

# Algorithms II

## Graphs

### Definition

A graph  $G = (V, E)$  where  $E \subseteq V \times V$

- In an **undirected graph**, edges are unordered pairs.
- A **weighted graph** has a **weighting function**  $E \rightarrow \mathbb{R}$  which could be positive, zero or negative.
- A graph is **complete** if  $E = V \times V$

### Representations of a Graph

| Adjacency matrix  | Adjacency list                                      |
|---|---|
| A $ V  \times  V $ matrix $\Theta( V ^2)$ in size. <ul style="list-style-type: none"><li>• If unweighted, each cell stores a 0 or 1</li><li>• If weighted, stores the weight</li><li>• If undirected, it is symmetric, so only half of the matrix will need to be stored.</li></ul> | List of holding a linked list of adjacent vertices. |
| $O(1)$ to check $(u, v) \in E$ .  | $O( V )$ to check $(u, v) \in E$ .                  |
| $O( V )$ to list neighbours.  | $O(\text{neighbour count})$ to list neighbours.     |
| $O( V ^2)$ to iterate over edges.   | $O( E )$ to iterate over edges.                     |
| More compact for dense graphs (1 bit per edge)  | More compact for sparse graphs                      |

- The **transpose of a graph**  $G^T = (V, E^T)$  represents a **reverse index**.
- The **square of a graph**  $G^2 = (V, E^2)$  where  $(u, v) \in E^2$  if there is a path from  $u$  to  $v$  consisting of at most 2 edges.

Two vertices are adjacent if they share a vertex.

### Definition

An **induced subgraph**  $G' = (V', E')$  where  $V' \subseteq V$  is where

$$\forall u, v \in V' : (u, v) \in E \iff (u, v) \in E'$$

A **clique** in a graph is any induced subgraph that is **complete**.

| Colouring problem | Description  |
|-------------------|--|
| Vertex colouring  | Assign colours to $v \in V$ so no adjacent vertices have same colour.    |
| Edge colouring    | ... no adjacent edges have the same colour.                              |
| Face colouring    | ... no two adjacent faces on a <b>planar graph</b> have the same colour. |

### Definitions

- A **planar graph** can be drawn on a plane so no two lines intersect.
- A **face** is a region bounded by edges.

## Breadth First Search

To work on cyclic graphs, mark vertices we have visited to prevent us from visiting twice.

```

for v in G.V:
    v.marked = false

Q = new Queue
Enqueue(Q, s)

while !QueueEmpty(Q):
    u = Dequeue(Q)
    if (!u.marked):
        u.marked = true
        for v in G.E.adj[u]:
            Enqueue(Q, v)

```

This algorithm is inefficient because it may add the same vertex multiple times to the queue, to fix this add a **pending** flag for the element.

The flags are replaced by any data structures where membership can be tested.

## Two Vertex Colourability

Input: connected, undirected graph

1. Run BFS to colour the first level as red, 2nd as black, etc  $O(|V|)$
2. Check if every adjacent vertices are of different colour  $O(|E|)$

Total cost:  $O(|V| + |E|)$ , for a complete graph, it is  $O(|E|)$

### Note

- The algorithm doesn't matter wherever you run it from.
- If the graph is not connected, that part will not be explored.

## Single-source All-destination Shortest Path

- Input: unweighted graph and a starting node  $s$
- Output: distance and shortest path to all nodes

Run BFS with:

- Source distance = 0, path = [ ]
- The output of any unreachable vertices have distance  $\infty$

If average path length is  $O(|V|)$ , then output is  $O(|V|^2)$

The path to a node is given by repeatedly visiting  $v.\pi$  until  $v.\pi = \text{NIL}$

### Algorithm 1: SSAD\_HOPCOUNT

```

1: function SSAD_HOPCOUNT(G, s)
2:   for v in V do
3:     v.pending ← false
4:     v.d ← ∞
5:     v.π ← NIL
6:   end
7:
8:   s.pending ← true
9:   s.d ← 0
10:  s.π ← NIL
11:
12:  Q ← new queue

```

---

```

13:    $Q \leftarrow \text{enqueue } s$ 
14:
15:   while  $Q$  not empty do
16:      $u \leftarrow \text{dequeue } Q$ 
17:     for  $v$  in  $E.\text{adj}[u]$  do
18:       if not  $v.\text{pending}$  then
19:          $v.\text{pending} \leftarrow \text{true}$ 
20:          $v.d \leftarrow u.d + 1$ 
21:          $v.\pi \leftarrow u$ 
22:          $Q \leftarrow \text{enqueue } v$ 
23:       end
24:     end
25:   end
26: end

```

---

### Proof of Correctness

- Goal: when SSAD\_HOPCOUNT terminates,  $v.d$  is the length of the shortest path from  $s$  to  $v$ .
- Let  $\delta(s, v)$  be the actual shortest path length from  $s$  to  $v$ .

