BFS and DFS

Josh Wilcox (jw14g24@soton.ac.uk)

April 10, 2025

Contents

1 Graph Recap

- A graph $G = (V, E)$ is a set of vertices V and a set of edges E connecting the vertices
- Each element in E is a pair (v, w) with $v, w \in V$
- If the pairs are **ordered** the graph is **directed**
  - Else, the graph is **undirected**
- Usually a **weight** is assigned to each edge

1.1 Paths

- A **Path** is a sequence of vertices $w_1, \ldots, w_n$ such that $(w_i, w_i + 1) \in E$
- Length of the path is the number of edges in it
- A path is said to be simple if all vertices are distinct
  - Except possibly the first and last
- A cycle is a path such that $w_1 = w_n$
  - Each edge in an **undirected** graph needs to be distinct for it to be a cycle

1.2 Degree

- The degree of a vertex $deg(v)$ is the number of edges incident on v
- Digraphs have **indegree** and **outdegree**
- For undirected graphs:

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

- For directed graphs

$$\sum_{v \in V} indeg(v) = \sum_{v \in V} outdeg(v) = |E|$$

2 Representing Graphs

- Two ways of representing:
  - Adjacency Matrix
  - Adjacency List

2.1 Adjacency List

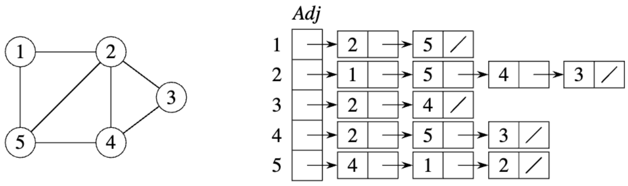
- Adjacency lists preffered due to their $\mathcal{O}(|E| + |V|)$ memory requirement
  - **Preferred** when the graph is **sparse**

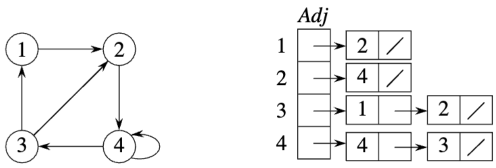
$$|E| << |V^2|$$

  - Remember the max number of edges in a graph is $n(n - 1)$

2.2 Adjacency Matrix

- $\mathcal{O}(|V^2|)$ in memory requirement
- Preferred when the graph is **dense**





Matrix

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

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

3 Breadth first search

- Start from the source s
  - Colour graph white except s which is black
- Discover all of the neighbours of s
- Given node `b` :
  - `v.d` is the distance (number of links) from `s` to `v`
  - `adj[v]` is the list of `v`'s neighbours
  - `v.p` is the predecessor of `v` in the path from `s` to `v`

3.1 BFS Initialisation

Algorithm 1 BFS Initialization

function BFSInit(G, s)
    for each $v \in V - \{s\}$ **do**
        v.color $\leftarrow$ WHITE
        v.d $\leftarrow$ 0
        v.p $\leftarrow$ NULL
    end for
    s.color $\leftarrow$ BLACK
    s.d $\leftarrow$ 0
    s.p $\leftarrow$ NULL
    $Q \leftarrow \emptyset$
    ENQUEUE(Q, s)
end function

- Initialize all vertices v in the graph G except the source s:
  - Set v.color to WHITE indicating they are unvisited.
  - Set v.d to 0, representing the distance from the source.
  - Set v.p to NULL, indicating no predecessor.
- Initialize the source vertex s:
  - Set s.color to BLACK indicating it is visited.

- Set s.d to 0, as the distance from itself is zero.
- Set s.p to NULL, as it has no predecessor.

- Initialize the queue Q as empty and enqueue the source vertex s.

