Table of Contents

1 Undirected Graph: The Basics

Definition 1.1: Graph

An (undirected) **graph** $G = (V, E)$ consists of two things:

- A collection V of vertices, or objects, to be connected.
- A collection E of edges, each of which connects a pair of vertices.

1.1 Exploring an Undirected Graph

We can make use of an `explore` algorithm to explore all vertices that can be reached from a particular vertex v. In particular, we can use a field `v.visited` to let us know which vertices we've already checked.

```
explore(v):
    v.visited <- true
    for each edge (v, w):
        if not w.visited:
            explore(w)
            w.prev <- v     // If we want to keep track of path taken
```

This algorithm runs in $\mathcal{O}(|V|)$ time. This is because we may potentially explore every vertex in the graph.

Theorem 1.1

If all vertices start unvisited, `explore(v)` marks as visited exactly the vertices reachable from `v`.

1.2 Depth First Search

`explore` only finds the part of the graph reachable from a single vertex. If you want to discover the entire graph, you may need to run it multiple times. This introduces an algorithm known as **depth-first search**:

```
DepthFirstSearch(v):
    Mark all v in G as unvisited.
    For v in G:
        if not v.visited:
            explore(v)
```

The final runtime is $\mathcal{O}(|V| + |E|)$.

1.3 Connected Components

Theorem 1.2

The vertices of a graph G can be partitioned into connected components so that v is is reachable from w if and only if they are in the same connected component.

We can use depth-first search to find the connected components:

```
explore(v):
    v.visited = true
    // CC is connected components
    v.CC = CCNum
    for each edge (v, w):
        if not w.visited:
```

```
            explore(w)

ConnectedComponents(G):
    CCNum = 0
    for each v in G:
        v.visited = false
    for each v in G:
        if not v.visited:
            CCNum++
            explore(v, CCNum)
```

This runs in $\mathcal{O}(|V| + |E|)$.

1.4 Pre- & Post-Order Numbers

We can augment DFS to keep track of what the algorithm does and how it does it. In particular, we can have a "clock" and note the time whenever:

- The algorithm visits a new vertex for the first time.
- The algorithm finishes processing a vertex.

These can be recorded as `v.pre` and `v.post`. The algorithm is as follows:

```
explore(v)
    v.visited = true
    v.pre = clock
    clock++
    For each edge (v, w)
        If not w.visited
            explore(w)
    v.post = clock
    clock++

DepthFirstSearch(G)
    clock = 1
    Mark all v in G as unvisited
    For v in G
        If not v.visited
            explore(v)
```

This runs in $\mathcal{O}(|V| + |E|)$.

1.4.1 Interpretation of Pre- & Post-Order Numbers

Proposition. *For vertices v, w, we can consider the intervals* `[v.pre, v.post]` *and* `[w.pre, w.post]`*. These intervals:*

1. *contain each other if v is an ancestor/descendant of w in the DFS tree.*
2. *are disjoint if v and w are cousins in the DFS tree.*
3. *never interleave (*`v.pre < w.pre < v.post < w.post`*).*

2 Directed Graph: The Basics

Definition 2.1

A **directed graph** is a graph where each edge has a direction. We say that it goes from v to w.

Often, we draw arrows on the edges to denote direction.

2.1 DFS on Directed Graph

We can use DFS on a directed graph. There are a few notes to consider:

- It's the same code.
- We only follow directed edges from v to w.
- The runtime is still $\mathcal{O}(|V| + |E|)$.
- `explore(v)` discovers all vertices reachable from v following only directed edges.

2.2 Topological Ordering

Essentially, a directed graph can be thought of as a graph of dependencies. An edge $v \mapsto w$ means that v should come before w. We can use something known as topological ordering to better understand this relationship.

Definition 2.2: Topological Ordering

A **topological ordering** of a directed graph is an ordering of the vertices so that for each edge (v, w), v comes before w in the ordering.

Remark: There can be multiple different topological orderings for the same graph.

2.2.1 Cycles

Definition 2.3: Cycle

A cycle in a directed graph is a sequence of vertices $v_1, v_2, \ldots, v_n$ so that there are edges:

$$(v_1, v_2), (v_2, v_3), \ldots, (v_n, v_1)$$

2.2.2 Obstacle

Proposition. *If G is a directed graph with a cycle, then G has no topological ordering.*

So, in other words, if G is a directed graph with at least one cycle *anywhere*, then it has no topological ordering.

