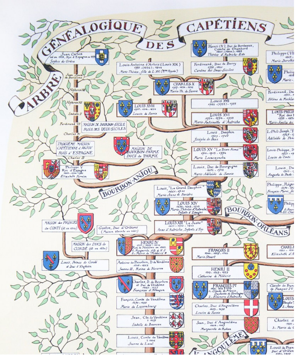
Fundamental Algorithmic Techniques VI

November 7, 2025

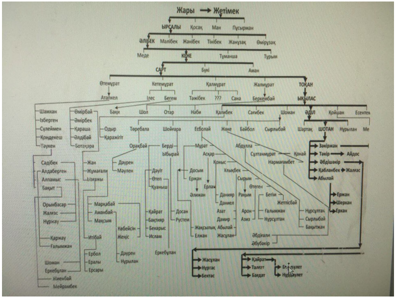
Outline

Graphs: Oldest Application

Early applications of graphs in historical contexts...

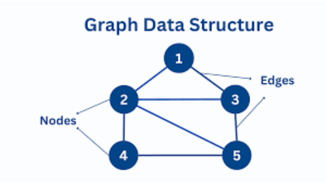


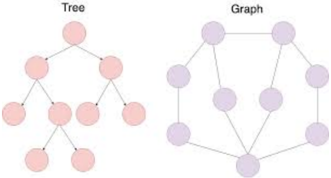
Capetian dynasty



Kazakh Clans

Introduction to Graphs: Basic Definitions





Tree & graph

Formal Definition

A (simple) graph is a pair of sets (V, E), where:

- V is a non-empty finite set of **vertices** (or **nodes**),
- E is a set of pairs of elements from V, called **edges**.

Undirected graph: Edges are unordered pairs (2-element sets). We write uv (or $\{u, v\}$) for the edge between u and v.

Directed graph: Edges are ordered pairs. We write $u \to v$ (or (u, v)) for the edge u to v.

Graph Basics: Subgraphs, Walks, and Connectivity

Subgraph: $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

Walk: sequence of vertices where consecutive vertices are adjacent.

Path: a walk with no repeated vertices.

Reachable: v is reachable from u if a path exists between them.

Connected: every pair of vertices is reachable.

Component: maximal connected subgraph.

Trees, Forests, and Spanning Subgraphs

Closed walk: starts and ends at same vertex.

Cycle: closed walk with no repeated vertices (except start/end).

Acyclic graph: contains no cycles $\rightarrow$ called a forest.

Tree: connected acyclic graph (i.e., one-component forest).

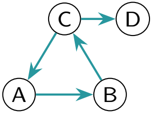
Spanning tree: subgraph that is a tree and includes **all** vertices of G.

G has a spanning tree $\iff$ G is connected.

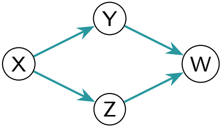
Spanning forest: spanning tree for each component.

Directed Graphs: Walks, Reachability, and DAGs

Directed walk: $v_0 \to v_1 \to \cdots \to v_\ell$ where each $(v_{i-1}, v_i) \in E$.
Directed path/cycle: no repeated vertices (except start/end in cycle).
v is reachable from u if a directed path $u \rightsquigarrow v$ exists.
Strongly connected: every vertex reachable from every other.
Directed Acyclic Graph (DAG): no directed cycles.



Cyclic digraph



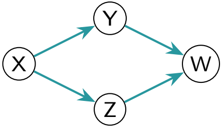
DAG (acyclic)

Directed Graphs: Walks, Reachability, Weighted & DAGs

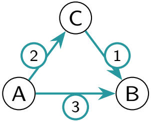
Directed walk: $v_0 \to v_1 \to \cdots \to v_\ell$ where each $(v_{i-1}, v_i) \in E$.
Directed path/cycle: no repeated vertices (except start/end in cycle).
v is reachable from u if a directed path $u \rightsquigarrow v$ exists.
Strongly connected: every vertex reachable from every other.
Directed Acyclic Graph (DAG): no directed cycles.

Unweighted graph: edges have no numerical values.
Weighted graph: each edge (u, v) has a weight $w(u, v) \in \mathbb{R}$.

For vertex v: $\deg^-(v) = |\{u : (u, v) \in E\}|$ (in-degree), $\deg^+(v) = |\{u : (v, u) \in E\}|$ (out-degree).

