Graphs

A graph is a pair $G = \langle V, E \rangle$, where

- $V = \{v_1, \ldots, v_n\}$ is a set of vertices, and
- $E = \{(v_i, v_j)\}$ is the set of edges.

Directed and Undirected Graphs. In directed graphs edges have start and end: if $(v, v') \in E$ is an edge in a directed graph $G = \langle V, E \rangle$, then $v$ is its start and $w$ is its end, and the direction of $(v, v')$ is from $v$ to $v'$.

In undirected graphs, edges do not have directions: $(v, v') = (w, v)$ for any edge $(v, v') \in E$.

Weighted Graphs. A vertex weighted graph $G = \langle V, E, w \rangle$ is a graph where $w : V \to \mathbb{R}$. Here, $w$ is the vertex weight function.

An edge weighted graph is a graph $G = \langle V, E, w \rangle$, where $w : E \to \mathbb{R}$. Here, $w$ is the edge weight function.

Graph Representation

Graphs have two typical representations as data structures: adjacency matrices and adjacency lists.

Adjacency matrix representation. A graph $G = \langle V, E \rangle$ where $V = \{v_1, \ldots, v_n\}$ is represented as a two-dimensional array $G[1..n, 1..n]$. $G[i, j] = 1$ if $(v_i, v_j) \in E$. $G[i, j] = 0$ otherwise.

A vertex weighted graph $G = \langle V, E, w \rangle$ is represented as a two-dimensional array $G[1..N][0..N]$, where $G[i][0] = w(v_i)$ and $g[i][j] = 1$ if $(v_i, v_j) \in G$ and $G[i, j] = 0$ otherwise for $j > 0$.

An edge weighted graph $G = \langle V, E, w \rangle$ is represented as a two-dimensional array $G[1..N][1..N]$, where $g[i][j] = w(v_i, v_j)$.

Adjacency matrices for undirected graphs are symmetric.

Adjacency list representation. A graph $G = \langle V, E \rangle$ is represented as an array $Adj[1..n]$ of lists. $Adj[v_i]$ contains all $v_j$, such that $(v_i, v_j) \in E$. If $G$ is vertex weighted, an additional array $w[1..n]$ is used to store vertex weights. If $G$ is edge weighted, then $Adj[v_i]$ stores pairs $(v_j, w(v_i, v_j))$.

Figures 1 and 2 show the adjacency list and adjacency matrix representations of undirected and directed graphs (respectively).

![Adjacency List and Matrix](image)

Figure 1: An undirected graph and its adjacency list and adjacency matrix representations.
Algorithms: Graph Traversal

Graph Traversal Problem. Given a graph $G = (V, E)$ and a node $s \in V$ (referred to as the source node), visit all nodes of the graph starting with the source node.

The output of a Graph Traversal Algorithm is a list of nodes in the order in which the algorithm visited them. (In a sense, "visit" in the context of graph traversal can be viewed as "access the label of the vertex").

Note: This is feasible is the graph is fully connected. If the graph is not fully-connected, the traversal problem can be restated as "visit all nodes in the graph connected to the source node".

Breadth-First Search

Breadth-First Search is a traversal technique which visits all yet unvisited neighbors of a node $v$ right after visiting $v$.

Node coloring. In the breadth-first search algorithm we use colors of nodes as node labels to represent current state of a node (visited, enqueued, unvisited). This is done for aesthetic reasons, numeric labels can be used instead.

Data Structures. Breadth-first search algorithm (BFS algorithm) maintains a queue of vertices.

BSF Algorithm. See Figure 3.

BFS: Analysis

Running time. $T(BFS) = O(|V| + |E|)$.

Notes: Initialization costs $O(V)$. For each node $v$ visited, its adjacency list $\text{Adj}[v]$ will be scanned once, leading to the overall $O(\sum_{v \in V} |\text{Adj}[v]|) = O(|E|)$.

Path. A path in a graph $G = (V, E)$ is a sequence $e_1, e_2, \ldots e_k$ of edges $e_i = (v_i, u_i) \in E$, such that $u_1 = v_2, u_2 = v_3, \ldots u_{k-1} = v_k$.

Shortest Path. A shortest path distance between two nodes $s$ and $v$ in a graph $G$, denoted $\delta(s, v)$ is the minimum number $k$ of edges on a path from $s$ to $v$.

A shortest path between $s$ and $v$ is any path whose length is equal to the shortest path distance between $s$ and $v$.

$v$ is reachable from $s$ if there exists at least one path in $G$ from $s$ to $v$.

Lemma 1. Let $G = (V, E)$ be a graph, $s \in V$ and $(u, v) \in E$. Then

$$\delta(s, v) \leq \delta(s, u) + 1.$$
Algorithm BFS(V, Adj, s)
begin
    foreach v ∈ V − {s} do //initialization
        v.color ← WHITE; //color
        v.d ← ∞; //distance from the source
        v.π ← NULL; //"parent" (in the traversal)
    endfor
    s.color ← GRAY;
    s.d ← 0;
    s.π ← NULL;
    Q ← ∅; //initialization of the queue
    Enqueue(Q, s);
    //Main loop
    while Q ≠ ∅ do
        u ← Dequeue(Q);
        foreach v ∈ Adj[u] do
            if v.color = WHITE then
                v.color ← GRAY;
                v.d ← u.d + 1;
                v.π ← u;
                Enqueue(Q, v);
            endif
        endfor
        u.color ← BLACK;
    endwhile
end

Figure 3: Breadth-First Search Algorithm pseudocode.

Proof. If u is reachable from s, then, of course v is reachable as well.
Case 1. u is on the shortest path from s to v. Then δ(s, v) = δ(s, u) + 1.
Case 2. u is NOT on the shortest path from s to v. Then δ(s, v) < δ(s, u) + 1.

