Advanced Graph Theory

Written by HADIOUCHE Azouaou.

Disclaimer

This document contains the lectures given by Dr. MEHDAOUI.

> To separate the contents of the course to actual additions or out of context information, a black band will be added by its side like the one on this comment.

Contents

Chapter 1

Fundamental Concepts

1.1. Definitions & Notations

1.1.1. Types Of Graphs

Definition 1.1.1.1 (Undirected Graph): *An undirected graph G is an ordered pair $G = (V, E)$ where V is a finite set called the vertex set (vertices) and $E \subset \{\{u, v\} \mid u, v \in V\}$ called the set of edges.*

Definition 1.1.1.2 (Directed Graph): *A directed graph G is an ordered pair $G = (V, A)$ where V is the set of vertices and $A \subset V \times V$ called the set of arcs or directed edges.*

Definition 1.1.1.3 (Unweighted Graph): *An unweighted graph G is a graph is a graph where all edges are considered equal.*

Definition 1.1.1.4 (Weighted Graph): *A weighted graph $G = (V, E)$ is a graph where there is a function $w : E \to \mathbb{R}$ where for each edge $e \in E$, the weight of e is $w(e)$.*

1.1.2. Concepts On Vertices & Edges

Definition 1.1.2.5 (Adjacent/Incident/Isolated): *Let $G = (V, E)$ a graph:*

1. *Two vertices $u, v \in V$ are adjacent or neighbors if $\{u, v\} \in E$.*
2. *A vertex v is called incident to an edge e if v is an endpoint of e.*
3. *A vertex with no incident edges is called isolated.*
4. *The order of G is $\# V$.*
5. *The size of G is $\# E$.*
6. *The open neighborhood of $v \in V$ is $\mathcal{N}(v) = \{u \in V \mid \{u, v\} \in E\}$.*
7. *The closed neighborhood of $v \in V$ is $[v] = \mathcal{N}(v) \cup \{v\}$.*
8. *A degree of a vertex v denoted $d(v)$ or $\deg(v)$ is the number of edges incident to v, that is $\deg(v) = \# N(v)$.*
9. *If G is directed then we define the following:*
   1. *The out neighborhood of $v \in V$ is $\mathcal{N}^+(v) = \{u \in V \mid (v, u) \in A\}$.*
   2. *The in neighborhood of $v \in V$ is $\mathcal{N}^-(v) = \{u \in V \mid (u, v) \in A\}$.*
   3. *The in/out degree are $\deg^-(v) = \# \mathcal{N}^-(v), \deg^+(v) = \# \mathcal{N}^+(v)$.*
10. *The number of edges incident to v with maximum degree of G, $\Delta(G) = \max_{v \in V} d(v)$ and minimum degree is $\delta(G) = \min_{v \in V} d(v)$.*

Lemma 1.1.2.6 (Handshake): *For an undirected graph $G = (V, E)$ then*

$$\sum_{v \in V} \deg(v) = 2 \# E$$

Proof. Every edge $\{u, v\} \in E$ contributes exactly 1 to $\deg(u)$ and 1 to $\deg(v)$ thus adding 2 to $\sum_{v \in V} \deg(v)$, so all edges counted twice in the sum of all edges. □

Theorem 1.1.2.7: *The number of vertices with odd degree is even.*

Proof. Let $V_o = \{v \in V \mid d(v) \text{ odd}\}$ and $V_e = \{v \in V \mid d(v) \text{ even}\}$, by the handshake lemma we have

$$2 \# E = \sum_{v \in V} \deg(v) = \sum_{v \in V_o} \deg(v) + \sum_{v \in V_e} \deg(v)$$

given that $2 \# E$ is even and $\sum_{v \in V_e} \deg(V)$ is even then necessarily $\sum_{v \in V_o} \deg(v)$ is even, and since the sum of odd numbers is even if and only if the number of

odd numbers is even then $\# V_o$ is even so there is an even number of odd degree vertices. □

1.1.3. Simple & Multigraphs

Definition 1.1.3.8 (Simple/Multi Graph): *A graph G is said to be simple if it has no loops and no multiple edges. A multigraph is a graph that may have loops and have multiple edges.*

Theorem 1.1.3.9: *In any simple graph G with order $n \geq 2$, there exists at least two vertices with same degree.*

Proof. The possible degrees in a simple graph is $\{0, ..., n-1\}$, however a graph cannot have a vertex with degree 0 and $n-1$ at the same time, so either it has vertices of degree $\{1, ..., n-1\}$ or $\{0, ..., n-2\}$, and since there are n vertices and $n-1$ choices for degrees, then by the pigeonhole principle, there are at least two vertices with the same degrees. □

