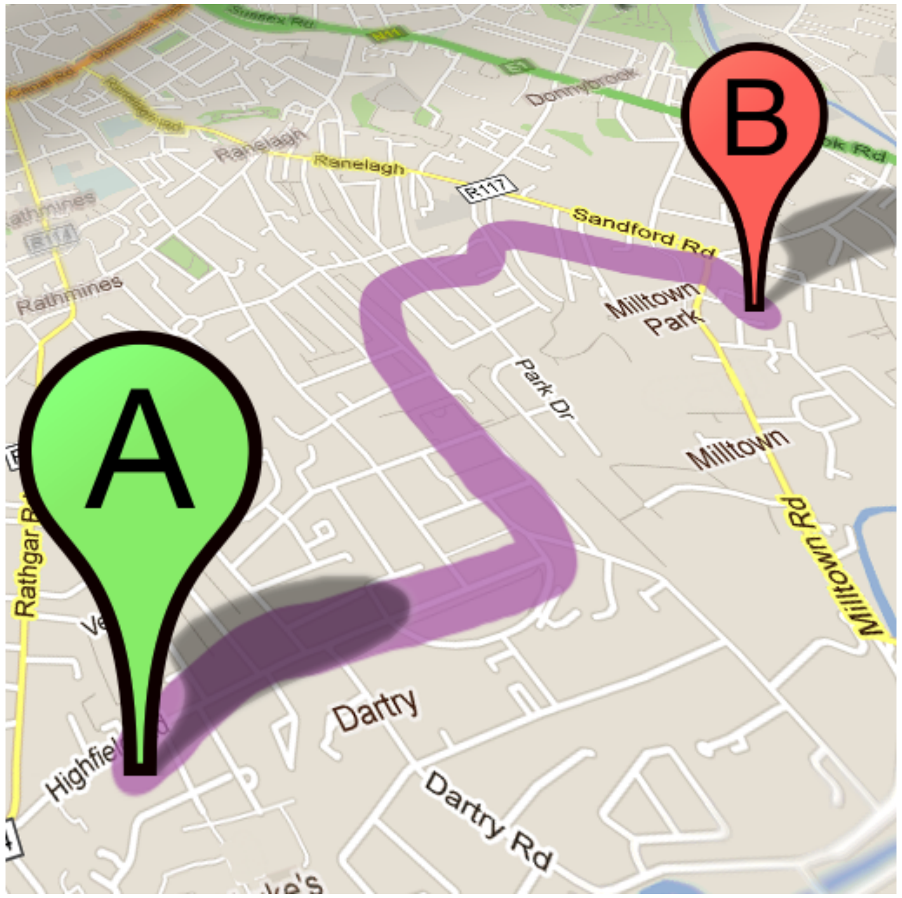
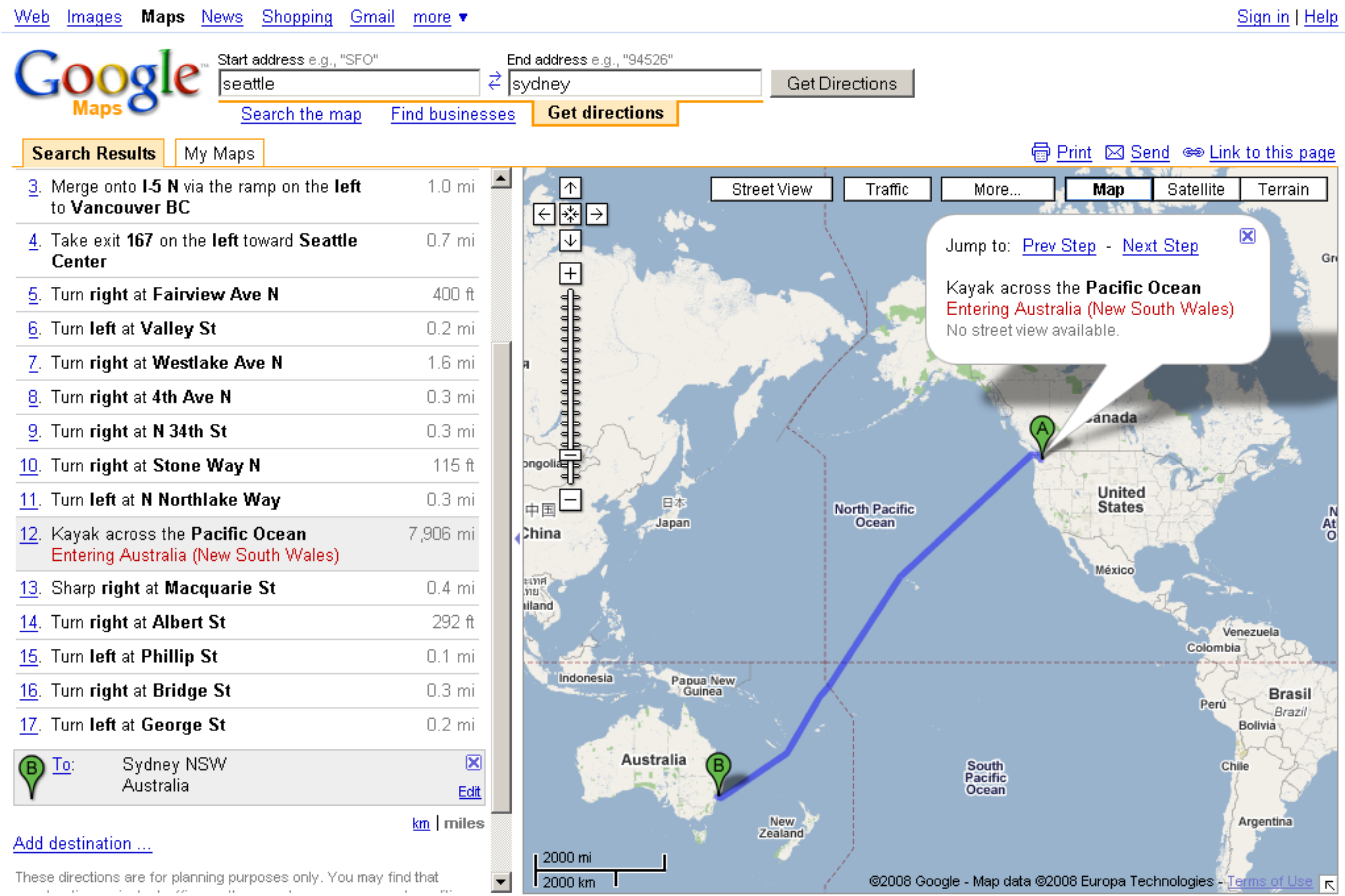
Honors Data Structures

Lecture 21: Shortest Paths.

4/11 & 4/13
2022

Daniel Bauer



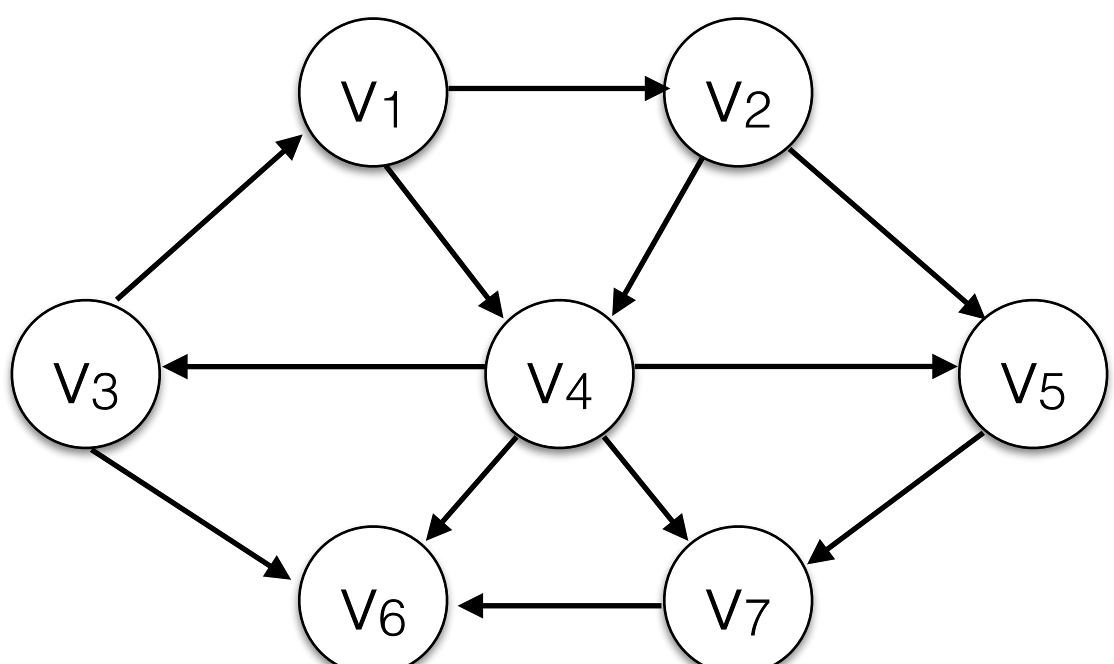

Web Images Maps News Shopping Gmail more ▾
Sign in | Help
Google Maps
Start address e.g., "SFO"
seattle
End address e.g., "94526"
sydney
Get Directions
Search the map
Find businesses
Get directions
Search Results
My Maps
Print
Send
Link to this page
3. Merge onto I-5 N via the ramp on the left to Vancouver BC 1.0 mi
4. Take exit 167 on the left toward Seattle Center 0.7 mi
5. Turn right at Fairview Ave N 400 ft
6. Turn left at Valley St 0.2 mi
7. Turn right at Westlake Ave N 1.6 mi
8. Turn right at 4th Ave N 0.3 mi
9. Turn right at N 34th St 0.3 mi
10. Turn right at Stone Way N 115 ft
11. Turn left at N Northlake Way 0.3 mi
12. Kayak across the Pacific Ocean 7,906 mi
Entering Australia (New South Wales)
13. Sharp right at Macquarie St 0.4 mi
14. Turn right at Albert St 292 ft
15. Turn left at Phillip St 0.1 mi
16. Turn right at Bridge St 0.3 mi
17. Turn left at George St 0.2 mi
To: Sydney NSW Australia
Edit
km | miles
Add destination ...
These directions are for planning purposes only. You may find that
Street View
Traffic
More...
Map
Satellite
Terrain
Jump to: Prev Step - Next Step
Kayak across the Pacific Ocean
Entering Australia (New South Wales)
No street view available.
Canada
United States
México
North Pacific Ocean
日本 Japan
中国 China
Indonesia
Papua New Guinea
Australia
New Zealand
South Pacific Ocean
Venezuela
Colombia
Perú
Brasil Brazil
Bolivia
Chile
Argentina
2000 mi
2000 km
©2008 Google - Map data ©2008 Europa Technologies - Terms of Use

Graph Traversals

- Different ways of exploring graphs:
  - Topological sort for Directed Acyclic Graphs.
  - Depth First Search (a generalization of pre-order traversal on trees to graphs, uses a Stack)
  - Breadth First Search (uses a Queue)
  - Dijkstra’s algorithm to find weighted shortest paths (uses a Priority Queue)

Finding Shortest Paths

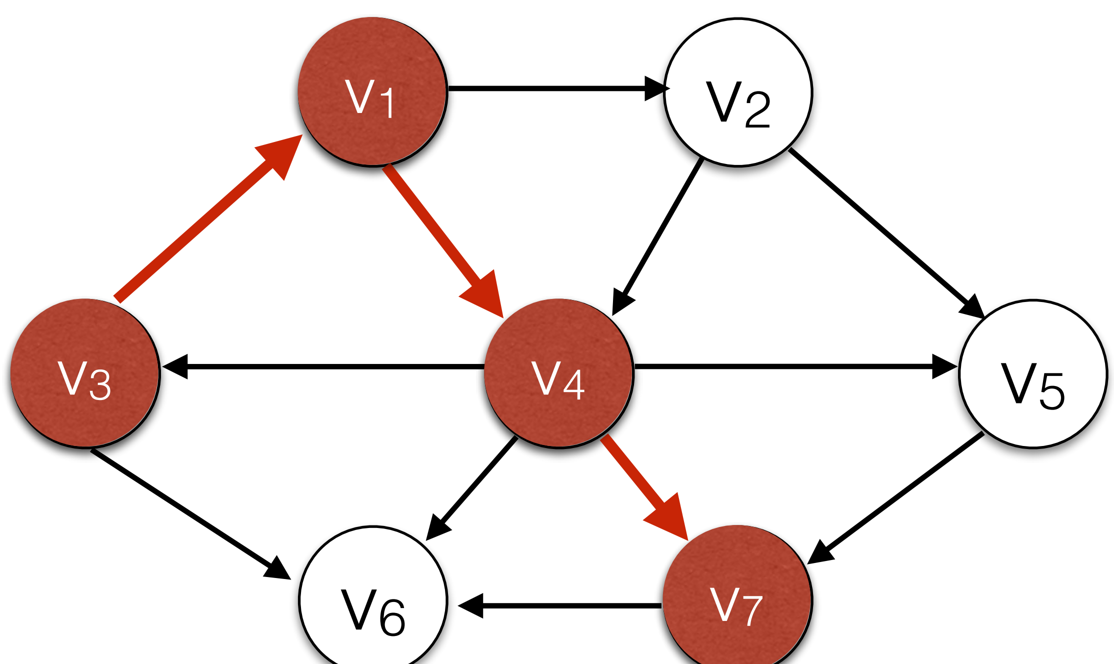
- Goal: Find the shortest path between two vertices s and t.



What is the shortest path between v_3 and v_7?

Finding Shortest Paths

- Goal: Find the shortest path between two vertices s and t.

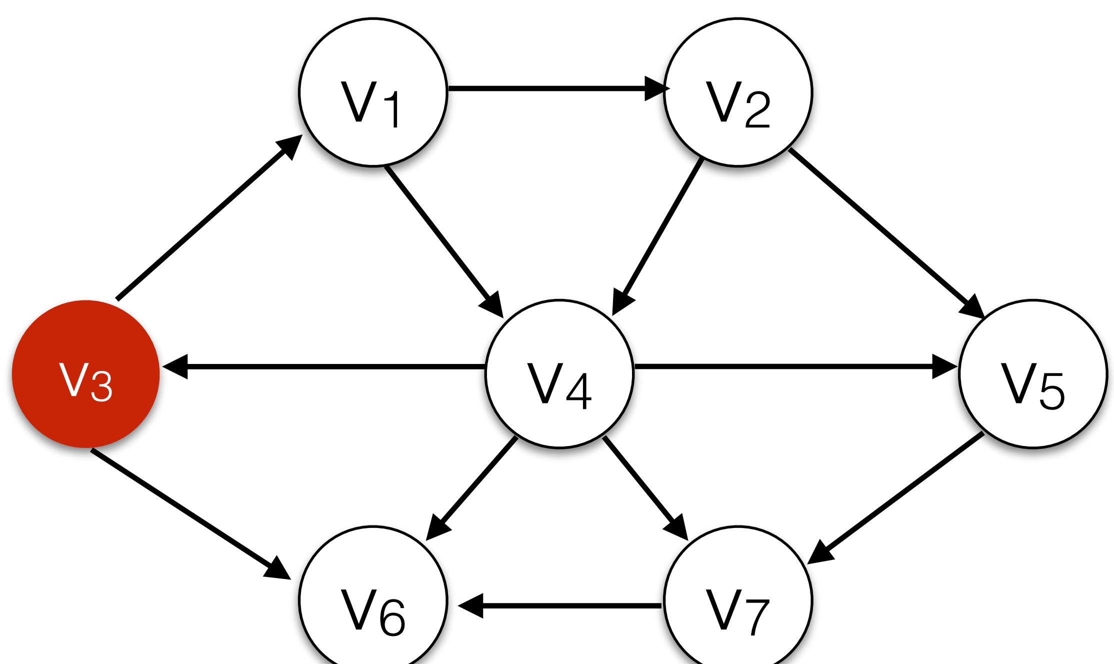


length 3

What is the shortest path between v_3 and v_7?

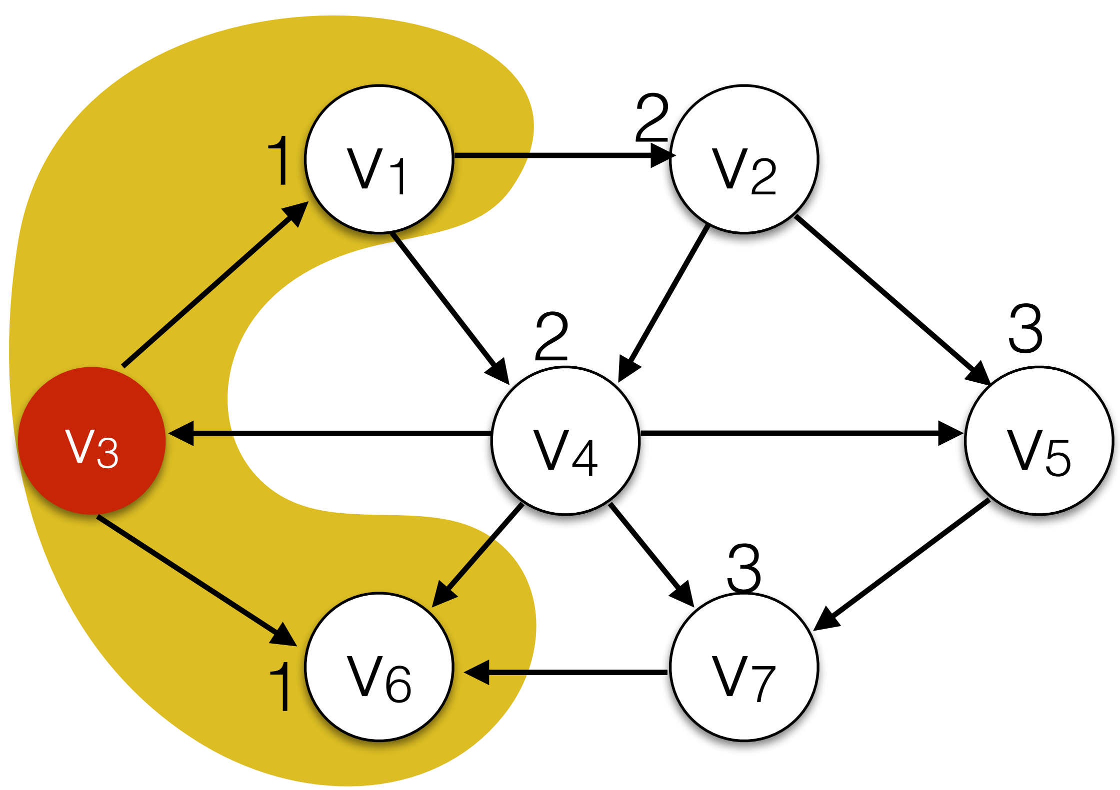
Finding Shortest Paths

- Goal: Find the shortest path between two vertices s and t.
- It turns out that finding the shortest path between s and ALL other vertices is just as easy. This problem is called **single-source shortest paths**.



