Data Structures in Java

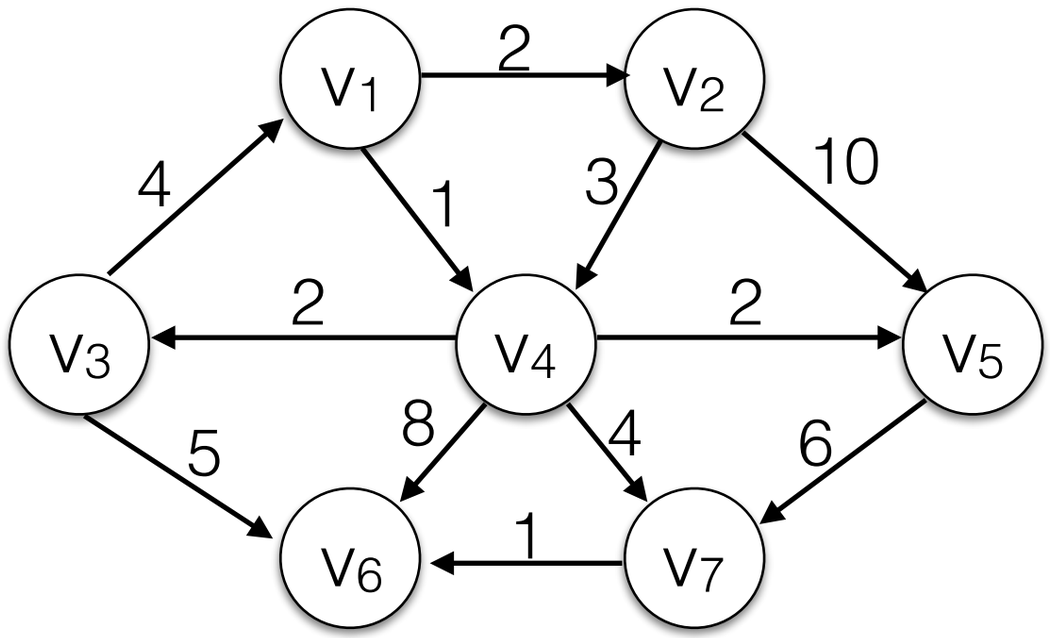
Lecture 19: Weighted Shortest Paths

11/20/2019

Daniel Bauer

Weighted Shortest Paths

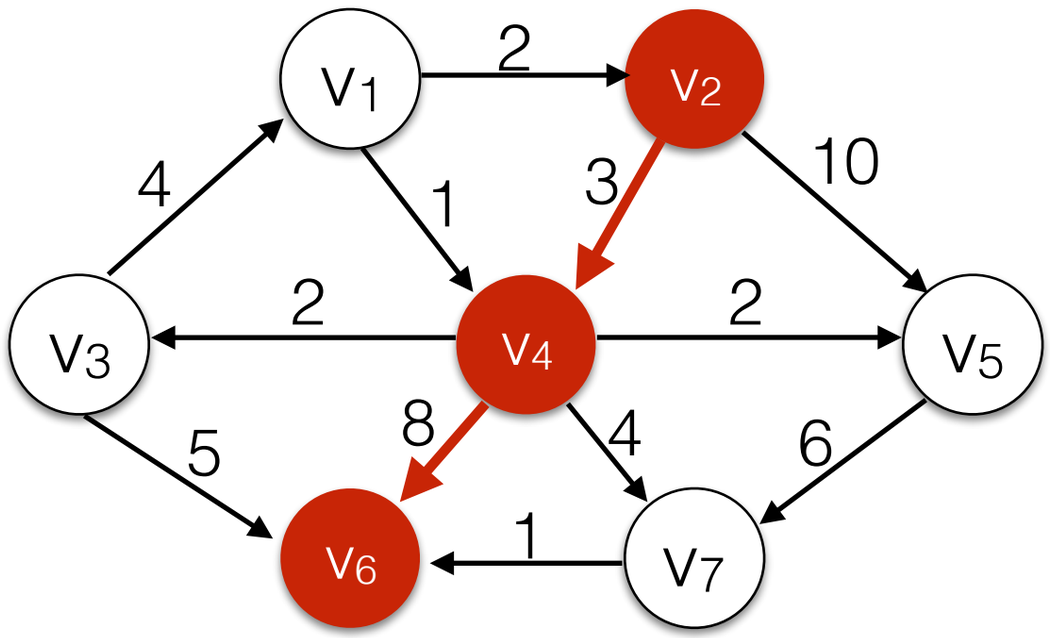
- Goal: Find the shortest path between two vertices s and t.



