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

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: \( O(V + E) \)
Strongly connected

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

Can we do the same thing?

Transpose of a graph

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

Running time to calculate \( G^T \)? \( \Theta(|V| + |E|) \)

Is it correct?

Strongly-Connected(\( G \))
- Run DFS-Visit or BFS from some node \( u \)
- If not all nodes are visited; return false
- Create graph \( G^T \)
- Run DFS-Visit or BFS on \( G^T \) from node \( u \)
- If not all nodes are visited; return false
- return true

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 \( G^T \), 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 \( u \)

\[ s \rightarrow \cdots \rightarrow u \rightarrow \cdots \rightarrow t \]
Runtime?

- Strongly-Connected(G)
  - Run DFS: Visit or BFS from some node u
  - If not all nodes are visited: return false
  - Create graph $G^r$
  - Run DFS: Visit or BFS on $G^r$ from node u
  - If not all nodes are visited: return false
  - return true

$O(|V| + |E|)$

Shortest paths

What is the shortest path from a to d?

BFS

What is the shortest path from a to d?
Dijkstra’s algorithm

```
Dijkstra(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← nil
4 dist[s] ← 0
5 Q ← MaxHeap(V)
6 while not Empty(Q)
7 s ← ExtractMin(Q)
8 for all edges (u, v) ∈ E
9 if dist[u] + w(u, v) < dist[v]
10 dist[v] ← dist[u] + w(u, v)
11 DecreaseKey(Q, v, dist[v])
12 prev[v] ← s
```

Dijkstra’s algorithm

```
prev keeps track of the shortest path
```

```
Dijkstra(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← nil
4 dist[s] ← 0
5 Q ← MaxHeap(V)
6 while not Empty(Q)
7 s ← ExtractMin(Q)
8 for all edges (u, v) ∈ E
9 if dist[u] + w(u, v) < dist[v]
10 dist[v] ← dist[u] + w(u, v)
11 DecreaseKey(Q, v, dist[v])
12 prev[v] ← s
```

Shortest paths

What is the shortest path from a to d?

```
A
B
C
D
E
```

Dijkstra’s algorithm

```
What is dist?

Dijkstra(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← nil
4 dist[s] ← 0
5 Q ← MaxHeap(V)
6 while not Empty(Q)
7 s ← ExtractMin(Q)
8 for all edges (u, v) ∈ E
9 if dist[u] > dist[v] + w(u, v)
10 dist[v] ← dist[u] + w(u, v)
11 DecreaseKey(Q, v, dist[v])
12 prev[v] ← s
```

Dijkstra’s algorithm

```
What is prev?

Dijkstra(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← nil
4 dist[s] ← 0
5 Q ← MaxHeap(V)
6 while not Empty(Q)
7 s ← ExtractMin(Q)
8 for all edges (u, v) ∈ E
9 if dist[u] > dist[v] + w(u, v)
10 dist[v] ← dist[u] + w(u, v)
11 DecreaseKey(Q, v, dist[v])
12 prev[v] ← s
```

Dijkstra’s algorithm

```
How does it work?

Dijkstra(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← nil
4 dist[s] ← 0
5 Q ← MaxHeap(V)
6 while not Empty(Q)
7 s ← ExtractMin(Q)
8 for all edges (u, v) ∈ E
9 if dist[u] > dist[v] + w(u, v)
10 dist[v] ← dist[u] + w(u, v)
11 DecreaseKey(Q, v, dist[v])
12 prev[v] ← s
```

Dijkstra’s algorithm

```
What is the run-time?

Dijkstra(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← nil
4 dist[s] ← 0
5 Q ← MaxHeap(V)
6 while not Empty(Q)
7 s ← ExtractMin(Q)
8 for all edges (u, v) ∈ E
9 if dist[u] > dist[v] + w(u, v)
10 dist[v] ← dist[u] + w(u, v)
11 DecreaseKey(Q, v, dist[v])
12 prev[v] ← s
```

Dijkstra’s algorithm

```
How do we get the shortest path?

Dijkstra(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← nil
4 dist[s] ← 0
5 Q ← MaxHeap(V)
6 while not Empty(Q)
7 s ← ExtractMin(Q)
8 for all edges (u, v) ∈ E
9 if dist[u] > dist[v] + w(u, v)
10 dist[v] ← dist[u] + w(u, v)
11 DecreaseKey(Q, v, dist[v])
12 prev[v] ← s
```
Dijkstra's algorithm

Dijkstra\(s)\ algorithm

1. Initialize: \(V\) = \{s\}, \(D_v = \infty\) for all \(v \in V\), \(\text{prev}_v = \text{null}\)
2. \(Q = \text{MaxHeap}\)
3. while \(Q\) is not empty:
   1. Remove the vertex \(u\) with the minimum \(D_u\)
   2. for each edge \((u, v) \in E\)
      1. if \(D_v > D_u + w(u, v)\)
      2. then \(D_v = D_u + w(u, v)\)
      3. \(\text{prev}_v = u\)