2.2.3 Directed Acyclic Graph

Suppose we want to focus on directed graphs with no cycles. This brings us to the following definition:

Definition 2.4

A **directed acylic graph** (DAG) is a directed graph which contains no cycles.

Remark: Every DAG has a topological ordering.

2.2.4 Existence of Orderings

Theorem 2.1

Let G be a (finite) DAG. Then, G has a topological ordering.

Lemma 2.1

Every finite DAG contains at least one sink.

Remark:

- A **sink vertex** is a vertex with no outgoing edges.
- A **source vertex** is a vertex with no incoming edges.

2.2.5 Algorithm

The algorithm for finding a topological ordering of a directed acyclic graph G is:

```
TopologicalOrdering(G)
    Run DFS(G) w/ Pre/Post Numbers
    Return Vertices in Reverse Postorder
```

The runtime is $\mathcal{O}(|V| + |E|)$.

2.3 Strongly Connected Components

Definition 2.5: Strongly Connected Components

In a directed graph G, two vertices v and w are in the same **strongly connected component** if v is reachable from w and w is reachable from v.

2.3.1 Equivalence Relation

Let $v \sim w$ if v is reachable from w and vice versa.

Proposition. *This is an equivalence relation. Namely:*

- $v \sim v$ *(v is reachable from itself).*
- $v \sim w \implies w \sim v$ *(relation is symmetric).*
- $u \sim v$ *and* $v \sim w \implies u \sim w$.

Essentially, when we have this equivalence relation, we can split a set into components (equivalence classes) so that $v \sim w$ if and only if v and w are in the same component.

2.3.2 Metagraph

Definition 2.6: Metagraph

The **metagraph** of a directed graph G is a graph whose vertices are the strongly connected components of G, where there is an edge between C_1 and C_2 if and only if G has an edge between some vertex of C_1 and some vertex of C_2.

Remark: If you're given the strongly connected components and the metagraph of a graph G, then you can you can figure out connectivity within the full graph.

2.3.3 Result of the Metagraph

Theorem 2.2

The metagraph of any directed graph is a DAG.

2.3.4 Computing Strongly Connected Components

Given a directed graph G, compute the SCCs of G and its metagraph.

- Find v in a sink SCC of G.
- Run `explore(v)` to find the component C_1.
- Repeat process on $G \setminus C_1$.

2.3.5 Result

Proposition. *Let C_1 and C_2 be SCCs of G with an edge from C_1 to C_2. If we run `DFS` on G, the largest postorder number of any vertex in C_1 will be larger than the largest postorder number in C_2.*

The reason why we care is because if v is the vertex with the largest postorder number, then:

- There is no edge to $SCC(V)$ from any other SCC.
- SCC is a source SCC.

However, we wanted a *sink* SCC. So, how do we relate these two?

- A sink is like a source, only with edges going in the opposite direction.

2.3.6 Reverse Graph

Definition 2.7: Reverse Graph

Given a directed graph G, the **reverse graph** of G (denoted G^R) is obtained by reversing the directions of all the edges of G.

Some properties of reverse graphs are:

- G and G^R have the same number of vertices and edges.
- $G = (G^R)^R$.
- G and G^R have the same SCCs.
- The sink SCCs of G are the source SCCs of G^R.
- The source SCCs of G are the sink SCCs of G^R.

2.3.7 Algorithm

```
SCCs(G)
    Run DFS(G^R), record postorders
    Mark all vertices as unvisited
    For v in V in reverse postorder
        If v not in a component yet     // if v is not visited
            explore(v) on G-components found,
            marking new component
```

This runs in $\mathcal{O}(|V| + |E|)$ time.

3 Breadth-First Search

Given a graph G with two vertices s and t in the same connected component, how do we find the *best* path from s to t? In fact, what do we mean by the best path?

- Least expensive.
- Best scenery.
- Shortest.

For now, we want the **fewest edges**.

3.1 Observation

Proposition. *If there is a path from vertices s to v with length at most d, then there is some w adjacent to v where there is a path a length at most $(d-1)$ from vertices s to w.*

3.2 Algorithm

```
BFS(G, s)
    For v in V, dist(v) = infinity
    Initialize Queue Q
    Q.enqueue(s)
    dist(s) = 0
    While Q is not empty
        u = front(Q)
        For (u, v) in E
            If dist(v) = infinity
                dist(v) = dist(u) + 1
                Q.enqueue(v)
                v.prev = u
```

- This runs in $\mathcal{O}(|V|+|E|)$ time.
- $d(v)$ means the distance of vertex v whereas $d(v,w)$ means the length of the edge between v and w.

