# GRAPHS: SHORTEST PATHS

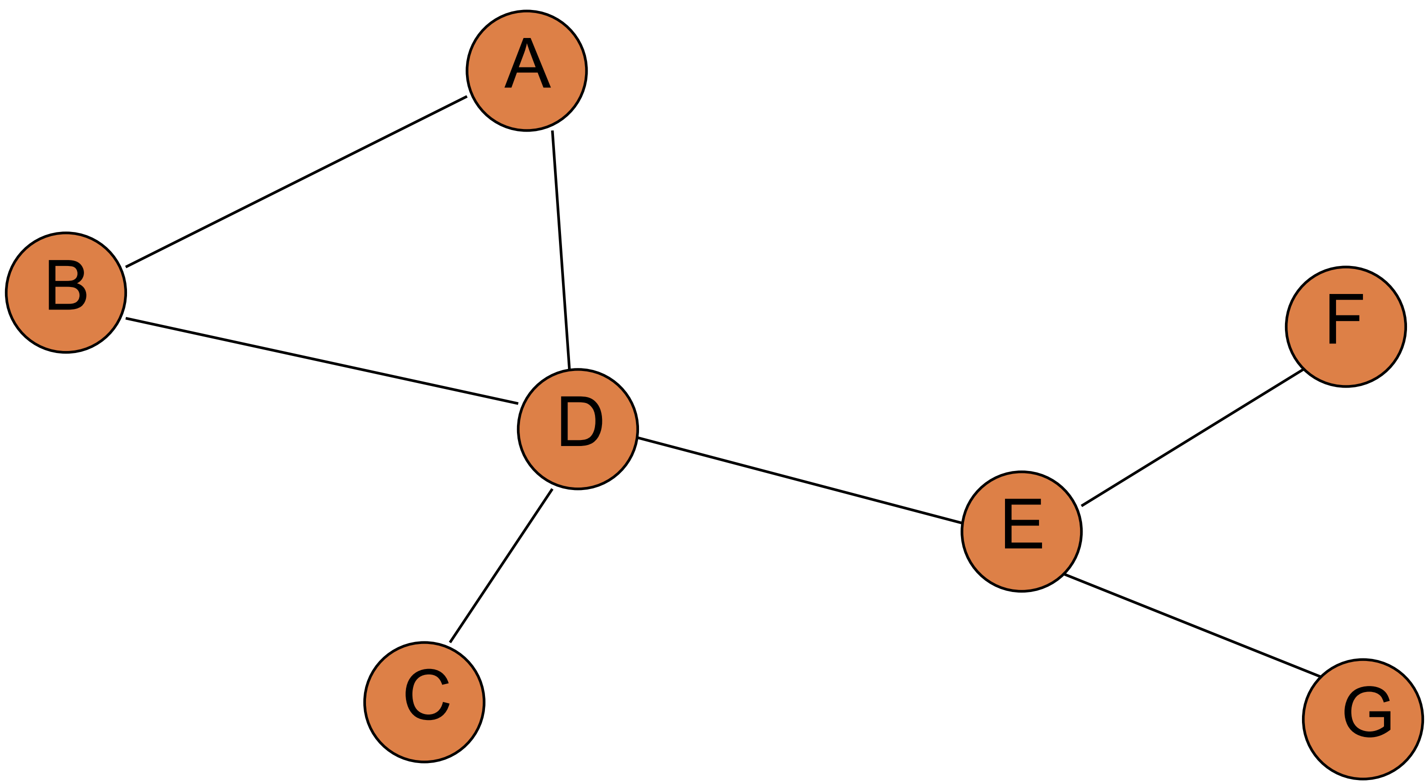
David Kauchak
CS 62 – Spring 2020

# Admin

Assignment 8

# Graphs

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



# Search

BFS: breadth first search

- Explores vertices in increasing distance (wrt number of edges) from the starting vertex
- Uses a queue to keep track of vertices to explore

DFS: depth first search:

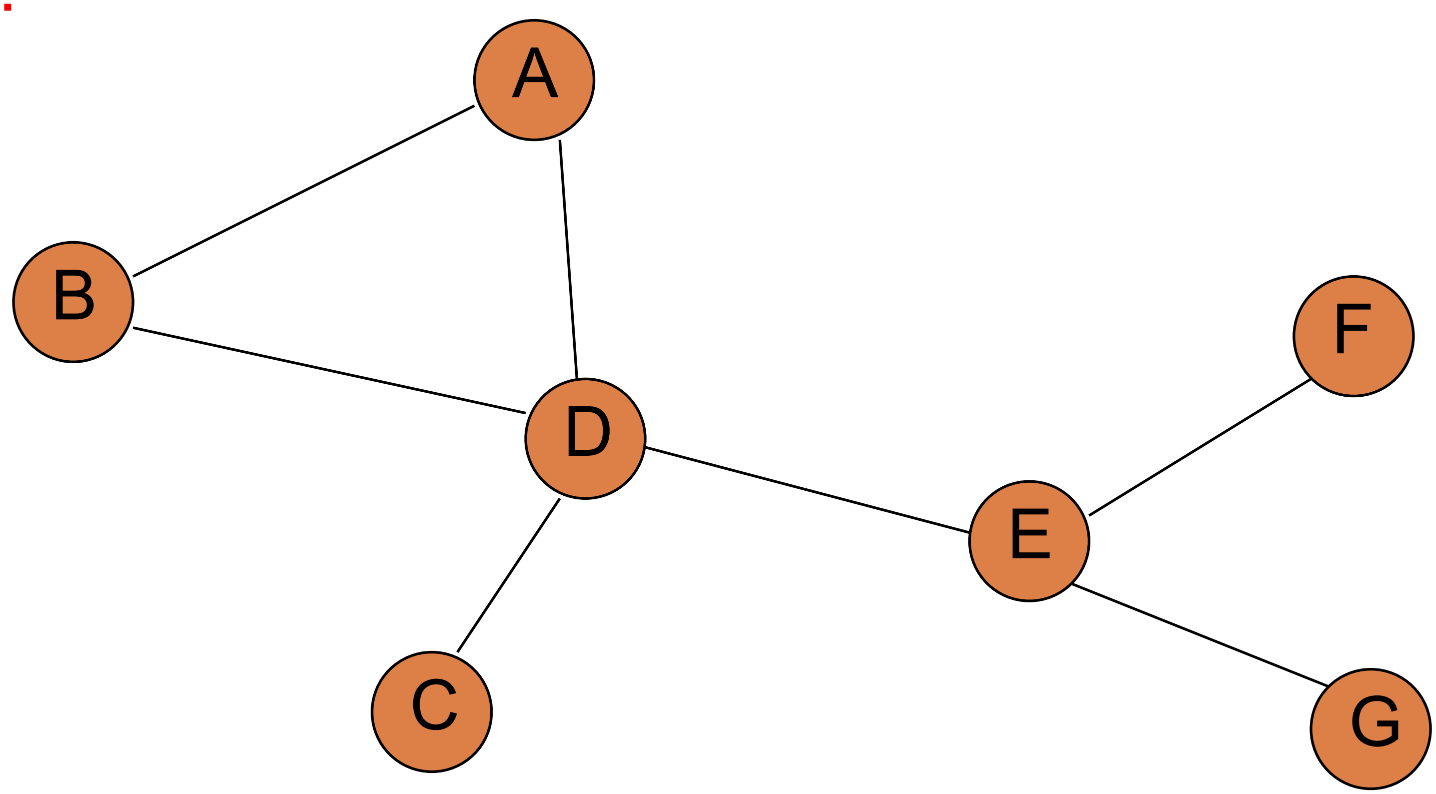
- Goes as far down a path first and then works its way back
- Two versions: stack and recursive version

Run-time: O(V + E)

# Connectedness

Connected – every pair of vertices is connected by a path

Algorithm?



# Connectedness

Connected – every pair of vertices is connected by a path

Pick any starting vertex u

Run DFS/BFS from u

For each vertex v:

    if !visited[v]

        return false

return true

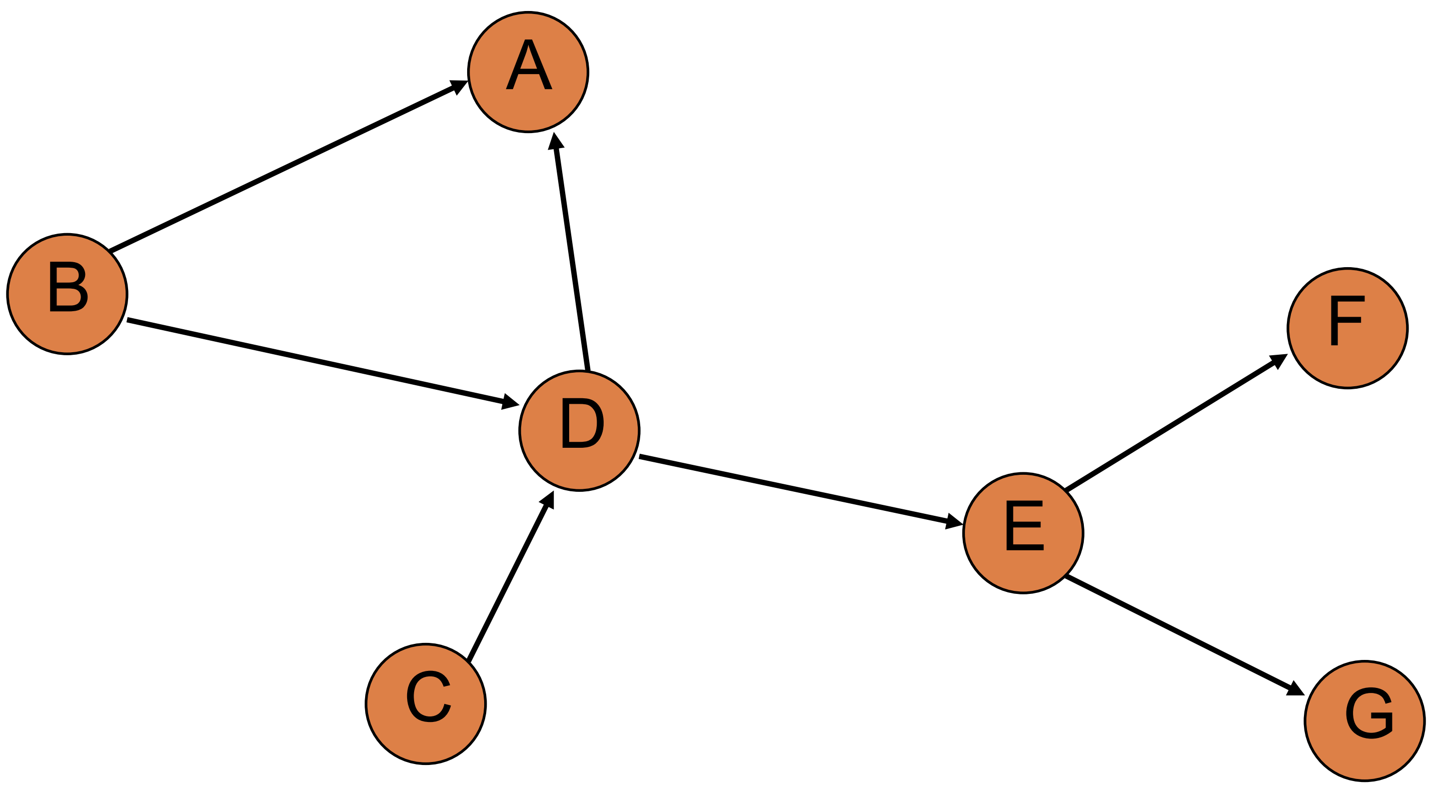
Why does this work?