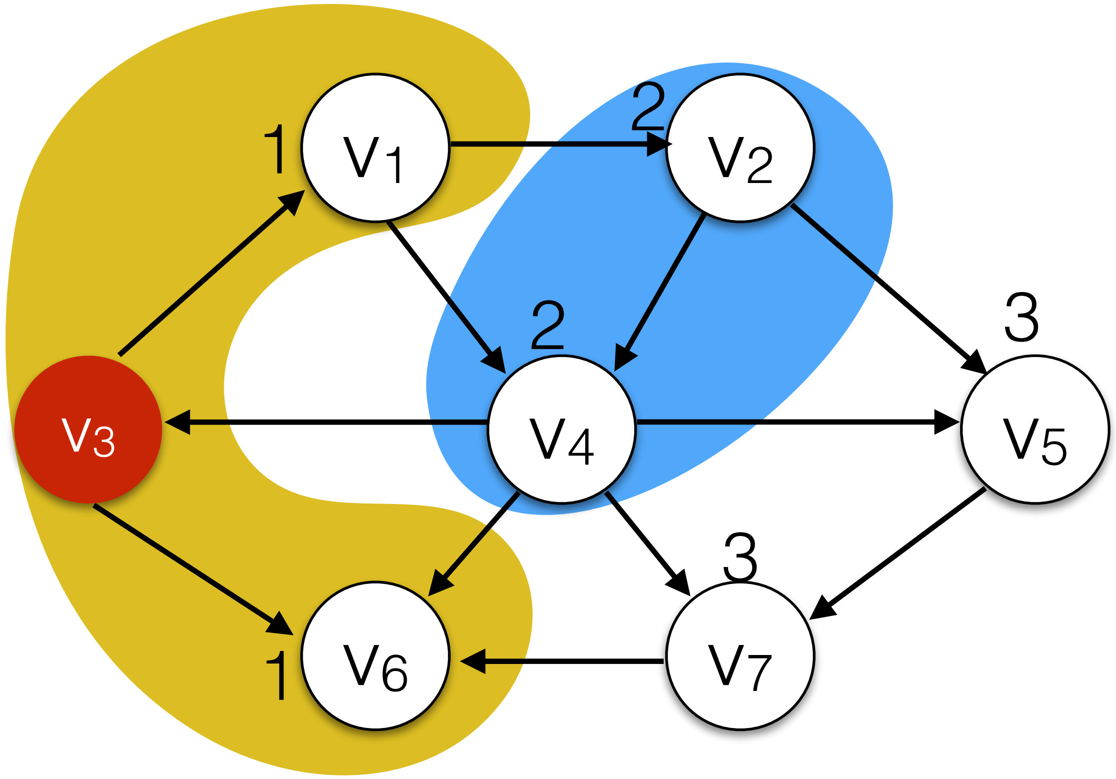
Finding Shortest Paths

- Goal: Find the shortest path between two vertices s and t.
- It turns out that finding the shortest path between s and ALL other vertices is just as easy. This problem is called **single-source shortest paths**.



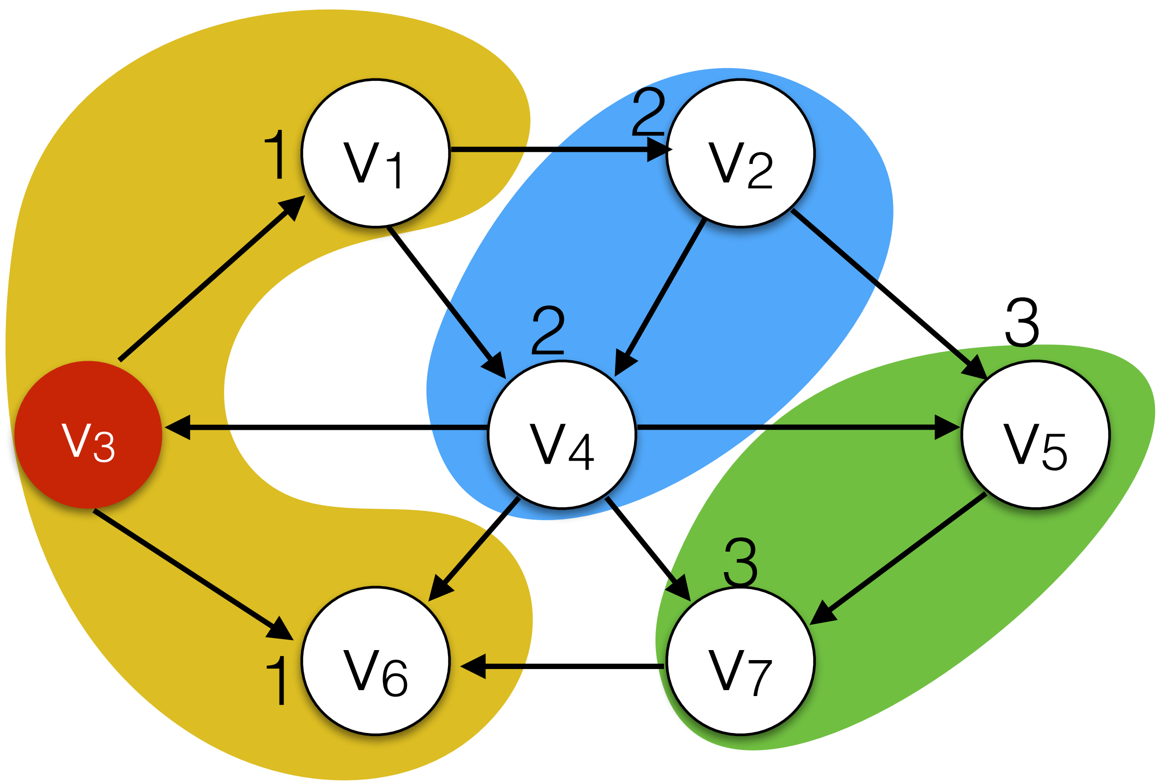
Finding Shortest Paths

- Goal: Find the shortest path between two vertices s and t.
- It turns out that finding the shortest path between s and ALL other vertices is just as easy. This problem is called **single-source shortest paths**.

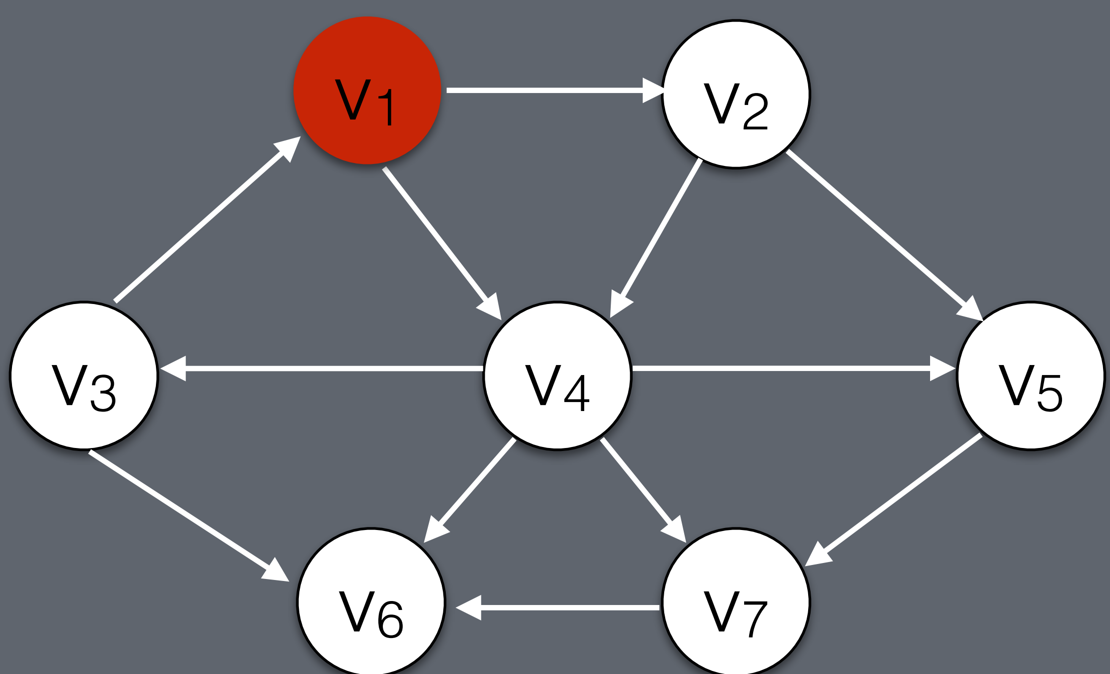


Finding Shortest Paths

- Goal: Find the shortest path between two vertices s and t.
- It turns out that finding the shortest path between s and ALL other vertices is just as easy. This problem is called **single-source shortest paths**.



Breadth First Search



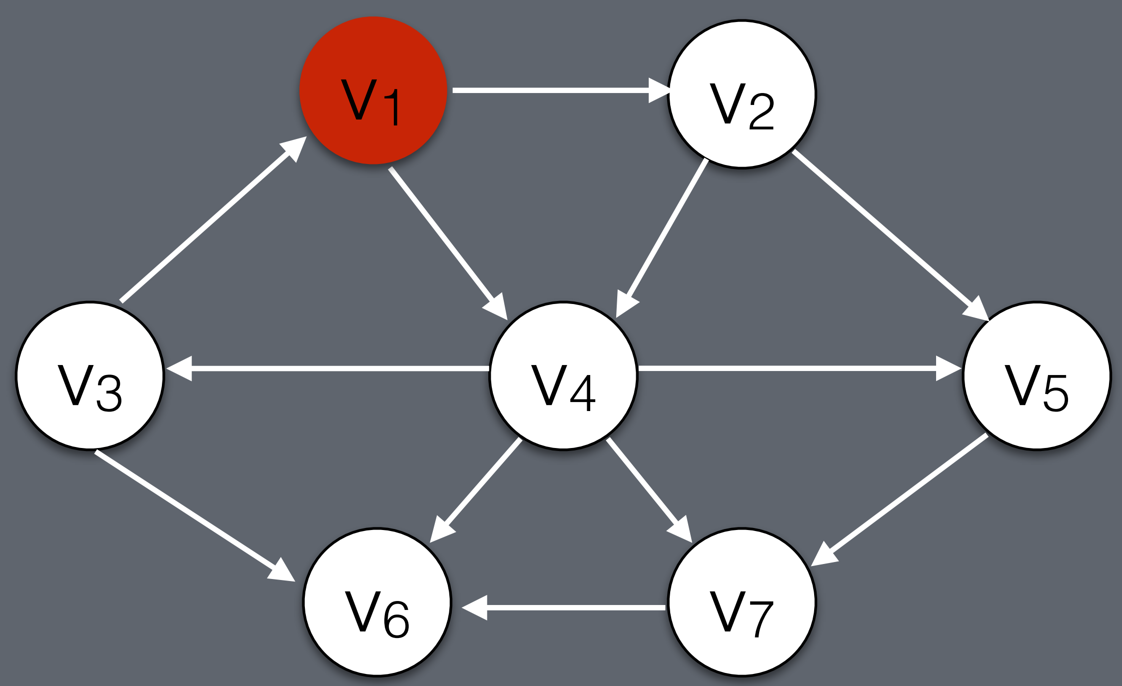
Queue **q**
q.enqueue(**start**)

while (**q** is not empty):
  u = **q**.dequeue()

  for each **v** adjacent to **u**:
    q.enqueue(**v**)

Queue: { V_1 }

Breadth First Search



Queue **q**
q.enqueue(**start**)

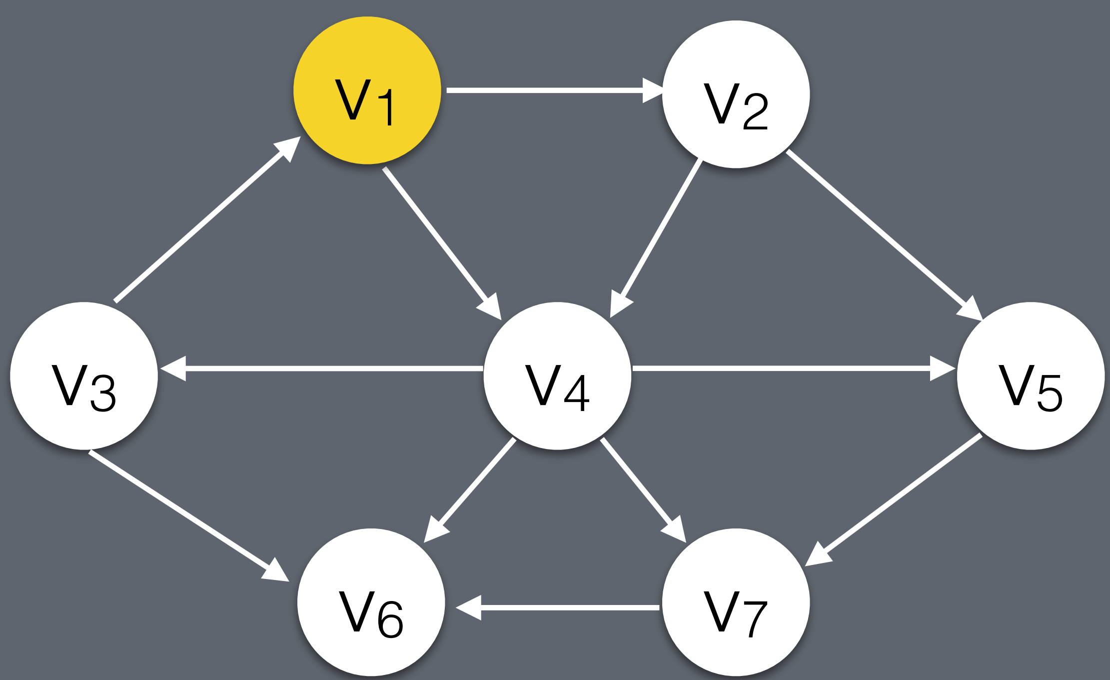
Set **discovered**

while (**q** is not empty):
  u = **q**.dequeue()

  for each **v** adjacent to **u**:
    if (v is not in **discovered**):
      q.enqueue(**v**)
      discovered.add(v)

Queue: { V_1 }

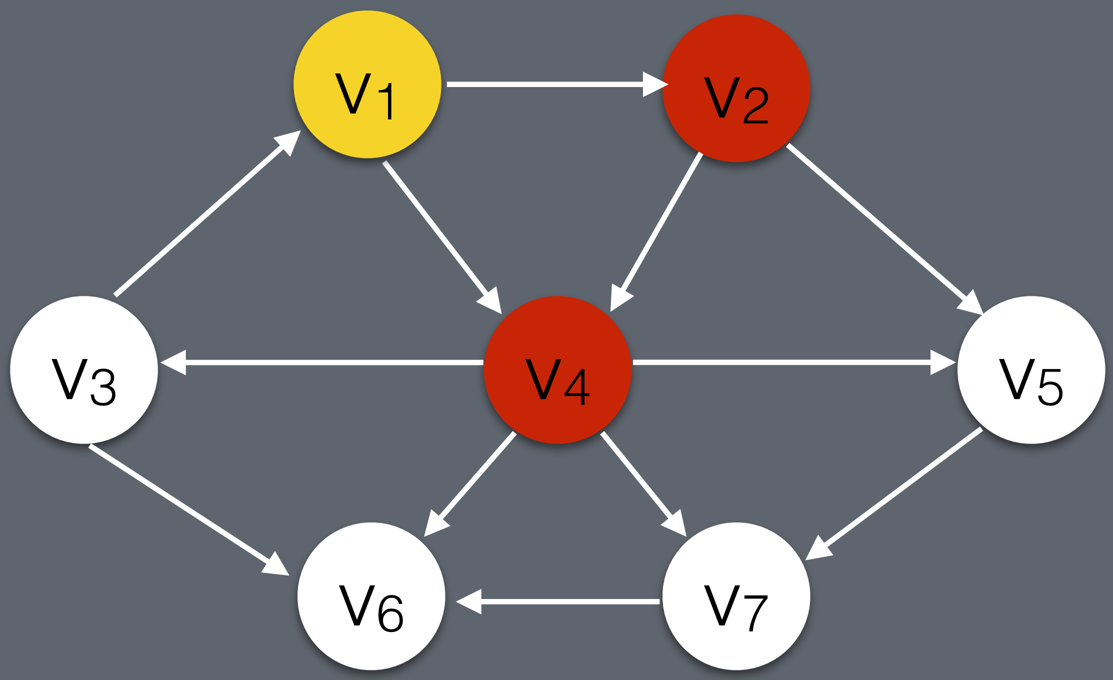
Breadth First Search




```
Queue q
q.enqueue(start)

Set visited

while (q is not empty):
  u = q.dequeue()

  for each v adjacent to u:
    if (v is not in visited):
      q.enqueue(v)
      visited.add(v)
```

