Programming in C/C++

- Graphs -

Graphs

Basics and refresher

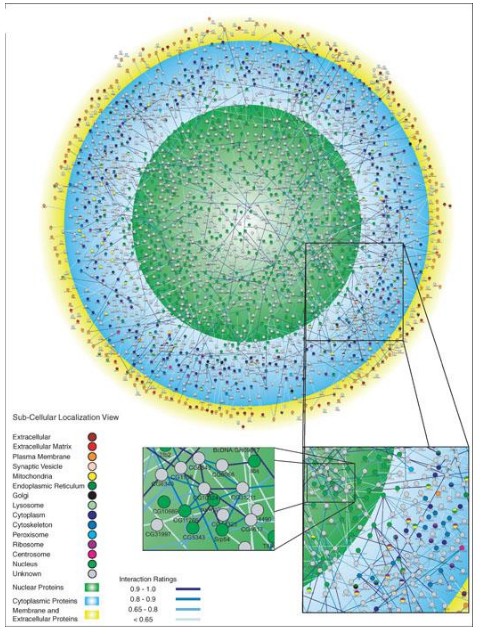
Adjacency Lists

Adjacency Matrices

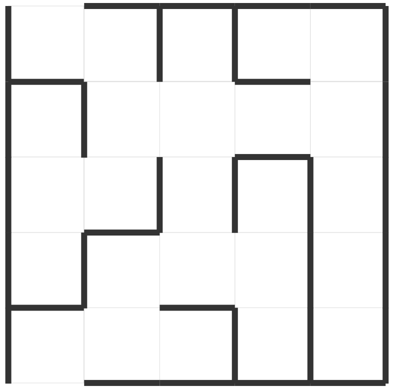
Traversal

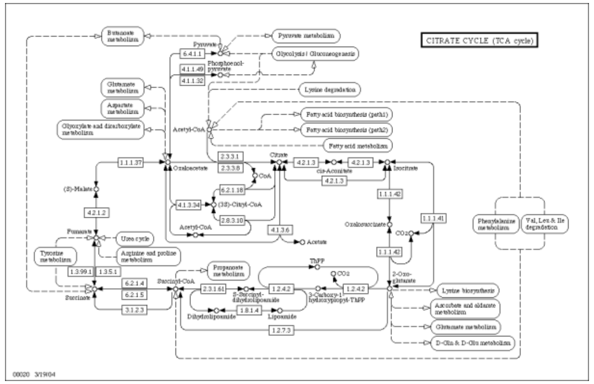
Graphs

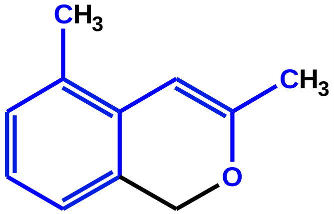
- Powerful structure to represent binary relations (= edges) between objects (= vertices)



- Examples:
  - Street or network maps
  - Social networks
  - Molecules, Biological networks, Pathways
  - Neighborhood relations e.g. adjacent pixel in image …

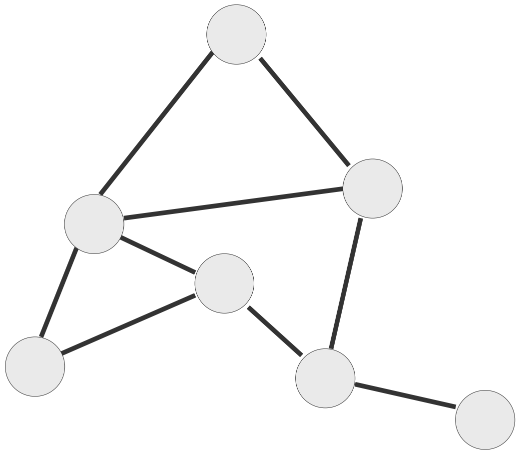




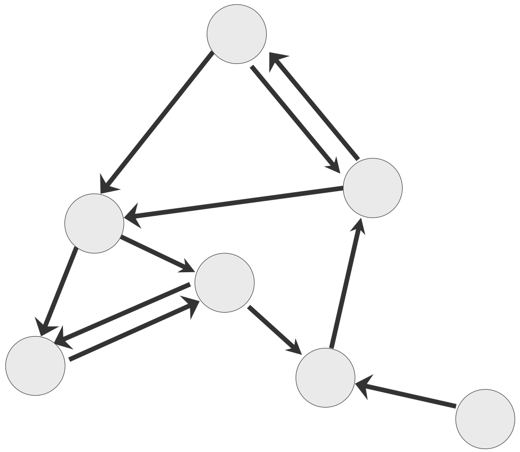


Definition – (Un)directed Graph

An **undirect Graph** is a Pair *G = (V, E)*, where
- *V* is a non-empty set of vertices (=nodes) and
- *E* a set of two-element subsets of *V* called *(edges)*.



