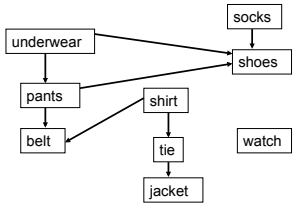
Graphs

David Kauchak
cs302
Spring 2012

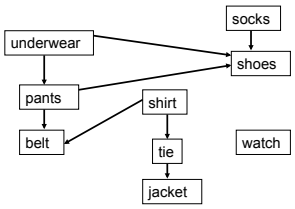
DAGs

Can represent dependency graphs



Topological sort

- A linear ordering of all the vertices such that for all edges (u,v) ∈ 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 it's dependent activities have happened



watch
underwear
pants
shirt
belt
tie
socks
shoes
jacket

Topological sort

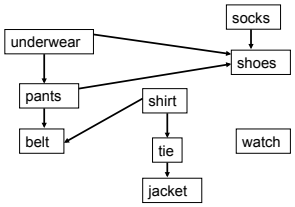
TOPOLOGICAL-SORT1(G)

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

TOPOLOGICAL-SORT1(G)

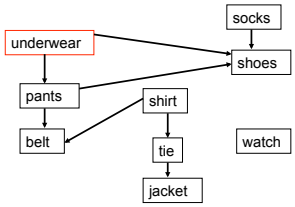
1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)



Topological sort

TOPOLOGICAL-SORT1(G)

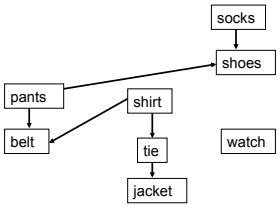
1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)



Topological sort

TOPOLOGICAL-SORT1(G)

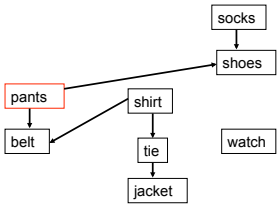
1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)



Topological sort

TOPOLOGICAL-SORT1(G)

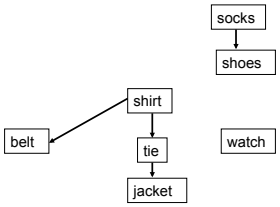
1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)



Topological sort

TOPOLOGICAL-SORT1(G)

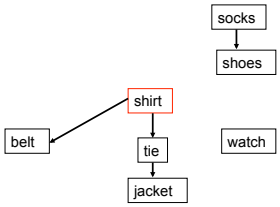
1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)



Topological sort

TOPOLOGICAL-SORT1(G)

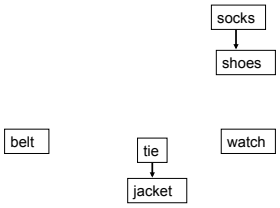
1 Find a node v with no incoming edges
2 Delete v from G
3 Add v to linked list
4 TOPOLOGICAL-SORT1(G)

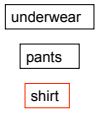


Topological sort

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)

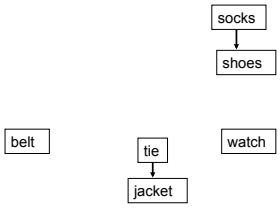


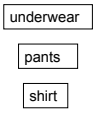


Topological sort

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)





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)

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(|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)
Find a node v with no incoming edges
Delete v from G
Add v to linked list
TOPOLOGICAL-SORT1(G)
```

Overall running time?

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

Can we do better?

```
TOPOLOGICAL-SORT1(G)
Find a node v with no incoming edges
Delete v from G
Add v to linked list
TOPOLOGICAL-SORT1(G)
```

Topological sort 2

```
TOPOLOGICAL-SORT2(G)
for all edges (u, v) ∈ E
        active[v] ← active[v] + 1
for all v ∈ V
        if active[v] = 0
                ENQUEUE(S, v)
while !EMPTY(S)
        u ← DEQUEUE(S)
        add u to linked list
        for each edge (u, v) ∈ E
                active[v] ← active[v] − 1
                if active[v] = 0
                        ENQUEUE(S, v)
```

Topological sort 2

```
TOPOLOGICAL-SORT2(G)
for all edges (u, v) ∈ E
        active[v] ← active[v] + 1
for all v ∈ V
        if active[v] = 0
                ENQUEUE(S, v)
while !EMPTY(S)
        u ← DEQUEUE(S)
        add u to linked list
        for each edge (u, v) ∈ E
                active[v] ← active[v] − 1
                if active[v] = 0
                        ENQUEUE(S, v)
```

Topological sort 2