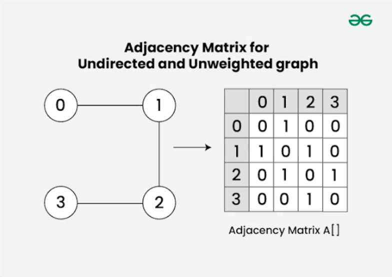


DAG (acyclic)

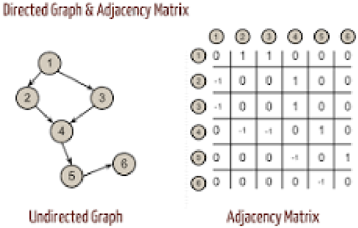


Weighted digraph

Graphs Representations: Adjacency Matrix



Undirected graph
(symmetric matrix)



Directed graph
(asymmetric matrix)

Adjacency matrices use $\mathcal{O}(V^2)$ space.
Efficient for dense graphs but obvious waste of memory for sparse.

$\Rightarrow$ Sparse Matrices

Graphs Representations: Sparse Matrix Representations

Sparse matrices store only non-zero values to save space. Three standard formats of size $\mathcal{O}(V_{non\ zero})$:

COO (Coordinate List)

Store triplets: (row, col, value)

i	j	val
0	2	5
1	0	3
2	2	7

Unsorted; simple to build

CSR (Compressed Sparse Row)

`values: [5,3,7]|`
`col_idx: [2,0,2]|`
`row_ptr: [0,1,2,3]|`
`Efficient row access; used for vector multiplication`

CSC (Compressed Sparse Column)

`values: [3,5,7]|`
`row_idx: [1,0,2]|`
`col_ptr: [0,1,1,3]|`
`Efficient column access; transpose of CSR`

COO = easy construction; CSR/CSC = efficient computation

Graphs: Basic Operations

Common operations on graph data structures:

`add_vertex(G, x)` Inserts a new vertex x into graph G.

`remove_vertex(G, x)` Removes vertex x and all its incident edges.

`add_edge(G, x, y)` Adds an edge between vertices x and y.

`remove_edge(G, x, y)` Removes the edge between x and y.

`adjacent(G, x, y)` Returns `true` if edge (x, y) exists.

`neighbors(G, x)` Returns list of vertices adjacent to x.

`get_vertex_value(G, x)` Retrieves the value stored at vertex x.

`set_vertex_value(G, x, v)` Sets the value of vertex x to v.

Graphs: Construction Operations

Operations to combine or transform graphs:

Graph Union Creates a new graph by combining two existing graphs G_1 and G_2. The most common method is the *disjoint union*, which keeps all vertices and edges from both graphs.

Graph Intersection Creates a new graph containing only the vertices and edges that are common to both G_1 and G_2.

Graph Join Creates a new graph by adding all possible edges that connect a vertex from G_1 to a vertex in G_2.

Traversal and Analysis Operations

Key algorithms for exploring and analyzing graph structure:

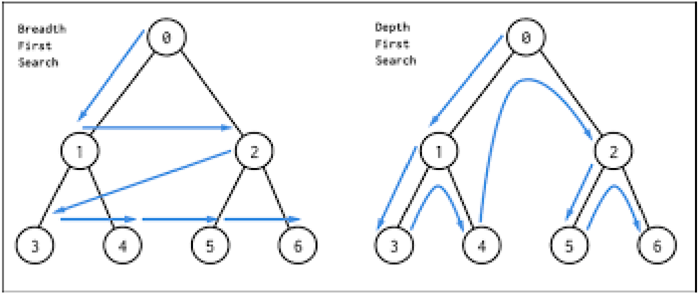
Graph Traversal Involves visiting every vertex in the graph. Common algorithms: Depth-First Search (DFS) and Breadth-First Search (BFS).

Shortest Path Finds the path with minimum total weight between two vertices in a weighted graph. Algorithms: Dijkstra's, Bellman-Ford, or Floyd–Warshall.

Connectivity Determines whether the graph is connected (undirected) or strongly/weakly connected (directed), and identifies connected components.

Topological Sort Arranges vertices of a directed acyclic graph (DAG) in linear order such that for every edge $u \to v$, u comes before v. Used in scheduling, build systems, and dependency resolution.

Depth-First Search vs Breadth-First Search



Breadth-First Search (BFS)

Explores all neighbors at the present depth before moving deeper.

Uses a **queue** (FIFO).

Depth-First Search (DFS)

Explores as far as possible along each branch before backtracking.

stack (recursion or explicit LIFO).