3.2 BFS Pseudocode

Algorithm 2 BFS Main Algorithm with Time Complexity Comments

```
function BFS(G, s)
    BFSInit(G, s)                          ▷ O(|V|): Initialize each vertex
    while Q ≠ ∅ do                         ▷ O(|V|) iterations overall, since each vertex is enqueued once
        u ← DEQUEUE(Q)                     ▷ O(1) per dequeue
        for each v ∈ Adj[u] do             ▷ Total over all iterations: |adj[u]| = O(|E|)
            if v.color = WHITE then
                v.color ← BLACK            ▷ O(1)
                v.d ← u.d + 1              ▷ O(1)
                v.p ← u                    ▷ O(1)
                ENQUEUE(Q, v)              ▷ O(1) per enqueue
            end if
        end for
    end while
end function
```

- The algorithm begins by initializing every vertex in the graph (except the source) with a default "unvisited" status (color WHITE), a distance of 0, and no predecessor.
- The source vertex (s) is immediately marked as visited (color BLACK), its distance remains 0, and it has no predecessor. It is then enqueued to begin the search.
- The main loop continues as long as there are vertices in the queue (i.e., vertices waiting to be explored).
- Within the loop, the vertex at the front of the queue (u) is dequeued, meaning it is now being actively processed.
- For every neighbor v of the vertex u:
  - If v has not been visited (its color is WHITE), it is marked as visited (color BLACK).
  - The distance for v is set to one more than the distance for u, reflecting that v is one step farther from the source.
  - The predecessor of v is set to u, so the path information is maintained.
  - v is then enqueued, so its neighbors will be explored in later iterations.
- This ensures that the algorithm visits vertices in layers, where each layer represents vertices at an increasing distance from the source.
- The process repeats until the queue is empty, meaning all vertices reachable from the source have been visited.

3.3 BFS time complexity

- Each vertex is enqueued and dequeued **at most once**
- Since the dequeue operations are $\mathcal{O}(1)$, so the total cost of all the dequeues is $\mathcal{O}(|V|)$
- Whenever a vertex is dequeues, we look through the adjacency list, which has a size of $|E|$
- Therefore the total cost of BFS is $\mathcal{O}(|V| + |E|)$
  - In the very worst case, we access every edge once, and every node once

3.4 Bipartite Graphs

- A graph $G = (V, E)$ is bipartite if V can be partitioned into two sets such that

$$(u, v) \in E \implies u \in V_1 \text{ and } v \in V_2,$$

or

$$(u, v) \in E \implies u \in V_2 \text{ and } v \in V_1.$$

3.4.1 Bipartite Graph Detection using BFS

One can detect if a graph is bipartite using a small modification of BFS:

- Instead of coloring a node black when it is discovered, we color it either red or blue.
- The algorithm starts by coloring the source node red and leaving all other nodes white, just as in the standard BFS initialization.
- While exploring the neighbors of a node u, each unvisited neighbor is assigned the opposite color of u (i.e., $\neg$Blue = Red and $\neg$Red = Blue).
- If a neighbor is already colored with the same color as u, then the graph is not bipartite.

Algorithm 3 Bipartite Graph Detection using BFS

```
function IsBipartite(G, s)
    BFSInit(G, s) ▷ Initialize vertices: set color WHITE, distance 0, and no predecessor; set source with initial color.
    while Q ≠ ∅ do                     ▷ Continue while there are vertices to process.
        u ← DEQUEUE(Q)                 ▷ Fetch the next vertex to explore.
        for all v ∈ Adj[u] do          ▷ Iterate over all the neighbors of u.
            if v.color = WHITE then    ▷ If v has not been visited yet.
                v.color ← ¬u.color     ▷ Assign v the opposite color of u.
                v.d ← u.d + 1          ▷ Set the distance of v to be one more than u's distance.
                v.p ← u                ▷ Record u as the predecessor of v.
                ENQUEUE(Q, v)          ▷ Enqueue v to process its neighbors later.
            else if u.color = v.color then ▷ If v is already visited and has the same color as u.
                return false           ▷ A same-color adjacent pair found; graph is not bipartite.
            end if
        end for
    end while
    return true                        ▷ No conflicting colors found; graph is bipartite.
end function
```

3.5 BFS Shortest Path

- The shortest-length distance from source node $s \in V$ t0 $v \in V$ can be denoted as $\delta(s, v)$
  - It measures the minimum **number of edges** between these two nodes
- BFS discovers every vertex reachable from the source s, for each of these verticies $v.d = \delta(s, v)$
  - Remember $v.d$ is the distance from the source
- The shortest length path from s to v consists of the shortest length path from s to $v.p$
- Therefore, we can easily find the shortest path between two nodes by iterating backwards through $v.p$