If we can get from u to every vertex then we know a path exists between all vertices.

Path from a to b: a – u – b

# Strongly connected

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



# Strongly connected

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

Pick any starting vertex u

Run DFS/BFS from u

For each vertex v:

    if !visited[v]

        return false

return true

Does this work?

# Strongly connected

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

```
Pick any starting vertex u
Run DFS/BFS from u

For each vertex v:
    if !visited[v]
        return false

return true
```

Does this work?

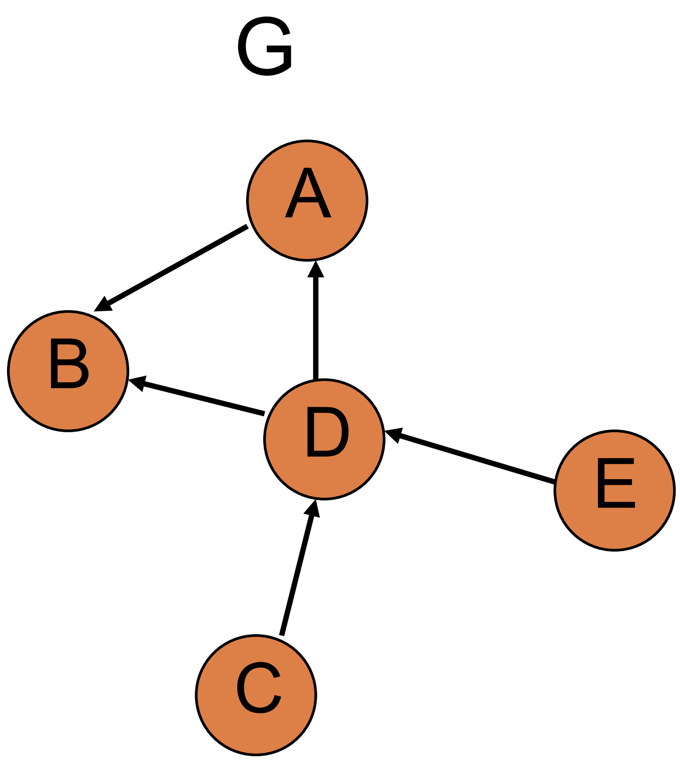
No!

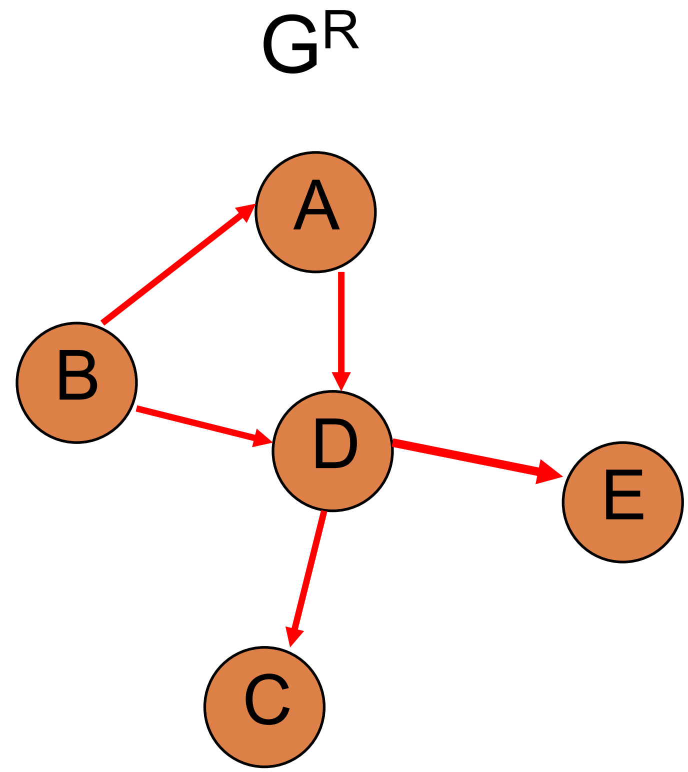
Path from a to b: a – u – b

We know we can get from u to b, but we don't know that we can get from a to s (directed graph!)

# Reverse of a graph

Given a graph G, we can calculate the reverse of a graph $G^R$ by reversing the direction of all the edges



# Strongly connected

Strongly-Connected(G)

- Run BFS/DFS from some node u
- If not all nodes are visited:
  return false
- Create graph $G^R$
- Run BFS/DFS on $G^R$ from node u
- If not all nodes are visited:
  return false
- return true

# Is it correct?

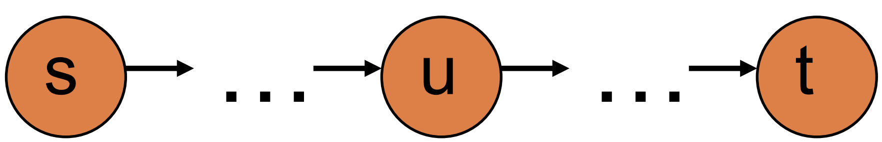
What do we know after the first search?

- Starting at u, we can reach every node

What do we know after the second search (reverse graph)?

- 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



# Run-times?

## Connectedness

```
Pick any starting vertex u
Run DFS/BFS from u

For each vertex v:
    if !visited[v]
        return false

return true
```

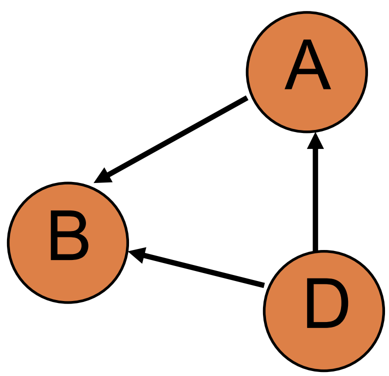
What is the run-time?

# 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

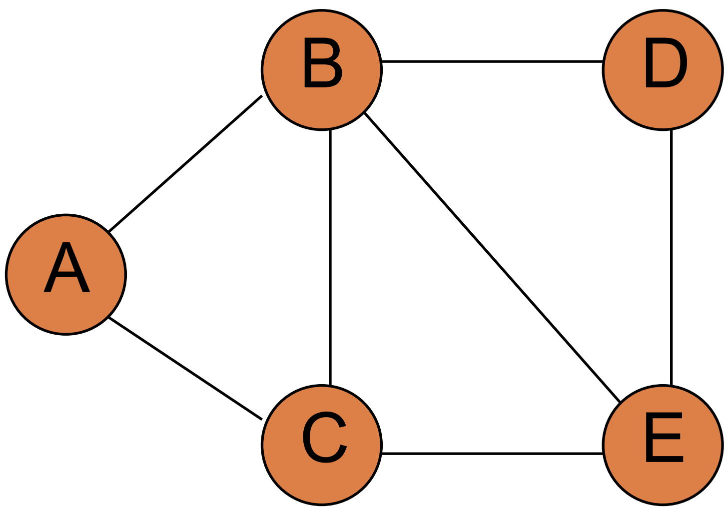
- BFS or DFS. If we reach a node we've seen already, then we've found a cycle

Directed graph

- Call TopologicalSort (more on this next week!)
- If the length of the list returned $\neq |V|$ then a cycle exists

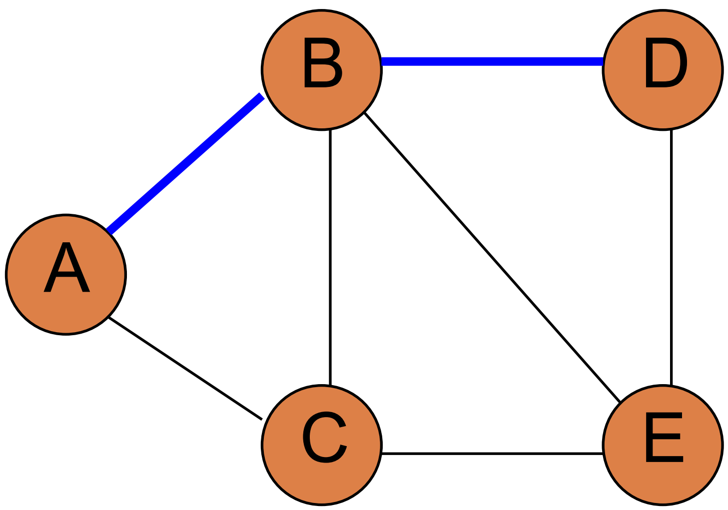
# Shortest paths

What is the shortest path from a to d?



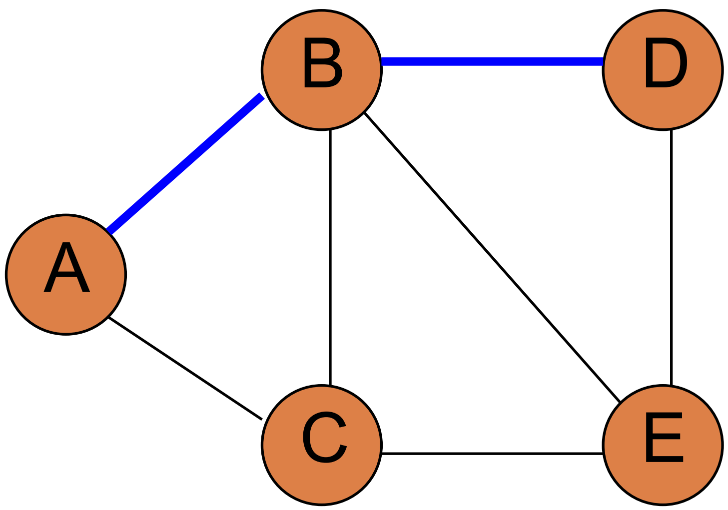
# Shortest paths

How can we find this?



# Shortest paths

BFS visits vertices in increasing distance!



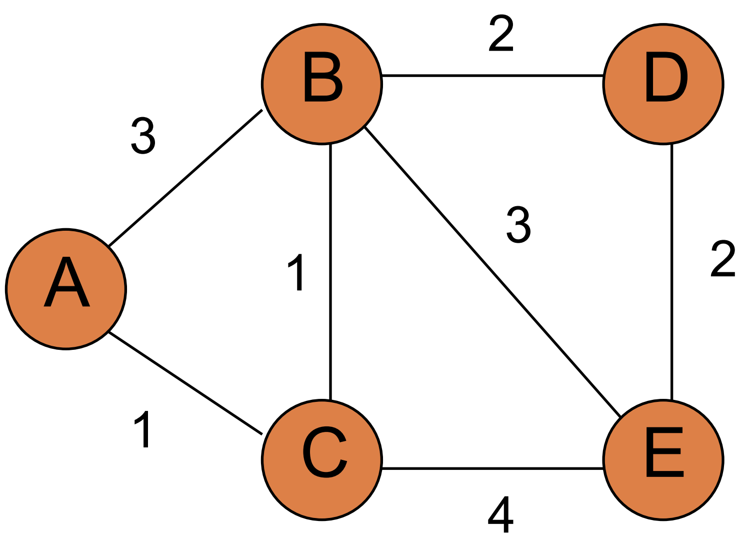
# BFS with distances

Look at ShortestPaths.bfsDistances in GraphExamples

https://github.com/pomonacs622020sp/LectureCode/tree/master/GraphExamples

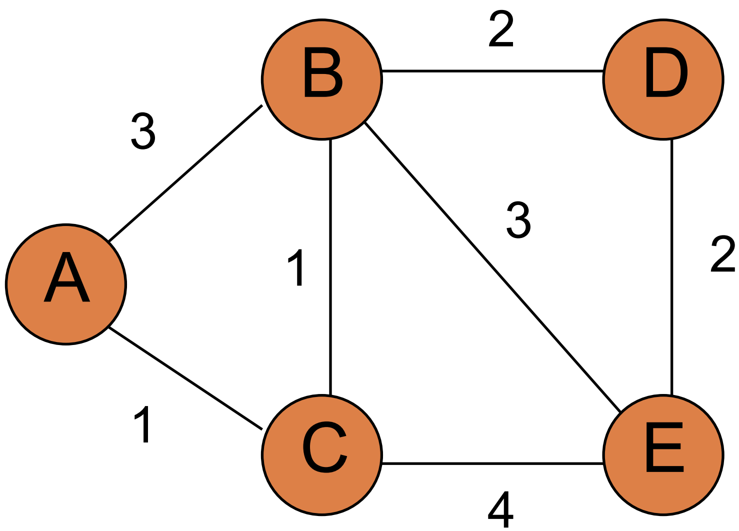
# Shortest paths

What is the shortest path from a to d?