Diagonal matrix

\[ A = \begin{bmatrix}
    0 & 5 & 0 & 2 & 1 \\
    5 & 0 & 4 & 0 & 0 \\
    0 & 4 & 0 & 2 & 0 \\
    2 & 0 & 2 & 0 & 3 \\
    1 & 0 & 0 & 3 & 0 \\
\end{bmatrix} \]
**Dijkstra(G, s)**

1. for all $v \in V$
2. $dist[v] \leftarrow \infty$
3. $prev[v] \leftarrow \text{null}$
4. $Q \leftarrow \text{MaxHeap}(V)$
5. while $\text{NotEmpty}(Q)$
6. $u \leftarrow \text{ExtractMin}(Q)$
7. for all edges $(u, v) \in E$
8. \quad if $dist[u] > dist[v] + w(u, v)$
9. \quad \quad $dist[u] \leftarrow dist[v] + w(u, v)$
10. \quad \quad $\text{DecreaseKey}(Q, u, dist[u])$
11. $prev[u] \leftarrow u$

**Heap**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>3</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>4</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

**Dijkstra(G, s)**

1. for all $v \in V$
2. $dist[v] \leftarrow \infty$
3. $prev[v] \leftarrow \text{null}$
4. $Q \leftarrow \text{MaxHeap}(V)$
5. while $\text{NotEmpty}(Q)$
6. $u \leftarrow \text{ExtractMin}(Q)$
7. for all edges $(u, v) \in E$
8. \quad if $dist[u] > dist[v] + w(u, v)$
9. \quad \quad $dist[u] \leftarrow dist[v] + w(u, v)$
10. \quad \quad $\text{DecreaseKey}(Q, u, dist[u])$
11. $prev[u] \leftarrow u$

**Heap**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>3</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>4</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

**Heap**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>3</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>4</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>
DIJKSTRA$(G, s)$
1. for all $v \in V$
2. $dist[v] \leftarrow \infty$
3. $prev[v] \leftarrow \text{null}$
4. $dist[s] \leftarrow 0$
5. $Q \leftarrow \text{MaxHeap}(V)$
6. while $\text{Empty}(Q)$
7. $u \leftarrow \text{ExtractMin}(Q)$
8. for all edges $(u, v) \in E$
9. \quad if $dist[u] + w(u, v) < dist[v]$
10. \quad $dist[v] \leftarrow dist[u] + w(u, v)$
11. \quad $prev[v] \leftarrow u$
12. \quad $\text{DecreaseKey}(Q, u, dist[u])$

Heap
$C \ 1$
$B \ \infty$
$D \ \infty$
$E \ \infty$

DIJKSTRA$(G, a)$
1. for all $v \in V$
2. $dist[v] \leftarrow \infty$
3. $prev[v] \leftarrow \text{null}$
4. $dist[a] \leftarrow 0$
5. $Q \leftarrow \text{MaxHeap}(V)$
6. while $\text{Empty}(Q)$
7. $u \leftarrow \text{ExtractMin}(Q)$
8. for all edges $(u, v) \in E$
9. \quad if $dist[u] + w(u, v) < dist[v]$
10. \quad $dist[v] \leftarrow dist[u] + w(u, v)$
11. \quad $prev[v] \leftarrow u$
12. \quad $\text{DecreaseKey}(Q, u, dist[u])$

Heap
$C \ 1$
$B \ \infty$
$D \ \infty$
$E \ \infty$

25

26

27

28
Dijkstra(G, s)
1 for all v ∈ V
2 dist[v] := ∞
3 prev[v] := nil
4 dist[s] := 0
5 Q := M[allest(V)]
6 while not EMPTY(Q)
7 u := EXTRACT-MIN(Q)
8 for all edges (u, v) ∈ E
9 if dist[u] + dist[u] + w(u, v) < dist[v]
10 dist[v] := dist[u] + dist[u] + w(u, v)
11 DECREASE-KEY(Q, v, dist[v])
12 prev[v] := u

Heap

Heap

Heap

Heap
11/3/22

Dijkstra(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← null
4 dist[s] ← 0
5 Q ← MaxHeap(V)
6 while not Empty(Q)
7 u ← ExtractMin(Q)
8 for all edges (u, v) ∈ E
9 if dist[v] > dist[u] + w(u, v)
10 dist[v] ← dist[u] + w(u, v)
11 DecreaseKey(Q, v, dist[v])
12 prev[v] ← u

Heap

B 2
D ∞
E ∞

A

B

C

D

E

Frontier?