```
TOPOLOGICAL-SORT2(G)
for all edges (u, v) ∈ E
        active[v] ← active[v] + 1
for all v ∈ V
        if active[v] = 0
                ENQUEUE(S, v)
while !EMPTY(S)
        u ← DEQUEUE(S)
        add u to linked list
        for each edge (u, v) ∈ E
                active[v] ← active[v] − 1
                if active[v] = 0
                        ENQUEUE(S, v)
```

Topological sort 2

```
TOPOLOGICAL-SORT2(G)
for all edges (u, v) ∈ E
        active[v] ← active[v] + 1
for all v ∈ V
        if active[v] = 0
                ENQUEUE(S, v)
while !EMPTY(S)
        u ← DEQUEUE(S)
        add u to linked list
        for each edge (u, v) ∈ E
                active[v] ← active[v] − 1
                if active[v] = 0
                        ENQUEUE(S, v)
```

Running time?

- How many times do we process each node?
- How many times do we process each edge?
- O(|V| + |E|)

```
TOPOLOGICAL-SORT2(G)
for all edges (u, v) ∈ E
        active[v] ← active[v] + 1
for all v ∈ V
        if active[v] = 0
                ENQUEUE(S, v)
while !EMPTY(S)
        u ← DEQUEUE(S)
        add u to linked list
        for each edge (u, v) ∈ E
                active[v] ← active[v] − 1
                if active[v] = 0
                        ENQUEUE(S, v)
```

Connectedness

Given an undirected graph, for every node u ∈ 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|)

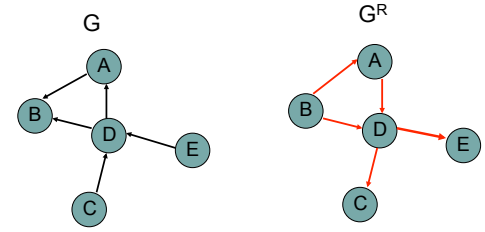
Strongly connected

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



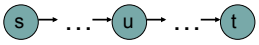
Running time to calculate G^R? O(|V| + |E|)

Strongly connected

```
STRONGLY-CONNECTED(G)
Run DFS or BFS from some node u
if not all nodes are visited
        return false
Create graph G^R by reversing all edge dirctions
Run DFS or BFS on G^R from node u
if not all nodes are visited
        return false
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



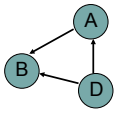
Runtime?

```
STRONGLY-CONNECTED(G)
Run DFS or BFS from some node u                    O(|V| + |E|)
if not all nodes are visited                       O(|V|)
        return false
Create graph G^R by reversing all edge directions  O(|V| + |E|)
Run DFS or BFS on G^R from node u                  O(|V| + |E|)
if not all nodes are visited                       O(|V|)
        return false
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



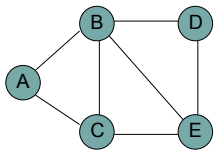
have to be careful

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

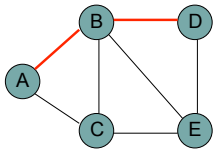
Shortest paths

- What is the shortest path from a to d?



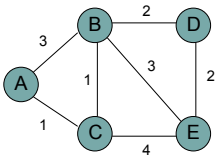
Shortest paths

- BFS



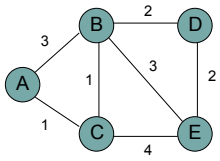
Shortest paths

- What is the shortest path from a to d?



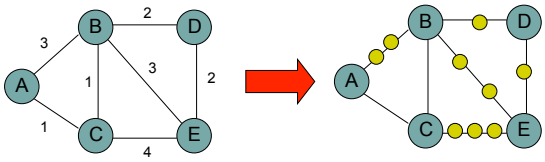
Shortest paths

- We can still use BFS



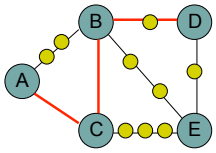
Shortest paths

- We can still use BFS



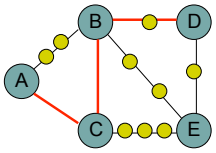
Shortest paths

- We can still use BFS



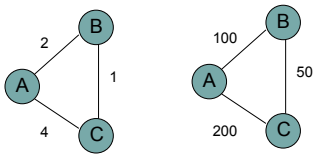
Shortest paths

- What is the problem?

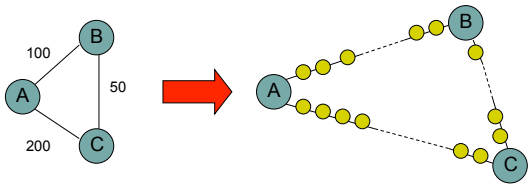


Shortest paths

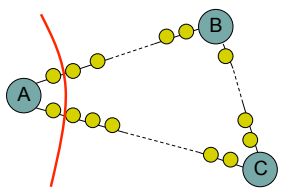
- Running time is dependent on the weights



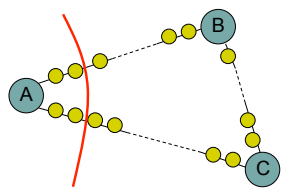
Shortest paths



Shortest paths

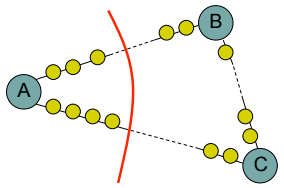


Shortest paths



Shortest paths

Nothing will change as we expand the frontier until we've gone out 100 levels



Dijkstra's algorithm

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Dijkstra's algorithm

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Dijkstra's algorithm

prev keeps track of the shortest path

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Dijkstra's algorithm

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Dijkstra's algorithm

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

Dijkstra's algorithm

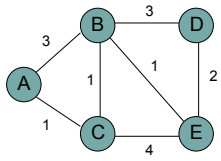
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

