

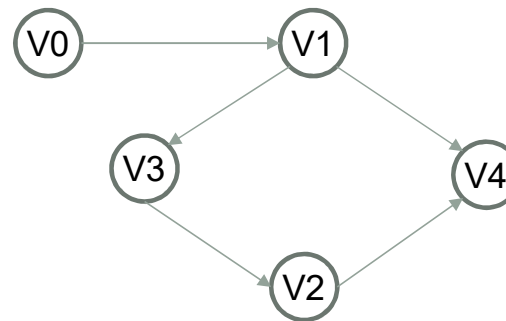
CSE 100: GRAPH

Announcements

- PA3
 - Checkpoint deadline 11:59pm on Thursday, November 29 (No slip days)
 - Final submission deadline 11:59pm on Thursday, December 6 (slip days allowed)
- Participation points
 - Will be out today. Please check csemoodle website.

BFS Traverse: Idea

- Input: an unweighted directed graph $G = (V, E)$ and a source vertex s in V
- Output: for each vertex v in V , a representation of the shortest path in G that starts in s and ends at v



Start at s . It has distance 0 from itself.

Consider nodes adjacent to s . They have distance 1. Mark them as visited.

Then consider nodes that have not yet been visited

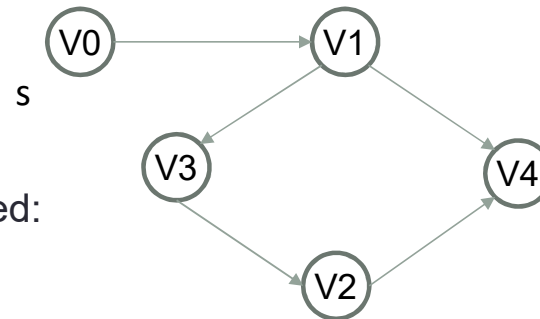
adjacent to those at distance 1. They have distance 2. Mark them as visited.

Etc. etc. until all nodes are visited.

BFS Traverse: Sketch of Algorithm

The basic idea is a breadth-first search of the graph, starting at source vertex s

- Initially, give all vertices in the graph a distance of INFINITY
- Start at s ; give s distance = 0
- Enqueue s into a queue
- While the queue is not empty:
 - Dequeue the vertex v from the head of the queue
 - For each of v 's adjacent nodes that has not yet been visited:
 - Mark its distance as 1 + the distance to v
 - Enqueue it in the queue



Queue:

BFS Traverse: Sketch of Algorithm

The basic idea is a breadth-first search of the graph, starting at source vertex s

- Initially, give all vertices in the graph a distance of INFINITY
- Start at s ; give s distance = 0
- Enqueue s into a queue
- While the queue is not empty:
 - Dequeue the vertex v from the head of the queue
 - For each of v 's adjacent nodes that has not yet been visited:
 - Mark its distance as $1 +$ the distance to v
 - Enqueue it in the queue

Questions:

- What data do you need to keep track of for each node?
- How can you tell if a vertex has been visited yet?
- This algorithm finds the length of the shortest path from s to all nodes. How can you also find the path itself?

BFS Traverse: Details

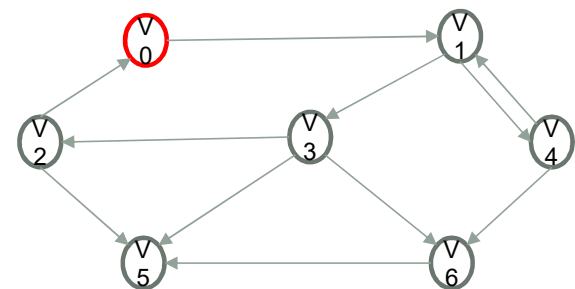
source

V0: dist=	prev=	adj: V1
V1: dist=	prev=	adj: V3, V4
V2: dist=	prev=	adj: V0, V5
V3: dist=	prev=	adj: V2, V5, V6
V4: dist=	prev=	adj: V1, V6
V5: dist=	prev=	adj:
V6: dist=	prev=	adj: V5

The queue (give source vertex dist=0 and prev=-1 and enqueue to start):

HEAD

TAIL



Representing the graph with structs

```
#include <iostream>
#include <limits>
#include <vector>
#include <queue>

using namespace std;

struct Vertex {
    vector<int> adj; // The adjacency list
    int dist;       // The distance from the source
    int index;      // The index of this vertex
    int prev;       // The index of the vertex previous in the path
};

vector<Vertex*> createGraph() {
...
}
```

Unweighted Shortest Path: C++ code

```
/** Traverse the graph using BFS */  
void BFSTraverse( vector<Vertex*> theGraph, int from )  
{  
    // assume code to initialize each Vertex's dist to INFINITY  
    queue<Vertex*> toExplore;  
    Vertex* start = theGraph[from];  
    // finish the code...
```

```
}
```

```
struct Vertex  
{  
    vector<int> adj;  
    int dist;  
    int index;  
    int prev;  
};
```