Queue: { }

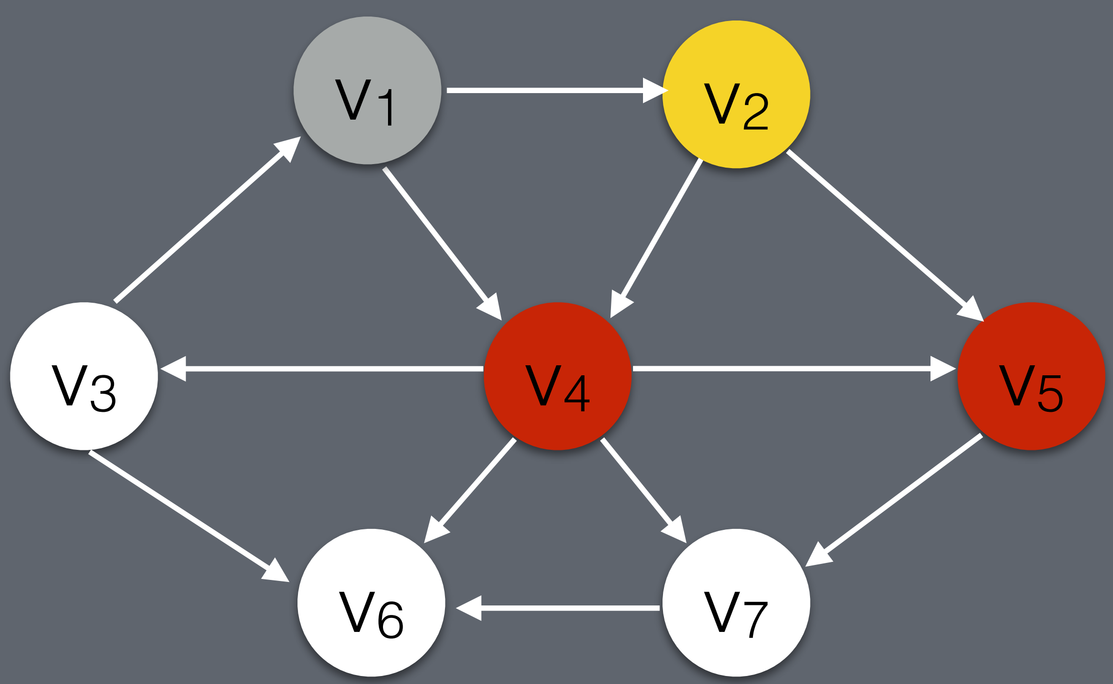
Breadth First Search




```
Queue q
q.enqueue(start)

Set visited

while (q is not empty):
  u = q.dequeue()

  for each v adjacent to u:
    if (v is not in visited):
      q.enqueue(v)
      visited.add(v)
```

Queue: { V_2, V_4}

Breadth First Search



Queue **q**
q.enqueue(**start**)

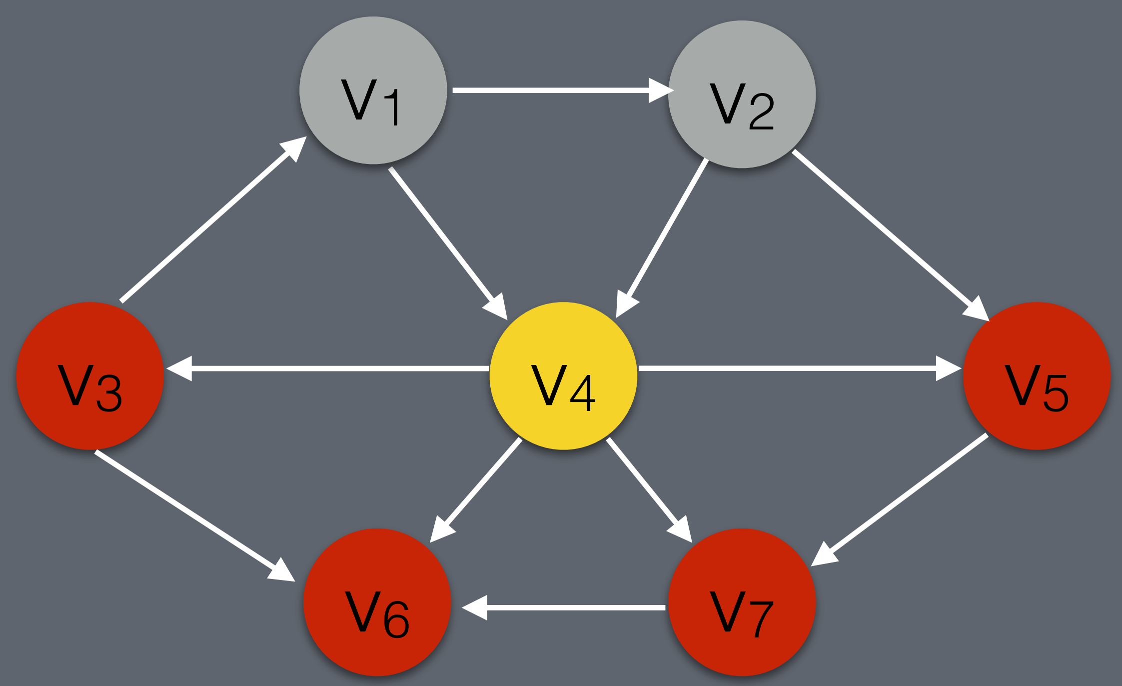
Set **visited**

while (**q** is not empty):
  u = **q**.dequeue()

  for each **v** adjacent to **u**:
    if (v is not in **visited**):
      q.enqueue(**v**)
      visited.add(v)

Queue: { V_4, V_5 }

Breadth First Search



Queue **q**
q.enqueue(**start**)

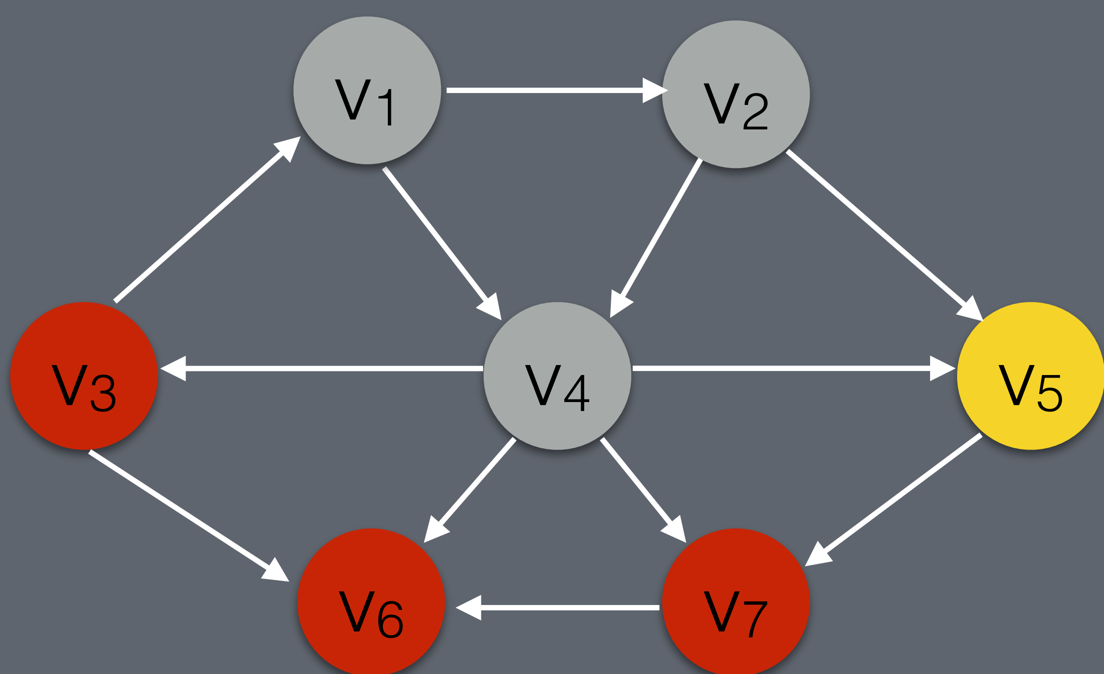
Set **visited**

while (**q** is not empty):
  u = **q**.dequeue()

  for each **v** adjacent to **u**:
    if (v is not in **visited**):
      q.enqueue(**v**)
      visited.add(v)

Queue: $\{V_5, V_3, V_6, V_7\}$

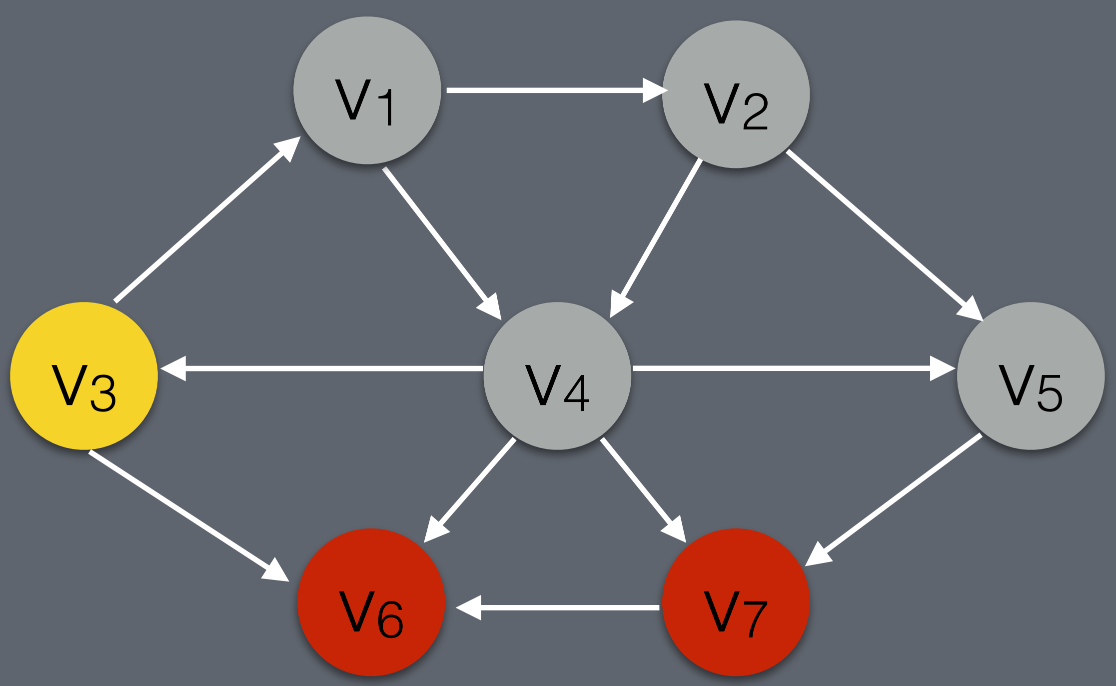
Breadth First Search




```
Queue q
q.enqueue(start)

Set visited

while (q is not empty):
  u = q.dequeue()

  for each v adjacent to u:
     if (v is not in visited):
       q.enqueue(v)
       visited.add(v)
```

Queue: $\{V_3, V_6, V_7\}$

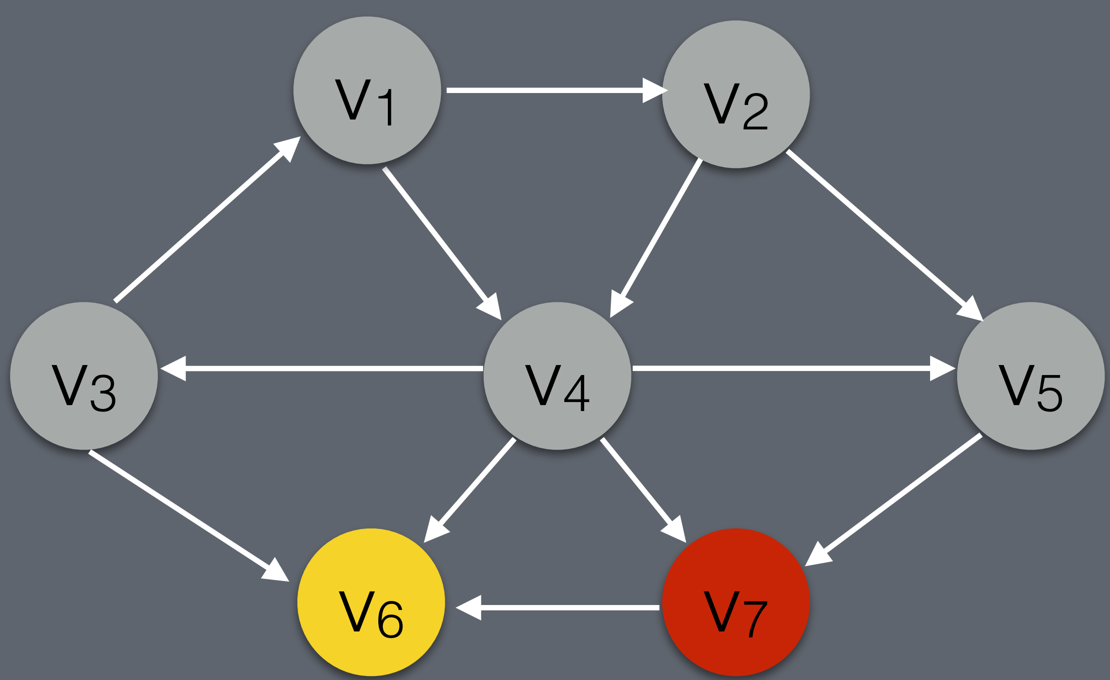
Breadth First Search




```
Queue q
q.enqueue(start)

Set visited

while (q is not empty):
  u = q.dequeue()

  for each v adjacent to u:
     if (v is not in visited):
       q.enqueue(v)
       visited.add(v)
```

Queue: $\{V_6, V_7\}$

Breadth First Search



Queue **q**
q.enqueue(**start**)

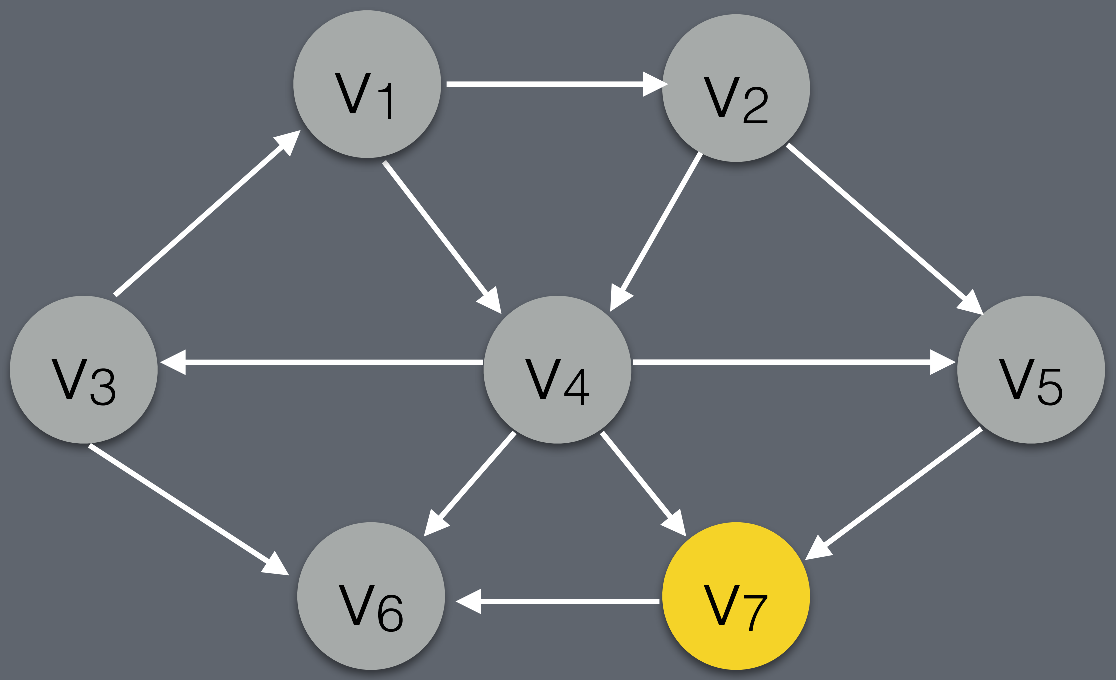
Set **visited**

while (**q** is not empty):
  u = **q**.dequeue()

  for each **v** adjacent to **u**:
    if (v is not in **visited**):
      q.enqueue(**v**)
      visited.add(v)

Queue: $\{V_7\}$

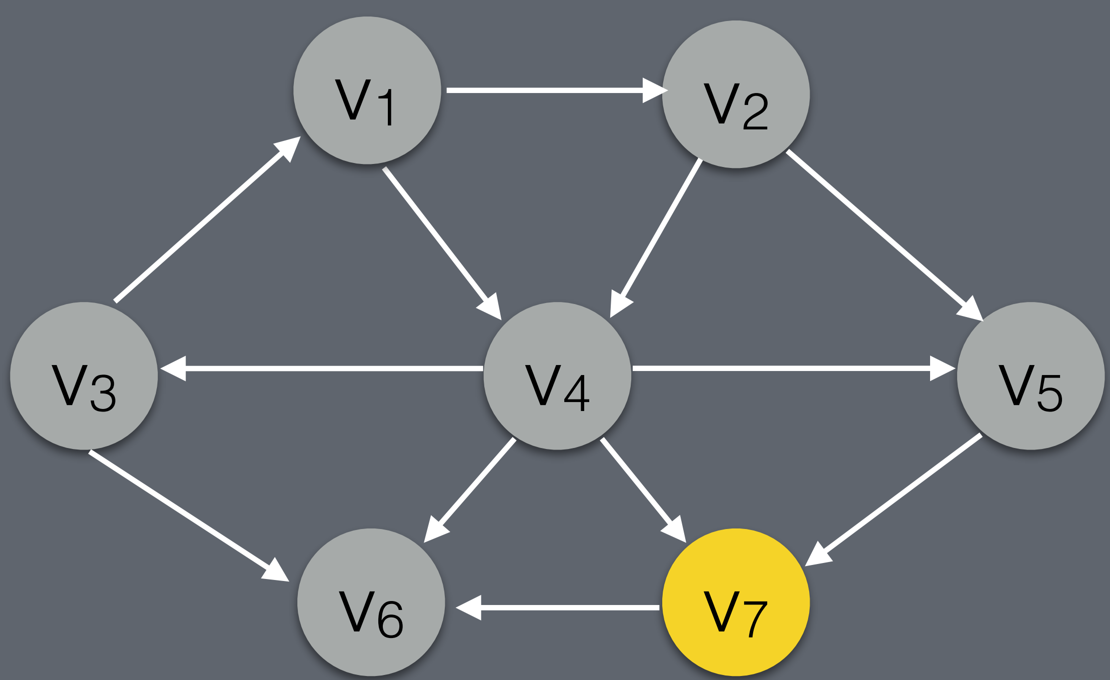
Breadth First Search




```
Queue q
q.enqueue(start)

Set visited

while (q is not empty):
  u = q.dequeue()

  for each v adjacent to u:
    if (v is not in visited):
      q.enqueue(v)
      visited.add(v)
```