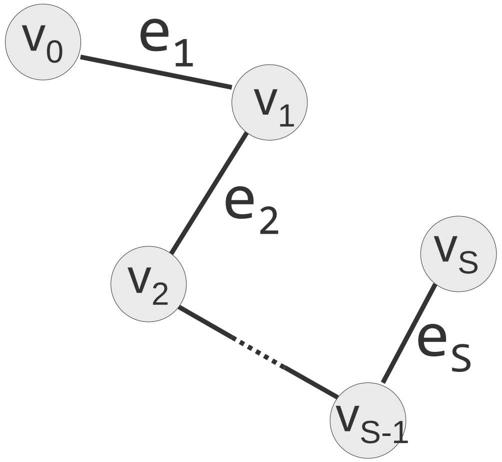
A **direct Graph** is a Pair *G = (V, E)*, where
- *V* is a non-empty set of vertices and
- $E \subseteq V \times V$ a set of ordered pairs.



Graphs – Properties and Terms

- Path: Sequence of edges $e_1, e_2, \ldots, e_S$ that connect nodes $v_0, v_1, v_2, \ldots, v_S$ such that

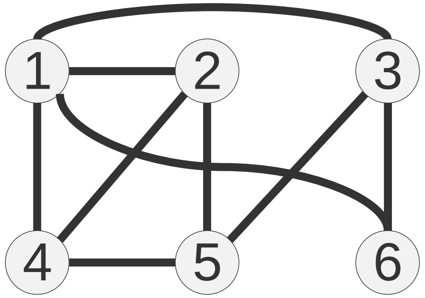
  e_1 connects v_0 and v_1,
  e_2 v_1 v_2,
  …
  e_S v_{S-1} v_S



We say: the path of length S connects v_0 and v_S.

- A **path** is called **simple** if no node appears twice.

- A **cycle** is a path that starts and ends at the same node.

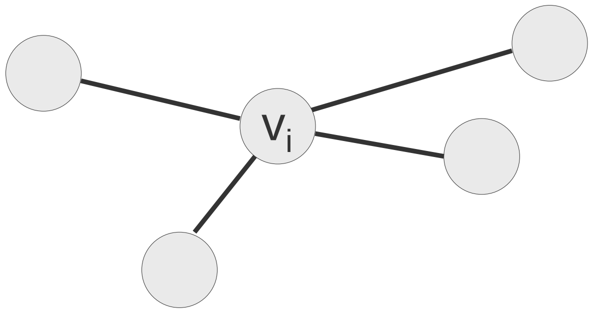


e.g.,: cycle 1,3,6,1 or 1,2,5,4,1

- A graph without cycles is called **acylic**.

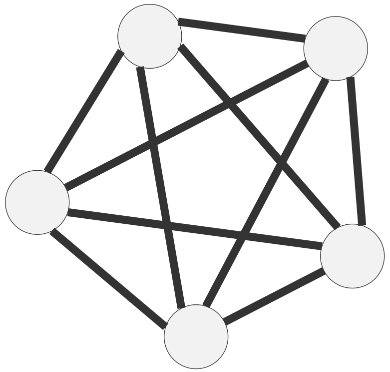
Graphs – Properties and Terms

- Degree of a node: number of edges going in or out of node v_i



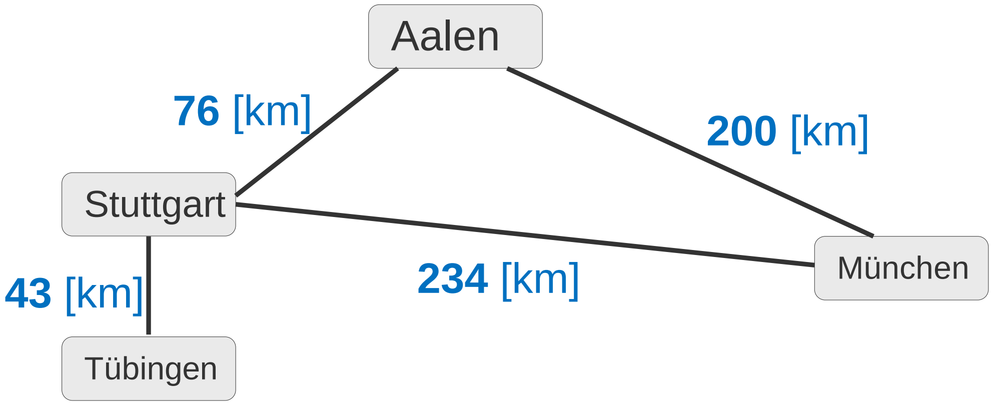
$\deg_G(v_i) = 4$

- Indegree / Outdegree of a node: number of directed edges going in / out of node v_i
- An edge that connects v_i and v_j is called **incident with** v_i **and** v_j**.**
- If v_i is a neighbor of v_j we say v_i is adjacent to v_j
- If each pair of vertices is connected by an edge the graph is complete and fully connected