Unweighted Shortest Path: C++ code

```
/** Traverse the graph using BFS */
void BFSTraverse( vector<Vertex*> theGraph, int from )
{
    // assume code to initialize each Vertex's dist to INFINITY
    queue<Vertex*> toExplore;
    Vertex* start = theGraph[from];
    start->dist = 0;
    toExplore.push(start);
    while ( !toExplore.empty() ) {

        Vertex* next = toExplore.front();
        toExplore.pop();
        vector<int>::iterator it = next->adj.begin();
        for ( ; it != next->adj.end(); ++it ) {
            Vertex* neighbor = theGraph[*it];
            if (next->dist+1 < neighbor->dist) {
                neighbor->dist = next->dist + 1;
                neighbor->prev = next->index;
                toExplore.push(neighbor);
            }
        }
    }
}
```

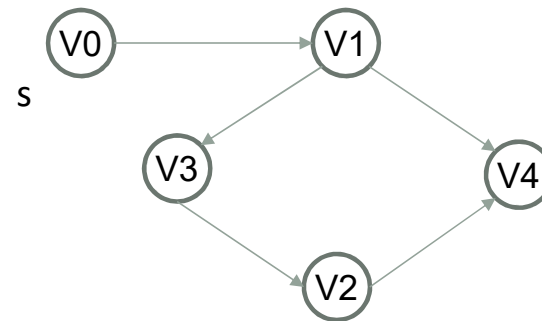
Unweighted Shortest Path: Running Time

The basic idea is a breadth-first search of the graph, starting at source vertex s

- Initially, give all vertices in the graph a distance of INFINITY
- Start at s ; give s distance = 0
- Enqueue s into a queue
- While the queue is not empty:
 - Dequeue the vertex v from the head of the queue
 - For each of v 's adjacent nodes that has not yet been visited:
 - Mark its distance as 1 + the distance to v
 - Enqueue it in the queue

What is the tightest worst-case time complexity (in terms of $|V|$ and $|E|$) of this algorithm?

- A. $O(|V|)$ B. $O(|E|)$ C. $O(|V| + |E|)$
D. $O(|V|^2)$ E. Other



Representing the graph with structs

```
#include <iostream>
#include <limits>
#include <vector>
#include <queue>

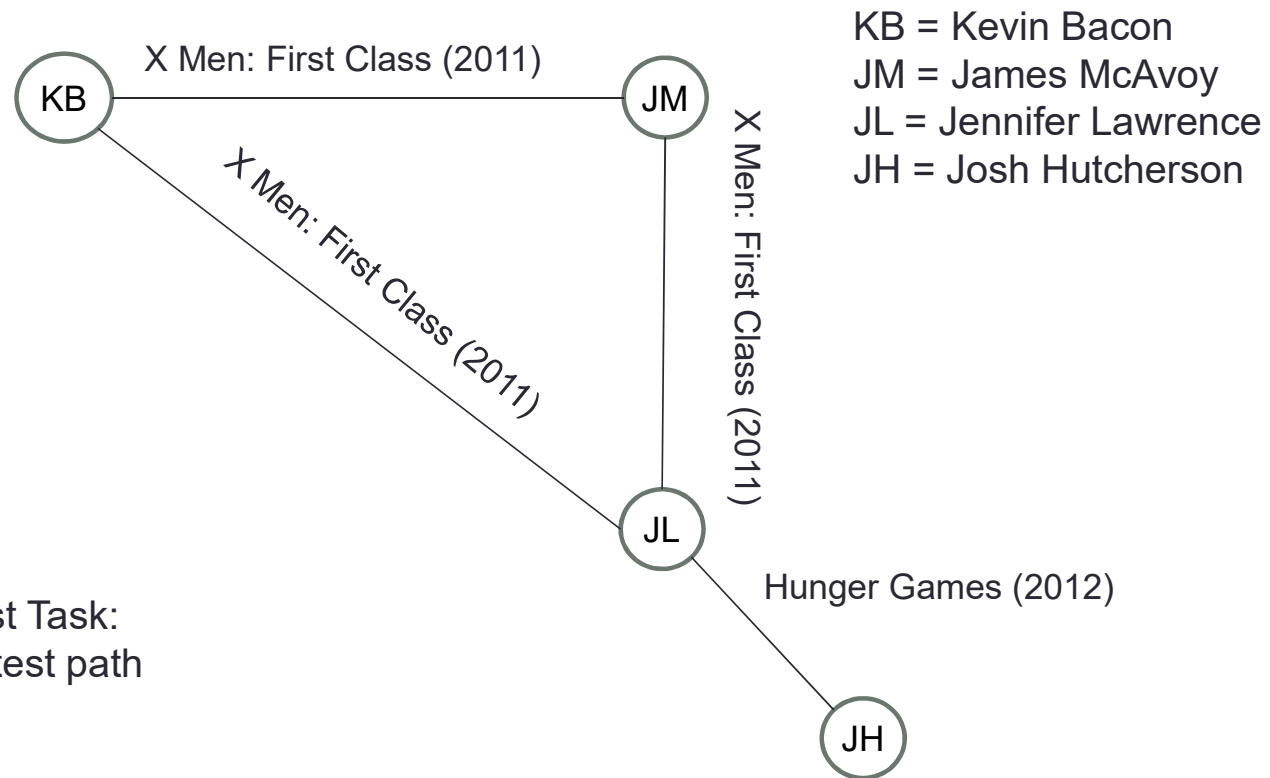
using namespace std;

struct Vertex
{
    vector<int> adj; // The adjacency list
    int dist;       // The distance from the source
    int index;      // The index of this vertex
    int prev;       // The index of the vertex previous in the path
};

vector<Vertex*> createGraph()
{ ... }
```

Your representation for PA3 will have some similarities and probably some differences.