Proposition 1.1.3.10: *There are $\binom{n}{2}$ maximum edges in a simple graph.*

Definition 1.1.3.11 (Walk/Trail): *Let $G = (V, E)$ be a graph*

1. *A walk of length k is a sequence $v_1, e_2, v_2, ..., e_k, v_k$ with $e_i = \{v_{i-1}, v_i\} \in E$.*
2. *A trail is a walk with no repeated edges.*
3. *A path is a walk with no repeated vertices.*
4. *A walk is closed if $v_0 = v_k$.*

Theorem 1.1.3.12: *If there is a walk from vertex u to v then there is also a path from u to v.*

Proof. By induction on the length of the walk n. For $n = 1$, it is trivial since any 1-walk is a path, now consider a walk of length n, $v_1, e_1, v_2, ..., v_{n-1}, e_n, v_n$, if $\forall i, j \in [\![1, n]\!], i \neq j \Rightarrow v_i \neq v_j$ then it is a path, otherwise there exists i, j such that v_i, v_j, we consider the new walk $v_1, e_1, v_2, ..., e_i, v_i, e_j, ..., e_n, v_n$, this walk has a strictly less length then the original, and thus by induction hypothesis, there is a walk from v_1 to v_n. □

Definition 1.1.3.13 (Circuit/Cycle):

1. *A circuit is a closed trail.*
2. *A cycle is a closed path.*
3. *A cycle of length k is denoted C_k, unless otherwise specified.*
4. *The length of the cycle is the number of edges in it.*
5. *A graph G is connected if for every pair of vertices $u, v \in V$ there exists a path from u to v, otherwise it is disconnected.*
6. *A subgraph H of G is a graph such that $V(H) \subset V(G)$ and $E(H) \subset E(G)$.*
7. *A connected component of G is a maximum connected subgraph of G.*
8. *A spanning subgraph H of G is a subgraph such that $V(H) = V(G)$.*

Theorem 1.1.3.14: *In a connected graph, any two longest paths share at least one vertex.*

Proof. Suppose that there is a graph where two longest paths do not share a vertex, denote them $v_1, e_1, ..., e_{n-1}, v_n$ and $v'_1, e'_1, ..., e'_{n-1}, v'_n$, given that the graph is connected, then there is a path from some v_i to v'_j that does not intersect with the previous paths. Consider the path that goes through the furthest endpoint v_1 or v_n from v_i, the path until v_i, then the path from v_i to v'_j and lastly to the furthest endpoint v'_1 or v'_n from v'_j. □

Definition 1.1.3.15 (Induced/Complement Graph):

1. *An induced subgraph G' is a subset $S \subset V$ and all edges from G that have both endpoints in S.*
2. *A complement graph $\overline{G}$ of a graph G has the same vertex set V, and $uv \in E\left(\overline{G}\right) \Leftrightarrow uv \notin E(G)$.*

1. $\# E(G) + \# E\left(\overline{G}\right) = \binom{n}{2}$.
2. The number of simple graphs of order n is $2^{\binom{n}{2}}$.
3. The number of spanning subgraphs of K_n with m edges is $\binom{\binom{n}{2}}{m}$.

4. In a graph with n vertices and m edges, the number of induced subgraphs of 2^n and the number of spanning subgraphs is 2^m.

1.2. Graph Representations

There are several ways to represent graphs in computer memory, each with different advantages for various algorithms and applications. We represent it efficiently in memory so that it does the basic operations like edge lookup and neighborhood efficiently and minimize space complexity.

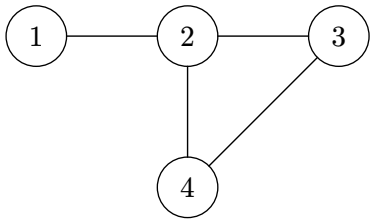
We assume throughout the course that $\#V(G) = n$ vertices and $\#E(G) = m$ edges, vertices are labeled from 1 to n. Graphs may be directed or undirected.

1.2.1. Adjacency Matrix

Let $G = (V, E)$ be a graph with $V = \{v_1, ..., v_n\}$, the adjacency matrix of G is an $n \times n$ matrix $A = (a_{ij})$ where

$$a_{ij} = \begin{cases} 1 \text{ if } v_i v_j \in E(G) \\ 0 \text{ otherwise} \end{cases}$$

Example: Here is a simple graph with its adjacency matrix



$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

For weighted graphs we take the adjacency matrix would be

$$a_{ij} = \begin{cases} w(v_i v_j) \text{ if } v_i v_j \in E \\ \text{None} \quad \text{otherwise} \end{cases}$$

If no edge exists, None, ∞ or 0 can be assigned depending on the application.