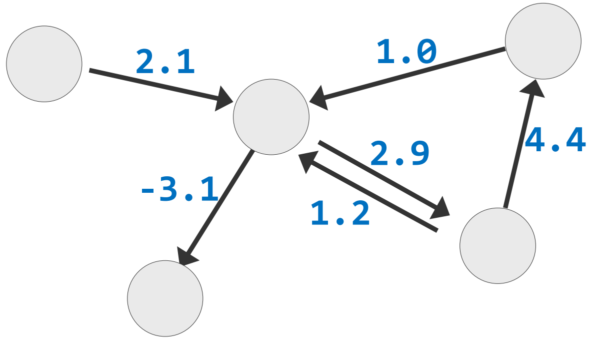


Definition – Weighted (un)directed Graph

An **weighted undirect Graph** is a Triplet $G = (V, E, w)$, where
- *(V,E) is an undirected Graph*
- w: E $\rightarrow$ M *is a mapping from edge to weight*

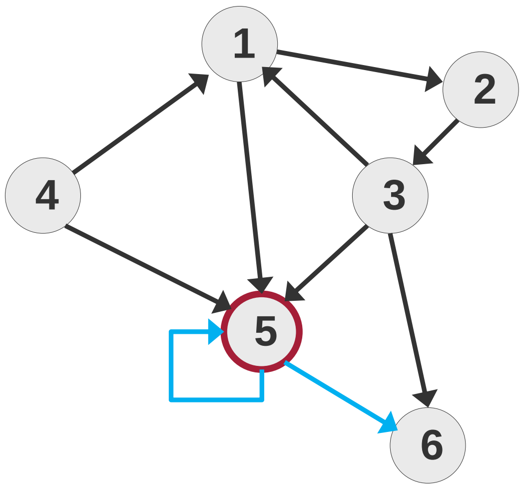


A **weighted directed Graph** is a Triplet $G = (V, E, w)$, where
- *(V, E) is a directed Graph*
- w: *E* $\rightarrow$ *M is a mapping from edge to weight*



Note: one can also define weights for nodes analogously.

Representing Graphs - Adjacency matrices



representation

j →

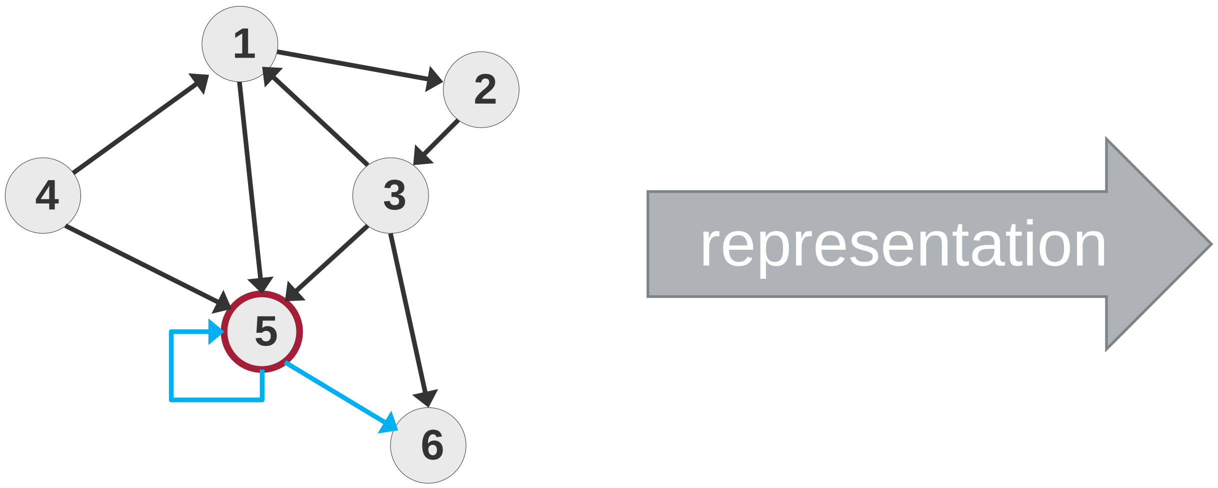
i ↓

	1	2	3	4	5	6
1	0	**1**	0	0	**1**	0
2	0	0	**1**	0	0	0
3	**1**	0	0	0	**1**	**1**
4	**1**	0	0	0	**1**	0
5	0	0	0	0	**1**	**1**
6	0	0	0	0	0	0

