Trees BFS

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)

Running time of Tree BFS

- How many times does it visit each vertex?
- How many times is each edge traversed?
- \(O(|V| + |E|)\)

Adjacency matrix
- For each vertex visited, how much work is done?
- \(O(|V|^2)\)
BFS Recursively

Hard to do!

BFS for graphs

What needs to change for graphs?

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

BFS(G, s)

1. for each \( v \in V \)
2. \( d[v] = \infty \)
3. \( u = \text{ENQUEUE}(Q) \)
4. \( \text{ENQUEUE}(Q, s) \)
5. while \( \text{EMPTY}(Q) \)
6. \( \text{VISIT}(u) \)
7. for each edge \((u, v) \in E\)
8. if \( d[u] = \infty \)
9. \( \text{ENQUEUE}(Q, v) \)
10. \( d[v] = d[u] + 1 \)

BFS(T)

1. \( \text{ENQUEUE}(Q, \text{ROOT}(T)) \)
2. while \( \text{EMPTY}(Q) \)
3. \( v = \text{DEQUEUE}(Q) \)
4. \( \text{VISIT}(v) \)
5. for all \( c \in \text{CHILDREN}(v) \)
6. \( \text{ENQUEUE}(Q, c) \)
BFS(G, s)
1. for each \( v \in V \):
2. \( \text{dist}[v] = \infty \)
3. \( \text{seen}[v] = false \)
4. \( \text{Enqueue}(Q, v) \)
5. while \( \text{NotEmpty}(Q) \):
6. \( u = \text{Dequeue}(Q) \)
7. \( \text{Visit}(u) \)
8. for each edge \( (u, v) \in E \):
9. \( \text{if} \ \text{dist}[v] = \infty \):
10. \( \text{Enqueue}(Q, v) \)
11. \( \text{dist}[v] = \text{dist}[u] + 1 \)

set all nodes as unseen

set the node as seen and record distance

check if the node has been seen
BFS(G, s)
1 for each v ∈ V
2     dist[v] = ∞
3 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

Q: A

BFS(G, s)
1 for each v ∈ V
2 dist[s] = 0
3 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

Q: E, B

BFS(G, s)
1 for each v ∈ V
2 dist[s] = 0
3 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

Q: D, E, B

BFS(G, s)
1 for each v ∈ V
2 dist[s] = 0
3 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

Q: E, B
BFSG, \( s \)
1. for each \( v \in V \)
2. \( d[v] = \infty \)
3. \( d[s] = 0 \)
4. Enqueue(Q, s)
5. while \( \text{Empty}(Q) \)
6. \( u = \text{Dequeue}(Q) \)
7. Visit(u)
8. for each edge \((u, v) \in E\)
9. if \( d[u] = \infty \)
10. Enqueue(Q, v)
11. \( d[v] = \min\{d[u] + 1\} \)

Diagram: A connected graph with vertices labeled from A to G and edges labeled with distances.

BFSG, \( s \)
1. for each \( v \in V \)
2. \( d[v] = \infty \)
3. \( d[s] = 0 \)
4. Enqueue(Q, s)
5. while \( \text{Empty}(Q) \)
6. \( u = \text{Dequeue}(Q) \)
7. Visit(u)
8. for each edge \((u, v) \in E\)
9. if \( d[u] = \infty \)
10. Enqueue(Q, v)
11. \( d[v] = \min\{d[u] + 1\} \)

Diagram: A connected graph with vertices labeled from A to G and edges labeled with distances.
BFS(G, s)
1. for each v ∈ V
2. \( \text{dist}[v] = \infty \)
3. \( \text{dist}[s] = 0 \)
4. \( \text{Enqueue}(Q, s) \)
5. while \( \text{Empty}(Q) \)
6. \( u \leftarrow \text{DEQUEUE}(Q) \)
7. \( \text{Visit}(u) \)
8. for each edge \((u, v) \in E\)
9. if \( \text{dist}[v] = \infty \)
10. \( \text{Enqueue}(Q, v) \)
11. \( \text{dist}[v] \leftarrow \text{dist}[u] + 1 \)

Q: F, C

BFS(G, s)
1. for each v ∈ V
2. \( \text{dist}[v] = \infty \)
3. \( \text{dist}[s] = 0 \)
4. \( \text{Enqueue}(Q, s) \)
5. while \( \text{Empty}(Q) \)
6. \( u \leftarrow \text{DEQUEUE}(Q) \)
7. \( \text{Visit}(u) \)
8. for each edge \((u, v) \in E\)
9. if \( \text{dist}[v] = \infty \)
10. \( \text{Enqueue}(Q, v) \)
11. \( \text{dist}[v] \leftarrow \text{dist}[u] + 1 \)
Is BFS correct?