Queue: {}

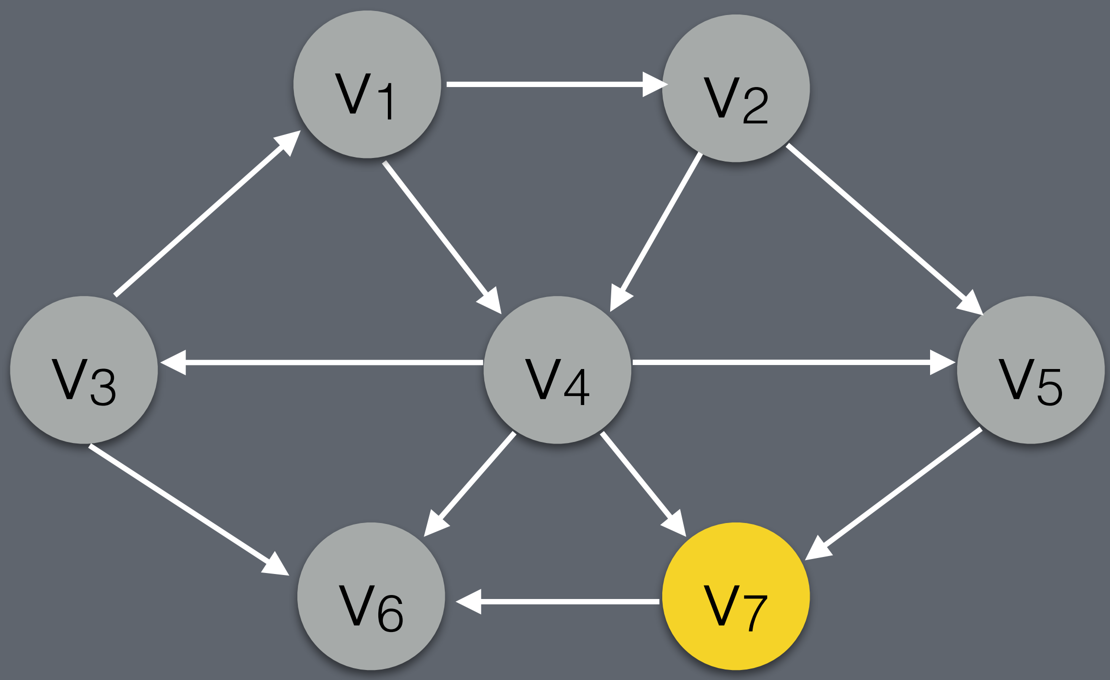
Breadth First Search




```
Queue q
q.enqueue(start)

Set visited

while (q is not empty):
  u = q.dequeue()

  for each v adjacent to u:
    if (v is not in visited):
      q.enqueue(v)
      visited.add(v)
```

Running time: $O(|V|+|E|)$

Breadth First Search



Running time: O(|V|+|E|)

Queue **q**
q.enqueue(**start**)

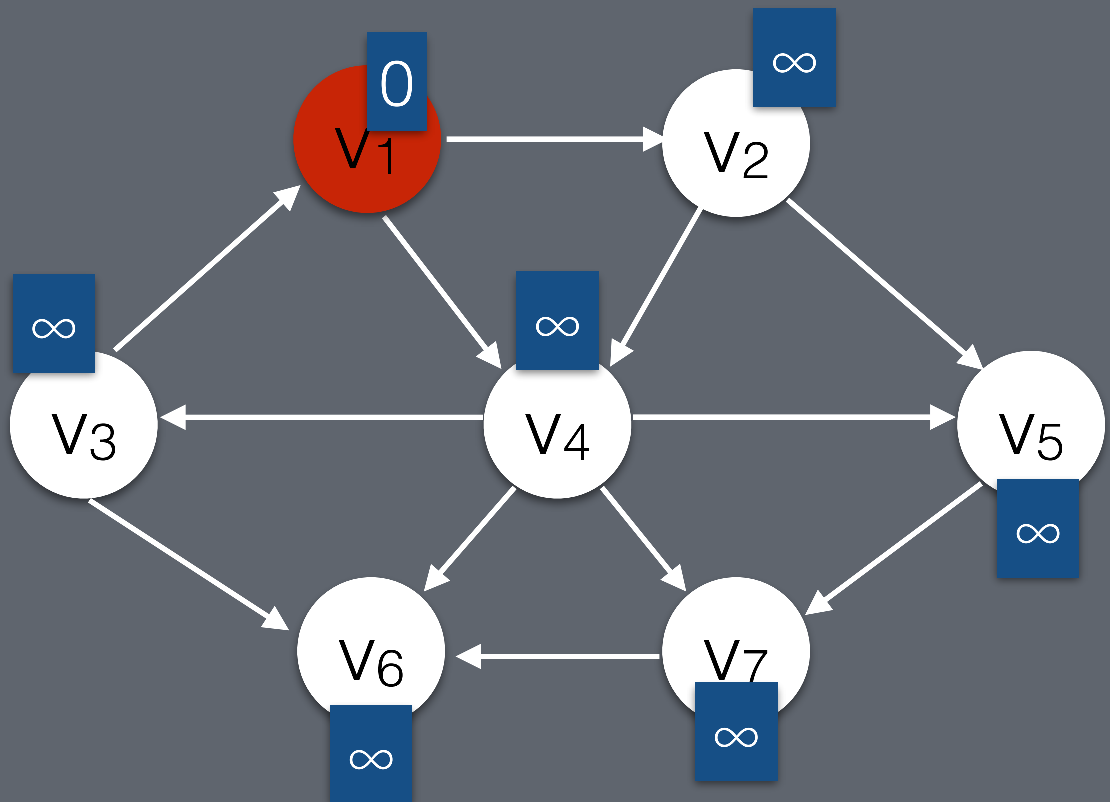
Set **visited**

while (**q** is not empty):

BFS will traverse the entire graph even without a visited set.

What happens if we use a stack (DFS)?

Unweighted Shortest Paths

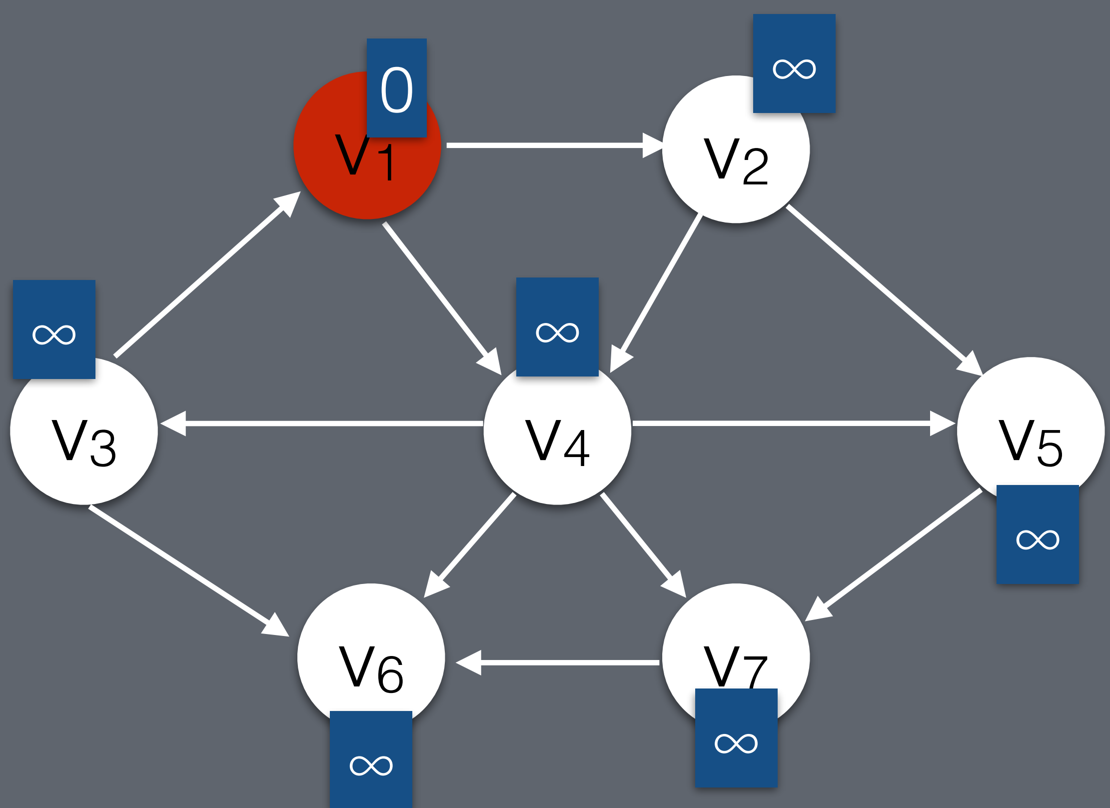


Queue: { V_1 }

for all **v**:
  v.cost = **∞**
  v.prev = **null**

start.cost = **0**

Unweighted Shortest Paths



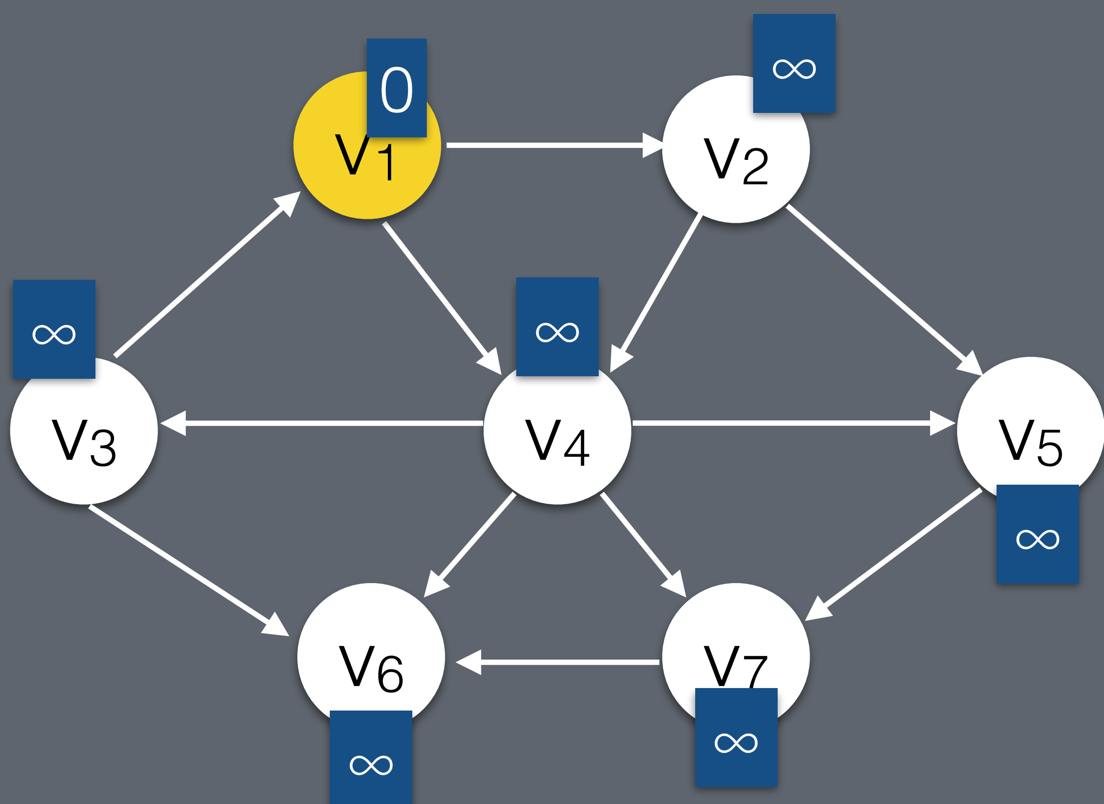
Queue: { V_1 }

for all **v**:
  v.cost = **∞**
  v.prev = **null**

start.cost = **0**

Queue **q**
q.enqueue(**start**)

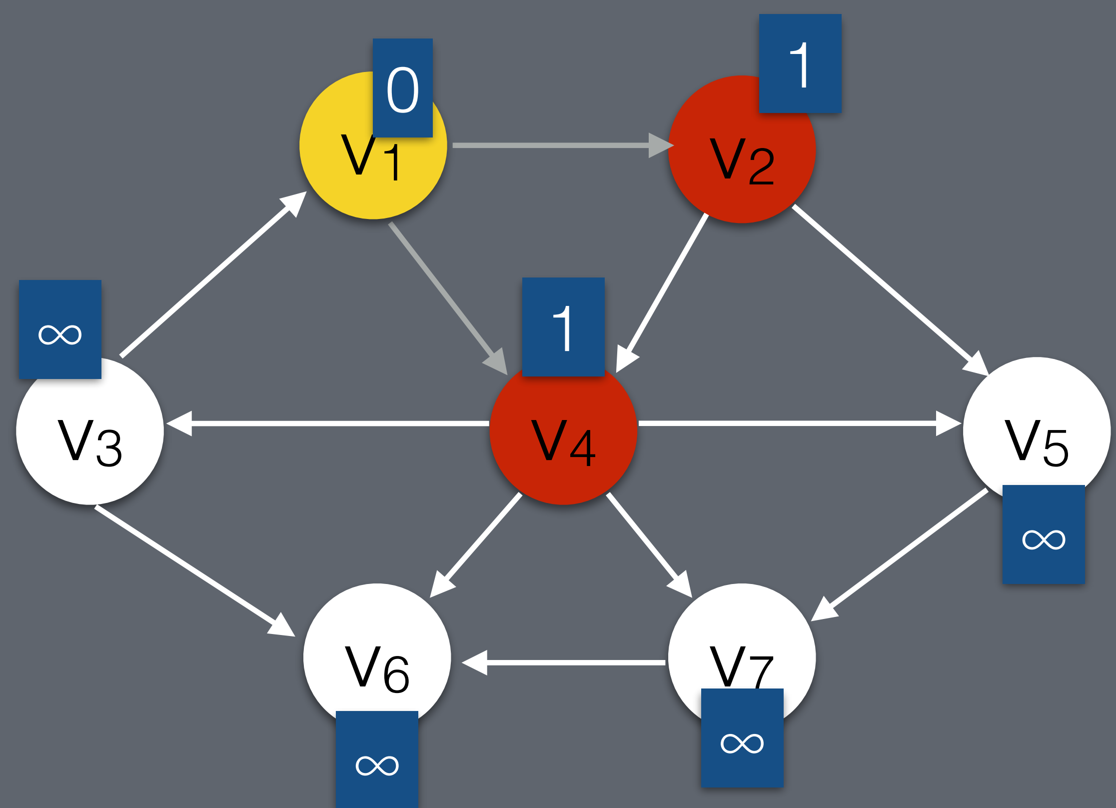
Unweighted Shortest Paths



Queue: { }

```
for all v:
  v.cost = ∞
  v.prev = null

start.cost = 0

Queue q
q.enqueue(start)

while (q is not empty):
  u = q.dequeue()

  for each v adjacent to u:
    if v.cost == ∞:
      v.cost = u.cost + 1
      v.prev = u
      q.enqueue(v)
```

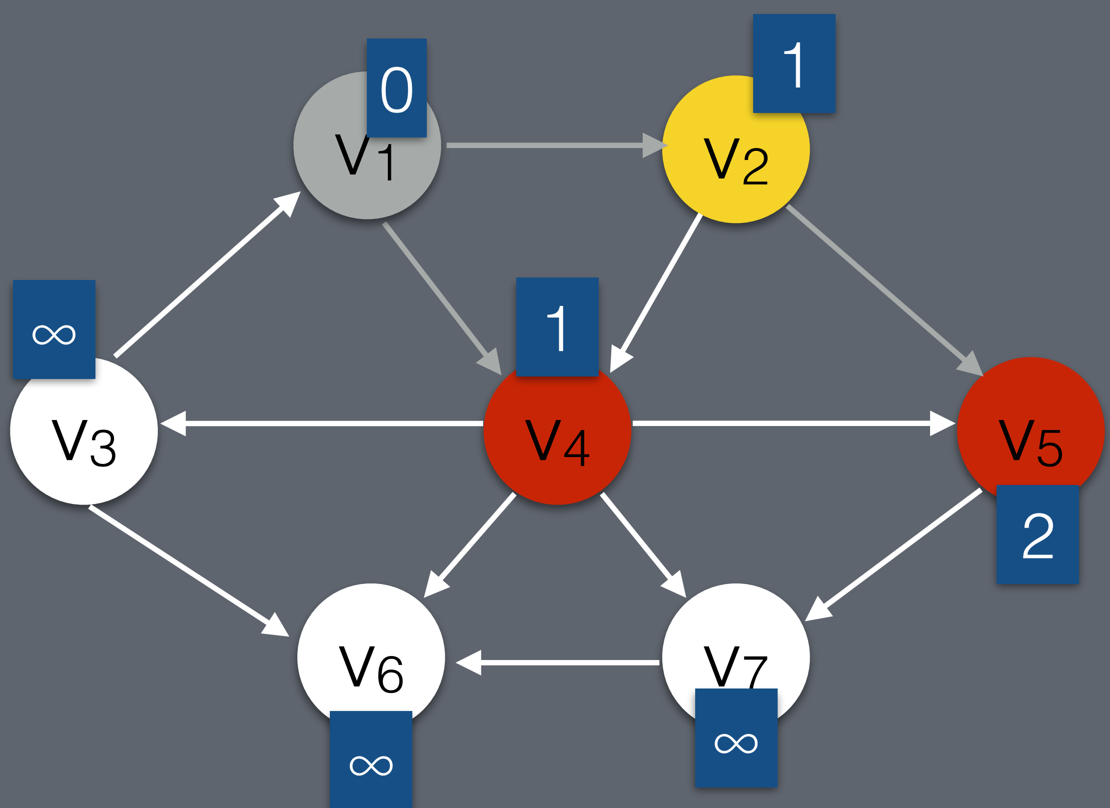
Unweighted Shortest Paths



Queue: { V_2, V_4}

```
for all v:
  v.cost = ∞
  v.prev = null

start.cost = 0

Queue q
q.enqueue(start)

while (q is not empty):
  u = q.dequeue()

  for each v adjacent to u:
    if v.cost == ∞:
      v.cost = u.cost + 1
      v.prev = u
      q.enqueue(v)
```