```
BFS(G, s)
for each v ∈ V
        dist[v] = ∞
dist[s] = 0
ENQUEUE(Q, s)
while !EMPTY(Q)
        u ← DEQUEUE(Q)
        VISIT(U)
        for each edge (u, v) ∈ E
                if dist[v] = ∞
                        ENQUEUE(Q, v)
                        dist[v] ← dist[u] + 1
```

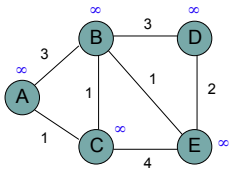
Single source shortest paths

- All of the shortest path algorithms we'll look at today are call "single source shortest paths" algorithms
- Why?

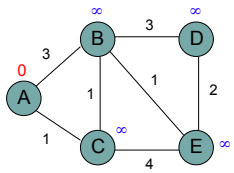
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```




```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



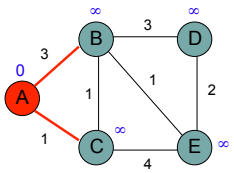
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

A 0
B ∞
C ∞
D ∞
E ∞

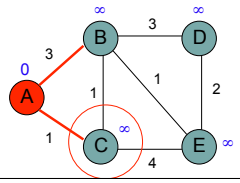
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B ∞
C ∞
D ∞
E ∞

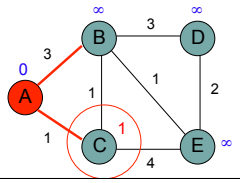
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B ∞
C ∞
D ∞
E ∞

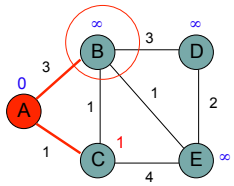
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

C 1
B ∞
D ∞
E ∞

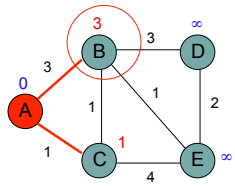
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

C 1
B ∞
D ∞
E ∞

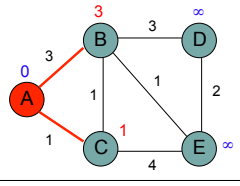
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

C 1
B 3
D ∞
E ∞

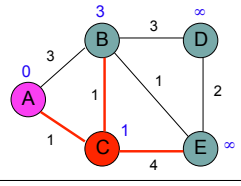
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

C 1
B 3
D ∞
E ∞

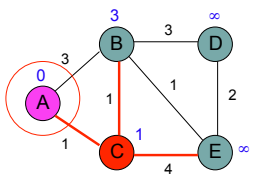
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B 3
D ∞
E ∞

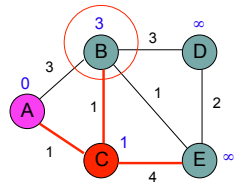
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B 3
D ∞
E ∞