Movie graphs: Representation hints



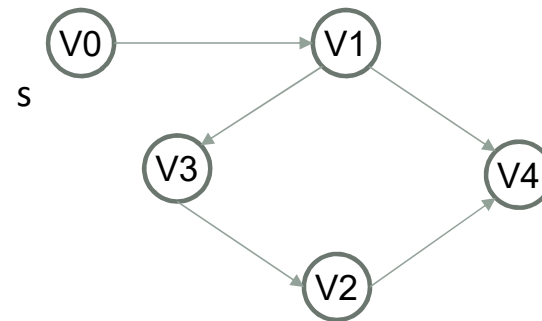
- PA3's First Task:
 - Shortest path

What is this algorithm??

The basic idea is a breadth-first search of the graph, starting at source vertex s

- Initially, give all vertices in the graph a distance of INFINITY
- Start at s ; give s distance = 0
- ~~Enqueue~~ Push s into a ~~queue~~ stack
- While the ~~queue~~ stack is not empty:
 - ~~Dequeue~~ pop the vertex v from the head of the ~~queue~~ top of the stack
 - For each of v 's adjacent nodes that has not yet been visited:
 - Mark its distance as 1 + the distance to v
 - ~~Enqueue~~ Push it on the ~~queue~~ stack

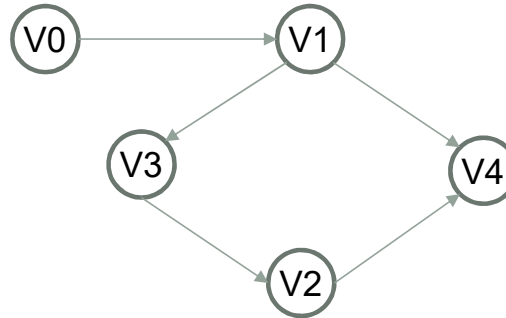
Stack:



A. BFS B. DFS shortest path C. DFS not shortest path D. Dijkstra's algorithm

Breadth First Search

- Explore all the nodes reachable from a given node before moving on to the next node to explore

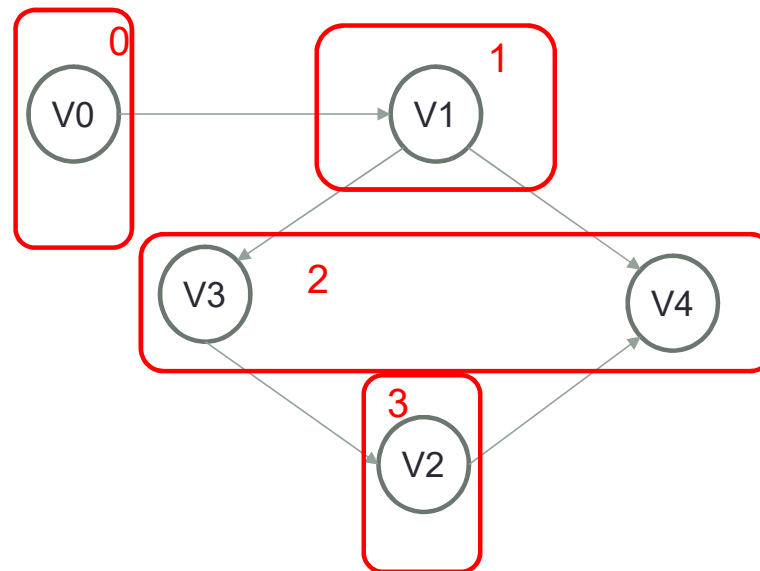


Does BFS always find the shortest path from the source to any node?

- A. Yes for unweighted graphs
- B. Yes for all graphs
- C. No

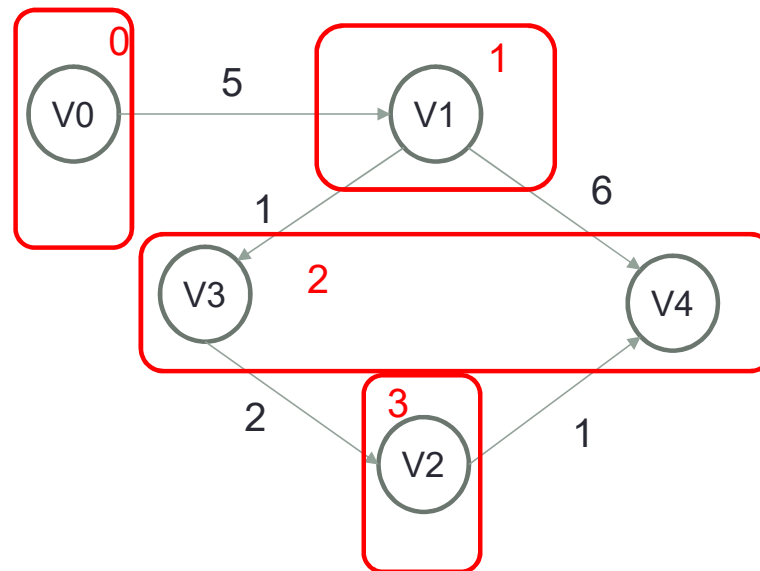
Why Did BFS Work for Shortest Path?