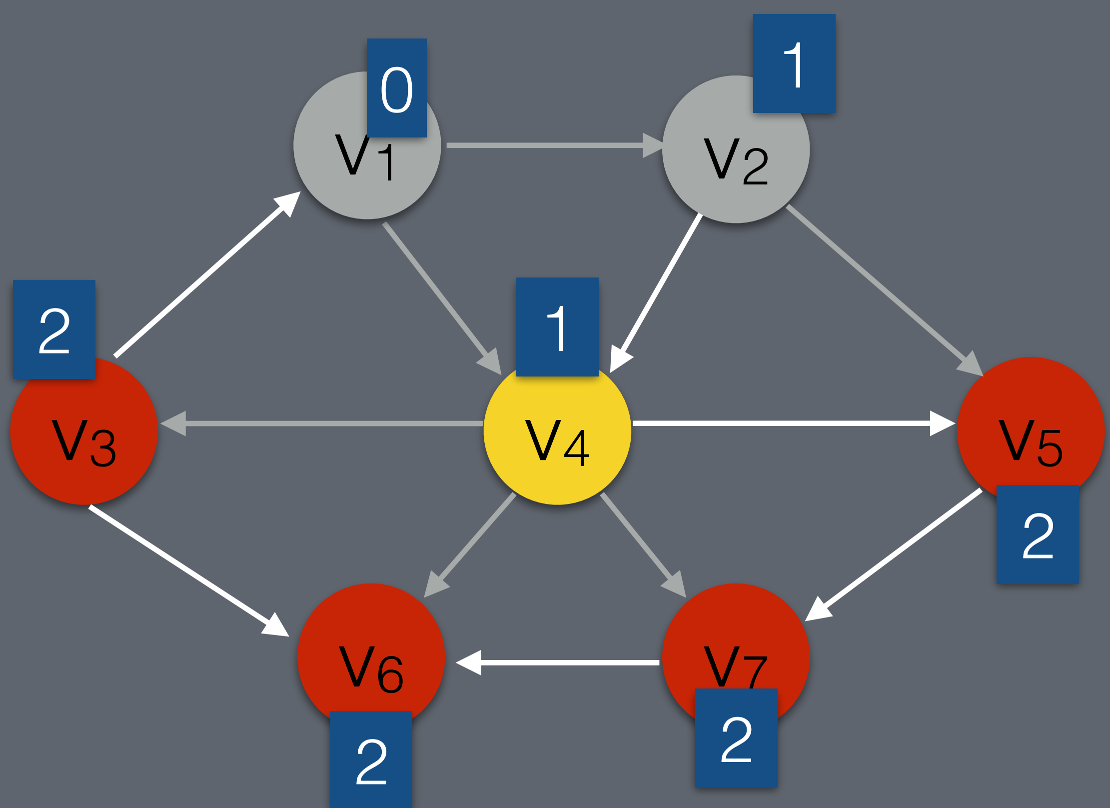
Unweighted Shortest Paths



Queue: { V_4, V_5 }

```
for all v:
  v.cost = ∞
  v.prev = null

start.cost = 0

Queue q
q.enqueue(start)

while (q is not empty):
  u = q.dequeue()

  for each v adjacent to u:
    if v.cost == ∞:
      v.cost = u.cost + 1
      v.prev = u
      q.enqueue(v)
```

Unweighted Shortest Paths

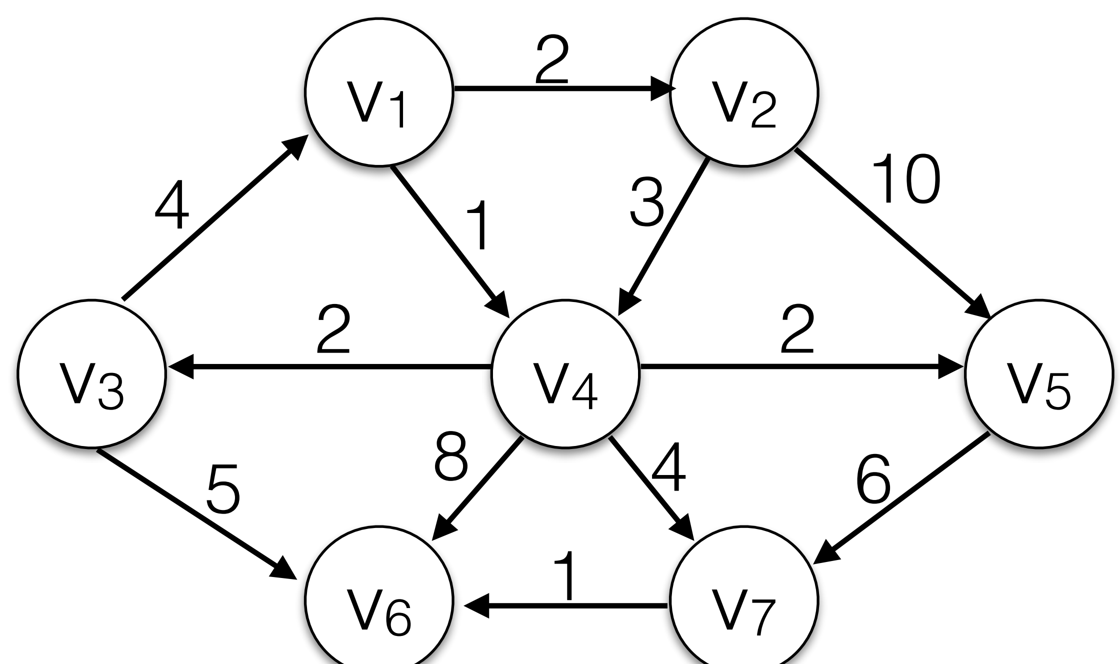


Queue: {V_5, V_3, V_6, V_7}

```
for all v:
  v.cost = ∞
  v.prev = null

start.cost = 0

Queue q
q.enqueue(start)

while (q is not empty):
  u = q.dequeue()

  for each v adjacent to u:
    if v.cost == ∞:
      v.cost = u.cost + 1
      v.prev = u
      q.enqueue(v)
```

Weighted Shortest Paths

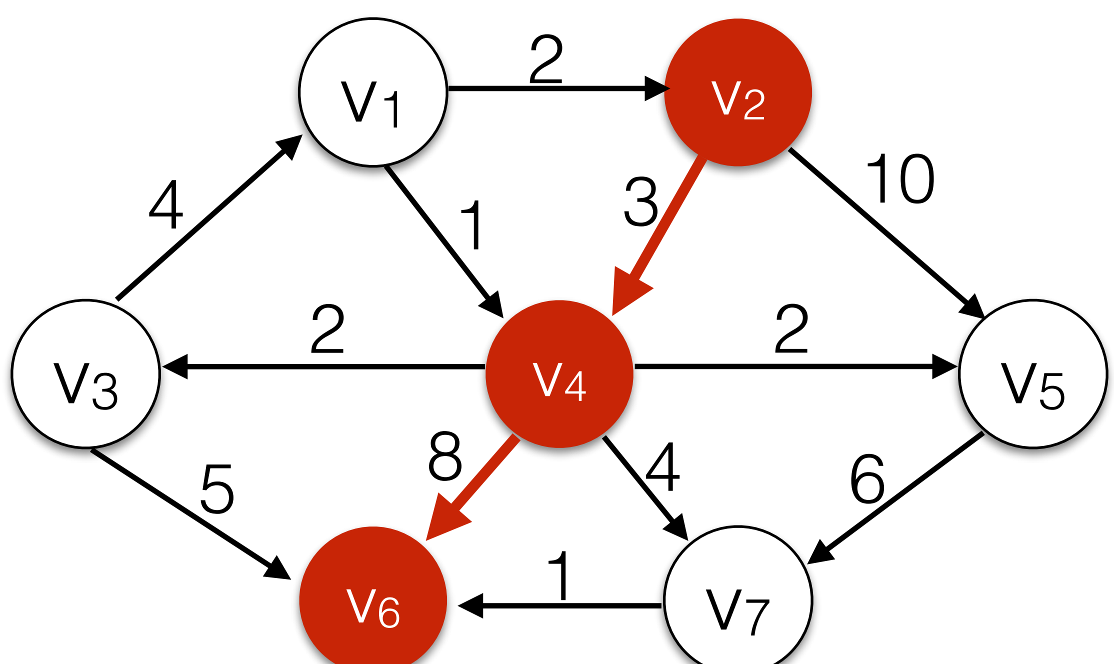
- Goal: Find the shortest path between two vertices s and t.



What is the shortest path between v_2 and v_6?

Weighted Shortest Paths

- Goal: Find the shortest path between two vertices s and t.
- Normal BFS will find this path.

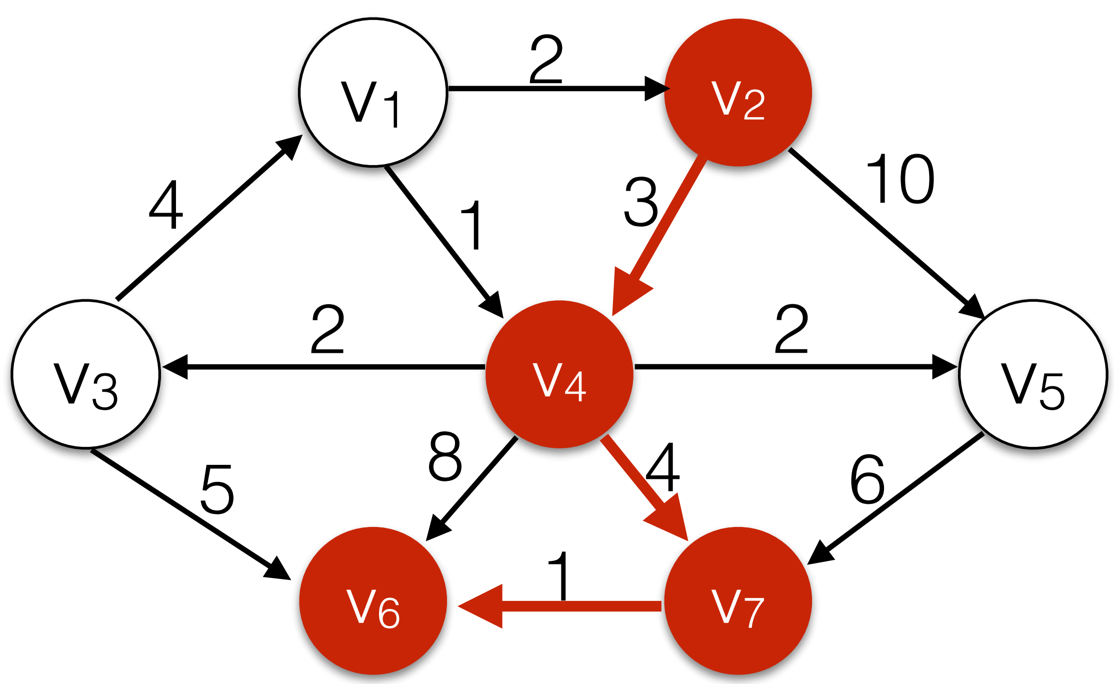


length 2
cost 11

What is the shortest path between v_2 and v_6?

Weighted Shortest Paths

- Goal: Find the shortest path between two vertices s and t.
- This path has a lower cost.

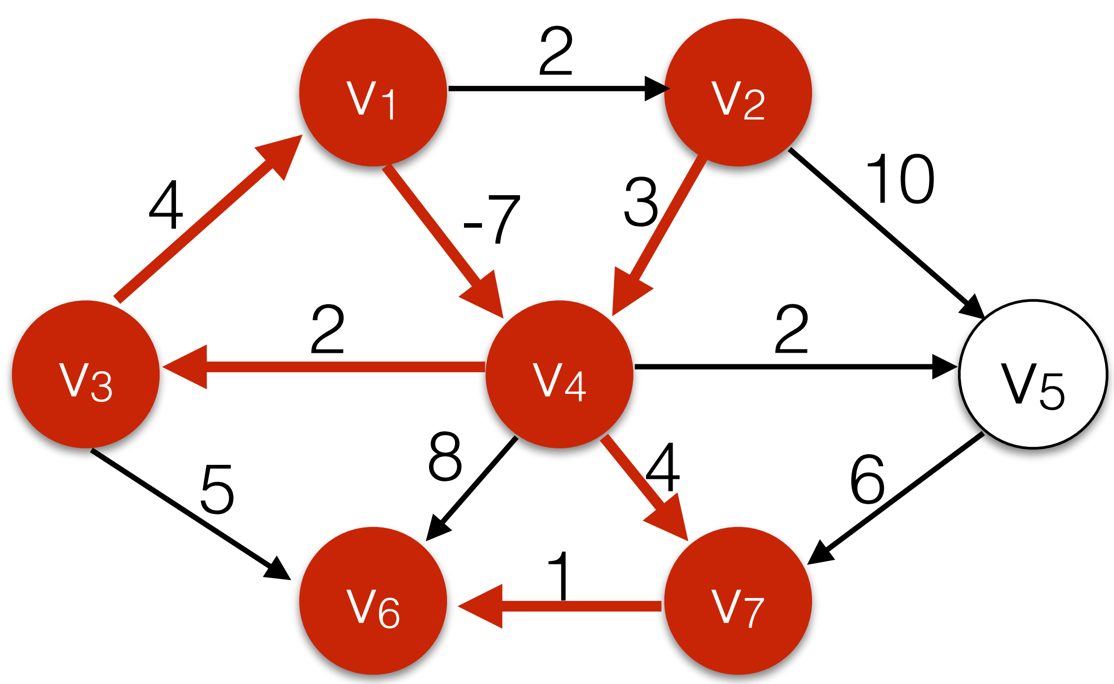


length 3
cost 8

What is the shortest path between v_2 and v_6?

Negative Weights

- We normally expect the shortest path to be simple.
- Edges with Negative Weights can lead to negative cycles.
- The concept of “shortest path” is then not clearly defined.



What is the shortest path between v_2 and v_6?

Dijkstra’s Algorithm for Weighted Shortest Path

Dijkstra’s Algorithm for Weighted Shortest Path

- Cost annotations for each vertex reflect the lowest cost *using only vertices visited so far.*

Dijkstra's Algorithm for Weighted Shortest Path

- Cost annotations for each vertex reflect the lowest cost *using only vertices visited so far.*
    - There might be a lower-cost path through other vertices that have not been seen yet.

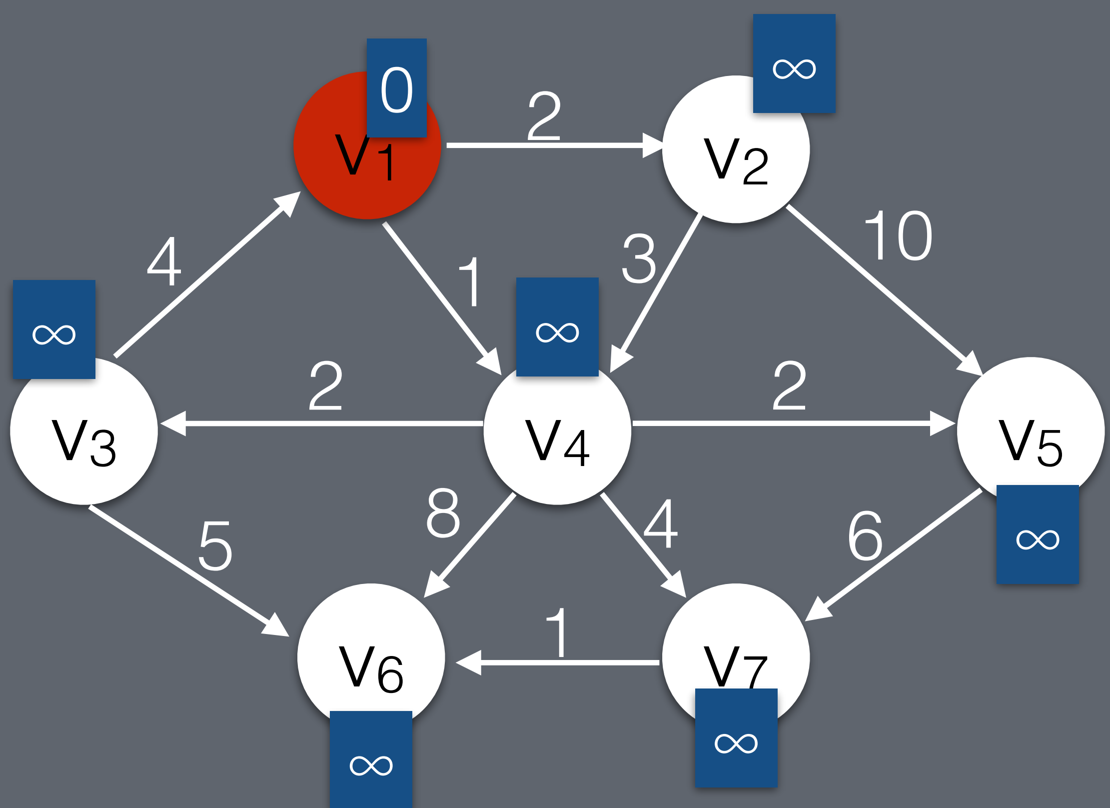
Dijkstra’s Algorithm for Weighted Shortest Path

- Cost annotations for each vertex reflect the lowest cost *using only vertices visited so far.*
  - There might be a lower-cost path through other vertices that have not been seen yet.
- Keep nodes on a **priority queue** and always expand the vertex with the lowest cost annotation first! ← This is a **greedy** algorithm

Dijkstra’s Algorithm for Weighted Shortest Path

- Cost annotations for each vertex reflect the lowest cost *using only vertices visited so far.*
  - There might be a lower-cost path through other vertices that have not been seen yet.
- Keep nodes on a **priority queue** and always expand the vertex with the lowest cost annotation first! ← This is a **greedy** algorithm
  - Intuitively, this means we will never overestimate the cost and miss lower-cost path.