If there is no path from  $s$  to  $v$ ,  $\delta(s, v) = \infty$

#### Lemma: 1

If  $(u, v) \in E$  then  $\delta(s, v) \leq \delta(s, u) + 1$

- Case  $u$  is unreachable:  $\delta(s, u) = \infty$ , so inequality holds.
- Case  $u$  reachable:
  - If the shortest path is through  $u$ , then  $(u, v)$  is shorter than any other edge from  $u$  to  $v$
  - Otherwise  $\delta(s, v) < \delta(s, u) + \delta(u, v) = \delta(s, u) + 1$

#### Lemma: 2

On termination,  $v.d \geq \delta(s, v)$  for all  $v \in V$

Induction hypothesis:  $\forall v \in V : v.d \geq \delta(s, v)$

- Base case: before the first while loop
  - $\delta(s, s) = 0$  and  $s.d = 0$  for source
  - $v.d = \infty$  for all other nodes
- $v.d$  is only updated if  $v$  is not pending

$$\begin{aligned}
 v.d &= u.d + 1 \\
 &\geq \delta(s, u) + 1 \\
 &= \delta(s, v)
 \end{aligned}$$

$v.d$  is then never changed again.

#### Lemma: 3

Inductive hypothesis: if  $Q = v_1, v_2, \dots, v_x$ , then  $v_x.d \leq v_1.d + 1$  and  $v_i.d \leq v_{i+1}.d$  for all  $i$

Dequeue:

- If dequeuing leaves  $Q$  empty, then vacuous.
- Otherwise  $v_x.d \leq v_1.d \leq v_2.d$

Enqueue: the new  $v_{x+1}.d = v_1.d + 1$ , then  $v_{x+1}.d \leq v_1.d + 1$  and  $v_x \leq v_{x+1}$

### Corollary

If  $v_a$  is enqueued before  $v_b$ , then  $v_a.d \leq v_b.d$  on termination.

- If  $v_a$  and  $v_b$  are in  $Q$  simultaneously, then  $v_a \leq v_b$
- Otherwise, apply transitivity

Suppose the algorithm doesn't work, then there is a minimum  $\delta(s, v)$  that has an incorrect  $v.d$  upon termination. This means  $v.d > \delta(s, v)$

- $v$  must be reachable from  $s$ , otherwise  $\delta(s, v) = \infty \geq v.d$  contradicts  $v.d > \delta(s, v)$

Let  $u$  be the node on the shortest path from  $s$  to  $v$  that comes immediately before  $v$

- We know  $\delta(s, u) = u.d$  is correct
- $v.d > \delta(s, u) + 1 = u.d + 1$

When  $u$  is dequeued, either

1.  $v$  is not yet queued,  $v.d = u.d + 1$ , but this contradicts  $v.d > u.d + 1$
2.  $v$  is enqueued but not yet dequeued  $v.d = w.d + 1 \leq u.d + 1$ , again contradiction
3.  $v$  has already been dequeued, then  $v.d \leq u.d$ , contradiction

Therefore there is no *first time* the algorithm goes wrong, it must be correct.

### Note

$v.\pi$  traces a path of length  $v.d$ , which is the shortest path.

All edges  $(v.\pi, v)$  forms a **predecessor subgraph** of  $G$  called the **breadth-first tree**.

- $V_{PSG} = \{v \in V \mid v.\pi \neq \text{NIL}\} \cup \{s\}$
- $E_{PSG} = \{(v.\pi, v) \mid v \in V \setminus \{s\}\}$
- $PSG = (V_{PSG}, E_{PSG})$

### Depth First Search

1. Pick a random vertex
  2. Explores everything reachable
  3. Repeat until all vertices have been visited
-