- Vertices are explored in order of their distance away from the source. So we are guaranteed that the first time we see a vertex we have found the shortest path to it.
- The queue (FIFO) assures that we will explore from all nodes in one level before moving on to the next.



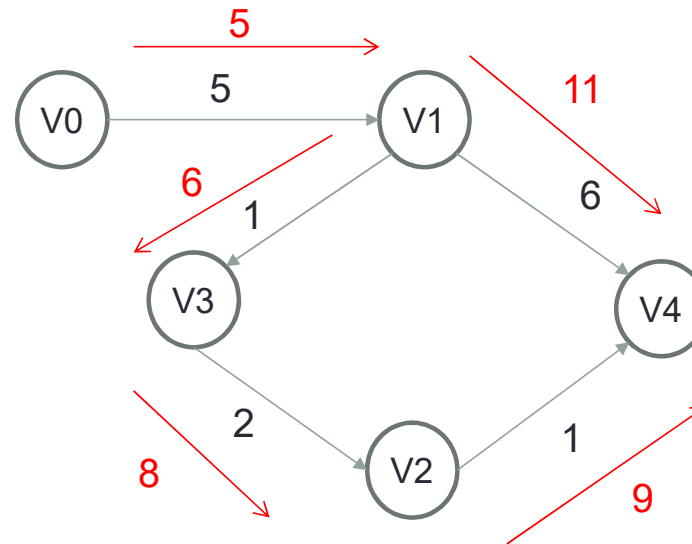
BFS on weighted graphs?

- Run BFS on this weighted graph. What are the weights of the paths that you find to each vertex from v0? Are these the shortest paths?



BFS on weighted graphs?

- In a weighted graph, the number of edges no longer corresponds to the length of the path. We need to decouple path length from edges, and explore paths in increasing *path length* (rather than increasing number of edges).
- In addition, the first time we encounter a vertex may, we may not have found the shortest path to it, so we need to delay committing to that path.



Dijkstra's Algorithm: Data Structures

- Maintain a sequence (e.g. an array) of vertex objects, indexed by vertex number
 - Vertex objects contain these 3 fields (and others):
 - “dist”: the cost of the best (least-cost) path discovered so far from the start vertex to this vertex
 - “prev”: the vertex number (index) of the previous node on that best path
 - “done”: a boolean indicating whether the “dist” and “prev” fields contain the final best values for this vertex, or not
- Maintain a priority queue
 - The priority queue will contain (*pointer-to-vertex*, *path cost*) pairs
 - *Path cost* is priority, in the sense that low cost means high priority
 - Note: multiple pairs with the same “*pointer-to-vertex*” part can exist in the priority queue at the same time. These will usually differ in the “*path cost*” part

Dijkstra's Algorithm

Nodes have:
prev
dist
done

Dijkstra(S):

Initialize: Priority queue (PQ), dist fields to infinity,
prev fields to -1, done fields to false

Set S's dist to 0

Enqueue {S, 0} onto the PQ

while PQ is not empty:

dequeue node v from front of queue

if (v is not done)

set v.done to true

for each of v's neighbors, w:

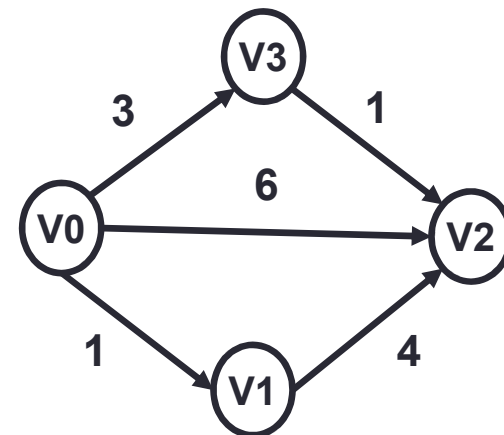
//distance to w through v is:

$c = v.\text{dist} + \text{edgeWeight}(v, w)$

if c is less than w.dist:

set w.prev = v and w.dist = c

enqueue {w, c} into the PQ



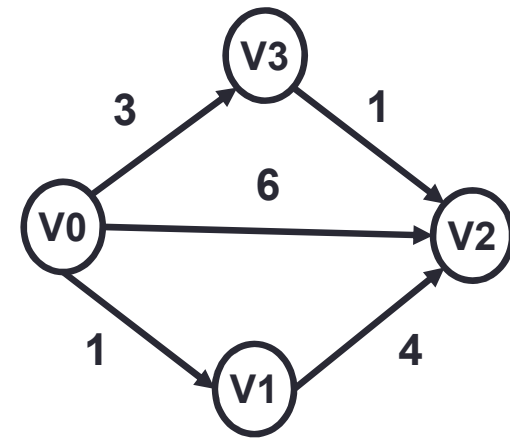
The array of vertices, which include dist, prev, and done fields (initialize dist to 'INF' and done to 'f'):

```
V0: dist=          prev=          done=
```

```
V1: dist=          prev=          done=
```

```
V2: dist=          prev=          done=
```

```
V3: dist=          prev=          done=
```

[illegible]

Dijkstra's Algorithm: Questions

Dijkstra(S):

Initialize: Priority queue (PQ), dist fields to infinity, prev fields to -1, done fields to false

Enqueue {S, 0} onto the PQ

while PQ is not empty:

dequeue node v from front of queue

if (v is not done)

set v.done to true

for each of v's neighbors, w:

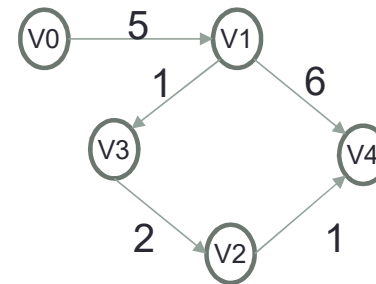
distance to w through v, $c = v.\text{dist} + \text{edgeWeight}(v, w)$

if c is less than w.dist:

set w.prev = v and w.dist = c

enqueue {w, c} into the PQ

When a node comes out of the priority queue, how do you know you've found the shortest path to the node?



