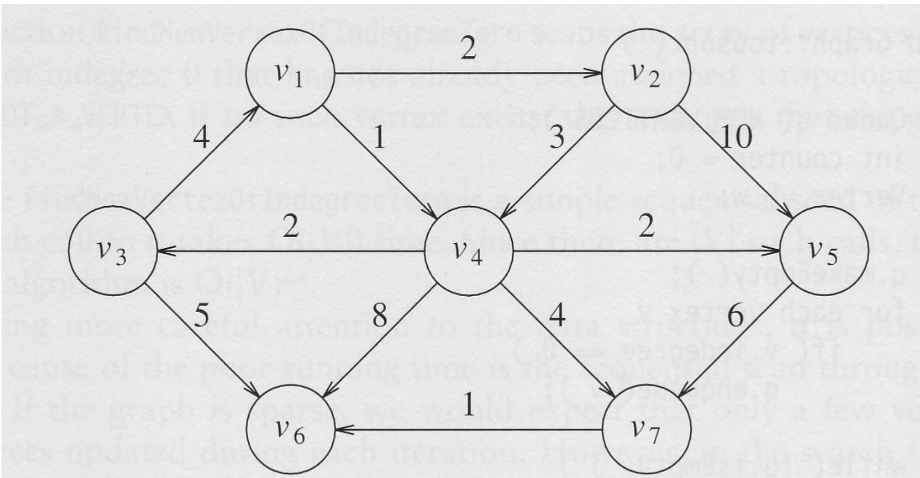
  Given a weighted graph $G = (V, E)$ and a vertex s, find the shortest weighted path from s to every other vertex in G

- If the graph has no negative weight edges

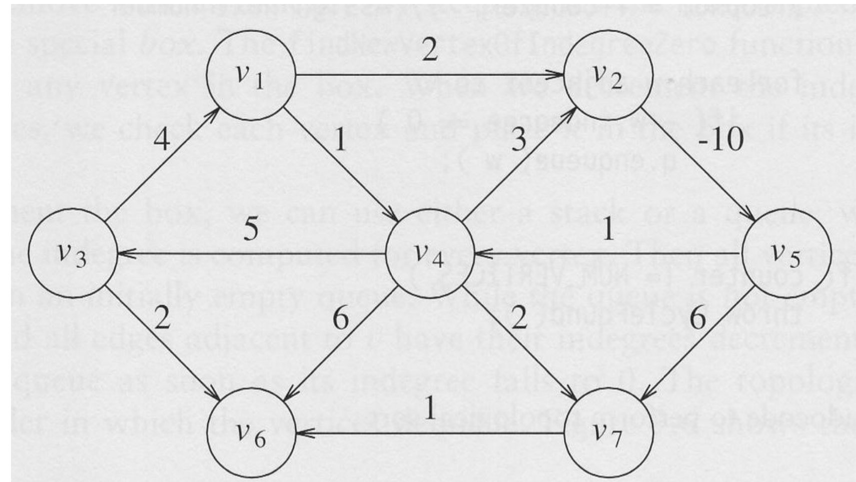
  => $T = O(|E| \log |V|)$

- If the graph has negative weight edges => $T = O(|E||V|)$

# Shortest Path Algorithms




Directed Graph




Graph with a negative-cost cycle

Shortest weighted path from $v_1$ to $v_6$ = $v_1v_4v_7v_6$

=> Cost = 6

Shortest unweighted path from $v_1$ to $v_6$ = $v_1v_4v_6$

=> Cost = 2

Path form $v_5$ to $v_4$ => cost = 1

But a shorter path exists!

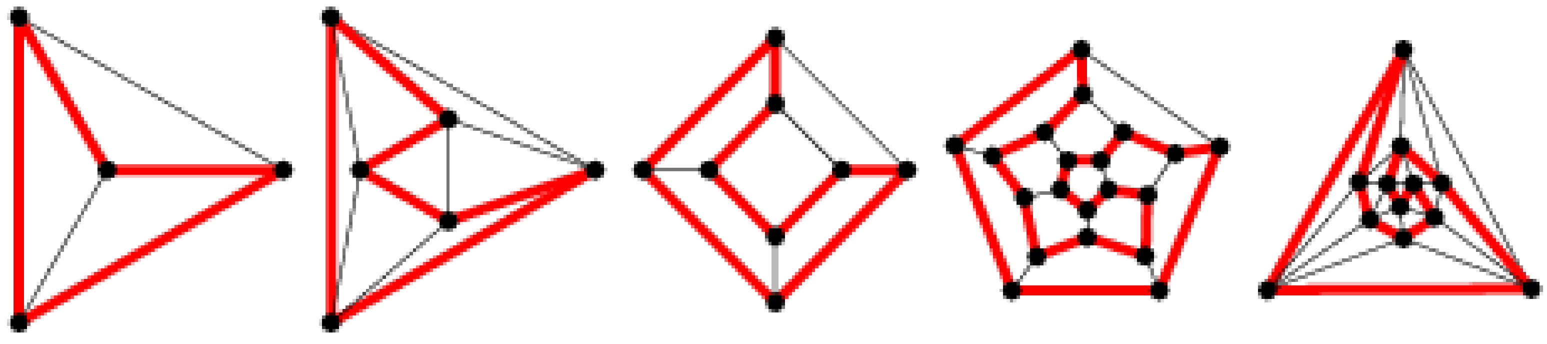
=>The shortest path is not defined when the graph has negative-cost cycles

# Hamiltonian Cycle Problem

- Problem: Determine a cycle in a graph that visits every vertex exactly once.

  Examples: all Platonic solids (*Timaeus [Τίμαιος]* ~350 BC) have Hamiltonian cycles.

Tetrahedron(3) - fire, octahedron(8) - air, cube(6)- earth, dodecahedron(12)- aether, icosahedron(20)-water



# Hamiltonian Cycle Problem

- Idea: Let V be the vertices in G and let n = |V|. Check all possible lists of n vertices without repetitions (all permutations of V ) to see if any of them forms a Hamiltonian cycle.
- Algorithm:

**for** each permutation
 p=(v(i(1)),v(i(2)),...,v(i(n))) of V **do**
  **if** (p forms a Hamiltonian cycle)
   **then return** "yes"
**return** "no“

- What has to be checked in order to see that p forms a Hamiltonian cycle?
- Why does this algorithm run in exponential time?

# Hamiltonian Cycle Problem

- Nobody knows of an algorithm that solves the Hamiltonian Cycle Problem in polynomial time.
- Can we check a solution in polynomial time?
- Suppose someone says that our graph has a Hamiltonian cycle, and provides us with a certificate: a Hamiltonian cycle.
- Can an algorithm verify this answer in polynomial time?
- The answer is yes. How ?
- NP is defined as the class of problems that can be verified in polynomial time, in other words, the problems for which there exists a certificate that can be checked by a polynomial-time algorithm.
- Hamiltonian Cycle is in the class NP.
- The name NP stands for "Nondeterministic Polynomial-time"
- It does not stand for "NonPolynomial" !