Adjacency matrix of a **directed graph with n vertices:**

- Matrix `A = [`a_{ij}`]`
- Entry a_{ij} is 1 if v_i has an edge to v_j
- For undirected graph, the matrix is symmetric
- For weighted graph, the entries are the weights

Representing Graphs - Adjacency matrices



j →

i ↓

	1	2	3	4	5	6
1	0	**1**	0	0	**1**	0
2	0	0	**1**	0	0	0
3	**1**	0	0	0	**1**	**1**
4	**1**	0	0	0	**1**	0
5	0	0	0	0	**1**	**1**
6	0	0	0	0	0	0

```cpp
int n; // number of nodes
vector<vector<int>> adjM(n, vector<int>(n, 0));

// … for unweighted edges
vector<vector<bool>> adjM(n, vector<bool>(n, 0));
```

Implementation:

- Even with a space efficient `vector<bool>` specialization in the STL. The representation is inefficient for very large graphs with few edges (sparse matrix with most entries 0).

Representing Graphs - Adjacency matrices

Inserting a **node** into an **adjacency matrix:**

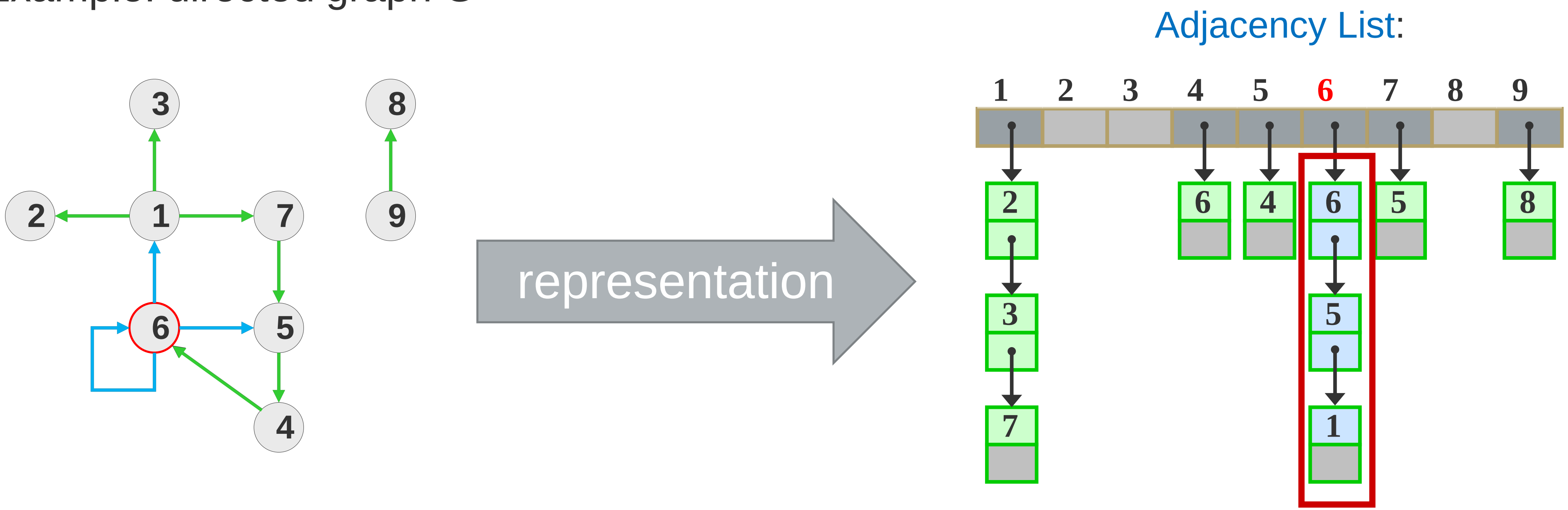
- Input
  - for **undirected** graphs: new node (with value, item) has a set of neighbor nodes;
  - for **directed** graphs: new node (with value) has a set of successor nodes and a set of predecessor nodes (and weighs)

- Append an item as a **new row** in the adjacency matrix
- For **each row**, a **new column** item must also be appended
- This can get inefficient very fast (vectors may grow often; a lot of reallocations happen).
- Adjacency lists can mitigate some of these problems and work well for sparse graphs.