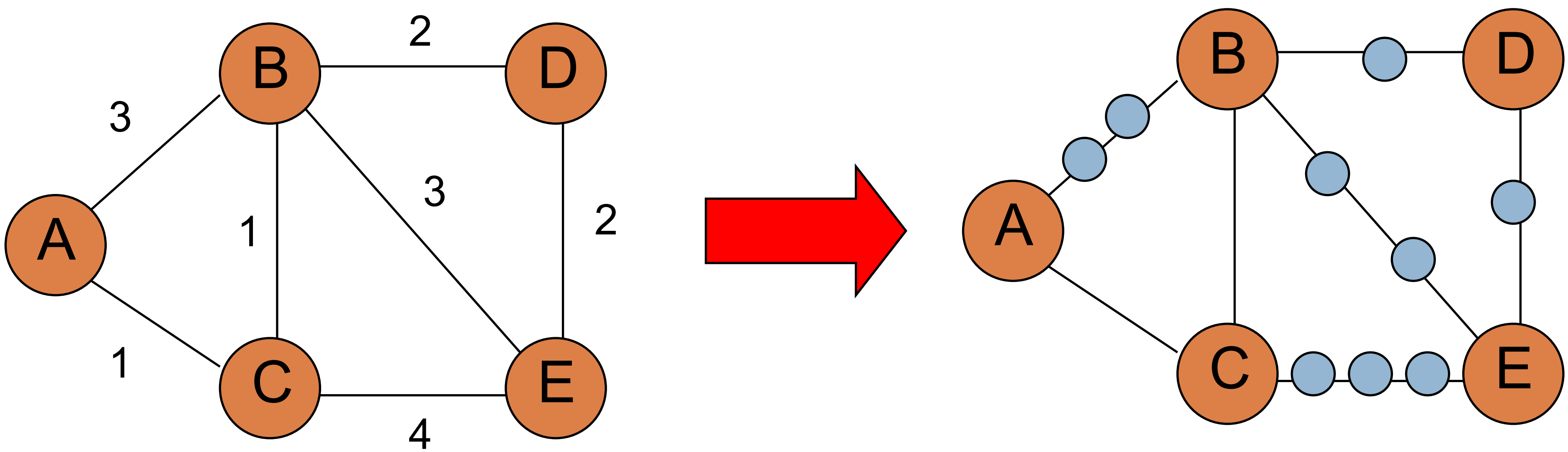
# Shortest paths

We can still use BFS



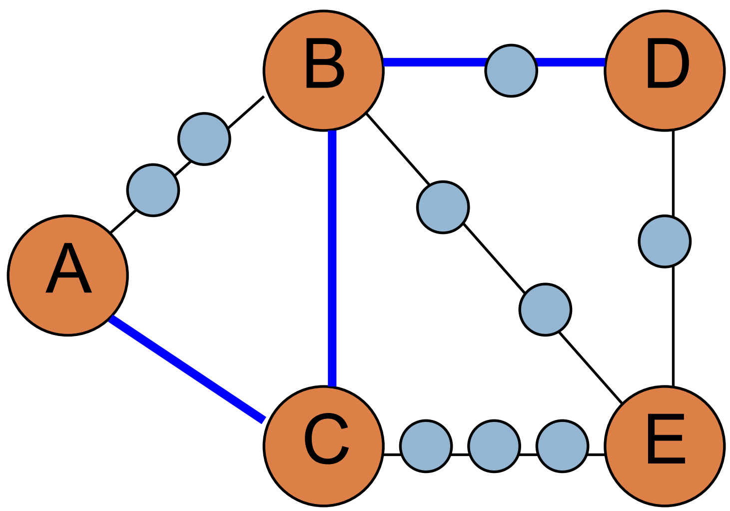
# Shortest paths

We can still use BFS



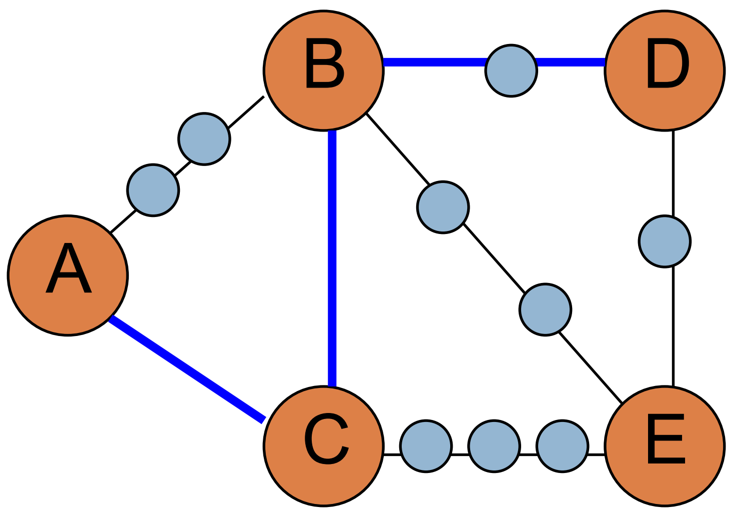
# Shortest paths

We can still use BFS



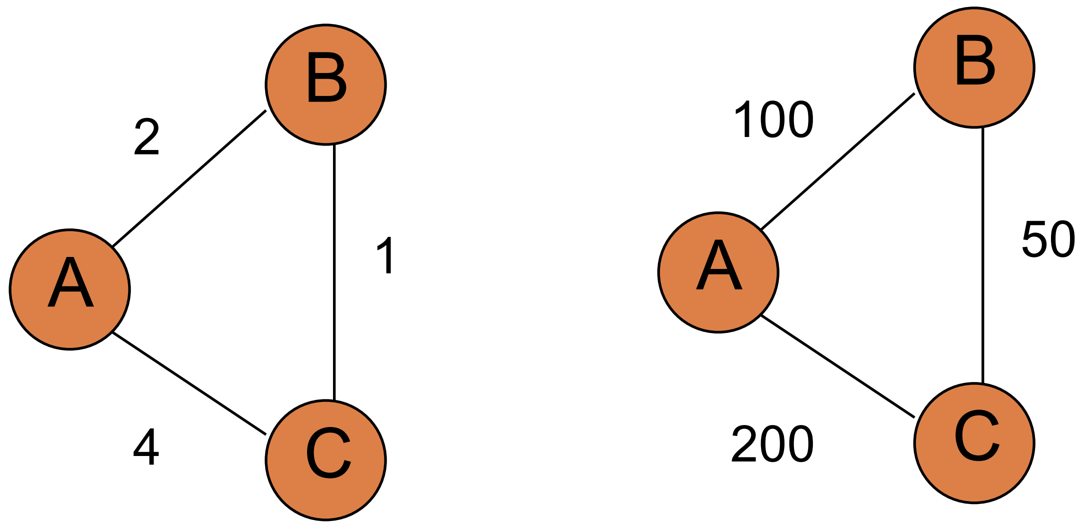
# Shortest paths

What is the problem?

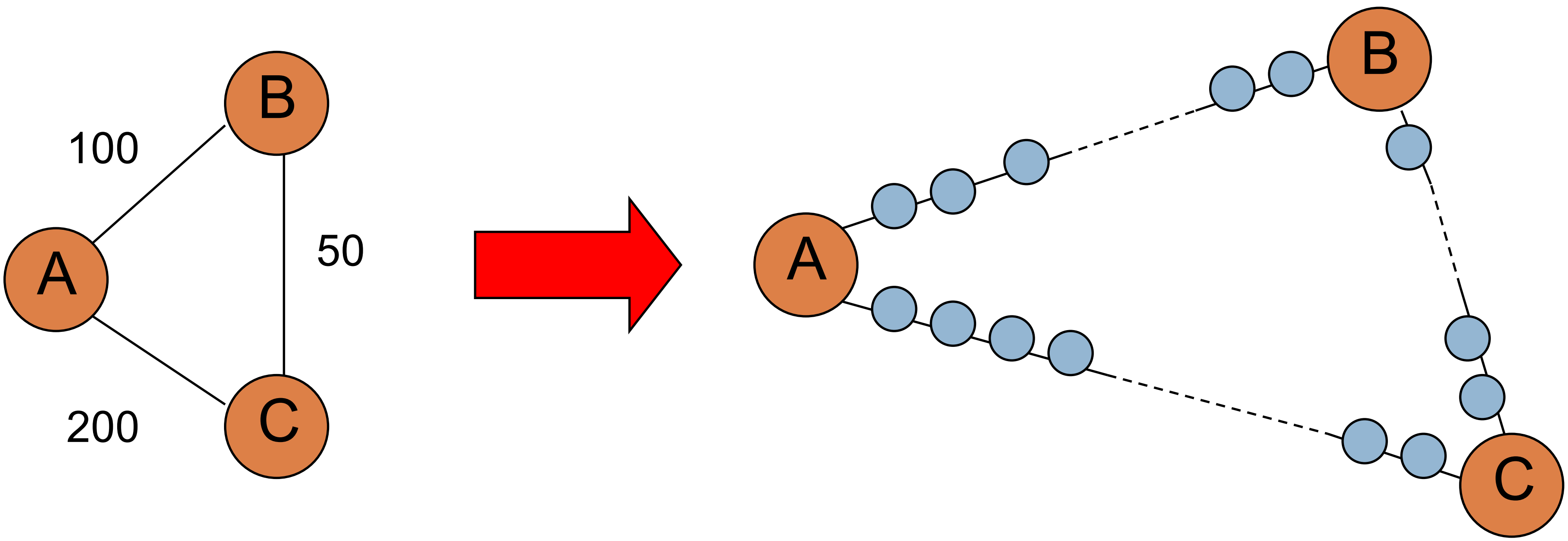


# Shortest paths

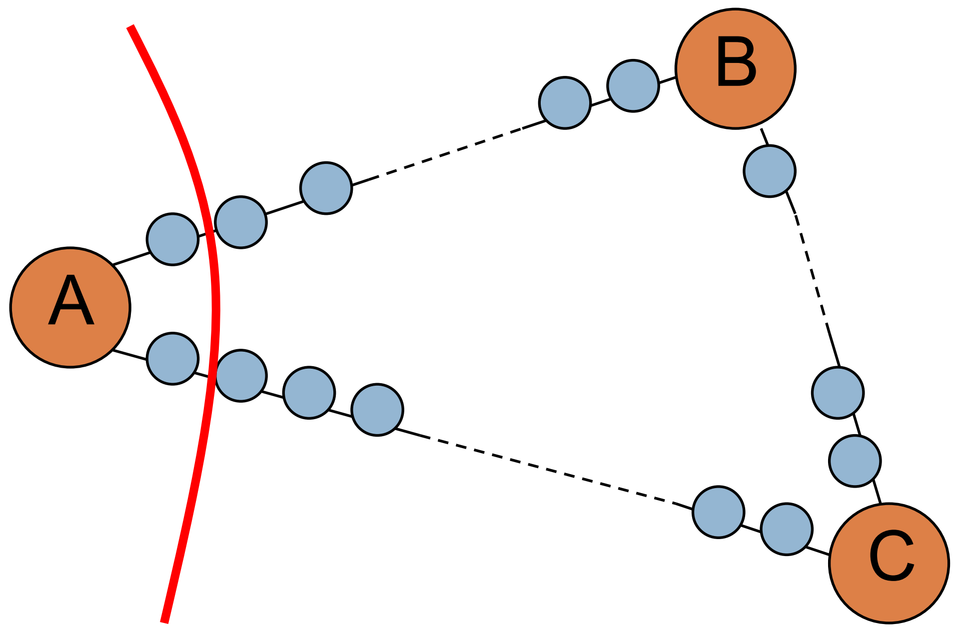
Running time is dependent on the weights!