1. Space complexity the adjacency matrix is $O(n^2)$.
2. Time to check if an edge exists is $O(1)$.
3. Time to find all neighbors of a vertex is $O(n)$.

1.2.2. Adjacency List

For a graph $G = (V, E)$ the adjacency list representation consists of an array Adj of $\#V$ lists, one for each vertex in V. For each $u \in V$, the adjacency list Adj[u] consists of all the vertices v such that $uv \in E$.

1. Space complexity $O(n + m)$.
2. Time to check if edge exists: $O(d)$ where d is the degree of the vertex.
3. Time to find all neighbors of a vertex: $O(d)$.

For a weighted graph, we just add a couple (v_j, w) if $v_i v_j \in E$ with $w = w(v_i v_j)$.

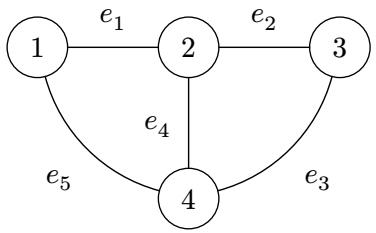
> **Definition 1.2.2.16 (Sparse/Dense Graph):** *Let $G = (V, E)$ a graph, we say that G is a sparse graph if $\#E \ll \#V^2$ and we call it dense if $\#E \approx \Theta(\#V^2)$.*

In sparse graphs the adjacency list is preferred, while in dense graphs the adjacency matrix is preferred.

1.2.3. Incidence Matrix

Given a graph $G = (V, E)$, the incidence matrix M is defined as follows

$$M_{ij} = \begin{cases} 1 \text{ if } v_i \text{ is incident to } e_j \\ 0 \text{ otherwise} \end{cases}$$

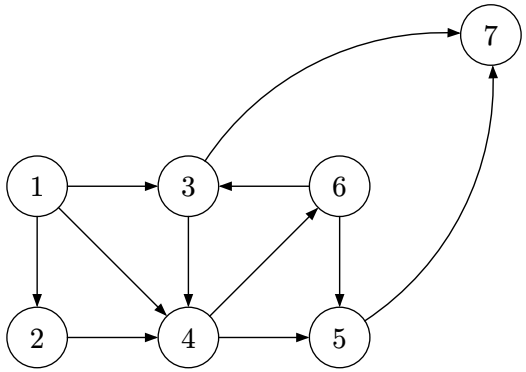


$$M = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

For a directed graph, we define the following representation

$$M_{ij} = \begin{cases} +1 \text{ if } e_j \text{ enters the vertex } v_i \\ -1 \text{ if } e_j \text{ leaves the vertex } v_i \\ 0 \quad \text{otherwise} \end{cases}$$

Example: Consider the following graph



We have the following representations

Adjacency matrix:

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Adjacency list:

$$\begin{aligned} &\text{Adj} = [\\ &\quad 1 \to \{2,3,4\}, 2 \to \{4\}, 3 \to \{4,7\}, \\ &\quad 4 \to \{5,6\}, 5 \to \{7\}, 6 \to \{5\}, 7 \to \{\} \\ &] \end{aligned}$$

Operation	Adjacency Matrix	Adjacency List	Incidence Matrix
Space	$O(n^2)$	$O(n+m)$	$O(n \cdot m)$
Check if edge exists	$O(1)$	$O(d(v))$	$O(m)$
Find all neighbors	$O(n)$	$O(d(v))$	$O(m)$
Add Edge	$O(1)$	$O(1)$	$O(n)$

1.3. Graph Traversal Algorithms

Graph traversal algorithms systematically explores the vertices of the graph. They are fundamental building blocks for many other graph algorithms.

1.3.1. Breadth-First Search

Fundamental graph traversal algorithm explores a graph layer by layer, starting from a source vertex, it visits all its directed neighbors (vertices at distance 1) before moving on to the neighbors of those neighbors, (vertices at distance 2) and so on. BFS uses a Queue data structure (First-In First-Out) to manage the order of exploration. The goal of BFS is to discover all vertices reachable from source s and compute the shortest path distance s to each reachable vertex.