Representing Graphs - Adjacency Lists

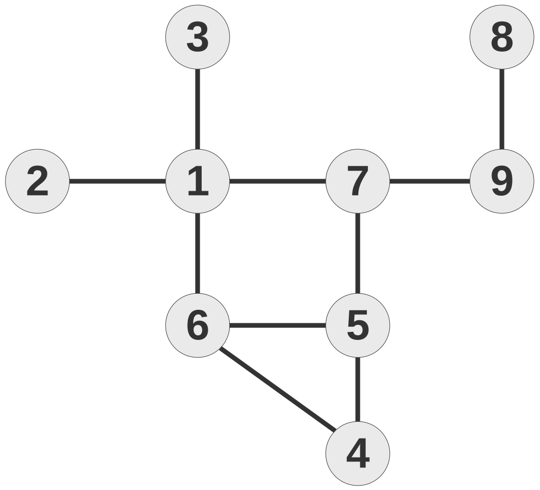
- A graph G(V, E) is
  - defined by a node array of length card{**V**} whose elements contain **pointers to lists of neighboring nodes** (nodes that can be reached directly through an outedge).

Example: directed graph G

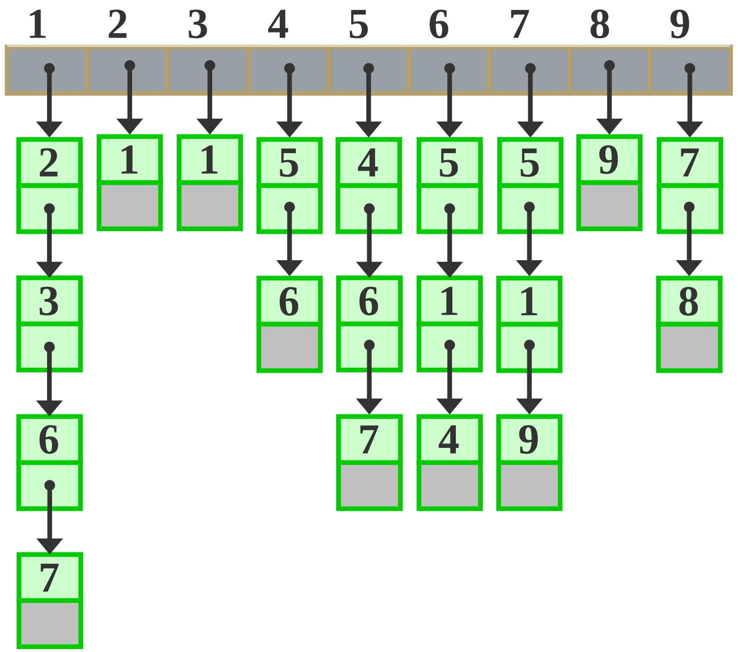


Representing Graphs - Adjacency Lists

Example: undirected graph G



Adjacency List:



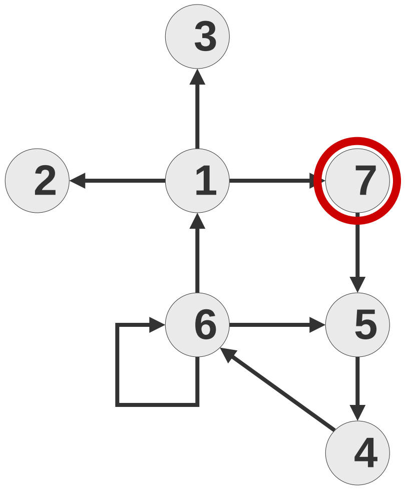
For **weighted graphs**, the weights are stored in the nodes of the lists.

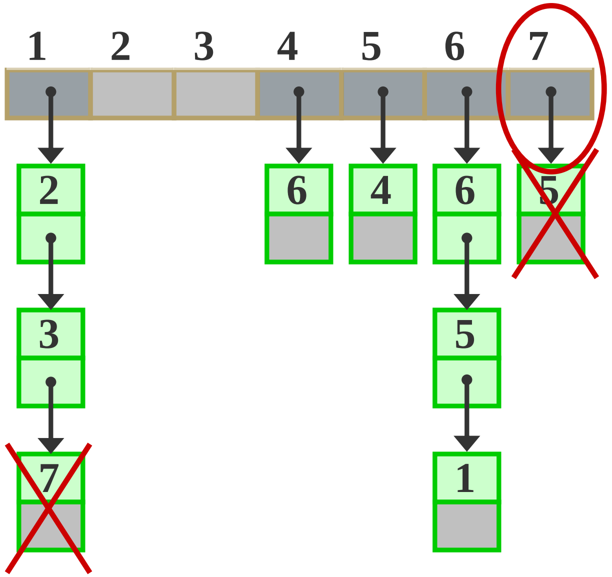
- Memory:
  - Average case: adjacency list O(|**V**| + |**E**|) vs. adjacency matrix O(|**V**| • |**V**|)
  - Worst case: both O(|**V**| • |**V**|)