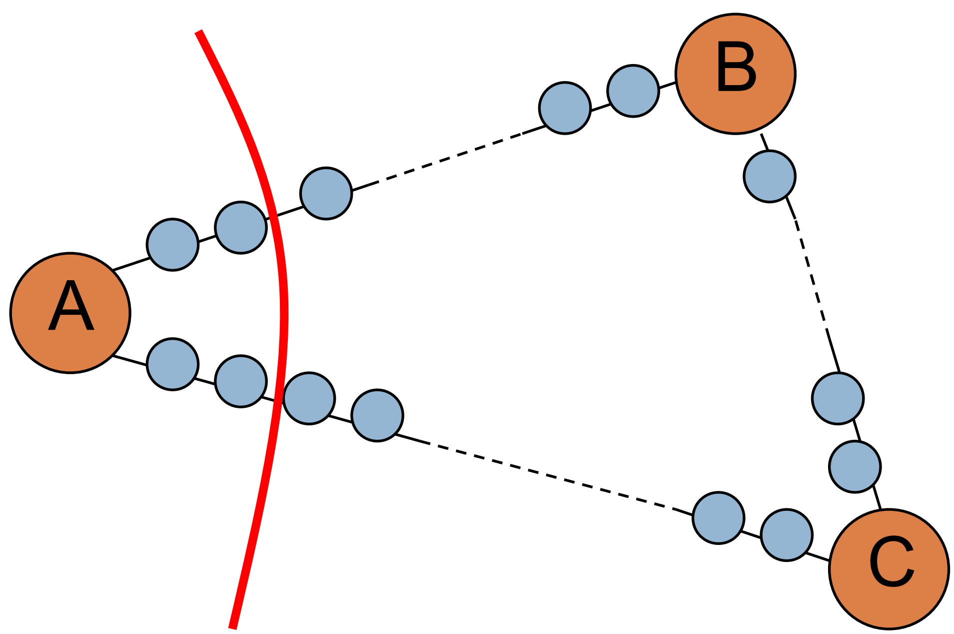
# Shortest paths



# Shortest paths

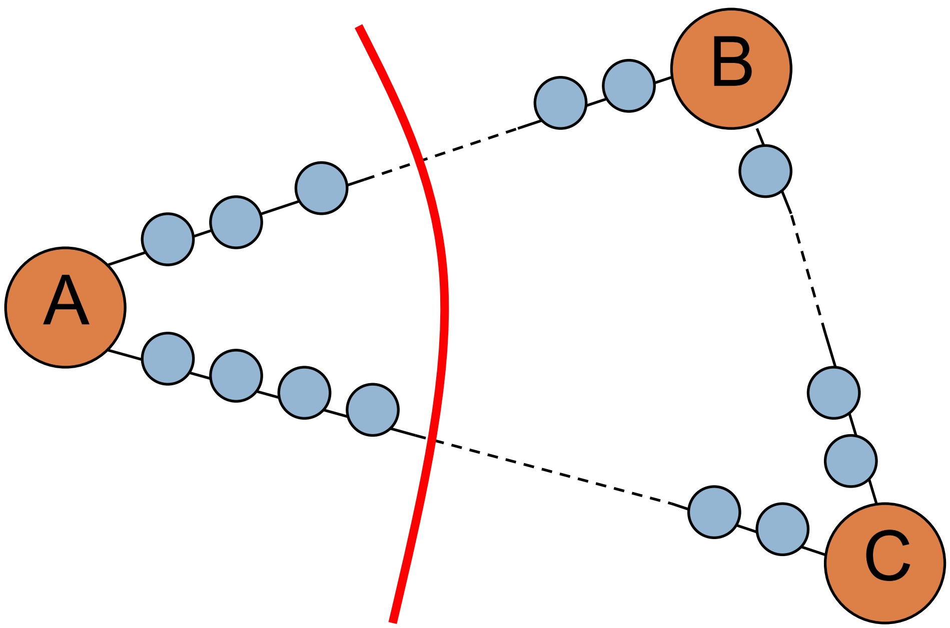


# Shortest paths



# Shortest paths

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



# Key idea

Explore the vertices in order of increasing distance from the starting vertex

Keep track of the distances to each vertex

If we find a better path, update that distance

# Dijkstra's high-level

Explore the vertices in order of increasing distance from the starting vertex

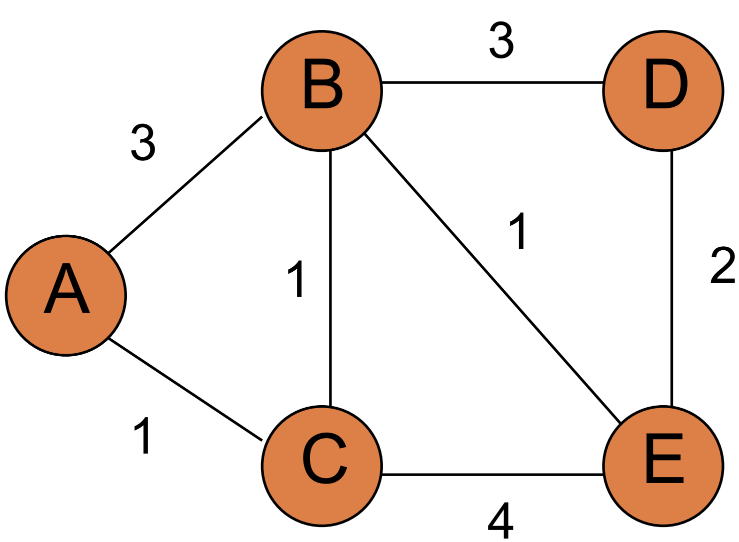
Use a priority queue to keep track of the shortest path found so far to a vertex

---

```
Initialize: distance to start = 0 and all others infinity

repeat
    get vertex v with shortest distance

    for each vertex, adj, adjacent to v (edge exists v -> adj)
        if path v -> adj is shortest then best path for adj so far
            update the distance for adj
            update the priority queue
```

Initialize: distance to start = 0 and all others infinity

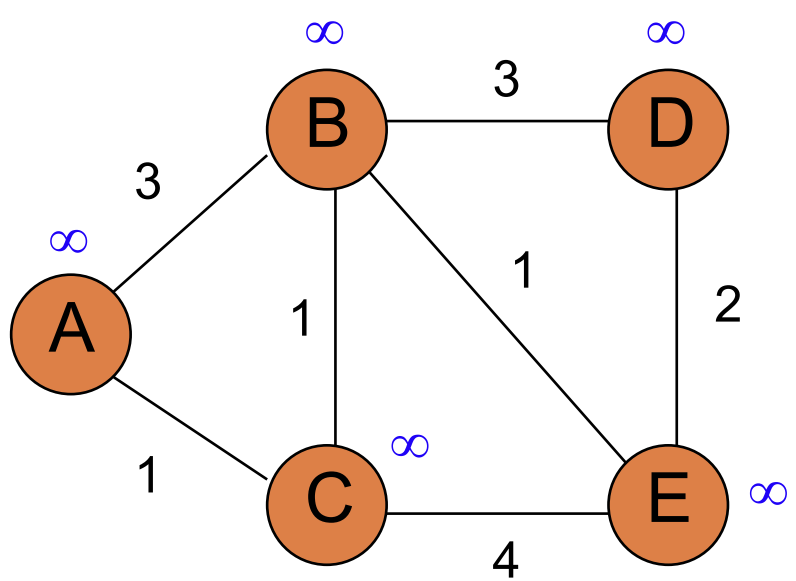
```
repeat
    get vertex v with shortest distance

    for each vertex, adj, adjacent to v (edge exists v -> adj)
        if path v -> adj is shortest then best path for adj so far
            update the distance for adj
            update the priority queue
```



Initialize: distance to start = 0 and all others infinity

repeat
    get vertex v with shortest distance

    for each vertex, adj, adjacent to v (edge exists v -> adj)
        if path v -> adj is shortest then best path for adj so far
            update the distance for adj
            update the priority queue



Initialize: distance to start = 0 and all others infinity

repeat

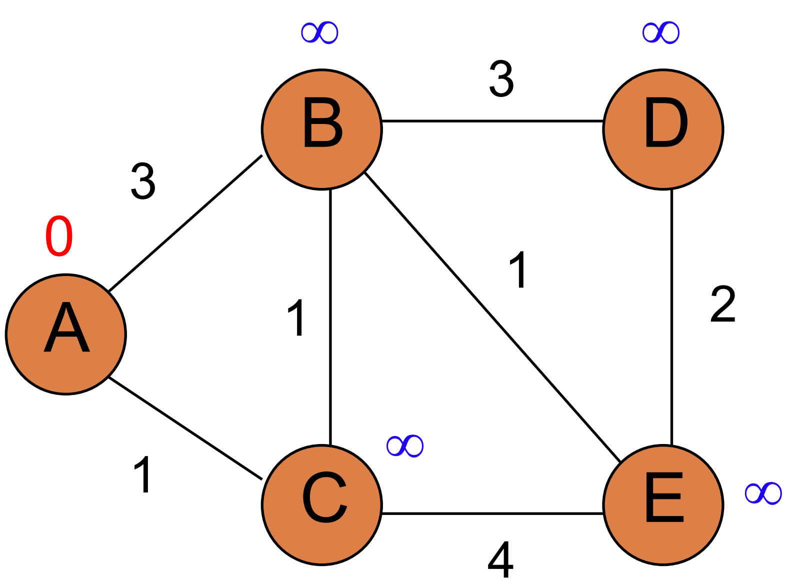
    get vertex v with shortest distance

    for each vertex, adj, adjacent to v (edge exists v -> adj)

        if path v -> adj is shortest then best path for adj so far

            update the distance for adj

            update the priority queue




**Heap**

A 0
B ∞
C ∞
D ∞
E ∞

Initialize: distance to start = 0 and all others infinity

repeat

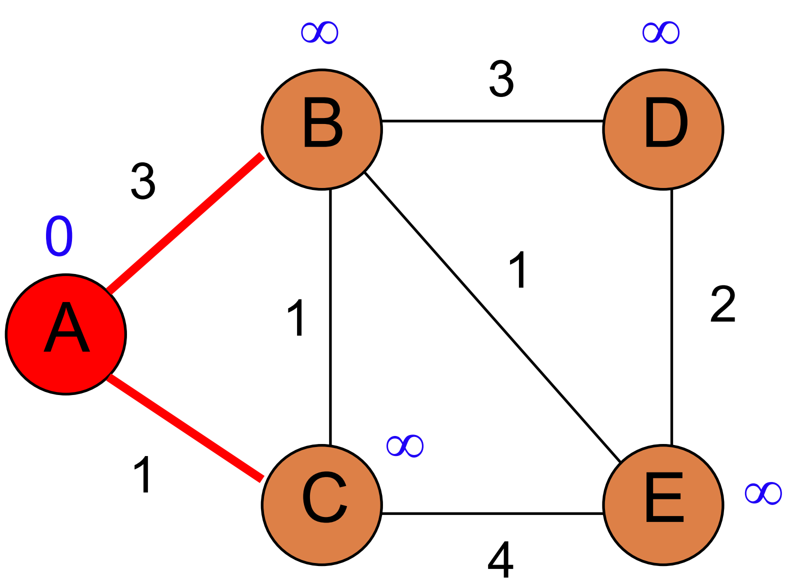
    get vertex v with shortest distance

    for each vertex, adj, adjacent to v (edge exists v -> adj)

        if path v -> adj is shortest then best path for adj so far

            update the distance for adj

            update the priority queue




**Heap**

| | |
|---|---|
| B | ∞ |
| C | ∞ |
| D | ∞ |
| E | ∞ |

Initialize: distance to start = 0 and all others infinity

repeat

get vertex v with shortest distance

for each vertex, adj, adjacent to v (edge exists v -> adj)