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.

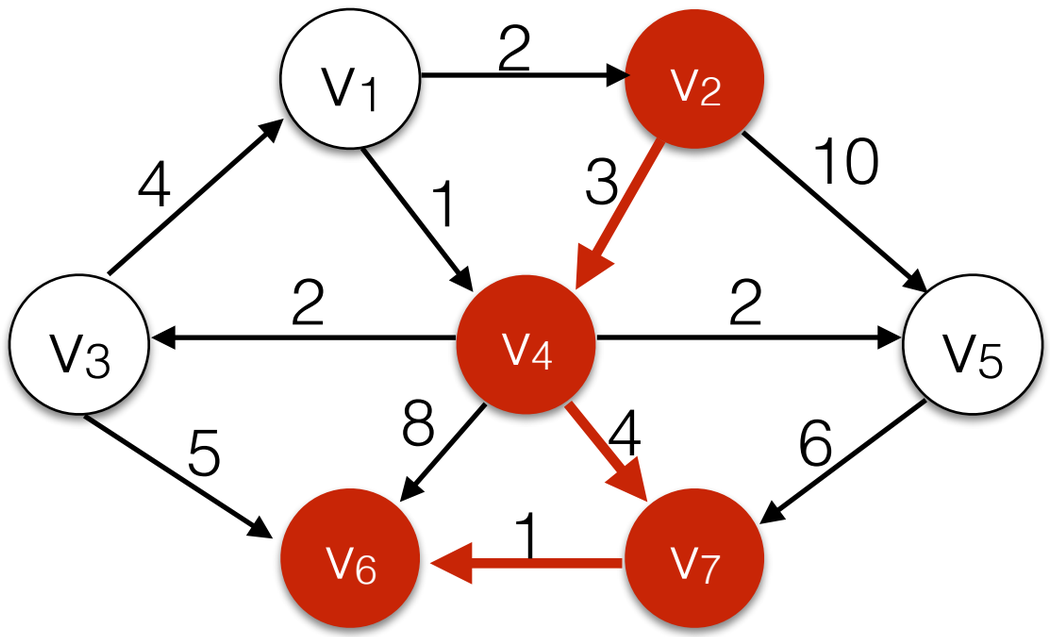


length 2
cost 11

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.

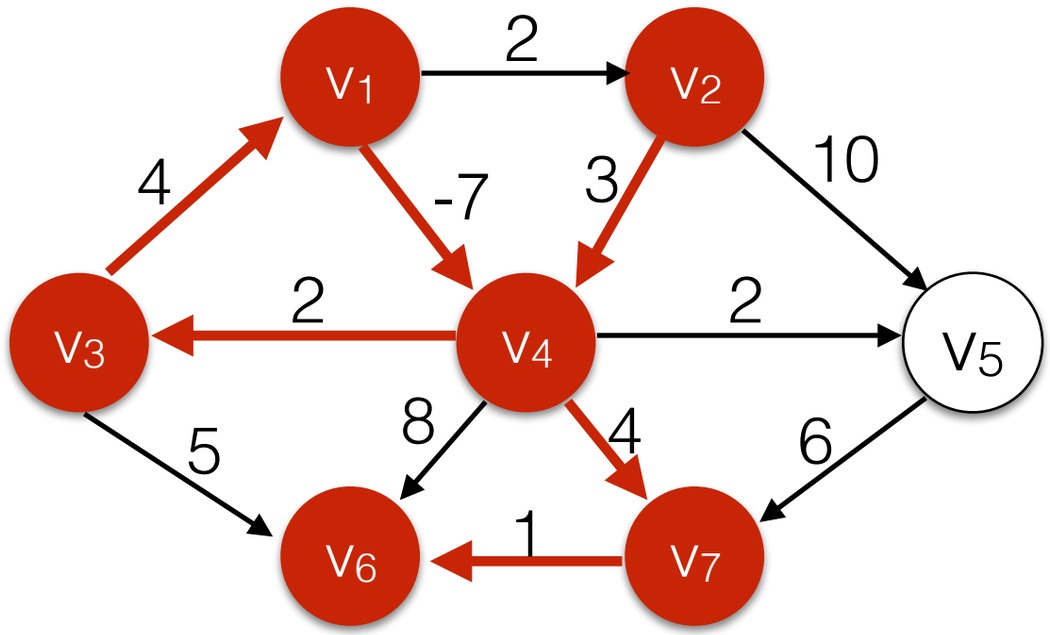


length 3
cost 8

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.