3.3 Differences Between DFS and BFS

- Similarities:
  - The way both algorithms process vertices is the same (`visited` for DFS vs. `dist < infinity` for BFS).
  - For each vertex, process all unprocessed neighbors.
- Differences:
  - DFS uses a stack to store vertices waiting to be processed.
  - BFS uses a queue.
- Big Effect:
  - DFS goes depth-first: very long path. Get a very "skinny" tree.
  - BFS is breadth first: visits all side paths. Get a very shallow tree since we process all of the neighbors.

4 Key Algorithms

Below are some key algorithms that we have discussed in lecture.

4.1 Algorithm: Explore

Category	**Answer**
Graph Type	Undirected, Directed
Runtime	$\mathcal{O}(\|V\| + \|E\|)$

- Explore a particular (strongly) connected component, or the entire graph if the graph is connected.

```
explore(v):
    v.visited <- true
    for each edge (v, w):
        if not w.visited:
            explore(w)
            w.prev <- v     // If we want to keep track of path taken
```

4.2 Algorithm: Depth-First Search

Category	**Answer**
Graph Type	Undirected, Directed
Runtime	$\mathcal{O}(\|V\| + \|E\|)$

```
DepthFirstSearch(v):
    Mark all v in G as unvisited.
    For v in G:
        if not v.visited:
            explore(v)
```

4.3 Algorithm: Connected Components

Category	**Answer**
Graph Type	Undirected
Runtime	$\mathcal{O}(\|V\| + \|E\|)$

- Find all connected components of an undirected graph.

```
explore(v, CCNum):
    v.visited = true
    // CC is connected components
    v.CC = CCNum
    for each edge (v, w):
        if not w.visited:
            explore(w)

ConnectedComponents(G):
    CCNum = 0
    for each v in G:
        v.visited = false
    for each v in G:
        if not v.visited:
            CCNum++
```

```
                    explore(v, CCNum)
```

4.4 Algorithm: Pre- & Post-Order Numbers

Category	Answer
Graph Type	Undirected, Directed
Runtime	$\mathcal{O}(\|V\| + \|E\|)$

- For a directed graph G, for any DFS on G, the vertex with highest post-order number lies in a source SCC. Regardless of where you start counting, the vertex with the largest post-order number in a directed graph is *a* source[1].
- If G is our graph and G^R is the reverse graph, then the vertex with the highest post-order number in the reverse graph is the *sink* of G.

```
explore(v)
    v.visited = true
    v.pre = clock
    clock++
    For each edge (v, w)
        If not w.visited
            explore(w)
    v.post = clock
    clock++

DepthFirstSearch(G)
    clock = 1
    Mark all v in G as unvisited
    For v in G
        If not v.visited
            explore(v)
```

4.5 Algorithm: Topological Ordering

Category	Answer
Graph Type	Directed (Acyclic)
Runtime	$\mathcal{O}(\|V\| + \|E\|)$

```
TopologicalOrdering(G)
    Run DFS(G) w/ Pre/Post Numbers
    Return Vertices in Reverse Postorder
```

4.6 Algorithm: Strongly Connected Components

Category	Answer
Graph Type	Directed
Runtime	$\mathcal{O}(\|V\| + \|E\|)$

- Used to compute the metagraph, which can then be used to find the source/sink components.

[1] Alternatively, a source vertex does not have any incoming edges.

```
SCCs(G)
    Run DFS(G^R), record postorders
    Mark all vertices as unvisited
    For v in V in reverse postorder
        If v not in a component yet     // if v is not visited
            explore(v) on G-components found,
            marking new component
```

4.7 Algorithm: Breadth-First Search

Category	**Answer**
Graph Type	Undirected, Directed
Runtime	$\mathcal{O}(\lvert V\rvert + \lvert E\rvert)$

- Finds the shortest distance between two vertices, assuming the edges are unweighted.
- Define $d(v)$ to be the (shortest) length of the path from the starting vertex to v.
- Define $d(v, w)$ to be the (shortest) length of the path from v to w.

```
BFS(G, s)
    For v in V, dist(v) = infinity
    Initialize Queue Q
    Q.enqueue(s)
    dist(s) = 0
    While Q is not empty
        u = front(Q)
        For (u, v) in E
            If dist(v) = infinity
                dist(v) = dist(u) + 1
                Q.enqueue(v)
```