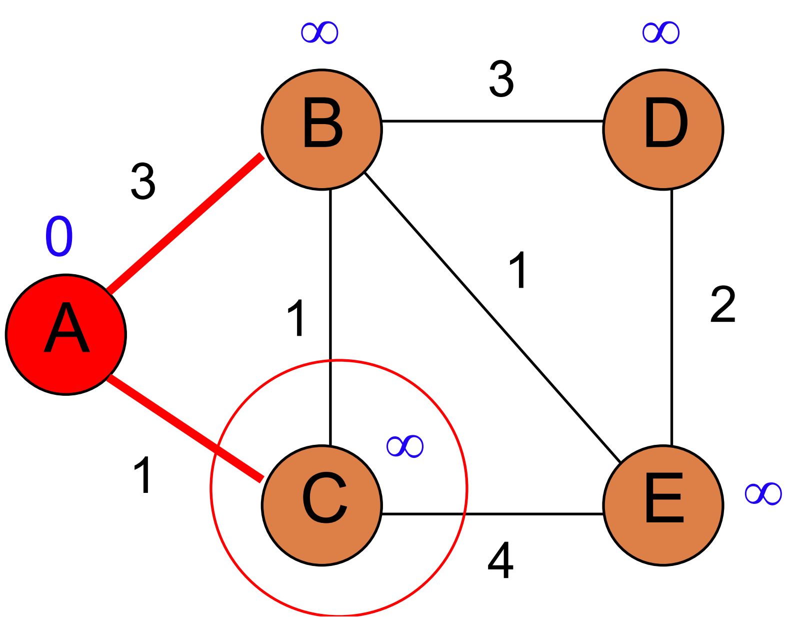
if path v -> adj is shortest then best path for adj so far

update the distance for adj

update the priority queue

Heap

| | |
|---|---|
| B | ∞ |
| C | ∞ |
| D | ∞ |
| E | ∞ |



Initialize: distance to start = 0 and all others infinity

repeat

get vertex v with shortest distance

for each vertex, adj, adjacent to v (edge exists v -> adj)

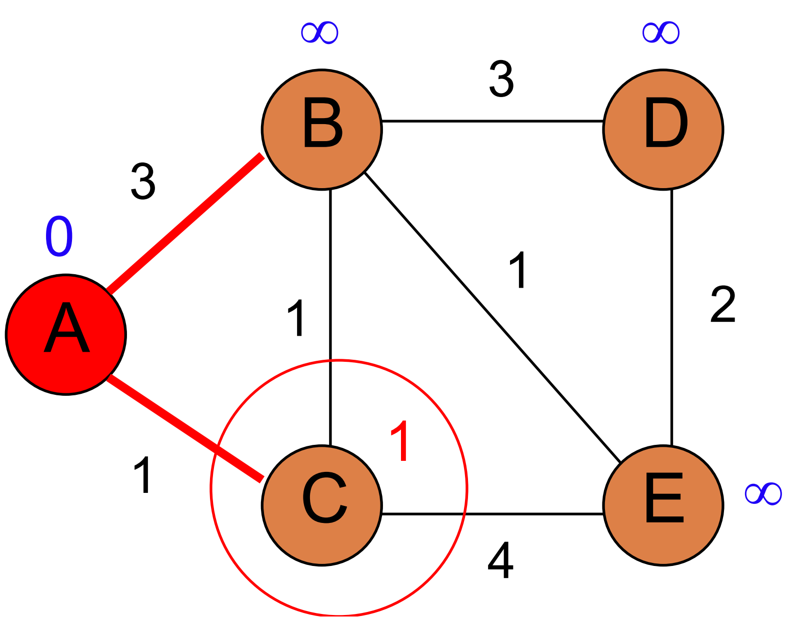
if path v -> adj is shortest then best path for adj so far

update the distance for adj

update the priority queue

Heap

C 1
B ∞
D ∞
E ∞



Initialize: distance to start = 0 and all others infinity

repeat

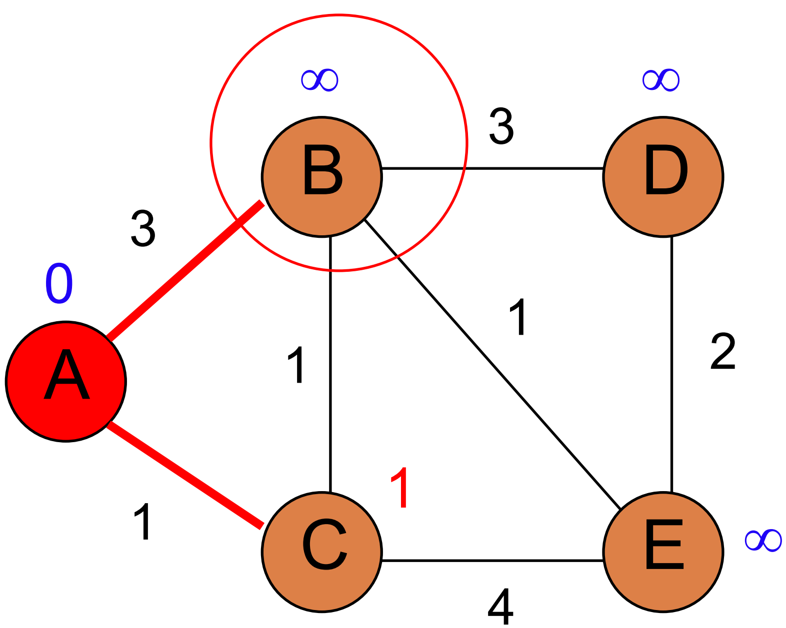
get vertex v with shortest distance

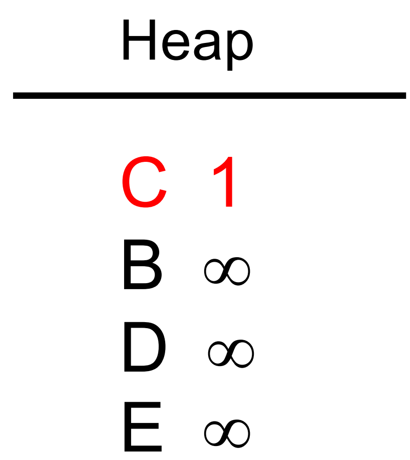
for each vertex, adj, adjacent to v (edge exists v -> adj)

if path v -> adj is shortest then best path for adj so far

update the distance for adj

update the priority queue



Initialize: distance to start = 0 and all others infinity

repeat

get vertex v with shortest distance

for each vertex, adj, adjacent to v (edge exists v -> adj)

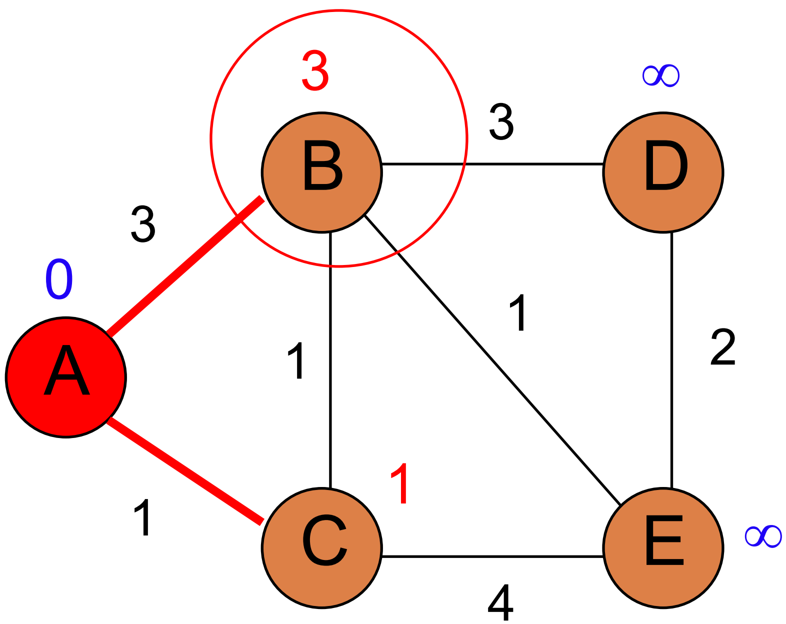
if path v -> adj is shortest then best path for adj so far