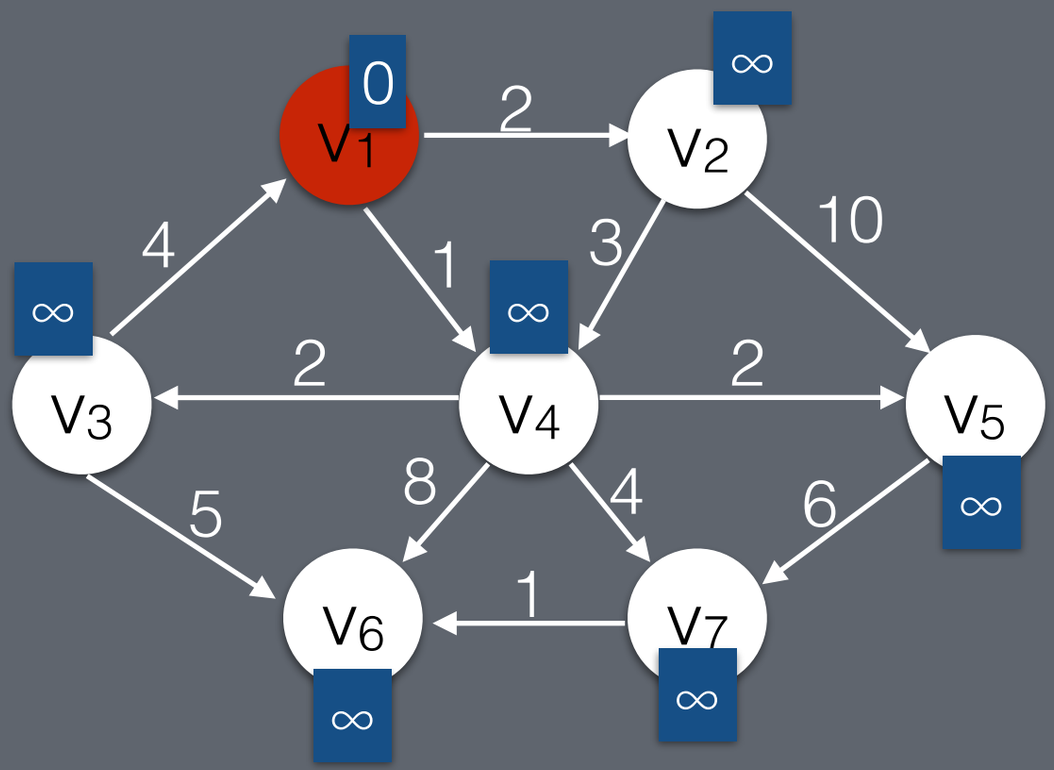
Dijkstra's Algorithm for Weighted Shortest Path

