What is a graph?

A graph is a set of vertices $V$ and a set of edges $(u,v) \in E$ where $u,v \in V$. 
Graphs

How do graphs differ? What are graph characteristics we might care about?

Different types of graphs

Undirected – edges do not have a direction

Directed – edges do have a direction

Weighted – edges have an associated weight
Different types of graphs

Weighted – edges have an associated weight

Terminology

Path – A path is a list of vertices $p_1, p_2, \ldots, p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$
A path is a list of vertices $p_1, p_2, \ldots, p_k$ where there exists an edge $(p_i, p_{i+1}) \in E$.

A simple path contains no repeated vertices (often this is implied).

Cycle – A subset of the edges that form a path such that the first and last node are the same.

Edges: B-A, A-D, D-B
Path: B, A, D, B

Not a cycle
Terminology

Cycle – A subset of the edges that form a path such that the first and last node are the same

Does this graph have a cycle?

Terminology

Cycle – A subset of the edges that form a path such that the first and last node are the same

not a cycle

Terminology

Cycle – A path $p_1, p_2, ..., p_k$ where $p_1 = p_k$

Connected – every pair of vertices is connected by a path

Is this graph connected?
Terminology

Connected – every pair of vertices is connected by a path

Is this graph connected?

Terminology

Not connected

Strongly connected (directed graphs) – Every two vertices are reachable by a path

Is this graph strongly connected?
Strongly connected (directed graphs) — Every two vertices are reachable by a path

Is this graph strongly connected?

A
B
C
D
E
F
G

not strongly connected

A
B
C
D
E
F
G
Terminology

Strongly connected (directed graphs) —
Every two vertices are reachable by a path

Different types of graphs

What is a tree (in our terminology)?

Different types of graphs

Tree — connected, undirected graph without any cycles

Different types of graphs

Tree — connected, undirected graph without any cycles

need to specify root
Different types of graphs

Tree – connected, undirected graph without any cycles

DAG – directed, acyclic graph

Complete graph – an edge exists between every node

Bipartite graph – a graph where every vertex can be partitioned into two sets $X$ and $Y$ such that all edges connect a vertex $u \in X$ and a vertex $v \in Y$
When do we see graphs in real life problems?

- Transportation networks (flights, roads, etc.)
- Communication networks
- Web
- Social networks
- Circuit design
- Bayesian networks

Representing graphs

**Adjacency list** — Each vertex \( u \in V \) contains an adjacency list of the set of vertices \( v \) such that there exists an edge \((u,v) \in E\)

```
A: B
B: A D
C: D
D: A B C E
E: D
```

```
A: B
B: A D
C: D
D: A B C E
E: D
```
Representing graphs

Adjacency matrix – A \(|V| \times |V|\) matrix A such that:

\[ a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases} \]

\[
\begin{array}{c|ccccc}
A & B & C & D & E \\
\hline
A & 0 & 1 & 0 & 1 & 0 \\
B & 1 & 0 & 0 & 1 & 0 \\
C & 0 & 0 & 0 & 1 & 0 \\
D & 1 & 1 & 1 & 0 & 1 \\
E & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]
Representing graphs

Adjacency matrix – $A_{|V| \times |V|}$ matrix $A$ such that:

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Is it always symmetric?

Adjacency list vs. adjacency matrix

Adjacency list
- Sparse graphs (e.g. web)
- Space efficient
- Must traverse the adjacency list to discover if an edge exists

Adjacency matrix
- Dense graphs
- Constant time lookup to discover if an edge exists
- Simple to implement
- For non-weighted graphs, only requires boolean matrix

Can we get the best of both worlds?

Sparse adjacency matrix

Rather than using an adjacency list, use an adjacency hashtable

A: hashtable [B,D]
B: hashtable [A,D]
C: hashtable [D]
D: hashtable [A,B,C,E]
E: hashtable [D]
Sparse adjacency matrix

- Constant time lookup
- Space efficient
- Not good for dense graphs, why?

Weighted graphs

- Adjacency list
  - store the weight as an additional field in the list

Graph algorithms/questions

- Graph traversal (BFS, DFS)
  - Shortest path from a to b
    - unweighted
    - weighted positive weights
    - negative/positive weights
  - Minimum spanning trees
  - Are all nodes in the graph connected?
  - Is the graph bipartite?
Breadth First Search (BFS) on Trees

TREEBFS(T)
1 ENQUEUE(Q, ROOT(T))
2 while !EMPTY(Q)
3 v ← DEQUEUE(Q)
4 VISIT(v)
5 for all c ∈ CHILDREN(v)
6 ENQUEUE(Q, c)

Tree BFS

TREEBFS(T)
1 ENQUEUE(Q, ROOT(T))
2 while !EMPTY(Q)
3 v ← DEQUEUE(Q)
4 VISIT(v)
5 for all c ∈ CHILDREN(v)
6 ENQUEUE(Q, c)

Q: A
Visited:

Tree BFS

Q: A
Visited: A
Tree BFS

TreeBFS(T)
1 ENQUEUE(Q, Root(T))
2 while ¬EMPTY(Q)
3 v ← DEQUEUE(Q)
4 VISIT(e)
5 for all c ∈ CHILDREN(e)
6 ENQUEUE(Q, c)

Q: B, D, E
Visited: A

57

Tree BFS

TreeBFS(T)
1 ENQUEUE(Q, Root(T))
2 while ¬EMPTY(Q)
3 v ← DEQUEUE(Q)
4 VISIT(e)
5 for all c ∈ CHILDREN(e)
6 ENQUEUE(Q, c)

Q: D, E
Visited: A B

58

Tree BFS

TreeBFS(T)
1 ENQUEUE(Q, Root(T))
2 while ¬EMPTY(Q)
3 v ← DEQUEUE(Q)
4 VISIT(e)
5 for all c ∈ CHILDREN(e)
6 ENQUEUE(Q, c)

Q: D, E, C, F
Visited: A B

59

Tree BFS

TreeBFS(T)
1 ENQUEUE(Q, Root(T))
2 while ¬EMPTY(Q)
3 v ← DEQUEUE(Q)
4 VISIT(e)
5 for all c ∈ CHILDREN(e)
6 ENQUEUE(Q, c)

Q: E, C, F
Visited: A B D

60
Q: E, C, F
Visited: A B D
Tree BFS

What order does the algorithm traverse the nodes?

BFS traversal visits the nodes in increasing distance from the root.

Tree BFS

What order does the algorithm traverse the nodes?

BFS traversal visits the nodes in increasing distance from the root.

Running time of Tree BFS

Adjacency list
- How many times does it visit each vertex?
- How many times is each edge traversed?
- \( \Theta(|V| + |E|) \) – for trees, i.e., assuming a connected graph

Adjacency matrix
- For each vertex visited, how much work is done?
- \( \Theta(|V|^2) \) – for trees, i.e., assuming a connected graph

Tree BFS

Does it visit all of the nodes?

If the graph is connected or strongly connected.

Tree BFS

The running time of Tree BFS is as follows:

1. \( \text{ENQUEUE}(Q, \text{ROOT}(T)) \)
2. \( \text{while} \ (\text{EMPTY}(Q)) \)
3. \( v \leftarrow \text{DEQUEUE}(Q) \)
4. \( \text{VISIT}(v) \)
5. \( \text{for all } c \in \text{CHILDREN}(v) \)
6. \( \text{ENQUEUE}(Q, c) \)
BFS Recursively

Hard to do!

TreeBFS(T)
1 Enqueue(Q, Root(T))
2 while !Empty(Q)
3 e ← Dequeue(Q)
4 Visit(e)
5 for all c ∈ Children(e)
6 Enqueue(Q, c)

BFS for graphs

What needs to change for graphs?

Need to make sure we don’t visit a node multiple times

A B D E C F G

BFS for graphs

What order will BFS visit starting at A (break ties to visit based alphabetically, with earlier first)?

A B D E C F G

BFS for graphs

What order will BFS visit starting at A (break ties to visit based alphabetically, with earlier first)?

A B D E C F G
BFS(G, s)
1 for each v ∈ V 
2 dist[v] = ∞
3 dist[s] = 0
4 ENQUEUE(Q, s)
5 while NOT EMPTY(Q)
6 s ← DEQUEUE(Q)
7 VISIT(s)
8 for each edge (s, v) ∈ E 
9 if dist[v] = ∞
10 ENQUEUE(Q, v)
11 dist[v] ← dist[s] + 1

---

The BFS(T)
1 ENQUEUE(Q, Root(T))
2 while NOT EMPTY(Q)
3 v ← DEQUEUE(Q)
4 VISIT(v)
5 for all e ∈ CHILDREN(v)
6 ENQUEUE(Q, e)

---

BFS(G, s)
1 for each v ∈ V 
2 dist[v] = ∞
3 dist[s] = 0
4 ENQUEUE(Q, s)
5 while NOT EMPTY(Q)
6 u ← DEQUEUE(Q)
7 VISIT(u)
8 for each edge (u, v) ∈ E 
9 if dist[v] = ∞
10 ENQUEUE(Q, v)
11 dist[v] ← dist[u] + 1

---

BFS(G, s)
1 for each v ∈ V 
2 dist[v] = ∞
3 dist[s] = 0
4 ENQUEUE(Q, s)
5 while NOT EMPTY(Q)
6 u ← DEQUEUE(Q)
7 VISIT(u)
8 for each edge (u, v) ∈ E 
9 if dist[v] = ∞
10 ENQUEUE(Q, v)
11 dist[v] ← dist[u] + 1

---

set all nodes as unseen

check if the node has been seen
BFS(G, s)
1 for each v ∈ V
2 dist[v] = ∞
3 dist[s] = 0
4 Enqueue(Q, s)
5 while !Empty(Q)
6 s = Dequeue(Q)
7 Visit(s)
8 for each edge (u, v) ∈ E
9 if dist[u] = ∞
10 Enqueue(Q, u)
11 dist[v] = dist[u] + 1