Removing an Element

- First, the element is **searched;** the element (node) is marked (e.g. pointer, index of the node in the node list).
- **Deleting the connections** of the marked node in the adjacency matrix or in the adjacency list.

- Example - Delete node 7 in **directed graph G** and **adjacency list:**





Algorithms on Graphs

Simple example algorithms:

- Determine if node can be reached (is there a path between v_i and v_j?)
- Determine if graph contains cycles
- Determining whether a given graph is **fully connected.**

Can easily be answered by **traversing the graph**.

More advanced example algorithms:

- **Shortest path problem**: finding the shortest path starting from a node S to a node Z (edges can be weighted)
- Determination of **minimum spanning trees**: Find the tree with the minimum cost (sum of weights) connecting all possible nodes.
- Travel/optimization problems (**traveling salesman problem**): Computation of a round-trip through all nodes. Nodes are visited only once, and the total cost of the path is minimized.

Depth-first Search in a Graph

When traversing a graph, two basic strategies are distinguished

- Depth-first search
- Breadth-first search

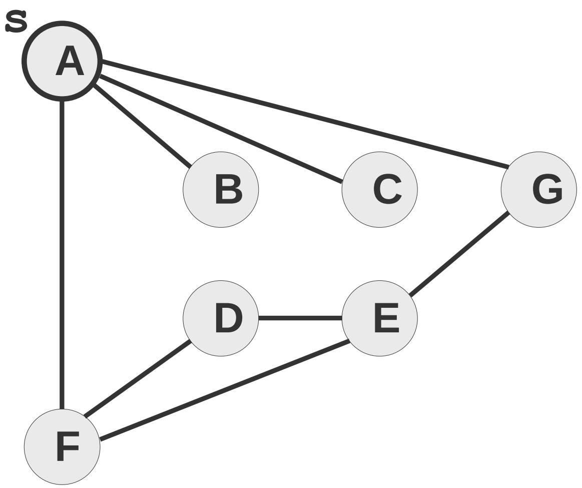
Depth-first search (recursive)

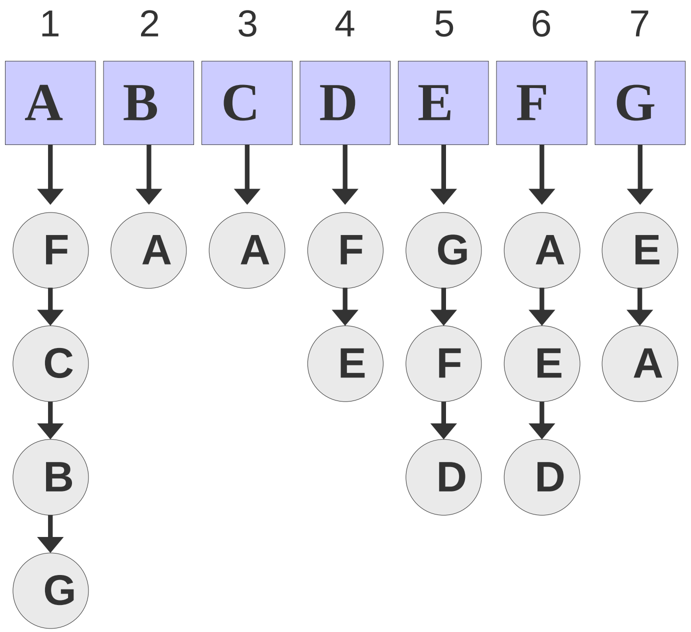
- Mark node s $\in$ **G**.
- Follow all incident edges and mark the visited node **v**
- If one encounters already visited nodes, this path is not explored further. Instead, an alternative edge e $\in$ nb(v) to another neighbor is followed
- If all edges starting from a current node have been traversed, the algorithm backtracks to the predecessor (to the node from which the current node was reached)
- Continue recursively

- **Note:** Non recursive versions can be implemented using a stack (LIFO) data structure