Lemma 2. For each node v ∈ V, in algorithm BFS v.d ≥ δ(s, d).

Proof. By induction. Base case: let n = s (the source node).

Lemma 3. Consider some state of the queue Q = (u₁, . . . , uᵣ) in the BFS algorithm. Then u₁.d ≤ u₂.d ≤ . . . ≤ uᵣ.d.

Proof. Induction on the number of Enqueue operations.

Theorem 1. Let G = (V, E) be a graph and s ∈ V. Then:
1. Algorithm BFS discovers all nodes in V reachable from s.
2. At the end of the algorithm, for each node v ∈ V, v.d = δ(s, v).
3. For each node v ≠ s, one of the shortest paths from s to v goes through the node v.π (and edge (v.π, v)).
Proof. By contradiction.

Depth-First Search

**Depth-First Search** traversal is a graph traversal technique that visits the neighbors of most recently visited node on each step.

**DFS node colors.** The Depth-First search (DFS) algorithm colors nodes as follows. Unvisited nodes are white; discovered nodes are gray and visited nodes are black.

**DFS timestamps.** Each node \( v \in V \) receives two *timestamps* during the DFS algorithm. The first timestamp, \( v.d \), records the step on which \( v \) was discovered (became gray). The second timestamp, \( v.f \) records the step on which \( v \) was visited (became black).

```
ALGORITHM DFS(V, Adj)
begin
  foreach \( v \in V \) do  //initialization
    \( v.color \leftarrow \text{WHITE} \);  //vertex color: unvisited
    \( v.\pi \leftarrow \text{NULL} \);  //"parent" (in the traversal)
  endfor
  time \leftarrow 0  //behaves as global variable
  //Main loop
  foreach \( v \in V \) do
    if \( v.color = \text{WHITE} \) then DFS_VISIT(V, Adj, v);
  endfor
end
```

```
ALGORITHM DFS_VISIT(V, Adj, u)
begin
  time \leftarrow time + 1;
  u.d \leftarrow time;
  u.color \leftarrow \text{GRAY};  //mark vertex as discovered
  foreach \( v \in Adj[u] \) do  //visit neighbors
    if \( v.color = \text{WHITE} \) then
      v.\pi \leftarrow u;
      DFS_VISIT(V, Adj, v);
    endif
  endfor
  u.color \leftarrow \text{BLACK};  //mark vertex as visited
  time \leftarrow time + 1;
  u.f \leftarrow time;
end
```

**DFS: Analysis**

**Running time.** \( T(DFS) = \Theta(|V| + |E|) \).

*Note.* Initialization takes \( \Theta(|V|) \) steps. On each call of \texttt{DFS\_VISIT}, with node \( v \) as input, at most \( |\text{Adj}[v]| \) of recursive calls will be made. So, the total number of recursive calls of \texttt{DFS\_VISIT} is \( \Theta(|E|) \).

**Predecessor subgraph.** Given a graph \( G \), its predecessor subgraph \( G_{\pi}^* = (V, E_{\pi}) \) contains only the edges \((v.\pi, v)\) for each \( v \in V \) generated during the DFS traversal.
Predecessor subgraph in DFS. A forest of trees. One tree per vertex \( v \in V \) that was retrieved in the main loop of Algorithm DFS and passed the if \( v.color = \text{WHITE} \) condition.

Parenthesis theorem. In any DFS order of the traversal of a graph \( G = \langle V, E \rangle \), for any two nodes \( u, v \in V \), one of the following three conditions holds:

1. \([u.d, u.f] \subset [v.d, v.f]\); \( u \) is a descendant of \( v \).
2. \([v.d, v.f] \subset [u.d, u.f]\); \( v \) is a descendant of \( u \).
3. \([u.d, u.f] \cap [v.d, v.f] = \emptyset \); \( u, v \) are not in ancestor-descendant relationship.

Proof. Consider two cases: \( u.d < v.d \) and \( u.d > v.d \). For each subcase, establish two possible outcomes.

White-path Theorem. In a depth-first forest of a graph \( G = \langle V, E \rangle \), vertex \( v \) is a descendant of vertex \( u \) iff at the time \( u.d \), there is a path from \( u \) to \( v \) which only encounters white nodes.

Proof. Prove in both directions. For the \( \Rightarrow \) direction, the white path is constructed. For the \( \Leftarrow \) direction, prove by contradiction.

Classification of edges. DFS algorithm splits edges in \( G \) into the following categories:

- **Tree edges.** Edges in the \( G_\pi \) depth-first forest of \( G \).
- **Back edges.** Edges \((u, v)\), where \( u \) is a descendant of \( v \).
- **Forward edges.** Edges \((u, v)\) not in \( G_\pi \), where \( v \) is a descendant of \( u \).
- **Cross edges.** All other edges.

Algorithms: Topological Sort on DAGs

Directed Acyclic Graphs (DAGs). A directed graph \( G = \langle V, E \rangle \) is acyclic if for any node \( v \in V \), there is no path from \( v \) back to \( v \).

Note, DAGs do not have back edges.

Topological Sort Problem. Given a directed acyclic graph \( G = \langle V, E \rangle \) a topological sort of \( G \) is a linear order \( < \) on the vertices from \( V \), such that:

if there is an edge \((u, v) \in E\), then \( u < v \).

The problem is to find a(ny) topological sort given a DAG \( G = \langle V, E \rangle \)

Algorithm. The algorithm for topological sort uses DFS:

run DFS(G), compute all \( v.f \)
sort \( V \) in ascending order by \( v.f \)
return sorted list of nodes