Does it visit all nodes reachable from the starting node?
Can you prove it?

Assume we "miss" some node 'u’, i.e. a path exists, but we don’t visit ‘u’

Is BFS correct?

Does it visit all nodes reachable from the starting node?
Can you prove it?

Find the last node along the path to ‘u’ that was visited

Why do we know that such a node exists?

Is BFS correct?

Does it visit all nodes reachable from the starting node?
Can you prove it?

We visited ‘z’ but not ‘w’, which is a contradiction, given the pseudocode

Is BFS correct?

Does it correctly label each node with the shortest distance from the starting node?

```plaintext
BFS(G, s)
1 for each v ∈ V
2 if dist[s] = ∞
3 then dist[v] = 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 then Enqueue(Q, v)
11 dist[v] ← dist[u] + 1
```
Is BFS correct?
Does it correctly label each node with the shortest distance from the starting node?
Assume the algorithm labels a node with a longer distance. Call that node ’u’

![Diagram of BFS process]

Is BFS correct?
Does it correctly label each node with the shortest distance from the starting node?
Find the last node in the path with the correct distance

![Diagram of BFS process]

Is BFS correct?
Does it correctly label each node with the shortest distance from the starting node?
Find the last node in the path with the correct distance

![Diagram of BFS process]

Runtime of BFS
Nothing changed over our analysis of TreeBFS

```plaintext
BFS(G, s)
1 for each v ∈ V
2 dist[v] = ∞
3 Enqueue(Q, s)
4 while not Empty(Q)
5 u = Dequeue(Q)
6 foreach v ∈ Children(u)
7 if dist[v] = ∞
8   Enqueue(Q, v)
9   dist[v] = dist[u] + 1

TreeBFS(T)
1 Enqueue(Q, Root(T))
2 while not Empty(Q)
3 u = Dequeue(Q)
4 for all v ∈ Children(u)
5   Enqueue(Q, v)
```
Runtime of BFS

Adjacency list: $O(|V| + |E|)$
Adjacency matrix: $O(|V|^2)$

Depth First Search (DFS)

Tree 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)
```
DFS on graphs

DFS(G)
1 for all \( v \in V \\
2 \text{visited}[v] \leftarrow false \\
3 \text{for all} \( v \in V \\
4 \text{if} \ \text{visited}[v] \\
5 \quad \text{DFS-Visit}(v)

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

mark all nodes as not visited until all nodes have been visited repeatedly call DFS-Visit.

What happened to the stack?
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?