Dijkstra(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← null
4 dist[s] ← 0
5 Q ← MaxHeap(V)
6 while not Empty(Q)
7 u ← ExtractMin(Q)
8 for all edges (u, v) ∈ E
9 if dist[v] > dist[u] + w(u, v)
10 dist[v] ← dist[u] + w(u, v)
11 DecreaseKey(Q, v, dist[v])
12 prev[v] ← u

Heap

B 2
E 5
D ∞

All nodes reachable from starting node within a given distance
Is Dijkstra’s algorithm correct?

Invariant: For every vertex removed from the heap, dist[v] is the actual shortest distance from s to v.

proof?

The only time a vertex gets visited is when the distance from s to that vertex is smaller than the distance to any remaining vertex.

Therefore, there cannot be any other path that hasn’t been visited already that would result in a shorter path.

Running time?

1 call to MakeHeap
Running time?

Dijkstra(G, s)
1 for all v ∈ V
dist[v] ← ∞
2 prev[v] ← nil
3 dist[s] ← 0
4 Q ← MakeQueue(V)
5 while not Empty(Q)
6 s ← ExtractMin(Q)
7 for all (v, w) ∈ E
8 if dist[v] > dist[u] + w(u, v)
9 dist[v] ← dist[u] + w(u, v)
10 DecreaseKey(Q, s, dist[s])
11 prev[v] ← s
85 |V| iterations
84

Running time?

Dijkstra(G, s)
1 for all v ∈ V
dist[v] ← ∞
2 prev[v] ← nil
3 dist[s] ← 0
4 Q ← MakeQueue(V)
5 while not Empty(Q)
6 s ← ExtractMin(Q)
7 for all (v, w) ∈ E
8 if dist[v] > dist[u] + w(u, v)
9 dist[v] ← dist[u] + w(u, v)
10 DecreaseKey(Q, s, dist[s])
11 prev[v] ← s
85 |V| calls
84

Running time?

Depends on the heap implementation

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Bin heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>MakeHeap</td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td>DecreaseKey</td>
<td>O(</td>
<td>E</td>
</tr>
<tr>
<td>Total</td>
<td>O(</td>
<td>V</td>
</tr>
</tbody>
</table>
Running time?

Depends on the heap implementation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>Bin heap</th>
<th>Fib heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>MakeHeap</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>$O(</td>
<td>V</td>
<td>^2)$</td>
</tr>
<tr>
<td>DecreaseKey</td>
<td>$O(</td>
<td>E</td>
<td>)$</td>
</tr>
<tr>
<td>Total</td>
<td>$O(</td>
<td>V</td>
<td>^2)$</td>
</tr>
</tbody>
</table>

Is this an improvement? If $|E| < |V|^2 / \log |V|$

What about Dijkstra’s on...?

Dijkstra’s algorithm only works for positive edge weights
Is Dijkstra’s algorithm correct?

Invariant: For every vertex removed from the heap, dist[v] is the actual shortest distance from s to v

- The only time a vertex gets visited is when the distance from s to that vertex is smaller than the distance to any remaining vertex.
- Therefore, there cannot be any other path that hasn’t been visited already that would result in a shorter path.

We relied on having positive edge weights for correctness!

Bounding the distance

Another invariant: For each vertex v, dist[v] is an upper bound on the actual shortest distance

\[
dist[v] = \min \{ dist[v], dist[u] + w(u,v) \}
\]

Can we ever go wrong applying this update rule?

- We can apply this rule as many times as we want and will never underestimate dist[v].

When will dist[v] be right?

- If u is along the shortest path to v and dist[u] is correct.
\[ \text{dist}[v] = \min \{\text{dist}[v], \text{dist}[u] + w(u, v)\} \]

\text{dist}[v] \text{ will be right if } u \text{ is along the shortest path to } v \text{ and } \text{dist}[u] \text{ is correct}

Consider the shortest path from \( s \) to \( v \)

What happens if we update all of the vertices with the above update?

\[ \text{dist}[v] = \min \{\text{dist}[v], \text{dist}[u] + w(u, v)\} \]

\text{dist}[v] \text{ will be right if } u \text{ is along the shortest path to } v \text{ and } \text{dist}[u] \text{ is correct}

What happens if we update all of the vertices with the above update?

\[ \text{dist}[v] = \min \{\text{dist}[v], \text{dist}[u] + w(u, v)\} \]

\text{dist}[v] \text{ will be right if } u \text{ is along the shortest path to } v \text{ and } \text{dist}[u] \text{ is correct}

What happens if we update all of the vertices with the above update?
\[ \text{dist}[v] = \min \{ \text{dist}[v], \text{dist}[u] + w(u,v) \} \]

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

How many times do we have to do this for vertex p, to have the correct shortest path from s?

\[ \text{dist}[v] = \min \{ \text{dist}[v], \text{dist}[u] + w(u,v) \} \]

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

How many times do we have to do this for vertex p, to have the correct shortest path from s?

\[ \text{dist}[v] = \min \{ \text{dist}[v], \text{dist}[u] + w(u,v) \} \]

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

How many times do we have to do this for vertex p, to have the correct shortest path from s?
\[ \text{dist}[v] = \min \{ \text{dist}[v], \text{dist}[u] + w(u,v) \} \]

\text{dist}[v] \text{ will be right if } u \text{ is along the shortest path to } v \text{ and } \text{dist}[u] \text{ is correct}

How many times do we have to do this for vertex \( p_i \) to have the correct shortest path from \( s \)?

