Algorithms on Graphs: Part I

Graphs

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* is a graph $G = \rangle V, E, w \langle$ where $w : V \longrightarrow \mathcal{R}$. Here, w is the *vertex weight function.*

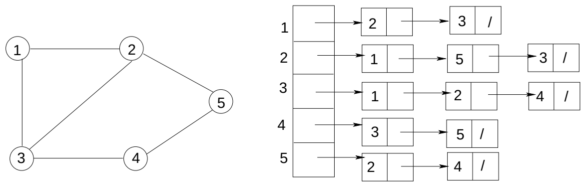
An *edge weighted graph* is a graph $G = \rangle V, E, w \langle$, where $w : E \longrightarrow \mathcal{R}$. Here, w is the *edge weight function.*

Graph Representation

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

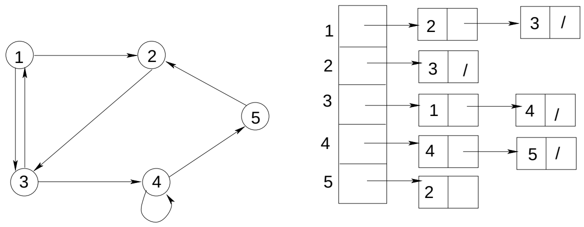
Adjacency matrix representation. A graph $G = \langle V, E \rangle$ where $V = \{v_1, \ldots, v_n\}$ is represented as a two-dimensional array $G[1..n, 1..n]$. $G[i, j] = 1$ **if** $(v_i, v_j) \in G$. $G[i, j] = 0$ otherwise.

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 $j > 0$.



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

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



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

Figure 2: An undirected graph and its adjacency list and adjacency matrix representations.

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.

Adjecency list representation. A graph $G = \langle V, E\rangle$ is represented as and 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).

Algorithms: Graph Traversal

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

```
ALGORITHM BFS(V,Adj,s)
begin
  foreach v ∈ V − {s} do      //initialization
    v.color ← WHITE;      //color
    v.d ← ∞;      //distance from the source
    v.π ←NIL;      //"parent" (in the traversal)
  endfor    s.color ← GRAY;
  s.d ← 0;
  s.π ← NIL;
  Q ← ∅;      //initialization of the queue
  Enqueue(Q);
             //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
```

BFS: Analysis

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

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

Path. A *path* in a graph $G = \langle V, E\rangle$ 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

$$\delta(s, v) \leq \delta(s, u) + 1.$$

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 $\delta(s, v) = \delta(s, u) + 1$.

Case 2. u is NOT on the shortest path from s to v. Then $\delta(s, v) < \delta(s, u)+1$.

Lemma 2. For each node $n \in V$, in algorithm BSF $v.d \geq \delta(s,d)$.

Proof. By induction.

Lemma 3. Consider some state of the queue $Q = (u_1, \ldots, u_r)$ in the BFS algorithm. Then $u_1.d \leq u_2.d \leq \ldots \leq u_r.d$.

Proof. Induction on the number of Enqueue operations.

Theorem 1. Let $G = \langle V, E \rangle$ be a graph and $s \in V$. Then:

1. Algorithm BFS discoveres all nodes in V *reachable* from s.
2. At the end of the algorithm, for each node $v \in V$, $v.d = \delta(s,d)$.
3. For each node $v \neq s$, one of the *shortest paths* from s to v goes through the node $v.\pi$ (and edge $(v.\pi, v)$).

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 ∈ V do     //initialization
    v.color ← WHITE;     //vertex color:  unvisited
    v.π ←NIL;     //"parent" (in the traversal)
  endfor   time ← 0     //behaves as global variable
//Main loop
  foreach v ∈ V do
    if v.color = WHITE then DFS_VISIT(V,Adj,v);
  endfor
end
```

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

DFS: Analysis

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

Note. Initialization takes $\Theta(|V|)$ steps. On each call of `DFS_VISIT`, with node v as input, at most $|Adj[v]|$ of recursive calls will be made.So, the total number of recursive calls of `DFS_VISIT` is $\theta(|E|)$

Predecssor subgraph. Given a graph G, its predecessor subgraph $G_\pi = \langle V, E_\pi \rangle$ contains only the edges $(v.\pi, v)$ for each $v \in V$.

Predecessor subgraph in DFS. A forest of trees.

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_π 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_π, 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 topoliogical 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

Minimal Spanning Trees

Spanning Tree. Let $G = \langle V, E, w \rangle$ be a (connected) edge-weighted undirected graph. A *spanning tree* of G is a subset $T \subseteq E$ of edges, such that $G_T = \langle V, T \rangle$ is *connected* and *acyclic*.

The weight of a spanning tree: $w(T) = \sum_{(u,v,)\in T} w(u, v)$.

Minimal Spanning Tree Problem. Given an undirected edge-weighted graph, find a spanning tree with minimum weight.

Greedy approach. (if we can make it work).

ALGORITHM GENERIC_MST(V,Adj,w)
begin
  $A \leftarrow \emptyset$;
  while A is not a spanning tree **do**
    find a safe edge$(u, v) \in E$ to include in A
    $A = A \cup \{(u, v\}$
  endwhile
  return A;
end

How to find a safe edge?

Cuts. A **cut** $S, V - S$ in an undirected graph $G = \langle V, E\rangle$ is a partition of V into two sets.

An edge (u, v) **crosses** the cut if $u \in S$ and $v \in V - S$ or vice versa.

A cut **respects** a set A of edges is no edge in A crosses the cut.

An edge (u, v) is a **light edge** crossing the cut is its weight is the minimum among all edges crossing the cut.

Theorem 3. Let $G = \langle E, V, w\rangle$ be a connected undirected edge-weighted graph. Let $A \subset E$ be in some *minimal spanning tree.* and let $(S, V - S)$ be some cut of G.

If $(S, V - S)$ respects A and (u, v) is a light edge crossing A, then (u, v) is safe for A.