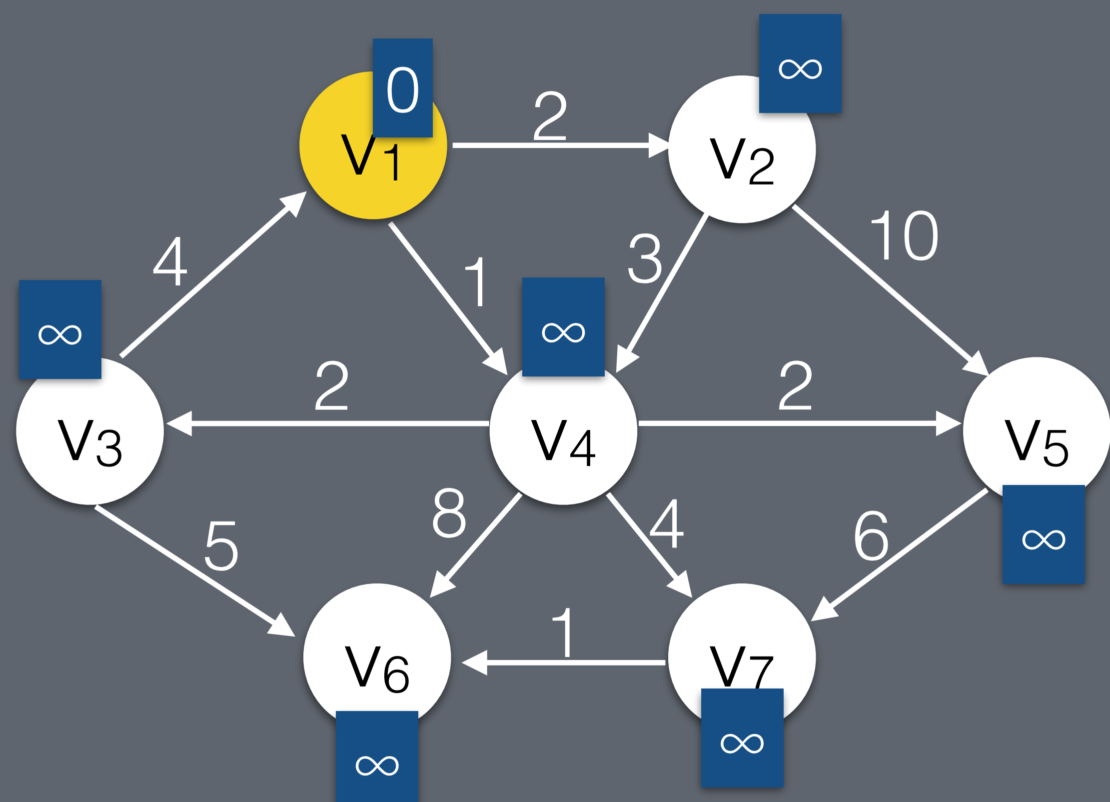
Dijkstra's Algorithm



for all **v**:
  v.cost = **∞**
  v.visited = **false**
  v.prev **= null**
start.cost = **0**

PriorityQueue **q**
q.insert(**start**)

Dijkstra's Algorithm

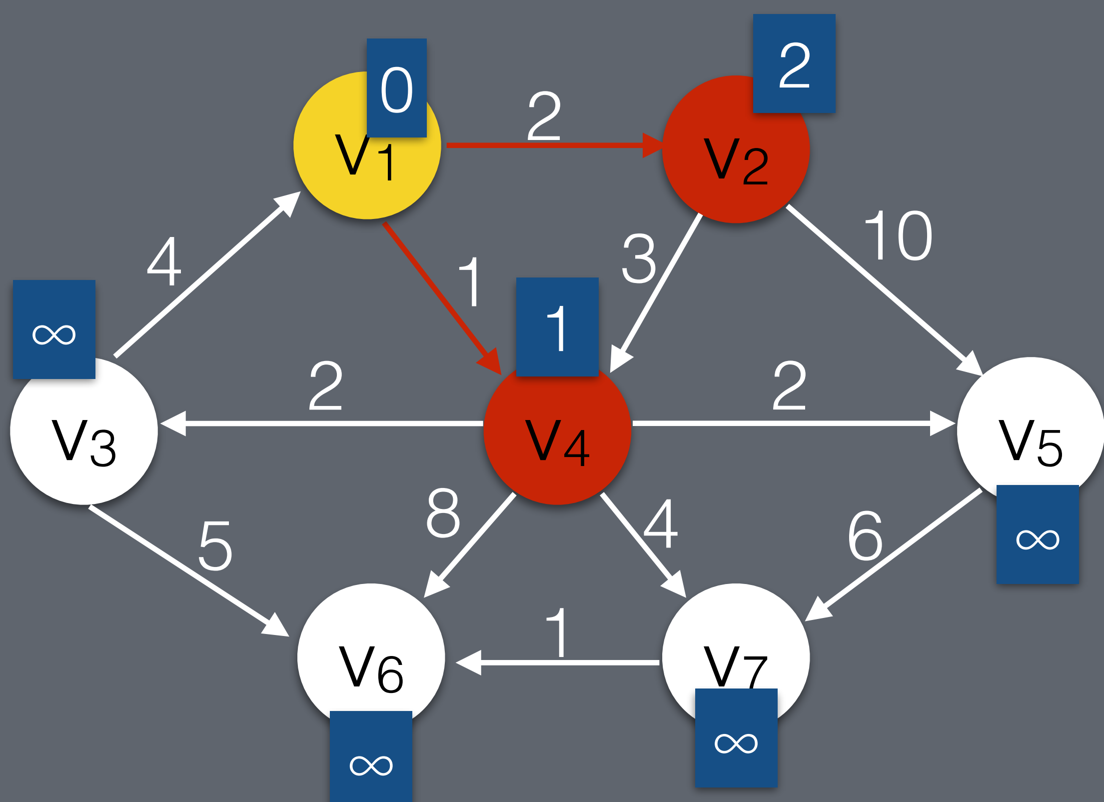



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
      c = u.cost + cost(u,v)
      if (c < v.cost):
        v.cost = c
        v.prev = u
        q.insert(v)
```

visit vertex **u**

discover and relax vertex **v**

Dijkstra's Algorithm

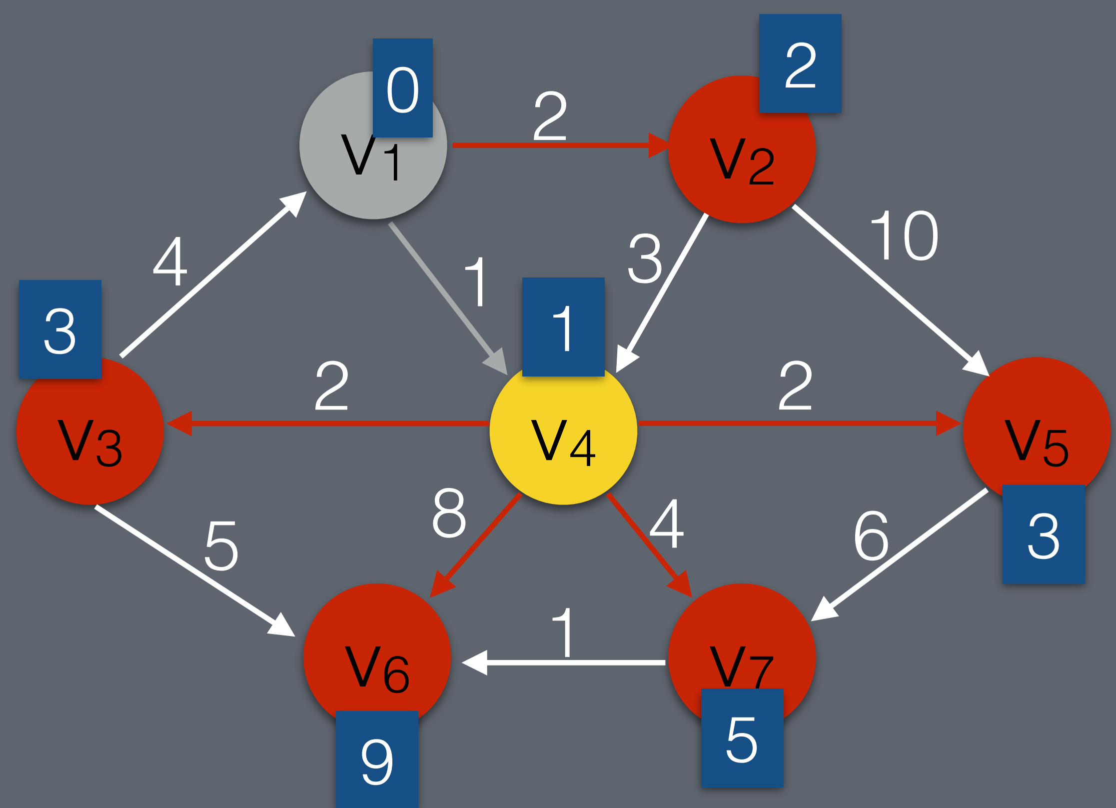



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
      c = u.cost + cost(u,v)
      if (c < v.cost):
        v.cost = c
        v.prev = u
        q.insert(v)
```

visit vertex **u**

discover and relax vertex **v**

Dijkstra's Algorithm

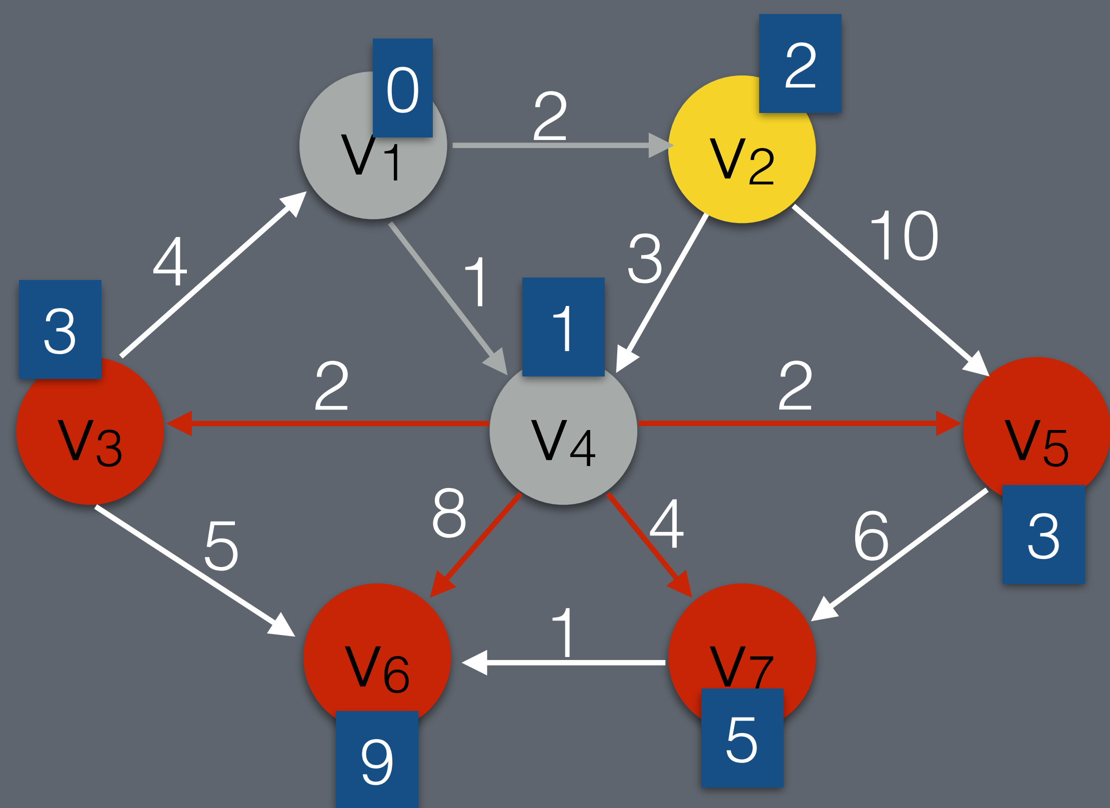



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
      c = u.cost + cost(u,v)
      if (c < v.cost):
        v.cost = c
        v.prev = u
        q.insert(v)
```

visit vertex **u**

discover and relax vertex **v**

Dijkstra's Algorithm

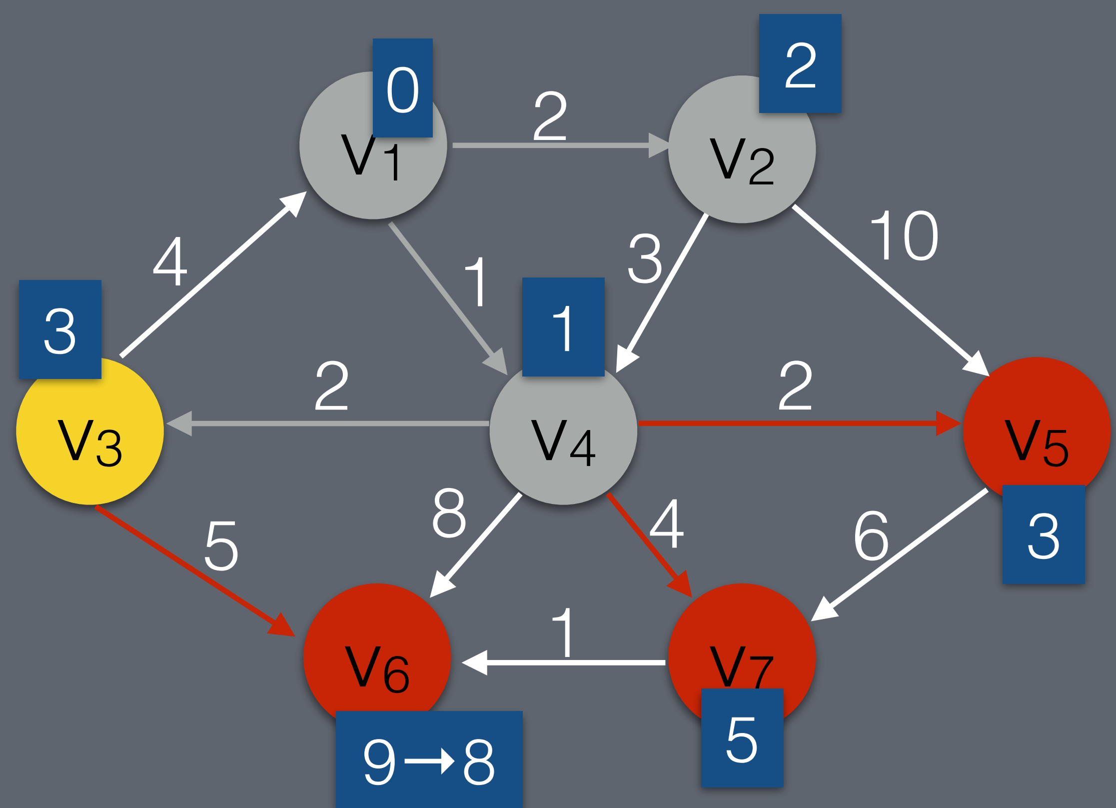



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
      c = u.cost + cost(u,v)
      if (c < v.cost):
        v.cost = c
        v.prev = u
        q.insert(v)
```

visit vertex **u**

discover and relax vertex **v**

Dijkstra's Algorithm



for all **v**:
  v.cost = **∞**
  v.visited = **false**
  v.prev **= null**
start.cost = **0**

PriorityQueue **q**
q.insert(**start**)