Algorithm 16.1: *Procedure BFS(G, s):*

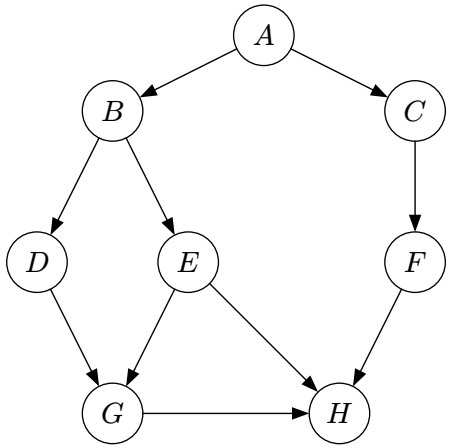
1. *Create a queue Q and a set visited.* ← $O(1)$
2. *Create a map distance with all values initialized to* ∞. ← $O(n)$
3. *visited.add(s)* ← $O(1)$
4. *distance[s]* ← *0* ← $O(1)$
5. *Q.enqueue(s)* ← $O(1)$
6. *while Q is not empty* ← $O(n)$
   1. *v* ← *Q.dequeue()* ← $O(1)$
   2. *Process vertex v.* ← $O(1)$
   3. *for each neighbor w of v* ← $O(m)$
      1. *if w is not visited then*
         1. *visited.add(w)* ← $O(1)$
         2. *distance[w] = distance[v] + 1* ← $O(1)$
         3. *Q.enqueue(w)* ← $O(1)$
      2. *endif*
   4. *endfor*

return distance

Global complexity with adjacency list is $O(n+m)$, and space complexity is $O(n)$.

Example: Let us apply BFS to the following graph $G = (V, E)$

$$V = \{A, B, C, D, E, F, G, H\}$$
$$E = \{(A, B), (A, C), (B, D), (B, E), (C, F)$$
$$(D, G), (E, G), (E, H), (F, H), (G, H)\}$$



Starting from vertex A we have

1. Initialization:
   - $Q = [A]$
   - $d[B] = \cdots = d[H] = \infty$
   - $P[A] = \cdots = P[G] = \text{None}$
2. Step 1: Dequeue A
   - Neighbors B, C
   - visited $= \{A, B, C\}$
   - $d[B] = d[C] = 1$
   - $P[B] = P[C] = A$
   - $Q = [B, C]$
3. Step 2: Dequeue B
   - Neighbors D, E
   - visited $= \{A, B, C, D, E\}$
   - $d[D] = d[E] = 2$
   - $P[D] = P[E] = B$
   - $Q = [C, D, E]$
4. Step 3: Dequeue C
   - Neighbors F
   - visited $= \{A, B, C, D, E, F\}$
   - $d[F] = 2$
   - $P[F] = C$
   - $Q = [D, E, F]$
5. Step 4: Dequeue D
   - Neighbors G
   - visited $= \{A, B, C, D, E, F, G\}$
   - $d[G] = 3$
   - $P[G] = D$
   - $Q = [E, F, G]$
6. Step 5: Dequeue E
   - Neighbors G, H
   - G already visited
   - visited $= \{A, B, C, D, E, F, G, H\}$
   - $d[H] = 3$
   - $P[H] = E$
   - $Q = [F, G, H]$
7. Step 6: Dequeue F
   - Neighbors H
   - H already visited
   - $Q = [G, H]$
8. Step 7: Dequeue G
   - Neighbors H
   - H already visited
   - $Q = [H]$
9. Step 8: Dequeue H
   - No neighbors
   - $Q = []$

Q is empty, algorithm stops. Which gives us the following results

v	A	B	C	D	E	F	G	H
$d(v)$	0	1	1	2	2	2	3	3
$P[v]$	None	A	A	B	B	C	D	E

Exercise: *Rewrite the BFS algorithm assuming the graph $G = (V, E)$ is represented by its adjacency matrix A and provide its complexity.*

Lemma 17: *Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be a fixed source point. For every edge $(u, v) \in E$, we have*

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

where $\delta(s, x)$ denotes the length of the shortest path.

Proof. If u is reachable from s, then there exists a shortest path from s to u of length $\delta(s, u)$, appending the edge (u, v) gives a path from s to v of length $\delta(s, u) +$

1. Since $\delta(s,v)$ is defined as the minimum length then necessarily $\delta(s,v) \leq \delta(s,u)+1$. If u is not reachable from s, then trivially $\delta(s,u)+1=\infty$ □

> **Lemma 18:** *Suppose that during the execution of BFS on a graph $G=(V,E)$, the queue Q contains the vertices $[v_1,...,v_r]$ where v_1 is the head of Q and v_r is the tail of Q. Then the following holds:*
> 1. $d(v_r) \leq d(v_1)+1$.
> 2. $\forall i \in [\![1,...,r-1]\!], d(v_i) \leq d(v_{i+1})$.

Proof. By induction on the number of queue operations.