Example – Depth First Search

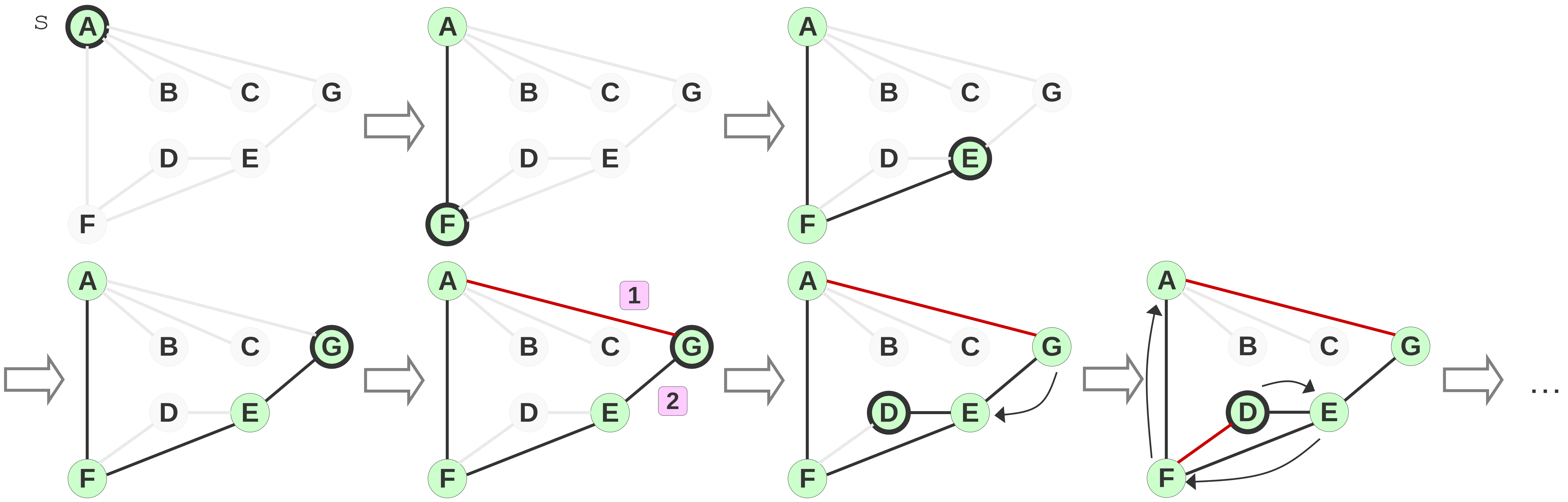
- For a graph G(V, E) and a distinguished start node $s \in G$
- Example graph and its representation (adjacency list).





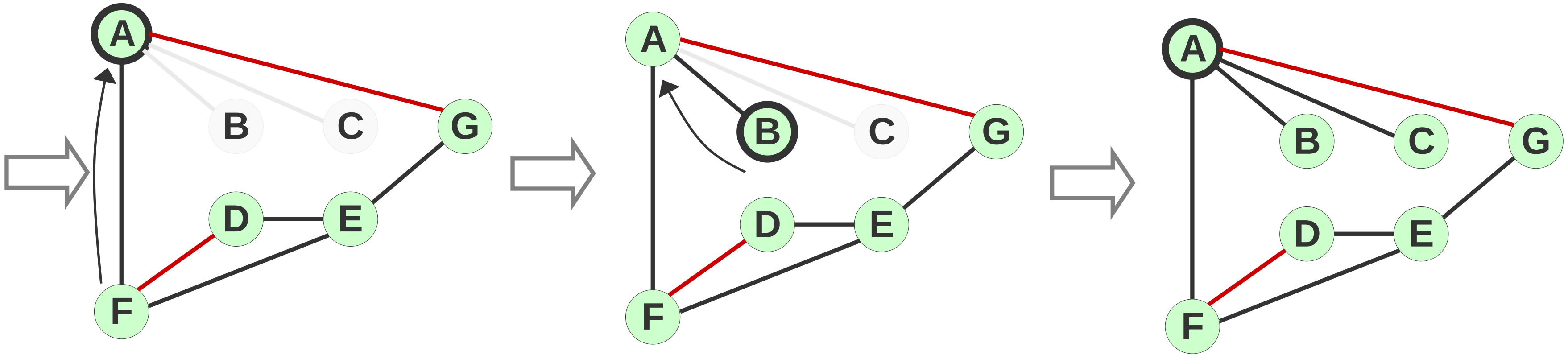
- The strategy of depth-first search runs from a node (starting with **s**) along one of the edges an edge $\mathbf{e_1} = \mathbf{v_{current}v_{NB}}$ starting from it,
- If node not yet visited, trace edge, etc.