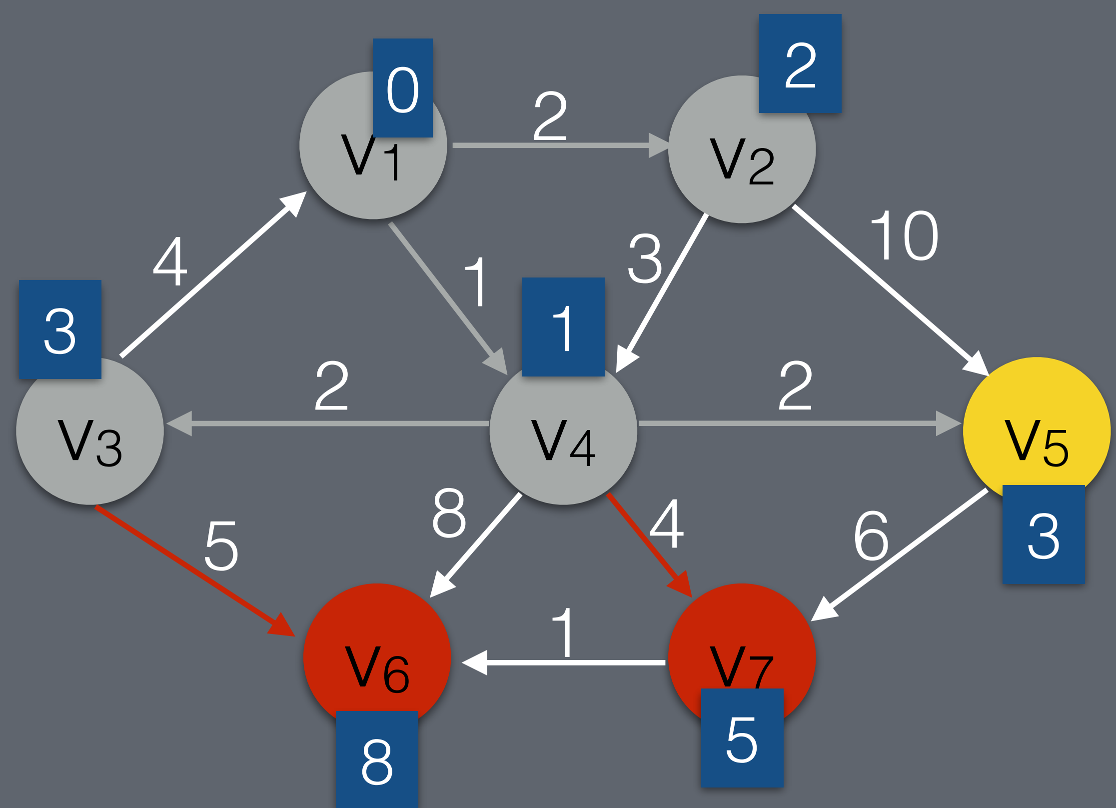
while (**q** is not empty):
  u = **q**.pollMin()
  u.visited **= true**

  for each **v** adjacent to **u**:
    if not **v**.visited:
      c = **u**.cost + cost(**u**,**v**)
      if (**c** < **v**.cost):
        v.cost = **c**
        v.prev **= u**
        q.insert(**v**)

visit vertex **u**

discover and relax vertex **v**

Dijkstra's Algorithm

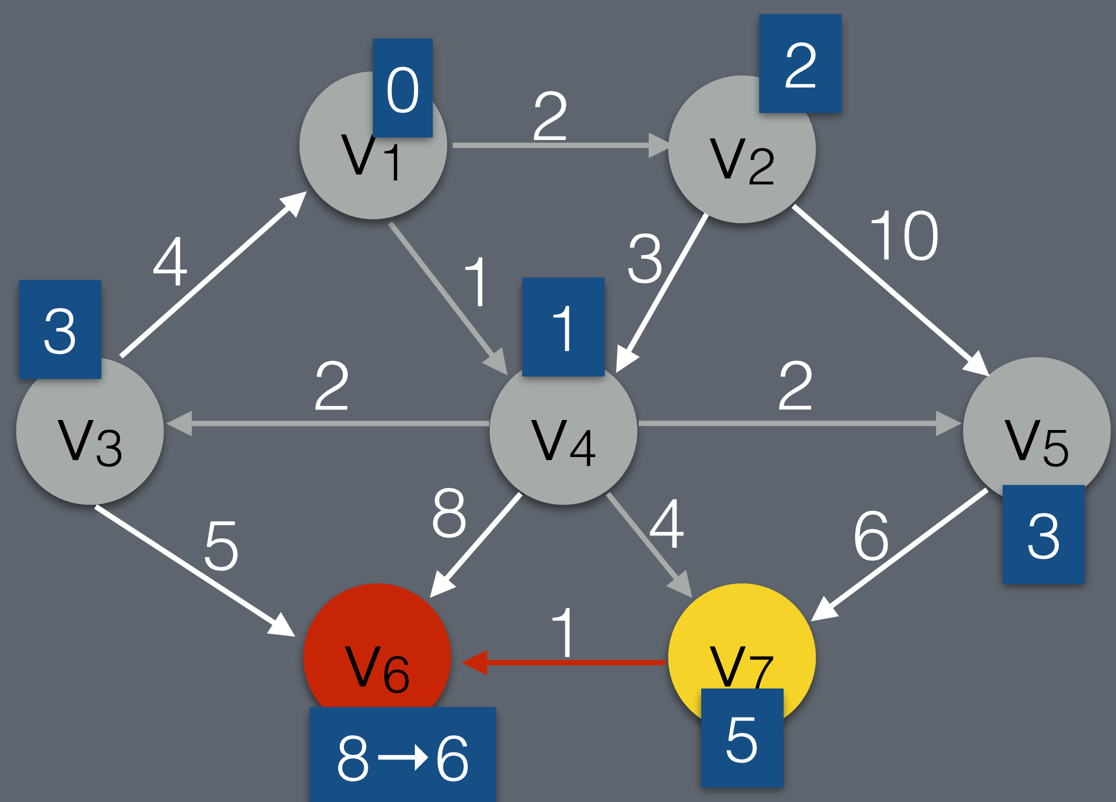



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
      c = u.cost + cost(u,v)
      if (c < v.cost):
        v.cost = c
        v.prev = u
        q.insert(v)
```

visit vertex **u**

discover and relax vertex **v**

Dijkstra's Algorithm

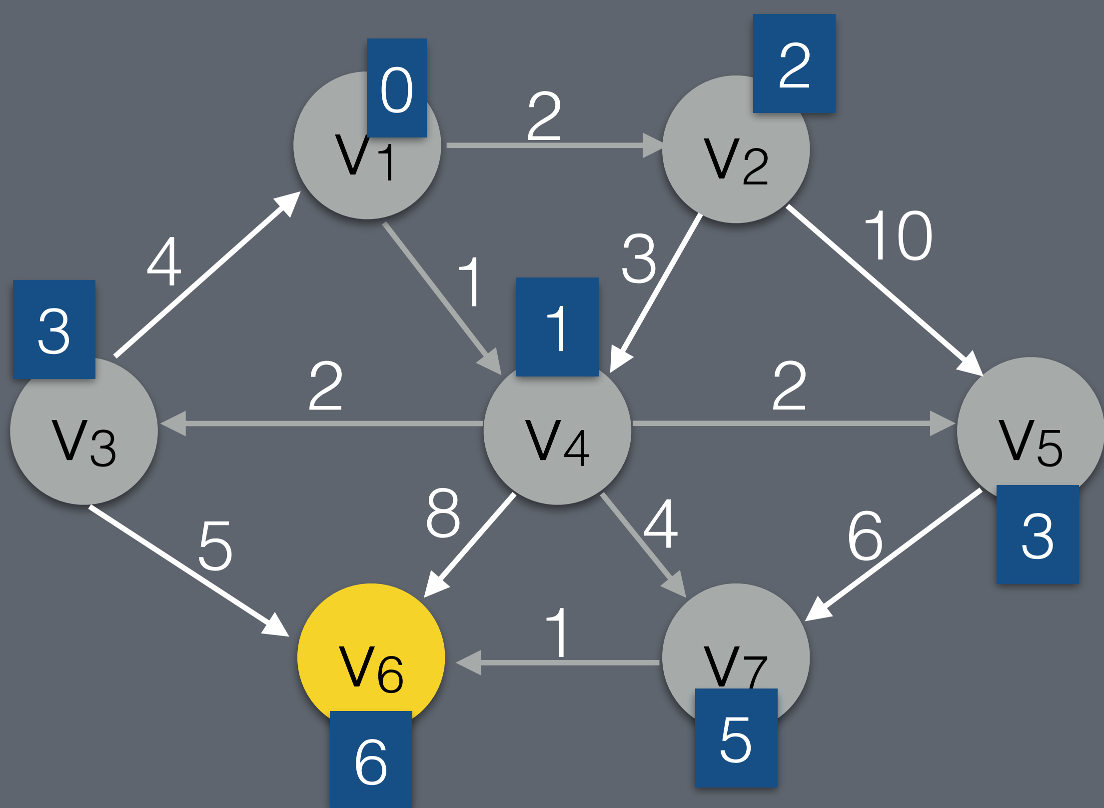



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
      c = u.cost + cost(u,v)
      if (c < v.cost):
        v.cost = c
        v.prev = u
        q.insert(v)
```

visit vertex **u**

discover and relax vertex **v**

Dijkstra's Algorithm



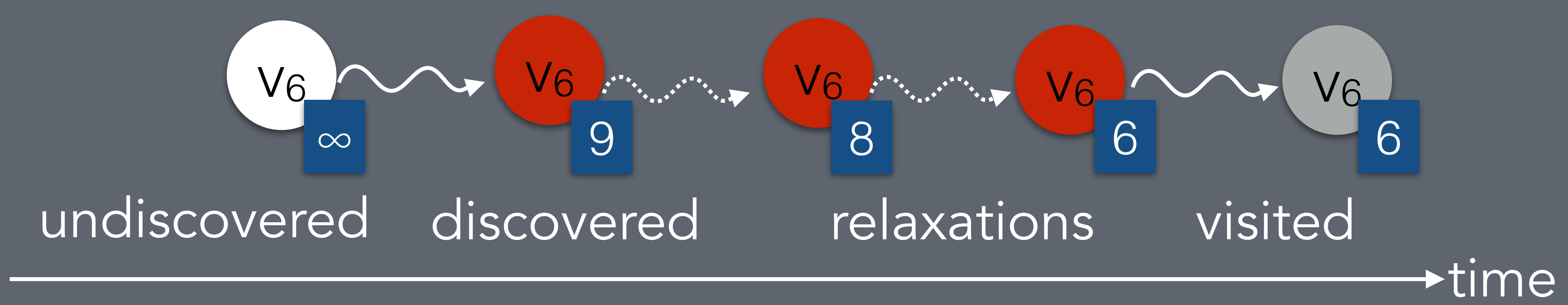

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

PriorityQueue q
q.insert(start)

