

## Graphs Cheatsheet

### Depth First Search (DFS)

---

```

def dfs(G, s):
    def explore_recursive(G, v):
        visited(v) = true
        previsit(v) # set the pre-order of v
        for each edge (v, u) in E:
            if not visited(u):
                explore_recursive(G, u)
        postvisit(v) # set the post-order of v

    def explore_iterative(G, v):
        st = stack()
        st.push(v)

        while st is not empty:
            u = st.pop()
            visited(u) = true

            for each edge (u, w) in E:
                if not visited(w):
                    st.push(w)

    # depending on how you want to DFS, you can use
    # either explore_recursive or explore_iterative below
    explore(G, s)
    for all v in V:
        if not visited(v):
            explore(G, v)

```

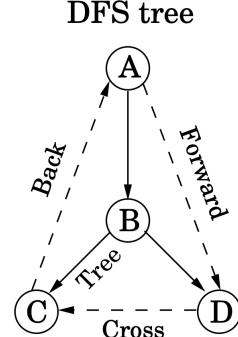
---

→ **Runtime of DFS:**  $O(|V| + |E|)$

**DFS Tree/Forest:** the tree/forest produced by the edges traversed during a given DFS

#### Edge Types:

- *Tree Edge:* leads to child; part of the DFS Tree/Forest
- *Forward Edge:* leads to a non-child descendant
- *Back Edge:* leads to an ancestor
- *Cross Edge:* leads to a node that's neither a descendant nor an ancestor



**Edge Type based on Pre/Post-orders:** an edge  $(u, v) \in E$  is a:

- *Tree or Forward Edge* if  $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$
- *Back Edge* if  $\text{pre}(v) < \text{pre}(u) < \text{post}(u) < \text{post}(v)$
- *Cross Edge* if  $\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$

## Topological Sort (Graph Linearization)

---

```
# Returns the topological order of vertices in G (acyclic)
def topo_sort(G):
    topo_order = []
    def explore(G, v):
        visited(v) = true
        for each edge (v, u) in E:
            if not visited(u):
                explore(G, u)
        topo_order.append(v) # note that topological order is reverse post-order!

    s = any arbitrary node in G
    explore(s)
    for all v in V:
        if not visited(v):
            explore(G, v)

    return topo_order[::-1]
```

---

## Breadth First Search (BFS)

---

```
def bfs(G, s):
    q = queue()
    q.push(v)

    while q is not empty:
        v = q.pop()
        visited(v) = true

        for each edge (v, u) in E:
            if not visited(u):
                q.push(u)
```

---

↪ **Runtime of BFS:**  $O(|V| + |E|)$

## Strongly Connected Components

A **strongly connected component** of  $G$  is a subset of vertices in which there is a path from every vertex to every other vertex.

### Kosaraju's Algorithm

Given a graph  $G = (V, E)$ , we can find all the SCCs as follows:

1. Run DFS on  $G^{\text{rev}}$  to get the post-order values of all vertices  $v \in V$ ; i.e. we compute  $\text{post}^{\text{rev}}(v)$  for all  $v \in V$ .
2. Run DFS on  $G$  starting at the vertex with the highest post-order in  $G^{\text{rev}}$  (that is unvisited), which must belong in the sink SCC of  $G$ . Throughout this DFS, we label each traversed vertex as part of the current SCC.
3. Repeat steps 2-3 until we've labeled all SCCs.

→ **Runtime of Kosaraju's:**  $O(|V| + |E|)$

### Dijkstra's Algorithm

→ Given a graph  $G$  with non-negative edge weights  $w(\cdot)$ , finds the shortest path lengths from  $s$  to all vertices

---

```
def dijkstra(G, s):
    for all v in V:
        dist(v) = infinity # distances
        par(v) = none # parents in shortest paths tree

    dist(s) = 0
    h = min_heap() # priority according to distance
    h.insert((s, 0))

    while h is not empty:
        v = h.delete_min()
        for each edge (v, u) in E:
            if dist(u) > dist(v) + w(v, u):
                dist(u) = dist(v) + w(v, u)
                par(u) = v
                h.decrease_key(u) # sets priority of u to be the updated dist(u)

    return dist, par
```

---

→ **Runtime of Dijkstra's:**

- $O((|E| + |V|) \log |V|)$  using a binary min-heap
- $O(|E| + |V| \log |V|)$  using a fibonacci min-heap

## Bellman-Ford Algorithm

→ Given a graph  $G$  (with possibly negative edge weights!), finds the shortest path lengths from  $s$  to all vertices

---

```

def bellman_ford(G, s):
    for all v in V:
        dist(v) = infinity
        par(v)

    # update all distances
    dist(s) = 0
    repeat |V| - 1 times:
        for each edge (v, u) in E:
            if dist(u) > dist(v) + w(v, u):
                dist(u) = dist(v) + w(v, u)
                par(u) = v

    # negative-cycle detection by trying to update a |V|th time
    for each edge (v, u) in E:
        if dist(u) > dist(v) + w(v, u): # detected negative-cycle!
            return None, None

    return dist, par

```

---

→ **Runtime of Bellman-Ford:**  $O(|V||E|)$ .