Q: A

BFS(G, s)
1 for each v ∈ V
2 dist[v] = ∞
3 dist[s] = 0
4 Enqueue(Q, s)
5 while !Empty(Q)
6 u = Dequeue(Q)
7 Visit(u)
8 for each edge (u, v) ∈ E
9 if dist[v] = ∞
10 Enqueue(Q, v)
11 dist[v] = dist[u] + 1

Q: B
BFS(G, s)
1 for each v ∈ V
2 dist[v] = ∞
3 dist[s] = 0
4 Enqueue(Q, s)
5 while !Empty(Q)
6 u ← DEQUEUE(Q)
7 Visit(u)
8 for each edge (u, v) ∈ E
9 if dist[v] = ∞
10 Enqueue(Q, v)
11 dist[v] ← dist[u] + 1

Q: D, E, B

0
B

1

C
A

D

E

F

G

∞
∞

0
B

1

C
A

D

E

F

G

∞
∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞

∞
BFS(G, s)
1 for each v ∈ V
2  
3 dist[v] = ∞
4 Enqueue(Q, s)
5 while !Empty(Q)
6 u ← Dequeue(Q)
7 Visit(u)
8 for each edge (u, v) ∈ E
9 if dist[v] = ∞
10 Enqueue(Q, v)
11 dist[v] ← dist[u] + 1

Q:

BFS(G, s)
1 for each v ∈ V
2  
3 dist[v] = ∞
4 Enqueue(Q, s)
5 while !Empty(Q)
6 u ← Dequeue(Q)
7 Visit(u)
8 for each edge (u, v) ∈ E
9 if dist[v] = ∞
10 Enqueue(Q, v)
11 dist[v] ← dist[u] + 1

Q:
Runtime of BFS

Adjacency list: $O(|V| + |E|)$
Adjacency matrix: $O(|V|^2)$ (we won’t assumed connectedness)

Depth First Search (DFS)

TreeDFS(T)
1 Push(S, Root(T))
2 while !Empty(S)
3 v ← Pop(S)
4 Visit(v)
5 for all $c ∈ Children(v)$
6 Push(S, c)

BFS(G, s)
1 for each $v ∈ V$
2 $dist[v] = ∞$
3 $dist[s] = 0$
4 Enqueue(Q, s)
5 while !Empty(Q)
6 $u ← Dequeue(Q)$
7 Visit(u)
8 for each edge $(u, v) ∈ E$
9 if $dist[v] = ∞$
10 Enqueue(Q, v)
11 $dist[v] = dist[u] + 1$
Depth First Search (DFS)

TreeDFS(T):
1. Push(S, Root(T))
2. while !empty(S)
3. v ← pop(S)
4. Visit(v)
5. for all c ∈ Children(v)
6. Push(S, c)

TreeBFS(T):
1. Enqueue(Q, Root(T))
2. while !empty(Q)
3. v ← Dequeue(Q)
4. Visit(v)
5. for all c ∈ Children(v)
6. Enqueue(Q, c)

Tree DFS

Stack

Tree DFS

Stack

Tree DFS

Stack
What does this assume about how we add them to the stack?

Added from right to left: E, then D, then B.
Tree DFS

102

Tree DFS

103

Tree DFS

104

Tree DFS

105
DFS on graphs

DFS(G)
1 for all $v \in V$
2 visited[$v$] $\leftarrow$ false
3 for all $v \in V$
4 if !visited[$v$]
5 DFS-Visit($v$)

DFS-Visit($v$)
1 visited[$v$] $\leftarrow$ true
2 PreVisit($v$)
3 for all edges $(u, v) \in E$
4 if !visited[$u$]
5 DFS-Visit($v$)
6 PostVisit($v$)

mark all nodes as not visited

What happened to the stack?

DFS(G)
1 for all $v \in V$
2 visited[$u$] $\leftarrow$ false
3 for all $v \in V$
4 if !visited[$v$]
5 DFS-Visit($v$)

DFS-Visit($v$)
1 visited[$u$] $\leftarrow$ true
2 PreVisit($v$)
3 for all edges $(u, v) \in E$
4 if !visited[$v$]
5 DFS-Visit($v$)
6 PostVisit($v$)

TreeDFS($T$)
1 Push(S.Root($T$))
2 while !Empty(S)
3 $v$ $\leftarrow$ Pop(S)
4 Visit($v$)
5 for all $e \in$ Children($v$)
6 Push(S.E($e$))
What does DFS do?

Finds connected components

Each call to DFS-Visit from DFS starts exploring a new set of connected components

Helps us understand the structure/connectedness of a graph

Is DFS correct?

Does DFS visit all of the nodes in a graph?

Running time?

Like BFS
- Visits each node exactly once
- Processes each edge exactly twice (for an undirected graph)
- \( \theta(|V|+|E|) \)

Connectedness

Given an undirected graph, for every node \( u \in V \), can we reach all other nodes in the graph?

Algorithm + running time

Run BFS or DFS-Visit (one pass) and mark nodes as we visit them. If we visit all nodes, return true, otherwise false.

Running time: \( O(|V| + |E|) \)