Depth-first Exploration



- Current node: bold outline
- Visited nodes: green
- Non-followed edge: **red**

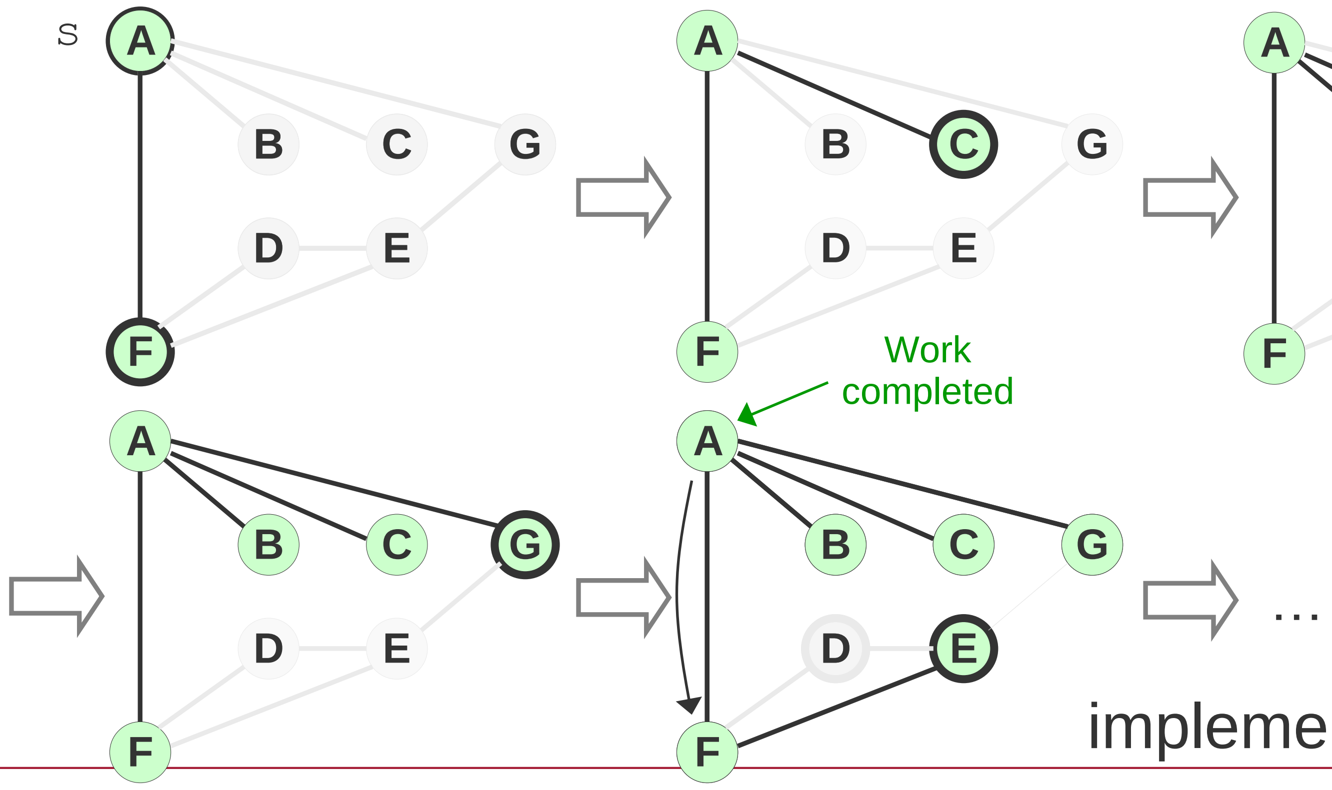
Depth-first Exploration



- Current node: bold outline
- Visited nodes: **green**
- Non-followed edge: **red**

Breadth-first Search

- Starting from a start node, all directly connected nodes are visited before going to the next lower level
- For the same example graph, exploration in breadth-first search is discussed (adjacency list of A: F - C - B - G)



implemented using a queue (FIFO)

Summary

- Both BFS and DFS can be easily modified to solve the reachability problem.
  - If the target node is reached during traversal, we are done else it is not reachable.

- Graph, Vertex, Edge, Basic terms

- Adjacency Lists

- Adjacency Matrices

- Traversal, DFS, BFS

Summary

- Graph representation
  - Adjacency matrix
  - Adjacency list
  - Drawbacks and benefits?

- Depth-first traversal vs. Breadth-first traversal
  - Stack vs. queue

Outlook:

- Shortest path algorithm
  - Dijkstra
  - Dynamic programming
- A*-Algorithm
  - Shortest path with heuristics