Dijkstra's Algorithm: Running time

Dijkstra(S):

Initialize: Priority queue (PQ), dist fields to infinity, prev fields to -1, done fields to false

Enqueue {S, 0} onto the PQ

while PQ is not empty:

 dequeue node v from front of queue

 if (v is not done)

 set v.done to true

 for each of v's neighbors, w:

 distance to w through v, $c = v.\text{dist} + \text{edgeWeight}(v, w)$

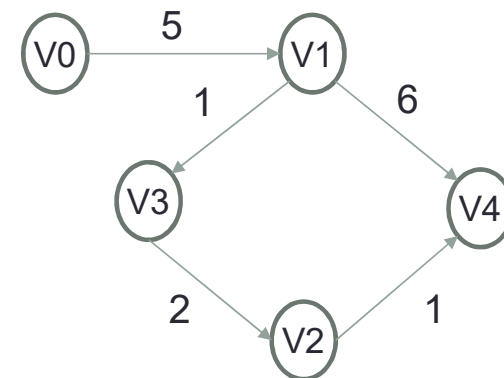
 if c is less than w.dist:

 set w.prev = v and w.dist = c

 enqueue {w, c} into the PQ

How long does the step in red take?

- A. $O(1)$
- B. $O(|V|)$
- C. $O(|E|)$
- D. $O(|V|+|E|)$
- E. Other



Dijkstra's Algorithm: Running time

Dijkstra(S):

Initialize: Priority queue (PQ), dist fields to infinity, prev fields to -1, done fields to false

Enqueue {S, 0} onto the PQ

while PQ is not empty:

 dequeue node v from front of queue

 if (v is not done)

 set v.done to true

 for each of v's neighbors, w:

 distance to w through v, $c = v.\text{dist} + \text{edgeWeight}(v, w)$

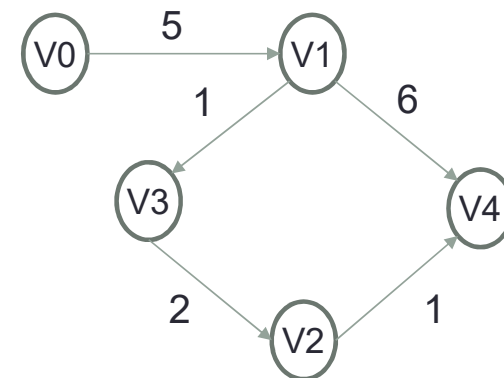
 if c is less than w.dist:

 set w.prev = v and w.dist = c

 enqueue {w, c} into the PQ

How long does the step in red take?

- A. $O(1)$
- B. $O(|V|)$
- C. $O(|E|)$
- D. $O(|V|+|E|)$
- E. Other



Walkthrough

Dijkstra(S, G):

Initialize: Priority queue (PQ), dist fields to infinity,
prev fields to -1, done fields to false

$O(|V|)$

Enqueue {S, 0} onto the PQ

$O(|1|)$

while PQ is not empty:

 dequeue node v from front of queue

 if (v is not done)

 set v.done to true

 for each of v's neighbors, w:

 distance to w through v, $c = v.dist + \text{edgeWeight}(v, w)$

 if c is less than w.dist:

 set w.prev = v and w.dist = c

 enqueue {w, c} into the PQ

Walkthrough

Dijkstra(S, G):

Initialize: Priority queue (PQ), dist fields to infinity,
prev fields to -1, done fields to false

Enqueue {S, 0} onto the PQ

while PQ is not empty:

 dequeue node v from front of queue

 if (v is not done)

 set v.done to true

 for each of v's neighbors, w:

 distance to w through v, $c = v.\text{dist} + \text{edgeWeight}(v, w)$

 if c is less than w.dist:

 set w.prev = v and w.dist = c

 enqueue {w, c} into the PQ

$O(|V|)$

~~$O(|V|)$~~

~~$O(|V|+1)$~~

Walkthrough

Dijkstra(S, G):

Initialize: Priority queue (PQ), dist fields to infinity,
prev fields to -1, done fields to false

Enqueue {S, 0} onto the PQ

while PQ is not empty:

dequeue node v from front of queue

if (v is not done)

set v.done to true

for each of v's neighbors, w:

distance to w through v, $c = v.\text{dist} + \text{edgeWeight}(v, w)$

if c is less than w.dist:

set w.prev = v and w.dist = c

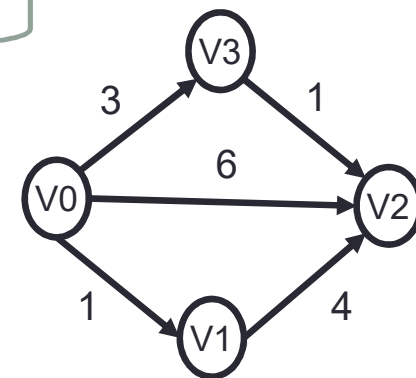
enqueue {w, c} into the PQ

$O(|V|)$

$O(?)$

Pairs of (node, cost) go into the priority queue. Can a node go into the priority queue more than once?