update the distance for adj

update the priority queue

Heap

C 1
B 3
D ∞
E ∞



Initialize: distance to start = 0 and all others infinity

repeat

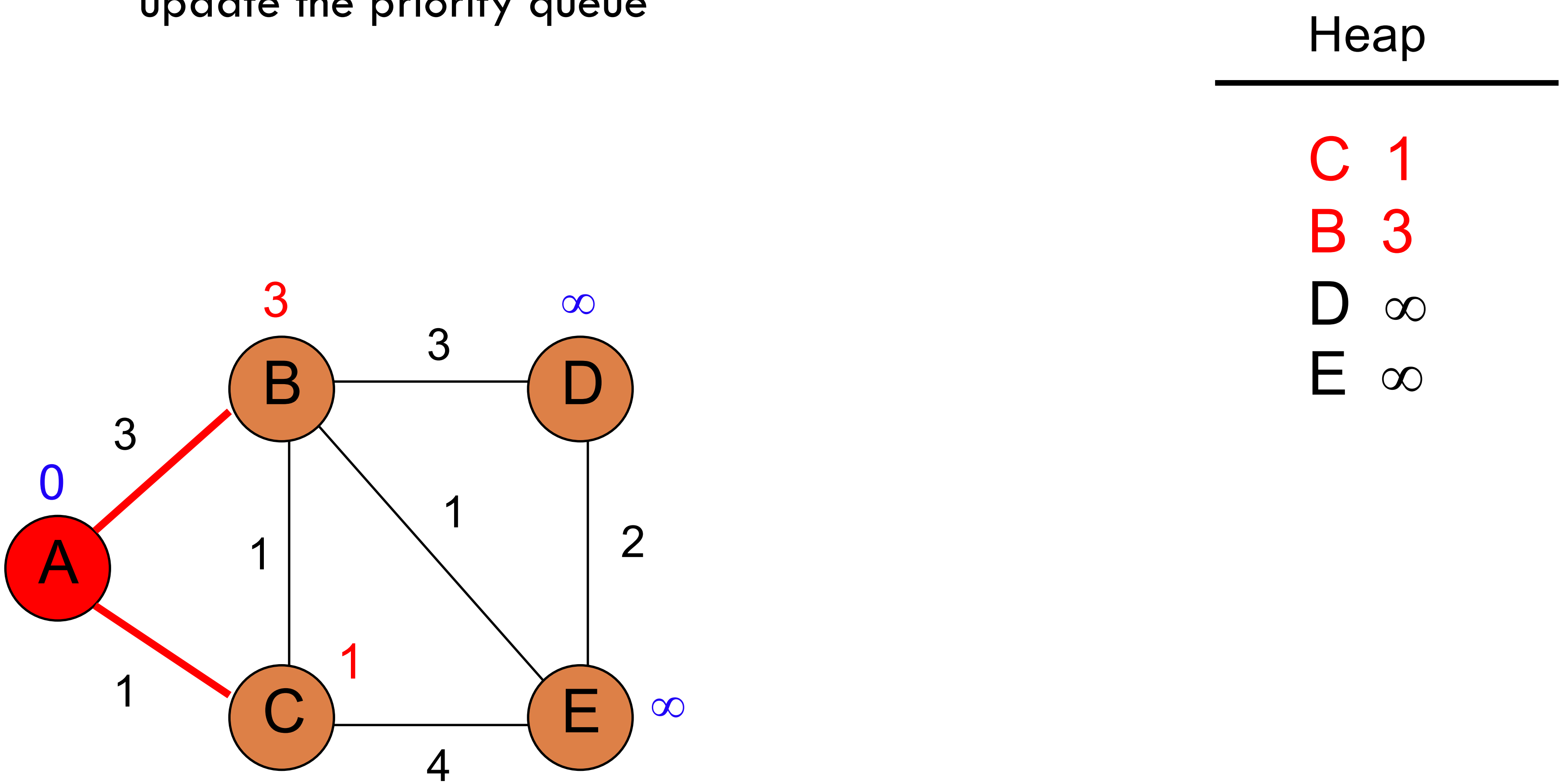
get vertex v with shortest distance

for each vertex, adj, adjacent to v (edge exists v -> adj)

if path v -> adj is shortest then best path for adj so far

update the distance for adj

update the priority queue



Initialize: distance to start = 0 and all others infinity

repeat

get vertex v with shortest distance

for each vertex, adj, adjacent to v (edge exists v -> adj)

if path v -> adj is shortest then best path for adj so far

update the distance for adj

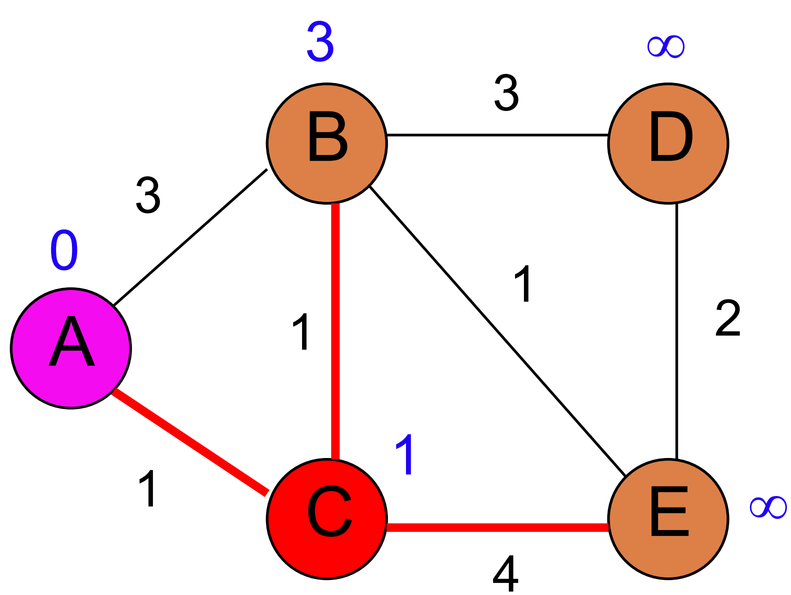
update the priority queue

Heap

B 3

D ∞

E ∞



Initialize: distance to start = 0 and all others infinity

repeat

    get vertex v with shortest distance

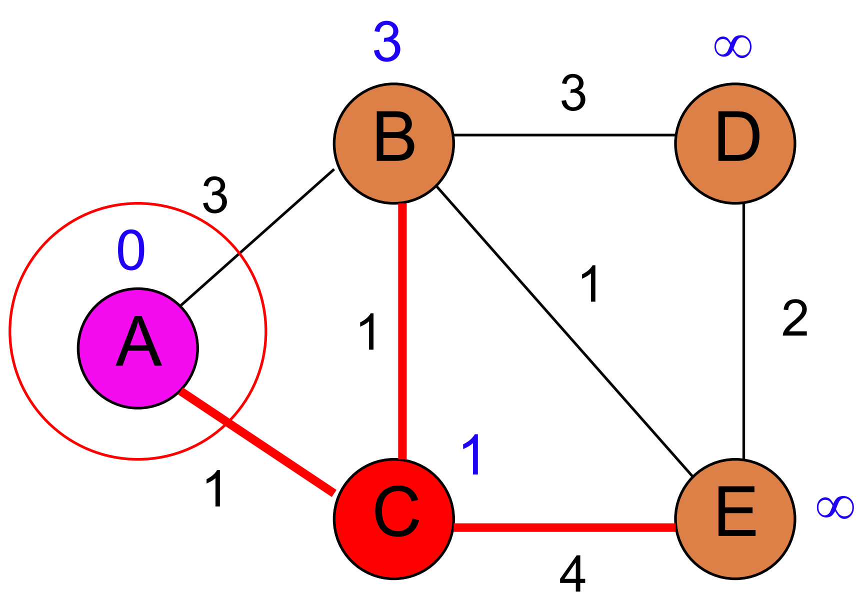
    for each vertex, adj, adjacent to v (edge exists v -> adj)

        if path v -> adj is shortest then best path for adj so far

            update the distance for adj

            update the priority queue

| Heap | |
|---|---|
| B | 3 |
| D | ∞ |
| E | ∞ |



Initialize: distance to start = 0 and all others infinity

repeat

get vertex v with shortest distance

for each vertex, adj, adjacent to v (edge exists v -> adj)

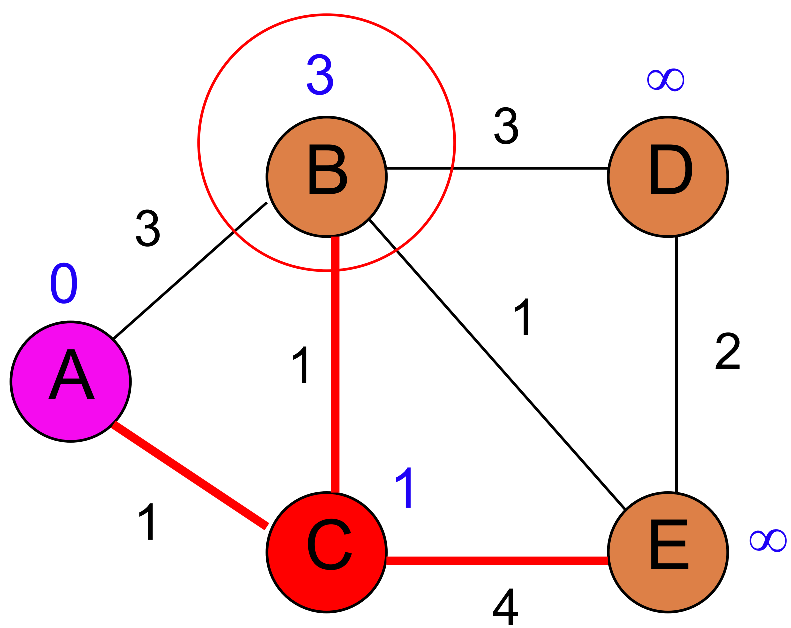
if path v -> adj is shortest then best path for adj so far

update the distance for adj

update the priority queue

Heap

| | |
|---|---|
| B | 3 |
| D | ∞ |
| E | ∞ |



Initialize: distance to start = 0 and all others infinity

repeat

get vertex v with shortest distance

for each vertex, adj, adjacent to v (edge exists v -> adj)

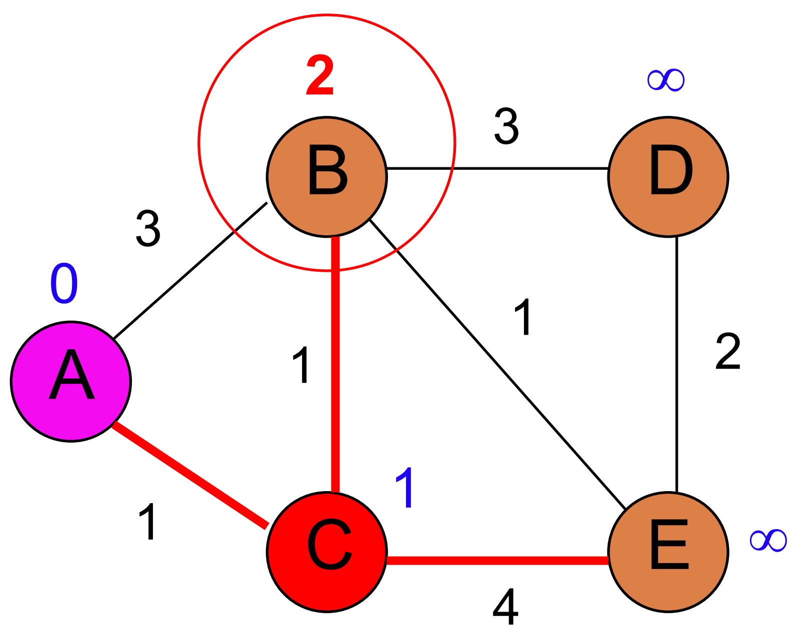
if path v -> adj is shortest then best path for adj so far

update the distance for adj

update the priority queue

Heap

B 2
D ∞
E ∞



Initialize: distance to start = 0 and all others infinity

repeat

get vertex v with shortest distance

for each vertex, adj, adjacent to v (edge exists v -> adj)

