Graphs made of:

**Vertices**
- $\leq 0$ per graph
- nodes, concepts, objects

**Edges**
- $\leq 0$ per graph
- links, connections, associates, relationships
- Connects two vertices or vertex to itself (**loop**)

**path** sequence of consecutive edges from one vertex to another

**undirected (bidirectional) graphs** symmetric

**directed graphs**

**Weights**
**weights** distance/cost between vertices

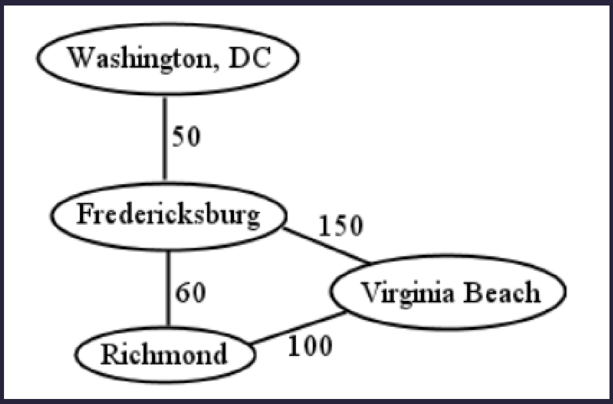


Figure 1: Weighted, undirected graph

**adjacent**

**connected**
- (vertices) at least one path between vertices
- (entire graphs) *every* node can be reached by others

**degree** number of edges connected to the vertex; in-degree/out-degree for arrows in/out

**cycle** path that begins and ends at same vertex

**DAG (directed, acyclic graph)**
- graph of dependencies
- **directed** means other nodes part of path prior to it

**acyclic** *no* kind of cycle can exist in the graph

## Breadth-first traversal
- Uses queue (FIFO)
- Finish level 1 first, then go to level 2
- Visit all direct children first *once*, then visit all second children *once*, etc.

**Queue Data Structure**

- Enque (add) node at q[−1]
- Deque (remove) node from q[0]

**Steps**

1. Mark and queue first node

While queue is not empty:

1. Pop from front
2. Do stuff
3. Mark as visited
4. Queue all unvisited adjacent nodes
5. Repeat

```
def bfs(root: Vertex):
    visited = []
    q = []

    while len(q) > 0:
        cur = q.pop()

        do_stuff(cur)
        for neighbor in graph[cur]:
            if neighbor not in visited:
                visited.append(neighbor)
                queue.append(neighbor)
```

## Depth-first traversal

- Go as deep as you can first, then go to next
- Uses **stack**

**Steps**

1. Mark and push first node onto stack

While stack is not empty:

1. pop(-1)
2. Do Stuff
3. Add all neighbors to visited

**Stack Data Structure**

- LIFO
- push_back()
- pop(-1)

```
def dfs(root_node):
    visited = set()
    stack = []
    stack.append(root_node)

    while len(stack) > 0:
        s = stack.pop(-1)

        if s not in visited:
            do_stuff(s)
            visited.add(s)
```

```
        for neighbor in s.neighbors:
            if neighbor not in visited:
                stack.append(neighbor)
```

## Dijkstra’s shortest-path algorithm

- Shortest path between just two nodes

## Prim’s minimal connecting edge set algorithm

- Shorted path between all nodes

```
while !allConnected:
    connect(closest_non_connected)
```