Graph Vocabulary

Graph Direction
- **Undirected graph** – edges have no direction and are two-way
  \[ V = \{ A, B, C \} \]
  \[ E = \{ (A, B), (B, C) \} \text{ inferred } (B, A) \text{ and } (C, B) \]

- **Directed graphs** – edges have direction and are thus one-way
  \[ V = \{ A, B, C \} \]
  \[ E = \{ (A, B), (B, C), (C, B) \} \]

Degree of a Vertex
- **Degree** – the number of edges containing that vertex
  \[ A : 1, B : 2, C : 1 \]
- **In-degree** – the number of directed edges that point to a vertex
  \[ A : 0, B : 2, C : 1 \]
- **Out-degree** – the number of directed edges that start at a vertex
  \[ A : 1, B : 1, C : 1 \]
Graph Vocabulary

Dense Graph – a graph with a lot of edges
$E \in \Theta(V^2)$

Sparse Graph – a graph with “few” edges
$E \in \Theta(V)$
Graph Vocabulary

Self loop – an edge that starts and ends at the same vertex

Parallel edges – two edges with the same start and end vertices

Simple graph – a graph with no self-loops and no parallel edges
**Graph Vocabulary**

**Walk** – A sequence of adjacent vertices. Each connected to next by an edge.

- $A, B, C, D$ is a walk.
- So is $A, B, A$

**Directed Walk** – must follow the direction of the edges

- $A, B, C, D, A$ is a directed walk.
- $A, B, A$ is not.

**Length** – The number of edges in a walk

- $(A, B, C, D)$ has length 3.
Graph Vocabulary

Path – A walk that doesn’t repeat a vertex. $A,B,C,D$ is a path. $A,B,A$ is not.

Cycle – path with an extra edge from last vertex back to first.

Be careful looking at other sources.
Some people call our “walks” “paths” and our “paths” “simple paths”
Use the definitions on these slides.
Paths and Reachability

Common questions:

- Is there a path between two vertices? (Can I drive from Seattle to LA?)
- What is the length of the shortest path between two vertices? (How long will it take?)
- List vertices that can reach the maximum number of nodes with a path of length 2.
- Can every vertex reach every other on a short path?
  - Length of the longest shortest path is the “diameter” of a graph
1. Traversal A-B-C-D-B-E is a ____________ (Walk, Path, Neither)

2. Traversal B-E-A-B is a ____________ (Walk, Path, Neither)

3. __________ does not contain a cycle. (Walk, Path)

4. Degree of Node B is __________
Implementing a Graph

Two main ways to implement a graph:

1. Adjacency Matrix
2. Adjacency List
Adjacency Matrix

Assign each vertex a number from 0 to \( V - 1 \)

Create a \( V \times V \) array of Booleans (or Int, as 0 and 1)

If \((x,y) \in E\) then \( \text{arr}[x][y] = \text{true} \)

Time complexity (in terms of \( V \) and \( E \))
- Get in–edges:
  - \( O(|V|) \)
- Get out–edges:
  - \( O(|V|) \)
- Decide if an edge \((u, w)\) exists:
  - \( O(1) \)
- Insert an edge:
  - \( O(1) \)
- Delete an edge:
  - \( O(1) \)

Space complexity:
- \( O(|V|^2) \)
Adjacency Matrix

Assign each vertex a number from 0 to \(V - 1\)

Create a \(V \times V\) array of Booleans (or Int, as 0 and 1)

If \((x,y) \in E\) then \(arr[x][y] = true\)

Time complexity (in terms of \(V\) and \(E\))
- Get in-edges: \(O(|V|)\)
- Get out-edges: \(O(|V|)\)
- Decide if an edge \((u, w)\) exists: \(O(1)\)
- Insert an edge: \(O(1)\)
- Delete an edge: \(O(1)\)

Space complexity: \(O(|V|^2)\)
Create a Dictionary of size $V$ from type $V$ to Collection of $E$

If $(x,y) \in E$ then add $y$ to the set associated with the key $x$

Time complexity:
- Get in-edges: $O(|V| + |E|)$
- Get out-edges: $O(|V|)$
- Decide if an edge $(u, w)$ exists:
- Insert an edge:
- Delete an edge:

Space complexity:
Create a Dictionary of size $V$ from type $V$ to Collection of $E$

If $(x,y) \in E$ then add $y$ to the set associated with the key $x$

Time complexity
- Get in-edges: $O(|V| + |E|)$
- Get out-edges: $O(1)$
- Decide if an edge $(u, w)$ exists: $O(1)$
- Insert an edge: $O(1)$
- Delete an edge: $O(1)$

Space complexity: $O(|V| + |E|)$
Graph Traversal

Traversing a tree:
start at root,
visit children in order

Traversing a graph:
Where to start?
Which nodes to visit and in which order?
Traversing a graph

Three collections: ‘unvisited’, ‘visited’, and ‘to be visited’. And a pointer to the ‘current’ vertex
1. Pick any vertex to start. The vertex you are currently processing is your ‘current’ vertex.
2. Put all neighbors of the current vertex in a “to be visited” collection
3. Mark the current vertex “visited”
4. Move onto next vertex in “to be visited” collection
5. Put all unvisited neighbors in “to be visited”
6. Move onto next vertex in “to be visited” collection
7. Repeat...
Traversing a graph

Breadth first search

Depth first search
BFS and DFS on Trees

Breadth first search
(Level order traversal)

Depth first search
(pre-order, in-order, post-order traversals)
Breadth First Search

```java
search(graph)
    toVisit.enqueue(first vertex)
    mark first vertex as visited
    while(toVisit is not empty)
        current = toVisit.dequeue()
        for (V : current.neighbors())
            if (v is not visited)
                toVisit.enqueue(v)
                mark v as visited
        finished.add(current)
```

Current node: I
Queue: B D E C F G H I
Finished: A B D E C F G H I
Breadth First Search Analysis

search(graph)
    toVisit.enqueue(first vertex)
    mark first vertex as visited
    while(toVisit is not empty)
        current = toVisit.dequeue()
        for (V : current.neighbors())
            if (v is not visited)
                toVisit.enqueue(v)
                mark v as visited
        finished.add(current)

Visited: A B D E C F G H I

How many times do you visit each node? 1 time each
How many times do you traverse each edge? Max 2 times each
- Putting them into toVisit
- Checking if they’re visited

Runtime? $O(V + 2E) = O(V + E)$ “graph linear”
Depth First Search (DFS)

BFS uses a queue to order which vertex we move to next

Gives us a growing “frontier” movement across graph

Can you move in a different pattern? Can you use a different data structure?

What if you used a stack instead?

\[
\text{dfs}(\text{graph})
\]

\[
\text{bfs}(\text{graph})
\]

```python
# functions toVisit.push(first vertex)
toVisit.push(first vertex)
mark first vertex as visited
while (toVisit is not empty)
current = toVisit.pop()
for (V : current.neighbors())
    if (V is not visited)
        toVisit.push(V)
        mark V as visited
finished.add(current)
```

```python
# functions

toVisit.enqueue(first vertex)
mark first vertex as visited
while (toVisit is not empty)
current = toVisit.dequeue()
for (V : current.neighbors())
    if (v is not visited)
        toVisit.enqueue(v)
        mark v as visited
finished.add(current)
```
Depth First Search

dfs(graph)
  toVisit.push(first vertex)
  mark first vertex as visited
  while(toVisit is not empty)
    current = toVisit.pop()
    for (V : current.neighbors())
      if (V is not visited)
        toVisit.push(v)
        mark v as visited
      finished.add(current)

How many times do you visit each node? 1 time each
How many times do you traverse each edge? Max 2 times each
- Putting them into toVisit
- Checking if they’re visited

Runtime? $O(V + 2E) = O(V + E)$ “graph linear"
1. Write the BFS traversal when starting at node A:

2. Write the DFS traversal when starting at node A:
BFS and DFS

- BFS and DFS have the same worst-case time and space complexity
- The differ in terms of the order they visit the vertices and as a result they provide some additional information about the graph. For example:
  - BFS is useful to find shortest paths
  - DFS is useful to check the presence of a cycle