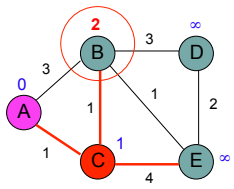
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B 3
D ∞
E ∞

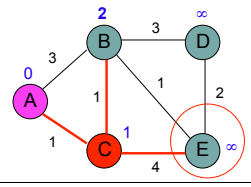
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B 2
D ∞
E ∞

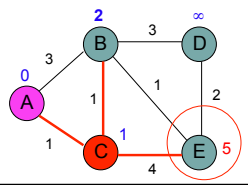
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B 2
D ∞
E ∞

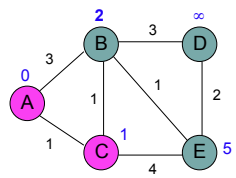
```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



Heap

B 2
E 5
D ∞

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

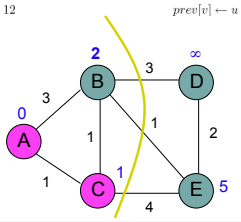


Heap

B 2
E 5
D ∞

Frontier?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

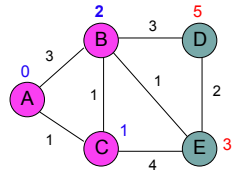


Heap

B 2
E 5
D ∞

All nodes reachable from starting node within a given distance

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```



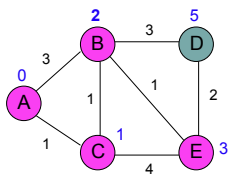
Heap

E 3
D 5

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

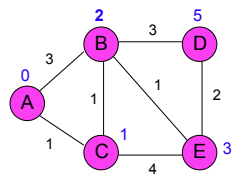
Heap

D 5



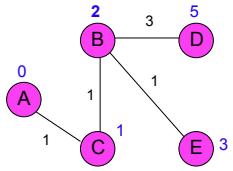

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap




```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Heap



Is Dijkstra's algorithm correct?

Invariant:

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Is Dijkstra's algorithm correct?

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

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

proof?

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

Running time?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

Running time?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

1 call to MakeHeap

Running time?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

|V| iterations

Running time?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

|V| calls

Running time?

```
DIJKSTRA(G, s)
for all v ∈ V
        dist[v] ← ∞
        prev[v] ← null
dist[s] ← 0
Q ← MAKEHEAP(V)
while !EMPTY(Q)
        u ← EXTRACTMIN(Q)
        for all edges (u, v) ∈ E
                if dist[v] > dist[u] + w(u, v)
                        dist[v] ← dist[u] + w(u, v)
                        DECREASEKEY(Q, v, dist[v])
                        prev[v] ← u
```

O(|E|) calls

Running time?

- Depends on the heap implementation

	1 MakeHeap	|V| ExtractMin	|E| DecreaseKey	Total
Array	O(|V|)	O(|V|^2)	O(|E|)	O(|V|^2)
Bin heap	O(|V|)	O(|V| log |V|)	O(|E| log |V|)	O((|V|+|E|) log |V|) O(|E| log |V|)

Running time?

- Depends on the heap implementation

	1 MakeHeap	|V| ExtractMin	|E| DecreaseKey	Total
Array	O(|V|)	O(|V|^2)	O(|E|)	O(|V|^2)
Bin heap	O(|V|)	O(|V| log |V|)	O(|E| log |V|)	O((|V|+|E|) log |V|) O(|E| log |V|)

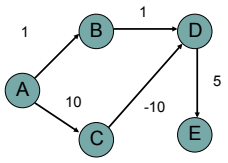
Is this an improvement? If |E| < |V|^2 / log |V|

Running time?

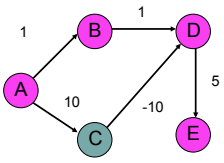
- Depends on the heap implementation

	1 MakeHeap	|V| ExtractMin	|E| DecreaseKey	Total
Array	O(|V|)	O(|V|^2)	O(|E|)	O(|V|^2)
Bin heap	O(|V|)	O(|V| log |V|)	O(|E| log |V|)	O((|V|+|E|) log |V|) O(|E| log |V|)
Fib heap	O(|V|)	O(|V| log |V|)	O(|E|)	O(|V| log |V| + |E|)

What about Dijkstra's on…?



What about Dijkstra's on…?



What about Dijkstra's on…?

Dijkstra's algorithm only works for positive edge weights