```
DFS(G)
1 for all v ∈ V
2 visited[v] ← false
3 for all v ∈ V
4 if !visited[v]
5 DFS-Visit(v)
```

Running time?

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

DAGs

Can represent dependency graphs
Topological sort

A linear ordering of all the vertices such that for all edges \((u,v) \in E\), \(u\) appears before \(v\) in the ordering.

An ordering of the nodes that "obeys" the dependencies, i.e., an activity can't happen until its dependent activities have happened.

Topological sort

1. Find a node \(v\) with no incoming edges.
2. Delete \(v\) from \(G\).
3. Add \(v\) to linked list.
4. Topological-sort1\((G)\).

Example:
- Underwear
- Pants
- Belt
- Shirt
- Tie
- Watch
- Socks
- Shoes
- Jacket

Graph:
- Underwear
  - Pants
  - Belt
  - Shirt
  - Tie
  - Watch
- Socks
- Shoes
- Jacket

Underwear
  - Pants
  - Belt
  - Shirt
  - Tie
  - Watch
- Socks
- Shoes
- Jacket
**Topological sort**

1. Find a node $v$ with no incoming edges
2. Delete $v$ from $G$
3. Add $v$ to linked list
4. Topological-Sort1($G$)

**Topological sort (example)**

1. Underwear
2. Pants
3. Belt
4. Shirt
5. Tie
6. Watch
7. Socks
8. Shoes
9. Jacket

**Topological sort**

1. Find a node $v$ with no incoming edges
2. Delete $v$ from $G$
3. Add $v$ to linked list
4. Topological-Sort1($G$)

**Topological sort (example)**

1. Underwear
2. Pants
3. Belt
4. Shirt
5. Tie
6. Watch
7. Socks
8. Shoes
9. Jacket

**Topological sort**

1. Find a node $v$ with no incoming edges
2. Delete $v$ from $G$
3. Add $v$ to linked list
4. Topological-Sort1($G$)

**Topological sort (example)**

1. Underwear
2. Pants
3. Belt
4. Shirt
5. Tie
6. Watch
7. Socks
8. Shoes
9. Jacket
Topological sort

1. Find a node with no incoming edges
2. Delete v from G
3. Add v to linked list
4. \textsc{Topological-Sort1}(G)

Running time?

1. Find a node with no incoming edges
2. Delete v from G
3. Add v to linked list
4. \textsc{Topological-Sort1}(G)

\textsc{Topological-Sort1}(G) \quad \text{Running time?} \quad \text{O}(|V|+|E|)
Running time?

**Topological-Sort1(G)**

1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. Topological-Sort1(G)

\( O(E) \) overall

Running time?

**Topological-Sort1(G)**

1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. Topological-Sort1(G)

How many calls? \( |V| \)

Running time?

**Topological-Sort1(G)**

1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. Topological-Sort1(G)

Overall running time?

\( O(|V|^2+|V||E|) \)

Can we do better?

**Topological-Sort1(G)**

1. Find a node \( v \) with no incoming edges
2. Delete \( v \) from \( G \)
3. Add \( v \) to linked list
4. Topological-Sort1(G)
Topological sort 2

**Topological-Sort(G)**
1. for all edges \((u, v) \in E\)
2. \(active[v] \leftarrow active[v] + 1\)
3. for all \(v \in V\)
4. if \(active[v] = 0\)
5. ENQUEUE(S, v)
6. while !EMPTY(S)
7. \(u \leftarrow \text{DEQUEUE}(S)\)
8. add \(u\) to linked list
9. for each edge \((u, v) \in E\)
10. \(active[v] \leftarrow active[v] - 1\)
11. if \(active[v] = 0\)
12. ENQUEUE(S, v)
Running time?

How many times do we process each node?
How many times do we process each edge?

\[ O(|V| + |E|) \]

Connectedness

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

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: \[ O(|V| + |E|) \]

Strongly connected

Given a directed graph, can we reach any node \( v \) from any other node \( u \)?