\( i \) times

\[ \min \{ v, u, w \} \]

\[ \text{dist}[v] = \min \{ \text{dist}[v], \text{dist}[u] + w(u,v) \} \]

\text{dist}[v] \text{ will be right if } u \text{ is along the shortest path to } v \text{ and } \text{dist}[u] \text{ is correct}

How many times do we have to do this for vertex \( p_i \) to have the correct shortest path from \( s \)?

\( i \) times

\[ \min \{ v, u, w \} \]

Bellman-Ford algorithm

\begin{verbatim}
Bellman-Ford(G, s)
1 for all \( v \in V \)
2 \( \text{dist}[v] \leftarrow \infty \)
3 \( \text{pre}[v] \leftarrow \text{null} \)
4 \( \text{dist}[s] \leftarrow 0 \)
5 for \( i \) from 1 to \( |V| - 1 \)
6 for all edges \( (u, v) \in E \)
7 if \( \text{dist}[u] > \text{dist}[u] + w(u,v) \)
8 \( \text{dist}[v] \leftarrow \text{dist}[u] + w(u,v) \)
9 \( \text{pre}[v] \leftarrow u \)
10 for all edges \( (u, v) \in E \)
11 if \( \text{dist}[u] > \text{dist}[u] + w(u,v) \)
12 return false
\end{verbatim}
Bellman-Ford algorithm

1. for all \( v \in V \)
2. \( \text{dist}[v] \leftarrow \infty \)
3. \( \text{pre}[v] \leftarrow \text{null} \)
4. \( \text{dist}[s] \leftarrow 0 \)
5. for \( i \leftarrow 1 \) to \(|V| - 1\)
6. for all edges \((u, v) \in E\)
7. if \( \text{dist}[u] > \text{dist}[v] + w(u, v) \)
8. \( \text{dist}[v] \leftarrow \text{dist}[u] + w(u, v) \)
9. \( \text{pre}[v] \leftarrow u \)
10. for all edges \((u, v) \in E\)
11. if \( \text{dist}[u] > \text{dist}[v] + w(u, v) \)
12. return false

Negative cycles

What is the shortest path from \( a \) to \( e \)?

Bellman-Ford algorithm

1. for all \( v \in V \)
2. \( \text{dist}[v] \leftarrow \infty \)
3. \( \text{pre}[v] \leftarrow \text{null} \)
4. \( \text{dist}[s] \leftarrow 0 \)
5. for \( i \leftarrow 1 \) to \(|V| - 1\)
6. for all edges \((u, v) \in E\)
7. if \( \text{dist}[u] > \text{dist}[v] + w(u, v) \)
8. \( \text{dist}[v] \leftarrow \text{dist}[u] + w(u, v) \)
9. \( \text{pre}[v] \leftarrow u \)
10. for all edges \((u, v) \in E\)
11. if \( \text{dist}[u] > \text{dist}[v] + w(u, v) \)
12. return false
How many edges is the shortest path from $s$ to:

A: 3
B: 5
Bellman-Ford algorithm

How many edges is the shortest path from s to:

A: 3

B: 5

D:

Iteration: 0

Bellman-Ford algorithm

Iteration: 1
Bellman-Ford algorithm

Iteration: 2

A has the correct distance and path

Iteration: 3

B has the correct distance and path

Iteration: 4
Correctness of Bellman-Ford

Loop invariant: After iteration \( i \), all vertices with shortest paths from \( s \) of length \( i \) edges or less have correct distances.

Bellman-Ford algorithm

Correctness of Bellman-Ford

Runtime of Bellman-Ford
Runtime of Bellman-Ford

```c
Bellman-Ford(G, s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← null
4 dist[s] ← 0
5 for i ← 1 to |V| − 1
6 for all edges (u, v) ∈ E
7 if dist[v] > dist[u] + w(u, v)
8 dist[v] ← dist[u] + w(u, v)
9 prev[v] ← u
10 for all edges (u, v) ∈ E
11 if dist[v] > dist[u] + w(u, v)
12 return false
```

Can you modify the algorithm to run faster (in some circumstances)?

Single source shortest paths

All of the shortest path algorithms we’ve looked at today are call “single source shortest paths” algorithms.

Why?