4 Depth First Search

- Depth first search tries to go as "deep" as possible whenever possible

- When all the neighbours of a node v are discovered, we can't go any further therefore **backtrack** to the predecessor and explore other nodes
- When we are done discovering the descendants of some source s and some nodes remain undiscovered then one of them is selected as source and the process is repeated.
- A **single run** of DFS is guaranteed to **visit all nodes**

4.1 DFS Initialization

Algorithm 4 DFS Initialization

```
function DFS(G)
    for each v ∈ V do                    ▷ Initialize all vertices
        v.color ← WHITE                  ▷ Mark all vertices as undiscovered
        v.p ← NULL                       ▷ No predecessor yet
    end for
    time ← 0                             ▷ Global time counter for tracking discovery and finish times
    for each v ∈ V do                    ▷ Process each vertex
        if v.color = WHITE then          ▷ If vertex is undiscovered
            DFS-VISIT(G, v)              ▷ Explore from this vertex
        end if
    end for
end function
```

- Unlike BFS which starts from a single source, DFS can explore the entire graph including disconnected components:
  - First, all vertices are marked as WHITE (undiscovered).
  - Each vertex is initialized with a NULL predecessor.
  - We set up a global time counter to keep track of discovery and finish times.
  - We then loop through all vertices in the graph.
  - If a vertex is undiscovered (WHITE), we start DFS exploration from that vertex.
  - This outer loop ensures we reach all disconnected components of the graph.

4.2 DFS-VISIT Algorithm

Algorithm 5 DFS-VISIT Algorithm

```
function DFS-VISIT(G, u)
    u.color ← GRAY                       ▷ Mark vertex as discovered but not finished
    time ← time + 1                      ▷ Increment the discovery time counter
    u.d ← time                           ▷ Record when vertex was discovered
    for each v ∈ Adj[u] do               ▷ Explore all neighbors
        if v.color = WHITE then          ▷ If neighbor is undiscovered
            v.p ← u                      ▷ Record u as predecessor of v
            DFS-VISIT(G, v)              ▷ Recursively explore from v (go deeper)
        end if
    end for
    u.color ← BLACK                      ▷ Mark vertex as finished when all neighbors are processed
    time ← time + 1                      ▷ Increment the finishing time counter
    u.f ← time                           ▷ Record when vertex was finished
end function
```

- The DFS-VISIT function implements the recursive depth-first exploration:
  - When we visit a vertex, we first mark it GRAY (discovered but not finished).
  - We record its discovery time by incrementing the global time counter.

  - We then explore each undiscovered neighbor recursively, going as "deep" as possible.
  - When all neighbors of a vertex have been explored (or it has no neighbors), we mark it BLACK (finished).
  - Finally, we record its finishing time.
- The color scheme helps detect different types of edges:
  - WHITE: Tree edges - edges to previously undiscovered vertices.
  - GRAY: Back edges - edges to ancestors (indicates cycles in undirected graphs).
  - BLACK: Forward/cross edges - edges to already completed vertices.
- The discovery time ($u.d$) and finishing time ($u.f$) create an interval for each vertex:
  - For any two vertices u and v, their intervals are either nested or disjoint.
  - This property is useful for determining relationships between vertices.

5 Topological Sorting

- A linear ordering of a **directed** graph $G = (V, E)$ such that for all edges u, v, u becomes before v in the ordering
- Useful for the scheduling of tasks based on their dependencies

5.1 DFS for Topological Sorting

- Run DFS on the graph
- Every time the processing of a node is finished, put it to the **front** of a linked list
- When the algorithm terminates, the resulting list is the topological sorting

5.2 Kosaraju Algorithm

- Setup an empty set C for all nodes that have been placed in their respective connected component
- Run DFS the first time on all the vertices to get the total finishing time on each
- Reverse the direction of all of the edges in the graph
- For each vertex in $v_i \in V$ in **reverse order** of finishing time calculated from the first DFS:
- The **strongly connected components** are the newly discovered nodes after each pass