- A. Yes
- B. No



Walkthrough

Dijkstra(S, G):

Initialize: Priority queue (PQ), dist fields to infinity,
prev fields to -1, done fields to false

$O(|V|)$

Enqueue {S, 0} onto the PQ

while PQ is not empty:

dequeue node v from front of queue

if (v is not done)

set v.done to true

for each of v's neighbors, w:

distance to w through v, $c = v.dist + \text{edgeWeight}(v, w)$

if c is less than w.dist:

set w.prev = v and w.dist = c

enqueue {w, c} into the PQ

$O(?)$

The total number of pairs that go into the priority queue is approximately which of the following (in the worst case):

A. $|V|$ (the number of nodes in the graph)

B. $|E|$ (the number of edges in the graph)

C. $|V| + |E|$

D. $|V| * |E|$

Walkthrough

Dijkstra(S, G):

Initialize: Priority queue (PQ), dist fields to infinity,
prev fields to -1, done fields to false

$O(|V|)$

Enqueue {S, 0} onto the PQ

while PQ is not empty:

dequeue node v from front of queue

if (v is not done)

set v.done to true

for each of v's neighbors, w:

distance to w through v, $c = v.dist + edgeWeight(v, w)$

if c is less than w.dist:

set w.prev = v and w.dist = c

enqueue {w, c} into the PQ

$O(?)$

So the while loop is making $O(|E|)$ insertions into a priority queue with size at most $O(|E|)$.

What is the total running time for the while loop?

A. $O(|E|)$

B. $O(|E| \log |E|)$

C. $O(|E| * |E|)$

D. Other

Walkthrough

Dijkstra(S, G):

Initialize: Priority queue (PQ), dist fields to infinity,
prev fields to -1, done fields to false

$$O(|V|)$$

Enqueue {S, 0} onto the PQ

while PQ is not empty:

← whole loop

$$O(|E| \log |E|)$$

 dequeue node v from front of queue

 if (v is not done)

 set v.done to true

 for each of v's neighbors, w:

 distance to w through v, $c = v.\text{dist} + \text{edgeWeight}(v, w)$

 if c is less than w.dist:

 set w.prev = v and w.dist = c

 enqueue {w, c} into the PQ

Walkthrough

Dijkstra(S, G):

Initialize: Priority queue (PQ), dist fields to infinity,
prev fields to -1, done fields to false

$$O(|V|)$$

Enqueue {S, 0} onto the PQ

while PQ is not empty:

← whole loop

$$O(|E| \log |E|)$$

 dequeue node v from front of queue

 if (v is not done)

 set v.done to true

 for each of v's neighbors, w:

 distance to w through v, $c = v.\text{dist} + \text{edgeWeight}(v, w)$

 if c is less than w.dist:

 set w.prev = v and w.dist = c

 enqueue {w, c} into the PQ

Overall:

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

Walkthrough

Dijkstra(S, G):

Initialize: Priority queue (PQ), dist fields to infinity,
prev fields to -1, done fields to false

$$O(|V|)$$

Enqueue {S, 0} onto the PQ

while PQ is not empty:

← whole loop

$$O(|E| \log |E|)$$

 dequeue node v from front of queue

 if (v is not done)

 set $v.done$ to true

 for each of v 's neighbors, w :

 distance to w through v , $c = v.dist + edgeWeight(v, w)$

 if c is less than $w.dist$:

 set $w.prev = v$ and $w.dist = c$

 enqueue $\{w, c\}$ into the PQ

**Because $|E| \leq |V|^2$ and
 $\log(|V|^2)$ is just $O(\log(|V|))$ we
could tighten to:
 $O(|E| \log |V| + |V|)$**

**Overall:
 $O(|E| \log |E| + |V|)$**

Unweighted Shortest Path: Running Time

BFS(S):

Initialize queue, set dist to INFINITY and prev to null for all nodes

Add S to queue and set S.dist to 0

while queue is not empty:

 dequeue node curr from head of queue

 set n.visited = true

 for each of curr's neighbors, n:

 if n.dist > curr.dist+1:

 set n.dist to curr.dist+1

 set n's prev to curr

 enqueue n to the queue

// When we get here then we're done exploring from S

What is the time complexity (in terms of $|V|$ and $|E|$) of this algorithm?

maps.google.com

Seattle

San Diego

**Driving
directions from
San Diego to
Seattle?**



maps.google.com

Seattle

San Diego

**Driving
directions from
San Diego to
Seattle?**



Driving directions from San Diego to Seattle?