while (q is not empty):
  u = q.pollMin()
  u.visited = true

  for each v adjacent to u:
    if not v.visited:
      c = u.cost + cost(u,v)
      if (c < v.cost):
        v.cost = c
        v.prev = u
        q.insert(v)
```

visit vertex **u**

discover and relax vertex **v**

Dijkstra's Algorithm

"Life Cycle" of a Vertex



Dijkstra's Running Time

- There are |E| insert and deleteMin operations.
- The maximum size of the priority queue is O(|E|). Each insert takes O(log |E|)

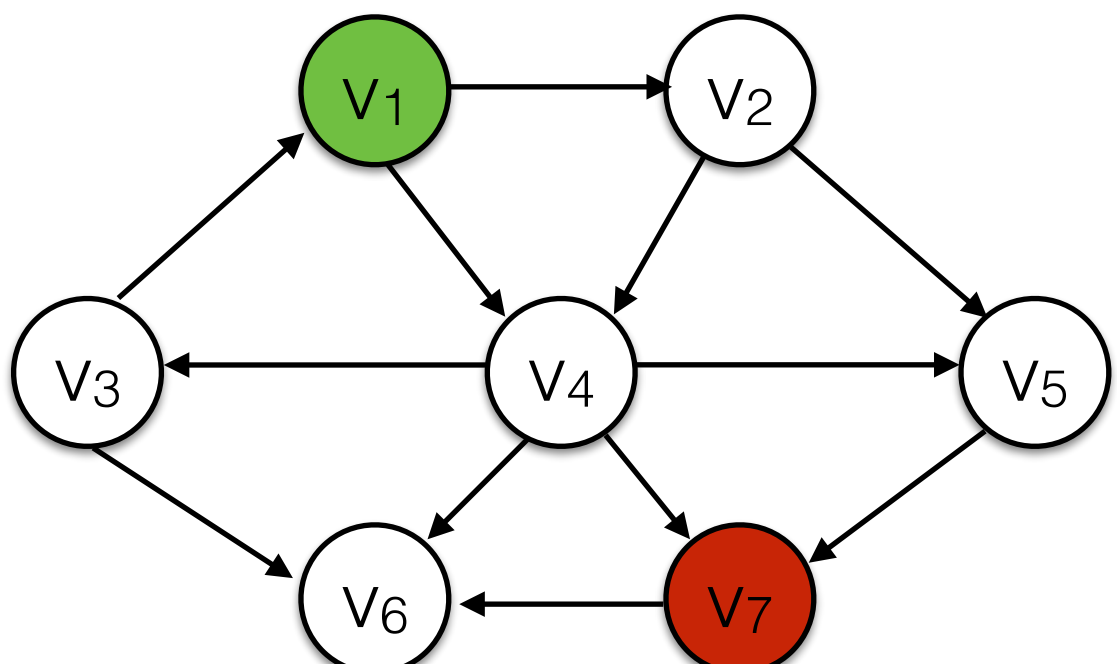
$O(|E| \log |E|)$
$= O(|E| \log |V|)$

$|E| \leq |V|^2$, so
$\log |E| \leq 2 \log |V| = O(\log |V|)$

A General View of Graph Search

Goals:

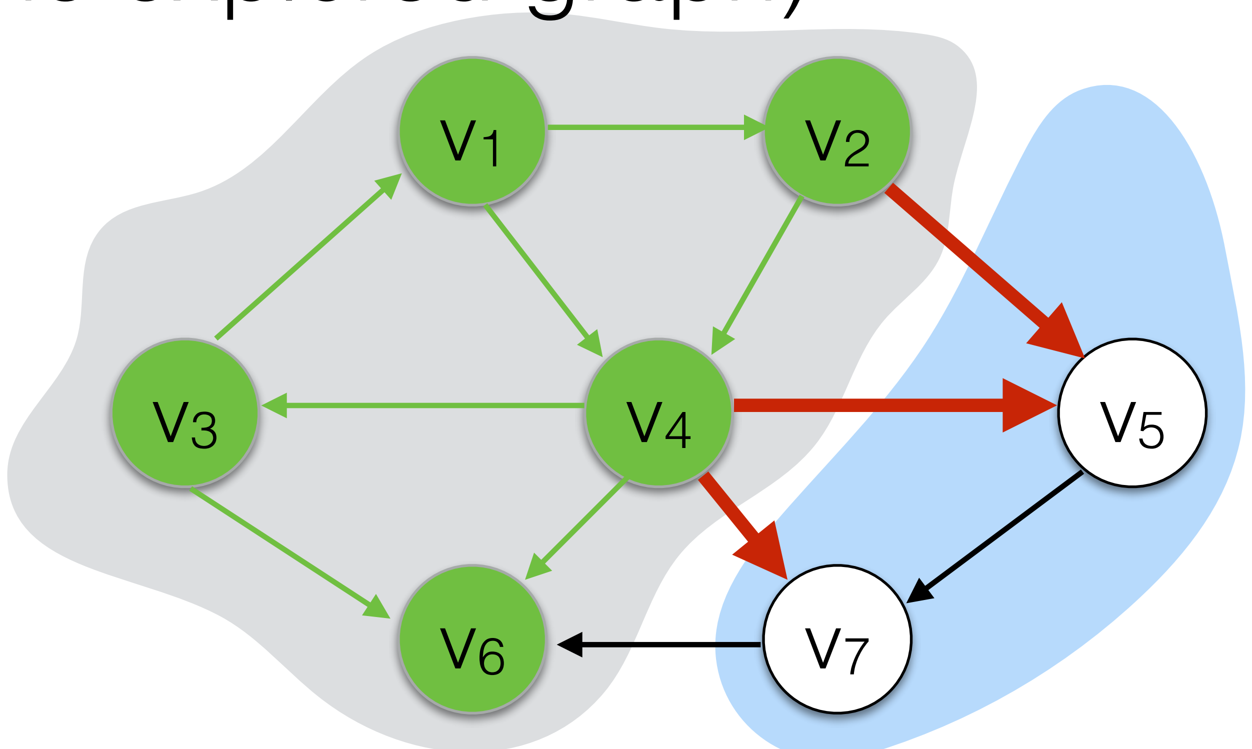
- Explore the graph systematically starting at *s* to
  - Find a vertex *t* / Find a path from *s* to *t*.
  - Find the shortest path from *s* to all vertices.
  - …



A General View of Graph Search

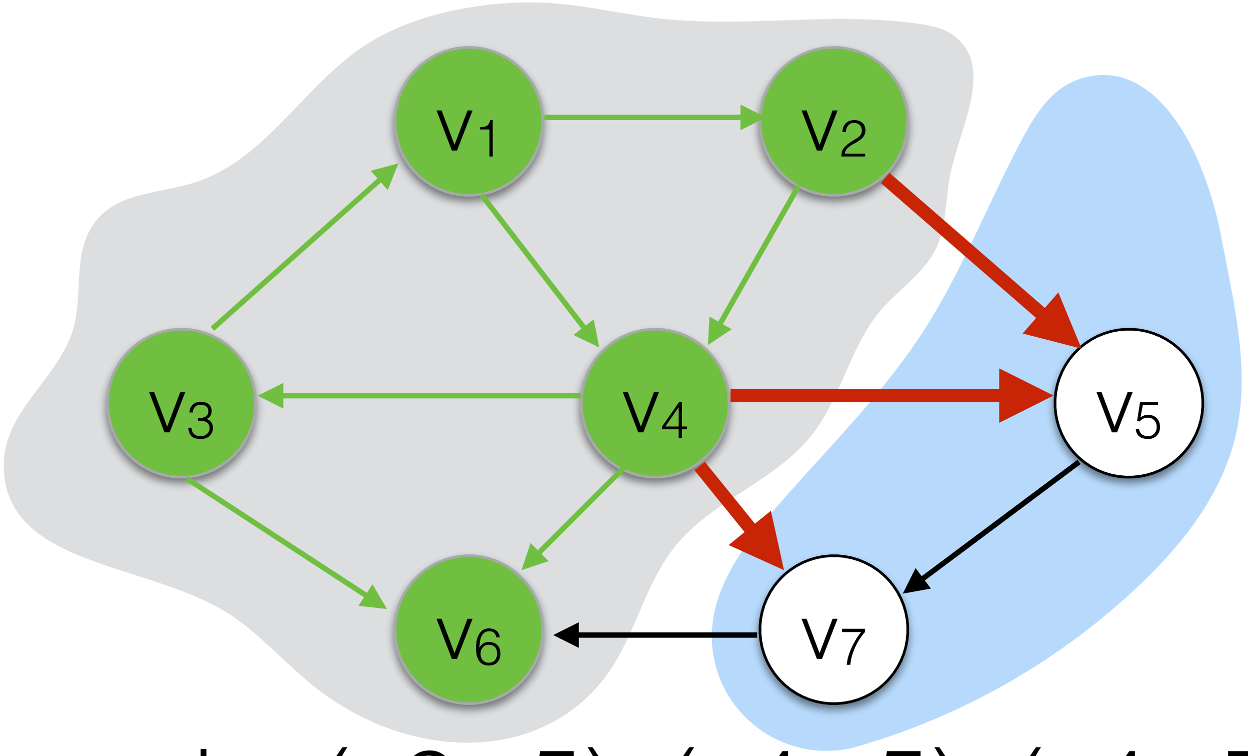
In every step of the search we maintain

- The part of the graph already explored.
- The part of the graph not yet explored.
- A data structure (an agenda) of *next* edges (adjacent to the explored graph).



Agenda: (v2,v5), (v4,v5), (v4,v7)

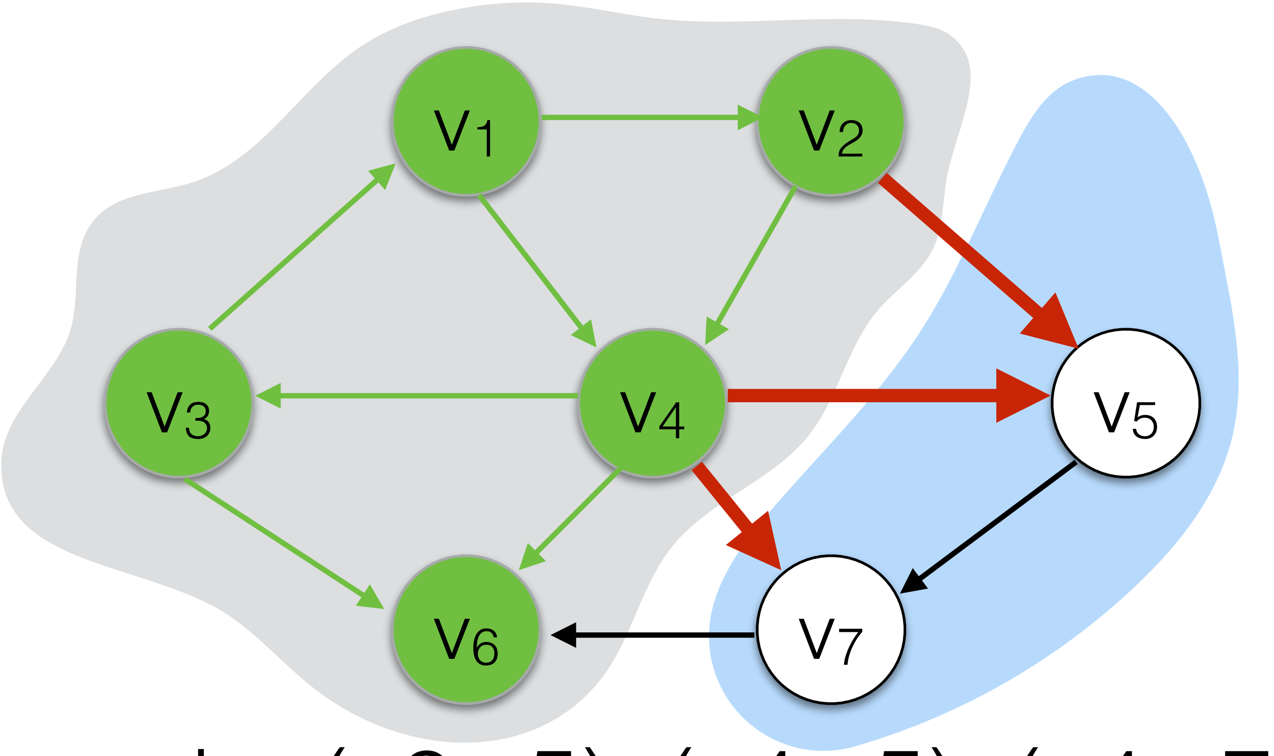
A General View of Graph Search



Agenda: (v2,v5), (v4,v5), (v4,v7)

A General View of Graph Search

The graph search algorithms discussed so far differ almost only in the type of agenda they use:

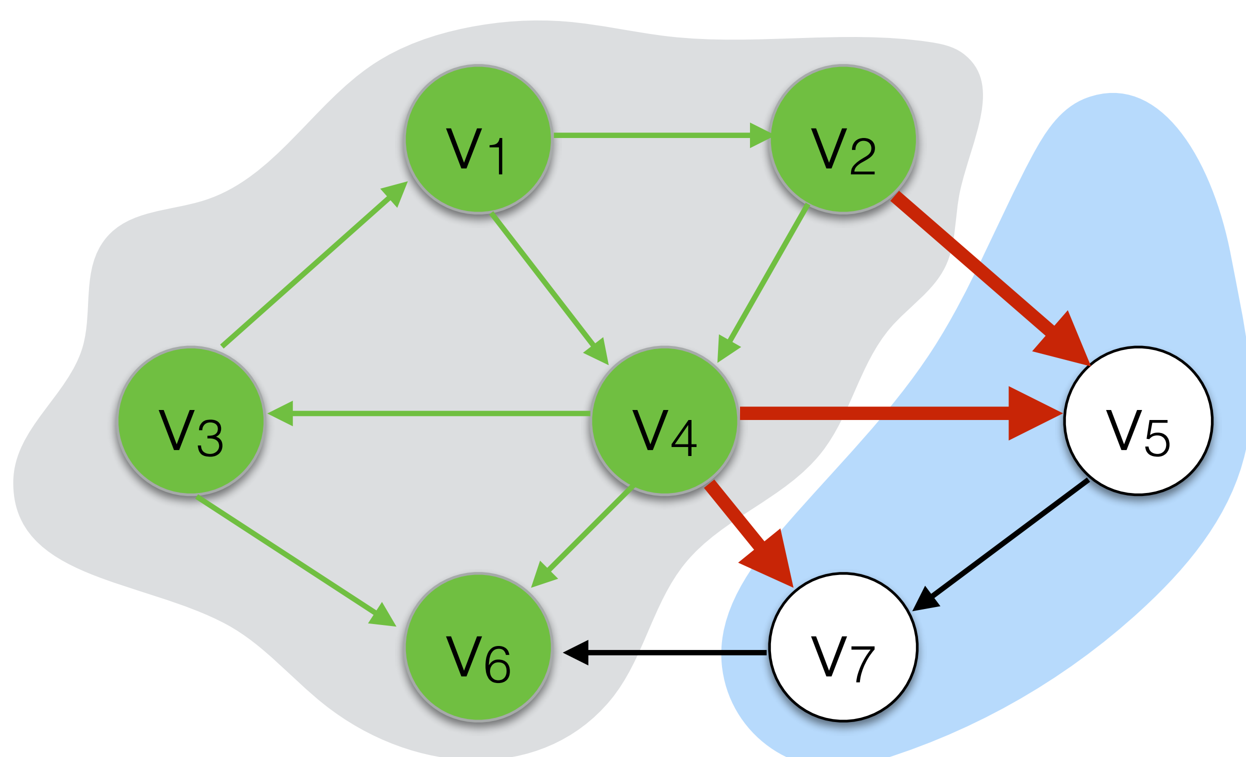


Agenda: (v2,v5), (v4,v5), (v4,v7)

A General View of Graph Search

The graph search algorithms discussed so far differ almost only in the type of agenda they use:

- unweighted shortest paths: breadth first, uses a queue.

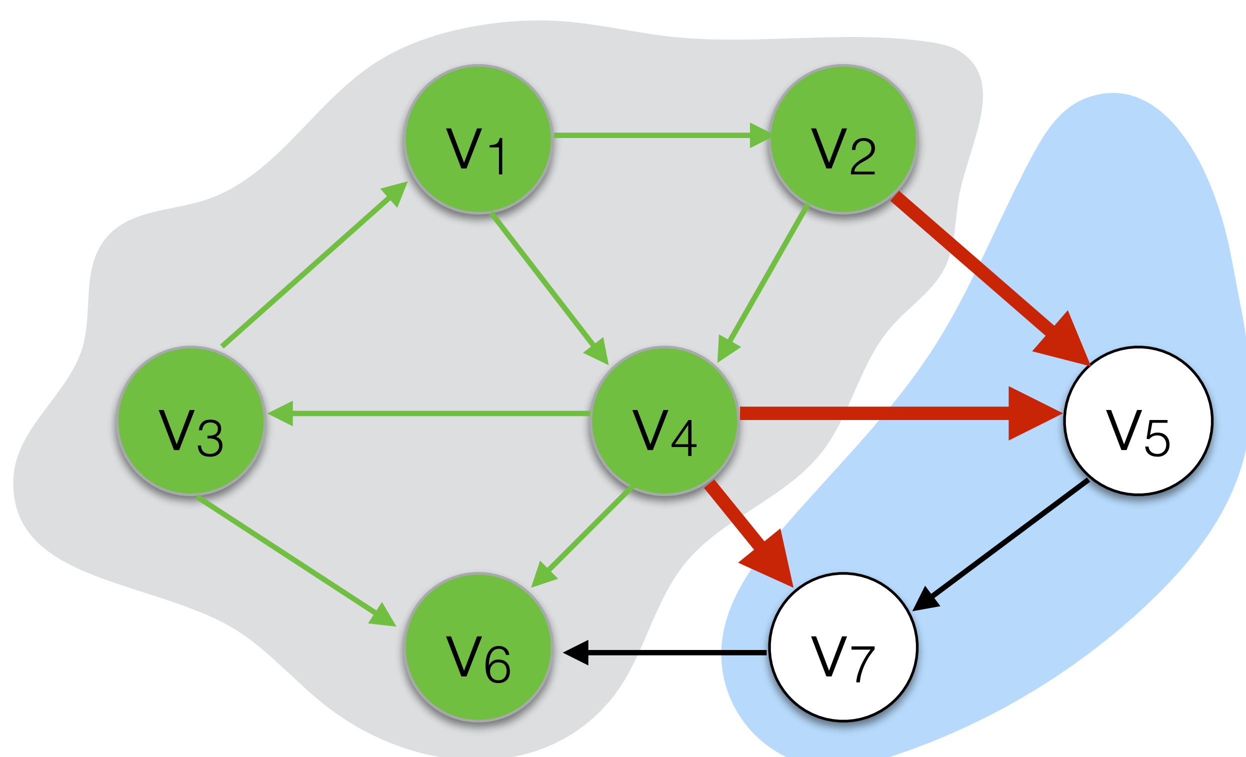


Agenda: (v2,v5), (v4,v5), (v4,v7)

A General View of Graph Search

The graph search algorithms discussed so far differ almost only in the type of agenda they use:

- unweighted shortest paths: breadth first, uses a queue.
- Dijkstra's: best first, uses a priority queue.

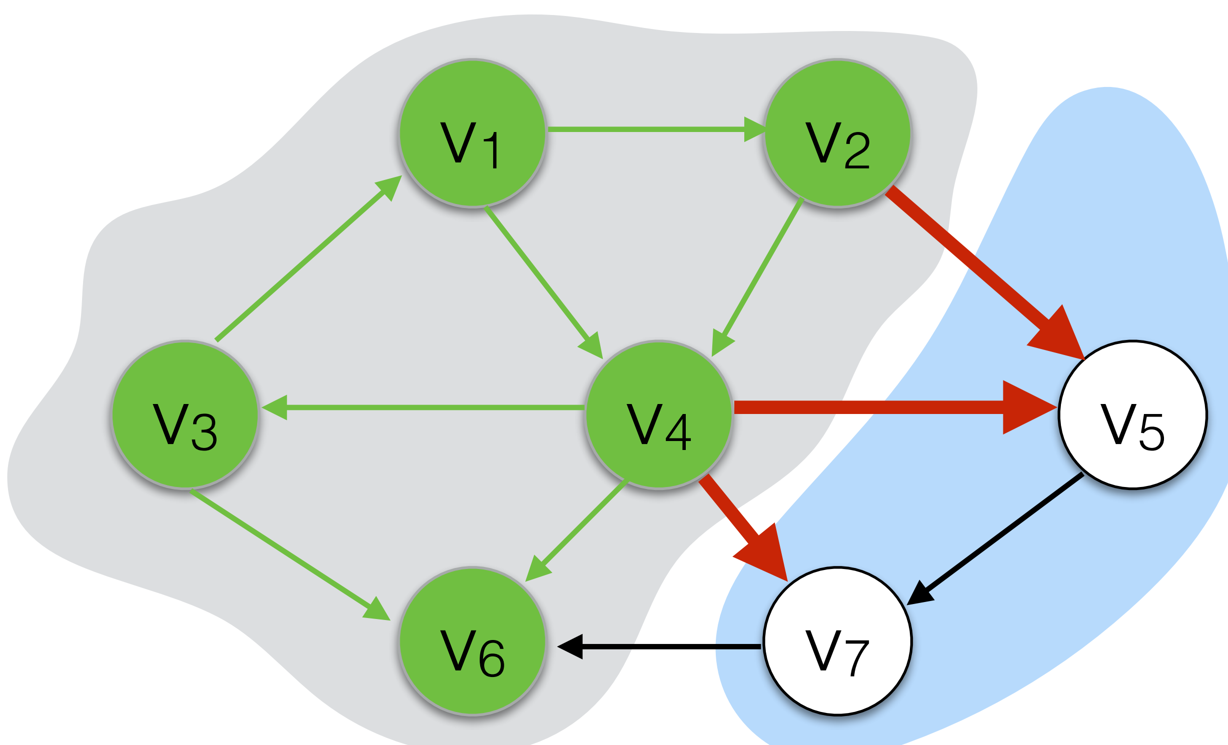


Agenda: (v2,v5), (v4,v5), (v4,v7)

A General View of Graph Search

The graph search algorithms discussed so far differ almost only in the type of agenda they use:

- unweighted shortest paths: breadth first, uses a queue.
- Dijkstra's: best first, uses a priority queue.
- Topological Sort: breadth first with constraint on items in the queue.



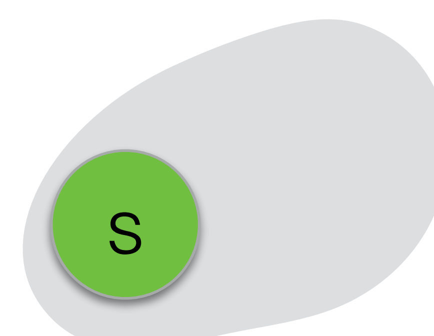
Agenda: (v2,v5), (v4,v5), (v4,v7)

Correctness of Dijkstra's Algorithm

- We want to show that Dijkstra's algorithm really finds the minimum path costs (we don't miss any shorter solutions by choosing the shortest edge greedily).

- Proof by induction on the set S of visited nodes.

- Base case:
  |S|=1. Trivial. Length shortest path is 0.

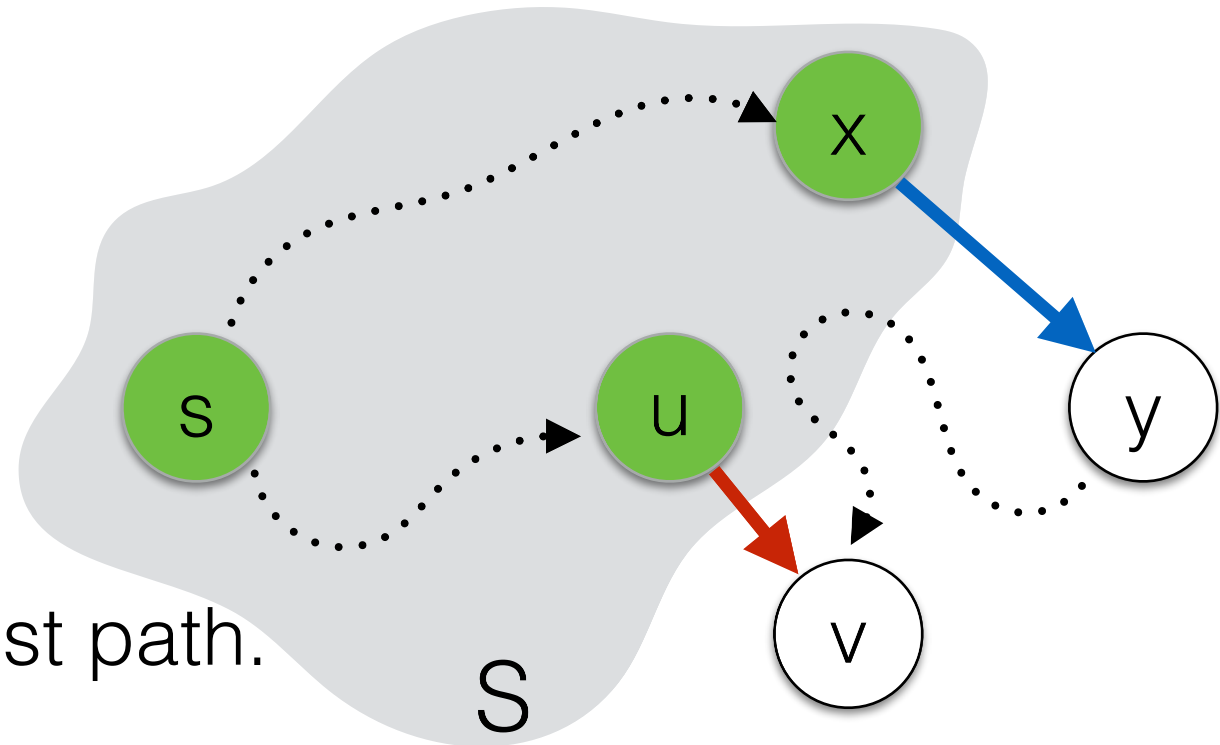


Correctness of Dijkstra's Inductive Step

- Assume the algorithm produces the minimal path cost from *s* for the subset S, |S| = k.

- Dijkstra's algorithm selects the next edge (u,v) leaving S.

- Assume there was a shorter path from s to v that does not contain (u,v).
    - Then that path must contain another edge (x,y) leaving S.
    - The cost of (x,y) is already higher than (u,v) because we didn't choose it before (u,v)



- Therefore (u,v) must be on the shortest path.