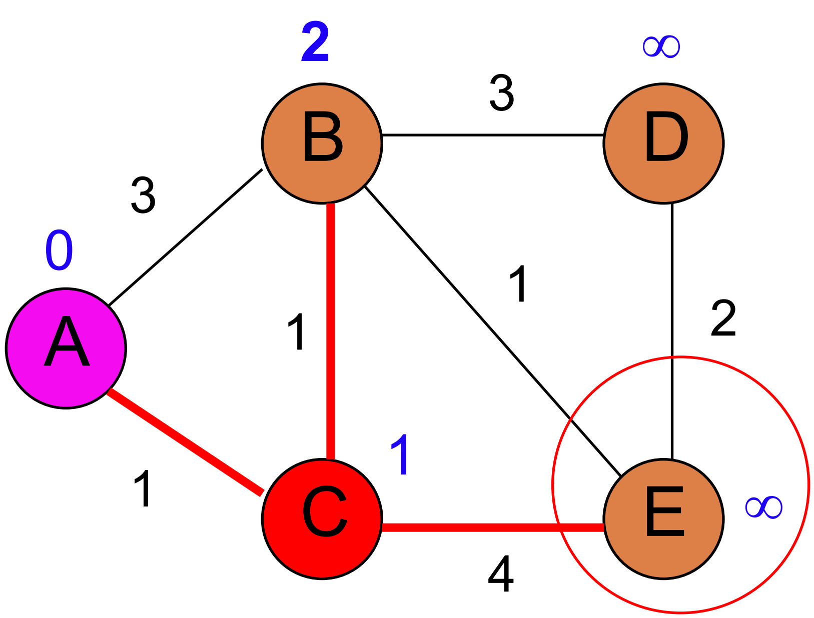
if path v -> adj is shortest then best path for adj so far

update the distance for adj

update the priority queue

Heap

| | |
|---|---|
| B | 2 |
| D | ∞ |
| E | ∞ |

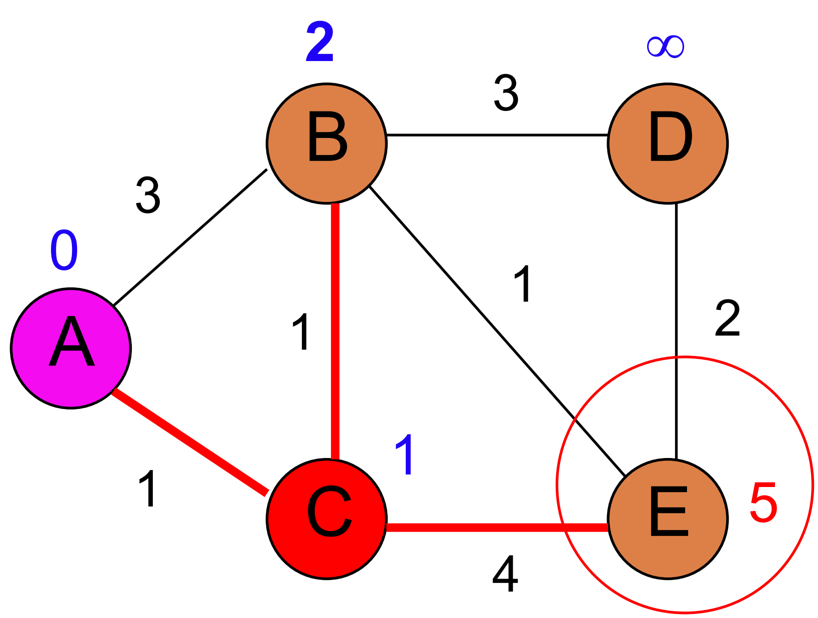


Initialize: distance to start = 0 and all others infinity

repeat
    get vertex v with shortest distance

    for each vertex, adj, adjacent to v (edge exists v -> adj)
        if path v -> adj is shortest then best path for adj so far
            update the distance for adj
            update the priority queue

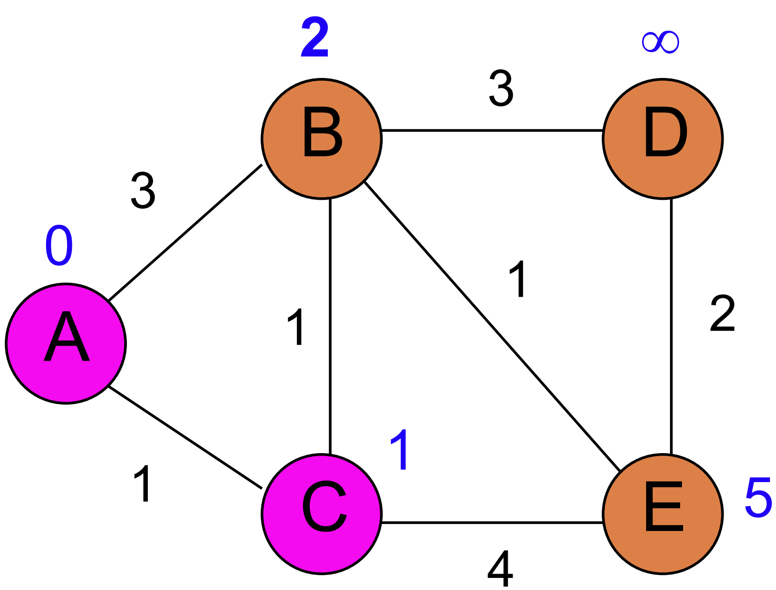
| Heap | |
|---|---|
| B | 2 |
| E | 5 |
| D | ∞ |



Initialize: distance to start = 0 and all others infinity

repeat
    get vertex v with shortest distance

    for each vertex, adj, adjacent to v (edge exists v -> adj)
        if path v -> adj is shortest then best path for adj so far
            update the distance for adj
            update the priority queue



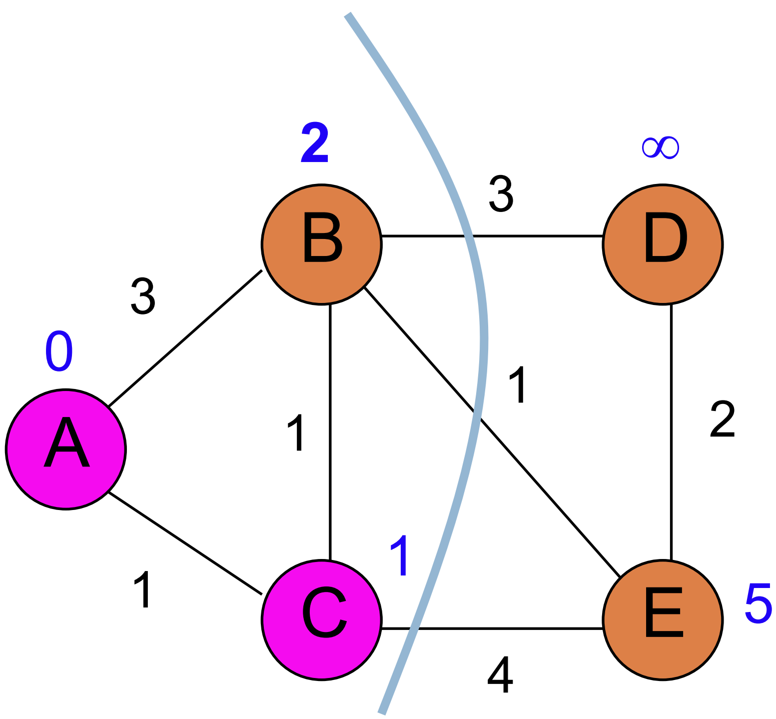

| Heap | |
|---|---|
| B | 2 |
| E | 5 |
| D | ∞ |

Frontier?

Initialize: distance to start = 0 and all others infinity

repeat
    get vertex v with shortest distance

    for each vertex, adj, adjacent to v (edge exists v -> adj)
        if path v -> adj is shortest then best path for adj so far
            update the distance for adj
            update the priority queue




| Heap | |
|---|---|
| B | 2 |
| E | 5 |
| D | ∞ |

All nodes reachable from starting node within a given distance

Initialize: distance to start = 0 and all others infinity

repeat

  get vertex v with shortest distance

  for each vertex, adj, adjacent to v (edge exists v -> adj)

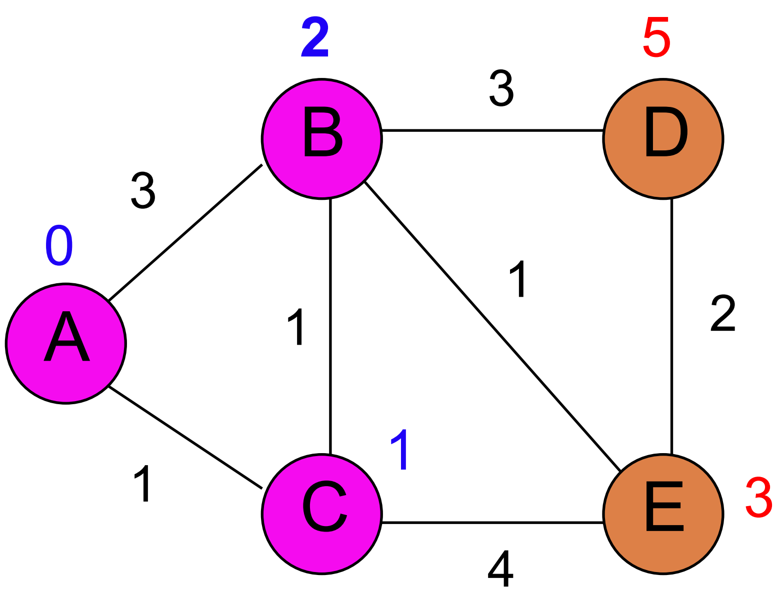
    if path v -> adj is shortest then best path for adj so far

      update the distance for adj

      update the priority queue

Heap

| | |
|---|---|
| E | 3 |
| D | 5 |



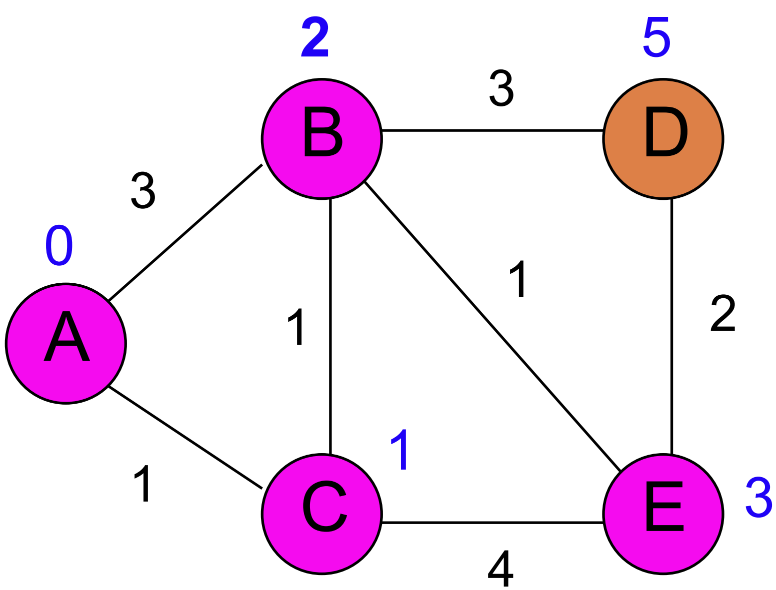
Initialize: distance to start = 0 and all others infinity

repeat
    get vertex v with shortest distance

    for each vertex, adj, adjacent to v (edge exists v -> adj)
        if path v -> adj is shortest then best path for adj so far
            update the distance for adj
            update the priority queue