Ideas?

Transpose of a graph

Given a graph \( G \), we can calculate the transpose of a graph \( G^R \) by reversing the direction of all the edges

Running time to calculate \( G^R \)? \[ O(|V| + |E|) \]
Strongly connected

**Strongly-Connected(G)**
1. Run DFS or BFS from some node u
2. If not all nodes are visited
   - return false
3. Create graph \( G^R \) by reversing all edge directions
4. Run DFS or BFS on \( G^R \) from node u
5. If not all nodes are visited
   - return false
8. return true

Is it correct?

What do we know after the first pass?
- Starting at u, we can reach every node

What do we know after the second pass?
- All nodes can reach u. Why?
  - We can get from u to every node in \( G^R \), therefore, if we reverse the edges (i.e. G), then we have a path from every node to u

Which means that any node can reach any other node. Given any two nodes s and t we can create a path through u

![Diagram showing a directed graph with nodes and edges]

Runtime?

**Strongly-Connected(G)**
1. Run DFS or BFS from some node u
2. If not all nodes are visited
   - return false
3. Create graph \( G^R \) by reversing all edge directions
4. Run DFS or BFS on \( G^R \) from node u
5. If not all nodes are visited
   - return false
8. return true

\[ O(|V| + |E|) \]
Detecting cycles

Undirected graph
- BFS or DFS. If we reach a node we’ve seen already, then we’ve found a cycle

Directed graph
- Call TopologicalSort
- If the length of the list returned ≠ |V| then a cycle exists

Shortest paths

What is the shortest path from a to d?

Shortest paths

BFS
Shortest paths
We can still use BFS

A
B
C
D
E

Shortest paths
What is the problem?

A
B
C
D
E

A
B
C
D
E

A
B
C
D
E
Shortest paths
Running time is dependent on the weights

A - B - C
\[
\begin{align*}
A & \quad B \\
\quad 4 & \quad \quad 2
\end{align*}
\]

A - C - B
\[
\begin{align*}
B & \quad C \\
100 & \quad \quad 50
\end{align*}
\]

A - B - C
\[
\begin{align*}
B & \quad C \\
100 & \quad \quad 50
\end{align*}
\]

A - C - B
\[
\begin{align*}
C & \quad B \\
200 & \quad \quad 100
\end{align*}
\]
Shortest paths
Nothing will change as we expand the frontier until we've gone out 100 levels

Dijkstra’s algorithm

Dijkstra(G, s)
1 for all \( v \in V \) 
2 \( dist[v] \leftarrow \infty \)
3 \( prev(v) \leftarrow null \)
4 \( dist[s] \leftarrow 0 \)
5 \( Q \leftarrow \text{MakeHeap}(V) \)
6 while !\( \text{Empty}(Q) \)
7 \( u \leftarrow \text{ExtractMin}(Q) \)
8 for all edges \((u, v) \in E\) 
9 if \( dist[u] + \text{weight}(u, v) < dist[v] \)
10 \( dist[v] \leftarrow dist[u] + \text{weight}(u, v) \)
11 \( \text{DecreaseKey}(Q, u, dist[v]) \)
12 \( prev[v] \leftarrow u \)

Dijkstra’s algorithm
prev keeps track of the shortest path

Dijkstra(G, s)
1 for all \( v \in V \) 
2 \( dist[v] \leftarrow \infty \)
3 \( prev(v) \leftarrow null \)
4 \( dist[s] \leftarrow 0 \)
5 \( Q \leftarrow \text{MakeHeap}(V) \)
6 while !\( \text{Empty}(Q) \)
7 \( u \leftarrow \text{ExtractMin}(Q) \)
8 for all edges \((u, v) \in E\) 
9 if \( dist[u] + \text{weight}(u, v) < dist[v] \)
10 \( dist[v] \leftarrow dist[u] + \text{weight}(u, v) \)
11 \( \text{DecreaseKey}(Q, u, dist[v]) \)
12 \( prev[v] \leftarrow u \)
Single source shortest paths

All of the shortest path algorithms we'll look at today are called "single source shortest paths" algorithms.

Why?
**Dijkstra(G, s)**

1. For all \( v \in V \)