1255
miles

San Diego

Seattle



maps.google.com

Seattle

**Dijkstra will find
the shortest
route. But how?**

**1255
miles**

San Diego



maps.google.com

Seattle

**Dijkstra will find
the shortest
route. But how?**

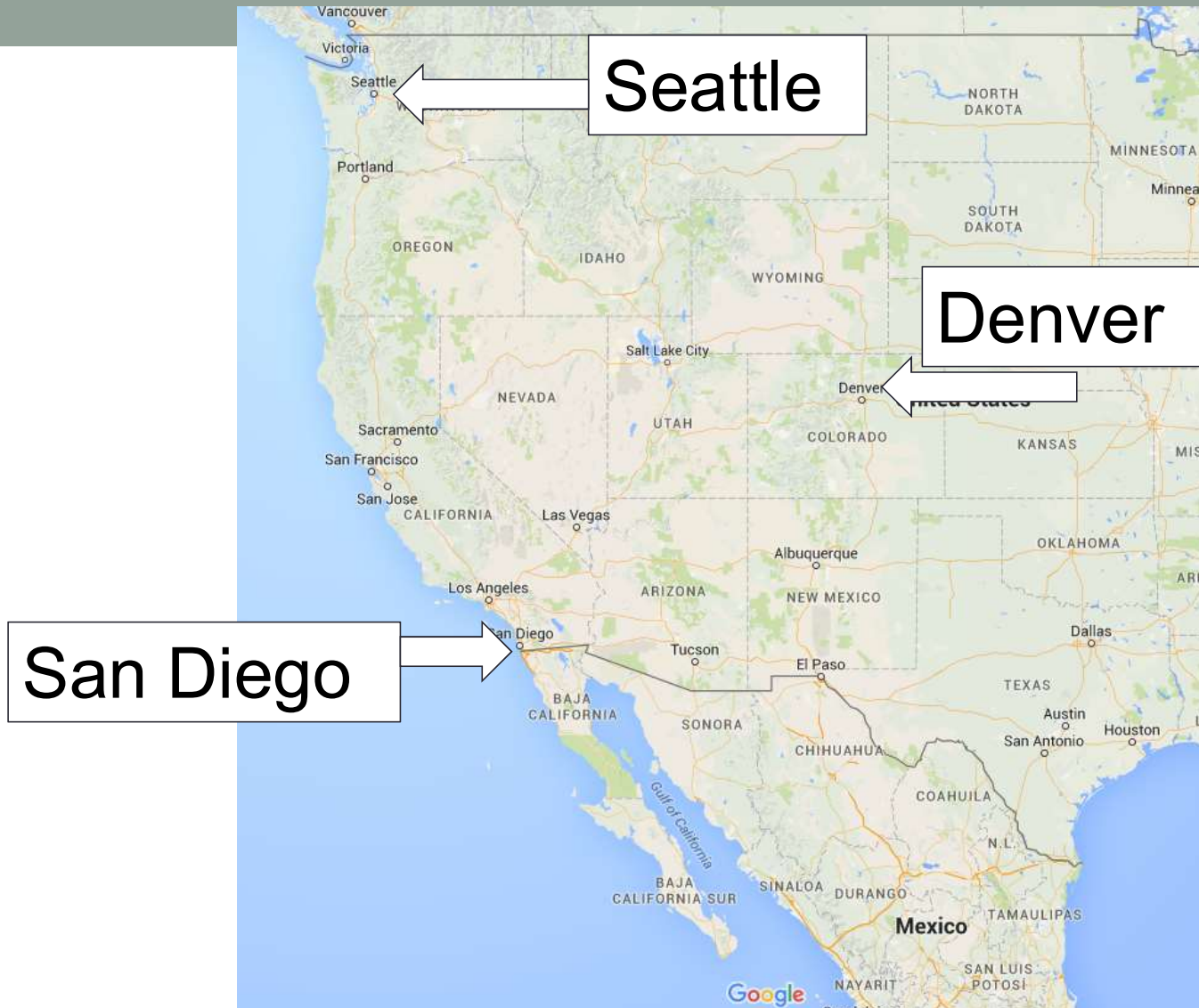
Denver

Dallas

San Diego

Mazatlán,
Mexico





**Would Dijkstra
have you
consider Denver
in finding the
path to Seattle?**

- A. Yes**
- B. No**
- C. Maybe**

Seattle

Denver

San Diego

**Why would
YOU have never
considered
Denver?**



Seattle

**Going East is
the wrong
direction!**

Denver

San Diego



Seattle

**We should
consider
distance from
target too!**

Denver

**Dijkstra only
considers
distance from
source**

San Diego



Dijkstra's Algorithm

- Priority Queue ordering is based on:

$g(n)$: the distance (cost) from start vertex to vertex n

A* Algorithm

- Priority Queue ordering is based on:

$g(n)$: the distance (cost) from start vertex to vertex n

AND

$h(n)$: the **heuristic estimated cost** from vertex n to goal vertex

A* Algorithm

- Priority Queue ordering is based on:

$g(n)$: the distance (cost) from start vertex to vertex n

AND

$h(n)$: the **heuristic estimated cost** from vertex n to goal vertex

$$f(n) = g(n) + h(n)$$

A* Algorithm

- Priority Queue ordering is based on:

$g(n)$: the distance (cost) from start vertex to vertex n

AND

$h(n)$: the **heuristic estimated cost** from vertex n to goal vertex

$$f(n) = g(n) + h(n)$$

Dijkstra can be seen as a special case where $h(n)=0$

A* Algorithm

- Priority Queue ordering is based on:

$g(n)$: the distance (cost) from start vertex to vertex n

AND

$h(n)$: the **heuristic estimated cost** from vertex n to goal vertex

$$f(n) = g(n) + h(n)$$

**Guaranteed to
find shortest
path IF estimate
is never an
overestimate**

maps.google.com

Seattle

1019
miles

**Underestimate:
use the exact
distance.**

1064
miles

Denver

San Diego

http://distancecalculator.globefeed.com/World_Distance_Calculator.asp

A* Algorithm

- Priority Queue ordering is based on:

$g(n)$ the distance (cost) from start vertex to vertex n

AND

$h(n)$ the **heuristic estimated cost** from vertex n to goal vertex

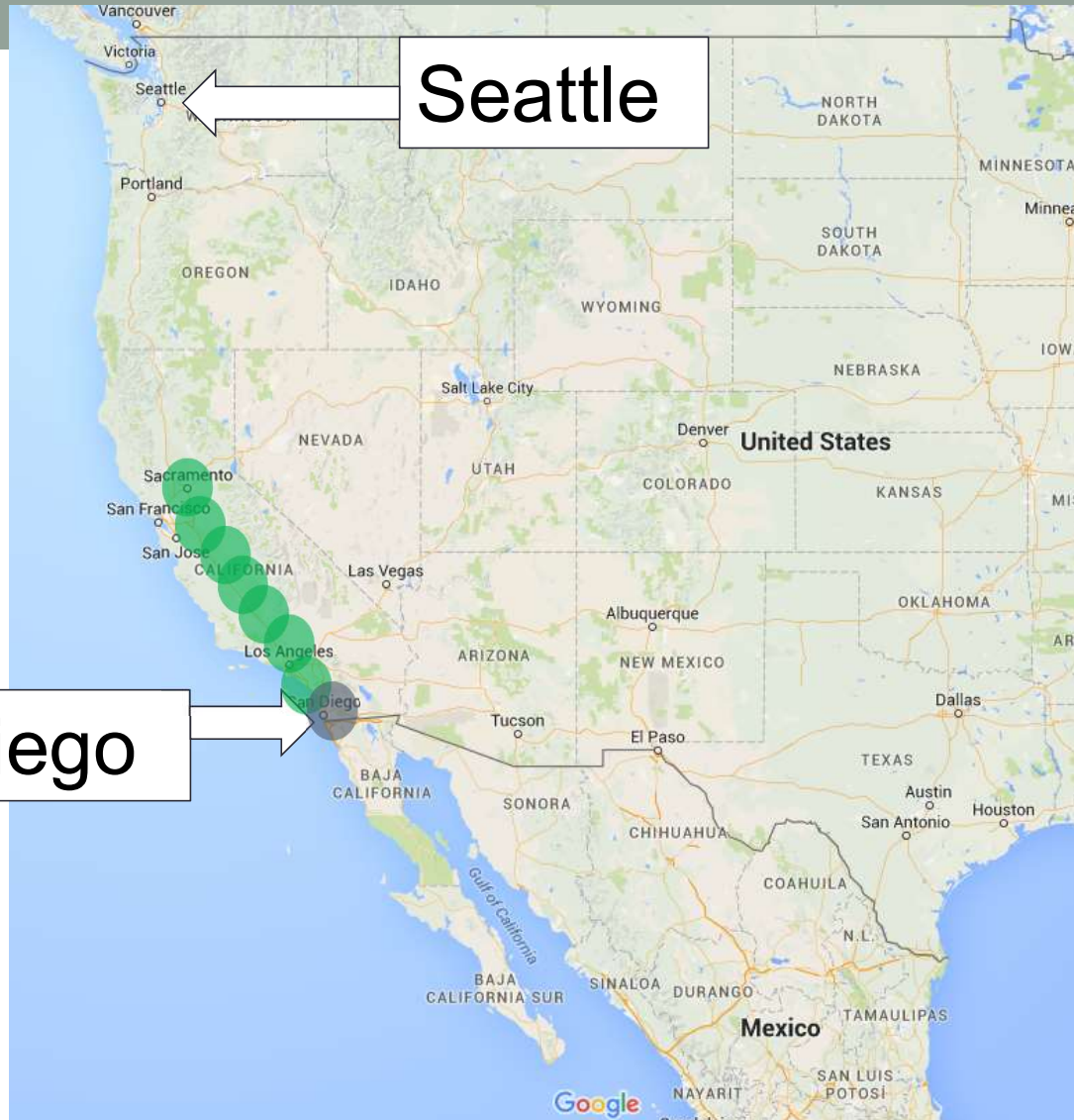
$$f(n) = g(n) + h(n)$$

maps.google.com

A*

Seattle

San Diego



maps.google.com

A*

Seattle

Sacramento

Las Vegas

San Diego



http://distancecalculator.globefeed.com/World_Distance_Calculator.asp

Seattle

A*

Sacramento

$$f(n) = 504 + 625 = 1129$$

Las Vegas

$$f(n) = 331 + 871 = 1202$$

San Diego

A* Algorithm

- Priority Queue ordering is based on:

$g(n)$ the distance (cost) from start vertex to vertex n

AND

$h(n)$ the heuristic estimated cost from vertex n to goal vertex

$$f(n) = g(n) + h(n)$$

**Just change the
priority function!**