Heap

D 5



Initialize: distance to start = 0 and all others infinity

repeat

get vertex v with shortest distance

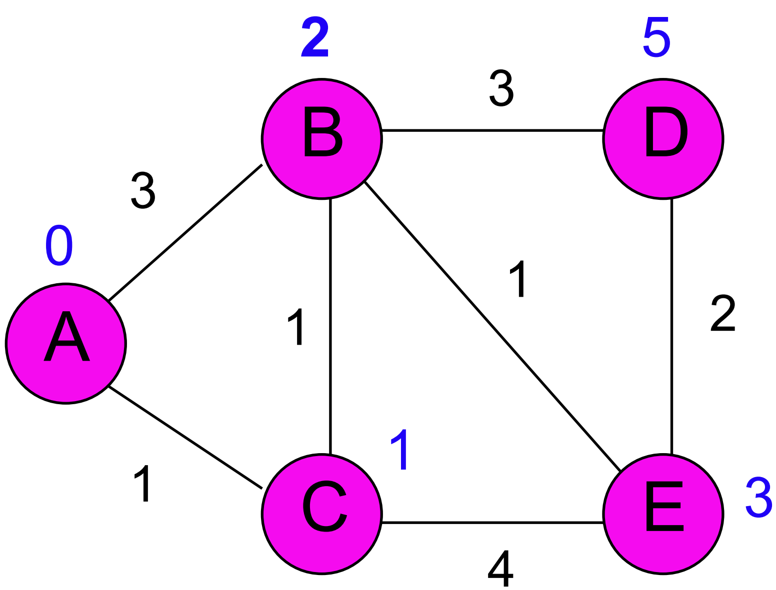
for each vertex, adj, adjacent to v (edge exists v -> adj)

if path v -> adj is shortest then best path for adj so far

update the distance for adj

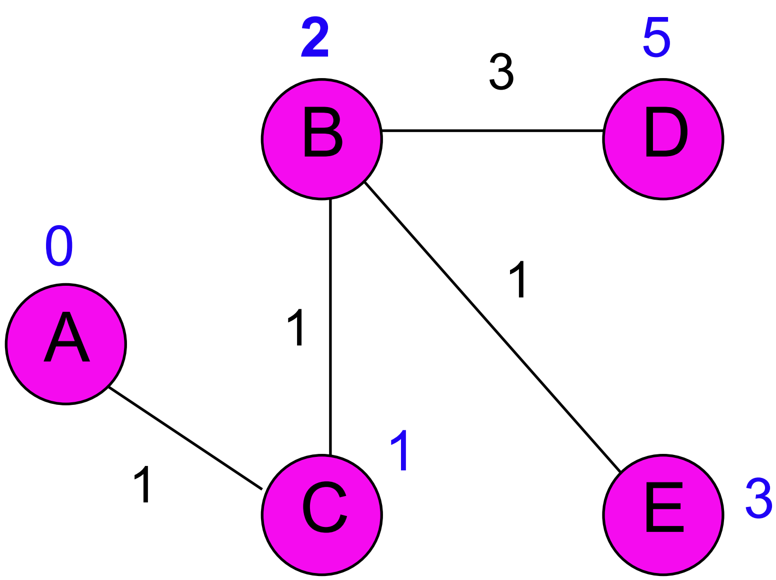
update the priority queue

Heap



```
Initialize: distance to start = 0 and all others infinity

repeat
    get vertex v with shortest distance

    for each vertex, adj, adjacent to v (edge exists v -> adj)
        if path v -> adj is shortest then best path for adj so far
            update the distance for adj
            update the priority queue
```

Heap



# Dijkstra's algorithm

```java
public static void dijkstra(WeightedGraph g, int start) {
    IndexMinPQ<Double> pq = new IndexMinPQ<Double>(g.numberOfVertices());
    int[] edgeTo = new int[g.numberOfVertices()];
    double[] distTo = new double[g.numberOfVertices()];

    for( int v = 0; v < g.numberOfVertices(); v++ ) {
        distTo[v] = Double.POSITIVE_INFINITY;
        pq.insert(v, Double.POSITIVE_INFINITY);
    }

    distTo[start] = 0.0;
    pq.decreaseKey(start, 0.0);

    // relax vertices in order of distance from s
    while( !pq.isEmpty() ) {
        int v = pq.delMin();

        for (WeightedEdge e : g.adj(v)) {
            int adj = e.to();

            if( distTo[v] + e.weight() < distTo[adj] ) {
                distTo[adj] = distTo[v] + e.weight();
                edgeTo[adj] = v;
                pq.decreaseKey(adj, distTo[adj]);
            }
        }
    }
}
```

# Dijkstra's algorithm

Dijkstra's

```
distTo[start] = 0.0;
pq.decreaseKey(start, 0.0);

while( !pq.isEmpty() ) {
    int v = pq.delMin();

    for (WeightedEdge e : g.adj(v)) {
        int adj = e.to();

        if( distTo[v] + e.weight() < distTo[adj] ) {
            distTo[adj] = distTo[v] + e.weight();
            edgeTo[adj] = v;
            pq.decreaseKey(adj, distTo[adj]);
        }
    }
}
```

BFS

```
q.addLast(start);
visited[start] = true;
distTo[start] = 0;

while( !q.isEmpty() ) {
    int v = q.removeFirst();

    for( int adj: g.adj(v) ) {
        if( !visited[adj] ) {
            visited[adj] = true;
            edgeTo[adj] = v;
            distTo[adj] = distTo[v] + 1;
            q.addLast(adj);
        }
    }
}
```

# Dijkstra example

Look at ShortestPaths.dijkstra in GraphExamples

https://github.com/pomonacs622020sp/LectureCode/tree/master/GraphExamples

# Why does it work?

When a vertex is removed from the priority queue, distTo[v] is the actual shortest distance from s to v

- The only time a vertex gets removed 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

# Why does it work?

When a vertex is removed from the priority queue, distTo[v] is the actual shortest distance from s to v

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

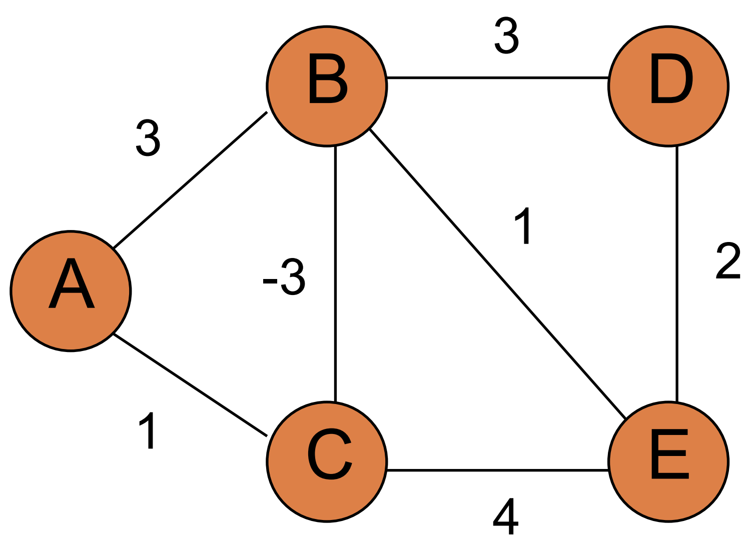
- Therefore, there cannot be any other path that hasn't been visited already that would result in a shorter path

Does this make any assumptions?

# What about this graph?

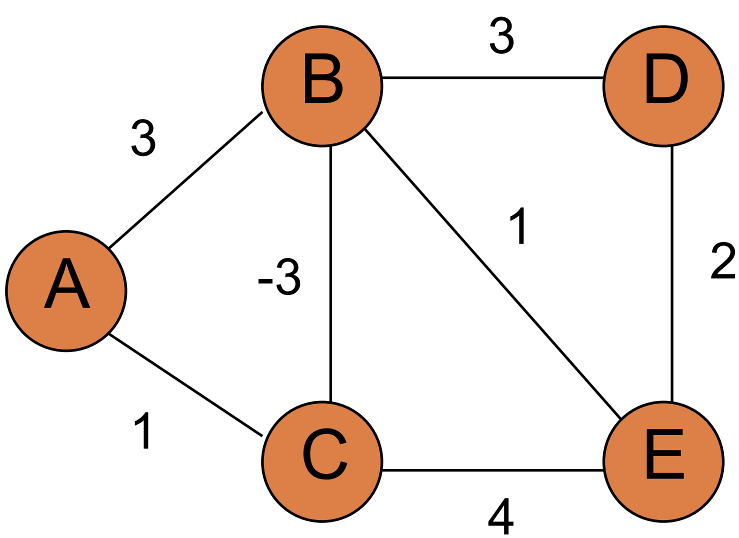
What's the shortest path from A to C?

What would Dijkstra's do?



# What about this graph?

Dijkstra's only works on graphs with positive edge weights



# Why does it work?

When a vertex is removed from the priority queue, distTo[v] is the actual shortest distance from s to v

- The only time a vertex gets removed 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

Assuming no negative edge weights!

# Relaxing an edge

This update is called “relaxing” an edge

```
if( distTo[v] + e.weight() < distTo[adj] ) {
    distTo[adj] = distTo[v] + e.weight();
    edgeTo[adj] = v;
    pq.decreaseKey(adj, distTo[adj]);
}
```

We can apply this to an edge as many times as we want

This idea is used in other shortest paths algorithms (e.g., Bellman-Ford)

```java
public static void fasterDijkstra(WeightedGraph g, int start) {
    IndexMinPQ<Double> pq = new IndexMinPQ<Double>(g.numberOfVertices());
    int[] edgeTo = new int[g.numberOfVertices()];
    double[] distTo = new double[g.numberOfVertices()];

    for( int v = 0; v < g.numberOfVertices(); v++ ) {
        distTo[v] = Double.POSITIVE_INFINITY;
    }

    distTo[start] = 0.0;
    pq.insert(start, 0.0);

    while( !pq.isEmpty() ) {
        int v = pq.delMin();

        for (WeightedEdge e : g.adj(v)) {
            int adj = e.to();

            if( distTo[v] + e.weight() < distTo[adj] ) {
                distTo[adj] = distTo[v] + e.weight();
                edgeTo[adj] = v;

                if( pq.contains(adj) ) {
                    pq.decreaseKey(adj, distTo[adj]);
                } else {
                    pq.insert(adj, distTo[adj]);
                }
            }
        }
    }
}
```

don’t insert everything into pq

only insert starting vertex

insert when we discover a vertex

# Run-time

```
public static void dijkstra(WeightedGraph g, int start) {
    IndexMinPQ<Double> pq = new IndexMinPQ<Double>(g.numberOfVertices());
    int[] edgeTo = new int[g.numberOfVertices()];
    double[] distTo = new double[g.numberOfVertices()];

    for( int v = 0; v < g.numberOfVertices(); v++ ) {
        distTo[v] = Double.POSITIVE_INFINITY;
        pq.insert(v, Double.POSITIVE_INFINITY);
    }

    distTo[start] = 0.0;
    pq.decreaseKey(start, 0.0);

    // relax vertices in order of distance from s
    while( !pq.isEmpty() ) {
        int v = pq.delMin();

        for (WeightedEdge e : g.adj(v)) {
            int adj = e.to();

            if( distTo[v] + e.weight() < distTo[adj] ) {
                distTo[adj] = distTo[v] + e.weight();
                edgeTo[adj] = v;
                pq.decreaseKey(adj, distTo[adj]);
            }
        }
    }
}
```

V calls

E calls

# Running time?

Depends on the heap implementation

| | V * delMin | E * decreaseKey | Total |
|---|---|---|---|
| Array | $O(\|V\|^2)$ | $O(\|E\|)$ | $O(\|V\|^2)$ |
| Bin heap | $O(\|V\| \log \|V\|)$ | $O(\|E\| \log \|V\|)$ | $O((\|V\|+\|E\|) \log \|V\|)$<br>$O(\|E\| \log \|V\|)$ |

# Running time?

Depends on the heap implementation

| | V * delMin | E * decreaseKey | Total |
|---|---|---|---|
| Array | $O(\|V\|^2)$ | $O(\|E\|)$ | $O(\|V\|^2)$ |
| Bin heap | $O(\|V\| \log \|V\|)$ | $O(\|E\| \log \|V\|)$ | $O((\|V\|+\|E\|) \log \|V\|)$<br>$O(\|E\| \log \|V\|)$ |
| Fib heap | $O(\|V\| \log \|V\|)$ | $O(\|E\|)$ | $O(\|V\| \log \|V\| + \|E\|)$ |

# Shortest paths

Dijkstra's: single source shortest paths for positive edge weight graphs

What is single source?

# Shortest paths

Dijkstra's: single source shortest paths for positive edge weight graphs

Many other variants:

- graphs with negative edges
- all pairs shortest paths
- …