2. \( d[v] \leftarrow \infty \)
3. \( prev[v] \leftarrow null \)
4. \( d[s] \leftarrow 0 \)
5. \( Q \leftarrow MaxHeap(V) \)
6. While \( \text{NotEmpty}(Q) \)
7. \( s \leftarrow \text{ExtractMin}(Q) \)
8. For all \( (u, v) \in E \)
9. \( \text{If } d[u] + w(u, v) < d[v] \)
10. \( d[v] \leftarrow d[u] + w(u, v) \)
11. \( \text{DecreaseKey}(Q, v, d[v]) \)
12. \( \text{prev}[v] \leftarrow u \)

**Heap**

- **A** 0
- **B** \( \infty \)
- **C** \( \infty \)
- **D** \( \infty \)
- **E** \( \infty \)

**Dijkstra(G, s)**

1. For all \( v \in V \)
2. \( d[v] \leftarrow \infty \)
3. \( prev[v] \leftarrow null \)
4. \( d[s] \leftarrow 0 \)
5. \( Q \leftarrow MaxHeap(V) \)
6. While \( \text{NotEmpty}(Q) \)
7. \( s \leftarrow \text{ExtractMin}(Q) \)
8. For all \( (u, v) \in E \)
9. \( \text{If } d[u] + w(u, v) < d[v] \)
10. \( d[v] \leftarrow d[u] + w(u, v) \)
11. \( \text{DecreaseKey}(Q, v, d[v]) \)
12. \( \text{prev}[v] \leftarrow u \)

**Heap**

- **B** \( \infty \)
- **C** \( \infty \)
- **D** \( \infty \)
- **E** \( \infty \)
Dijkstra(G, s)
1 for all v ∈ V
2 dist[v] = ∞
3 prev[v] = null
4 dist[s] = 0
5 Q = ExtractMin(Q)
6 while Q ̸= ∅
7 s = ExtractMin(Q)
8 for all neighbors (v, c) of s in G
9 if dist[i] > dist[s] + w(s, c)
10 dist[i] = dist[s] + w(s, c)
11 prev[i] = s

Heap

C 1
B 3
D ∞
E ∞
Dijkstra(G, s)
1 for all v ∈ V
2 dist(v) ← ∞
3 prev(v) ← nil
4 dist(s) ← 0
5 Q ← PriorityQueue()
6 while Q is nonempty
7 s ← ExtractMin(Q)
8 for all nodes (u, v) ∈ E
9 if dist(s) + w(u, v) < dist(v)
10 then dist(v) ← dist(s) + w(u, v)
11 prev(v) ← s

Heap

Dijkstra(G, s)
1 for all v ∈ V
2 dist(v) ← ∞
3 prev(v) ← nil
4 dist(s) ← 0
5 Q ← PriorityQueue()
6 while Q is nonempty
7 s ← ExtractMin(Q)
8 for all nodes (u, v) ∈ E
9 if dist(s) + w(u, v) < dist(v)
10 then dist(v) ← dist(s) + w(u, v)
11 prev(v) ← s

Heap
**Dijkstra(G, s)**
1. For all $v \in V$
2. $d[v] \leftarrow \infty$
3. $\text{prev}[v] \leftarrow \text{null}$
4. $d[s] \leftarrow 0$
5. $Q \leftarrow \text{Initialize}(V, d[s])$
6. While $\text{Empty}(Q)$
7. $u \leftarrow \text{ExtractMin}(Q)$
8. For all $(u, v) \in E$
9. $d[u] \leftarrow d[u] + w(u, v)$
10. $Q \leftarrow \text{DecreaseKey}(Q, u, d[u])$
11. $\text{prev}[v] \leftarrow u$

**Frontier?**
Dijkstra(G, s)
1. for all v ∈ V
2. dist[v] = ∞
3. prev[v] = null
4. dist[s] = 0
5. Q = MaintainF(V)
6. while Q ≠ EmptyQ
7. s = ExtractMin(Q)
8. for all edges (s, v) ∈ E
9. if dist[v] > dist[s] + w(s, v)
10. dist[v] = dist[s] + w(s, v)
11. InsertMinHeap(Q, v, dist[v])
12. prev[v] = s

All nodes reachable from starting node within a given distance

Dijkstra(G, s)
1. for all v ∈ V
2. dist[v] = ∞
3. prev[v] = null
4. dist[s] = 0
5. Q = MaintainF(V)
6. while Q ≠ EmptyQ
7. s = ExtractMin(Q)
8. for all edges (s, v) ∈ E
9. if dist[v] > dist[s] + w(s, v)
10. dist[v] = dist[s] + w(s, v)
11. InsertMinHeap(Q, v, dist[v])
12. prev[v] = s
Dijkstra(G, s)
1  for all x ∈ V
2    dist[x] ← ∞
3    prev[x] ← null
4    dist[s] ← 0
5    Q ← MainHeaps()
6  while not Empty(Q)
7    u ← ExtractMin(Q)
8    for all adj(u, v) ∈ E:
9      if dist[u] + w(u, v) < dist[v]
10         dist[v] ← dist[u] + w(u, v)
11         prev[v] ← u
12         InsertHeaps(u, v, dist[v])

Heap

A
   |   |
 /   |   \
B    E
 /   |   \  
C    1  1
 |   |   |
|---|---|---|
1  1  3

Heap

E
   |   |
 /   |   \
B    E
 /   |   \  
C    1  1
 |   |   |
|---|---|---|
1  1  3