- Base case: initially, the queue contains the source vertex s, the lemma is trivial.
- Assume the lemma holds before queue operations, we must show that it continues to hold after a dequeue.
  - Dequeue operation: if the head v_1 is dequeued then v_2 becomes the new head (unless the queue becomes empty, in which case the statement remains true), by induction hypothesis we have that $d(v_1) \leq d(v_2)$ since $d(v_r) \leq d(v_1)+1$ we obtain that $d(v_r) \leq d(v_2)+1$ and all the other inequalities remain.
  - Enqueue operation: suppose a vertex v is discovered while scanning the adjacency list of some vertex, hence $d(v)=d(u)+1$ and enqueue v at the tail of the queue. At the moment, u has already been dequeued. By the induction hypothesis the current head v_1 satisfies $d(u) \leq d(v_1)$. Therefore, $d(v)=d(u)+1 \leq d(v_1)+1$. Furthermore, the previous tail $d(v_r) \leq d(u)+1=d(v)$. Therefore, the lemma holds for all queue operations.

Which proves this lemma. □

> **Theorem 19:** *Let $G=(V,E)$ be an unweighted graph for any vertex $v \in V$ reachable from the source s, the distance $d[v]$ assigned by the BFS algorithm is equal to the shortest path distance $\delta(s,v)$.*

Proof. Using induction on the shortest path $\delta(s,v)=k$, because for $k=0$ the only vertex v such that $\delta(s,v)=0$ is s itself. BFS initialize $d[s]=0$ thus $d[s]=\delta(s,s)=0$. By induction, assume that forall vertices u with $\delta(s,u)=k$, BFS assign correctly $d[u]=k$, consider a vertex v such that $\delta(s,v)=k+1$. By definition of shortest distance there exists a vertex x such that $(u,v) \in E$ and $\delta(s,u)=k$. By inductive step $d[u]=k$. When x is dequeued, the algorithm examines all neighbors of v is undiscovered. BFS sets $d[v]=d[u]+1$. Since BFS explores vertices in non-decreasing order of distance, v cannot have been discovered at a distance smaller than $k+1$ (otherwise a shortest path to v would be contradicting $\delta(s,v)=k+1$. Thus, the first time v is reached, it is assigned $d[v]=k+1$. □

> **Exercise:** *The diameter of a tree $T=(V,E)$ is defined as*
>
> $$\max_{u,v \in V} \delta(u,v)$$
>
> *that is the longest of all shortest path distances in the tree. Give an efficient algorithm to compute the diameter of a tree and analyze its complexity.*

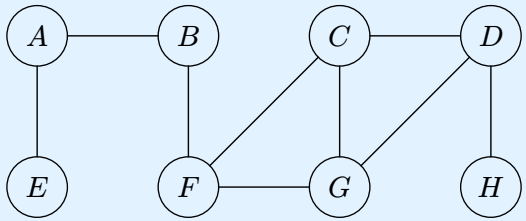
The algorithm is as follows:

- **Algorithm:** BFS-Modified(T, s)
  1. create a queue Q, and distances d
  2. initialize d for all vertices to 0
  3. d[s] ← 0
  4. Q.enqueue(s)
  5. while Q is not empty do
     1. v ← Q.dequeue()
     2. for each neighbor w of v do
        1. if d[w] = ∞ then
           1. d[w] ← d[v] = 1
           2. Q.enqueue(w)
        2. endif
     3. endfor
  6. endwhile
  7. let x be a vertex maximizing d[x]
  8. return $(x, \text{d}[x])$
- **Algorithm:** Tree-Diameter(T)
  1. chose an arbitrary vertex $v \in V$
  2. (u, _) ← BFS-Modified(T, s)
  3. (v, d) ← BFS-Modified(T, u)
  4. return d

The complexity of the pseudocode $O(n)$.

The BFS builds a BFS-Tree as it searches the graph. The tree corresponds to the p attributes. For a graph $G = (V, E)$ with sources. Let $G_p = (V_p, E_p)$ where $V_p = \{v \in V \mid p(v) \neq \text{Nil}\} \cup \{s\}$ and $E_p = \{(p(v), v) \mid v \in V_p \setminus \{s\}\}$. The predecessor subgraph G_q is a BFS tree if V_p consists of the vertices reachable from s and for all $v \in V_p$, the subgraph G_p contains a unique simple path from s to v, that is the shortest path from s to v in G. A BFS-Tree is in fact a tree, since it is connected by construction and $|E_p| = |V_p| - 1$.

> **Exercise:** *Show that if a graph is connected and it has $n - 1$ edges then the graph is a tree.*

> **Exercise:** *Run the BFS algorithm on the following graph and give the BFS tree starting from vertex A.*
>
> 
>

Going through the same procedure as the previous example to get