- 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

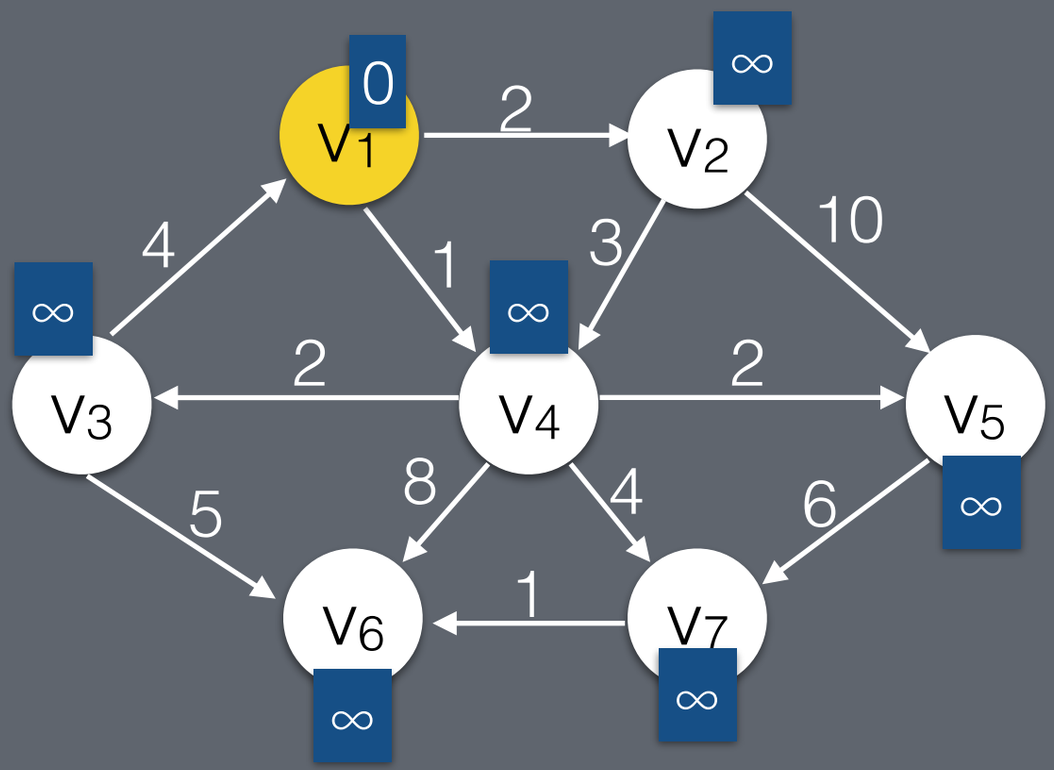
- Cost annotations for each vertex reflect the lowest cost *using only vertices visited so far.*
  - There might be a lower-cost path through other vertices that have not been seen yet.
- Keep nodes on a **priority queue** and always expand the vertex with the lowest cost annotation first! ← This is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

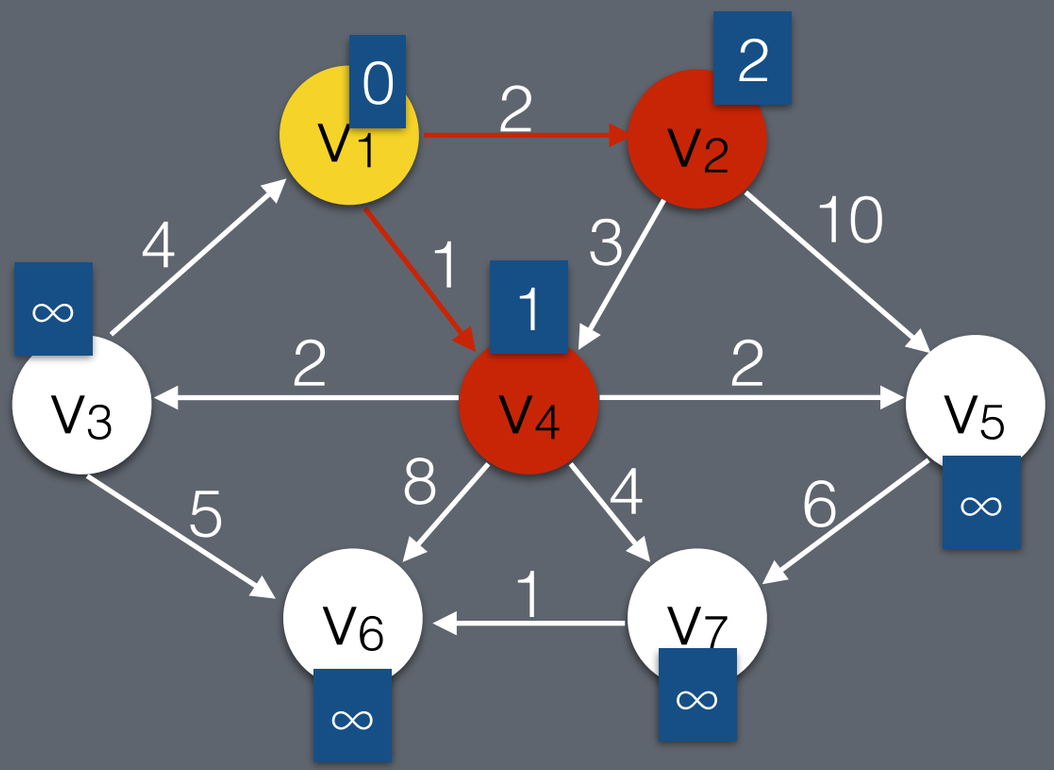



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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

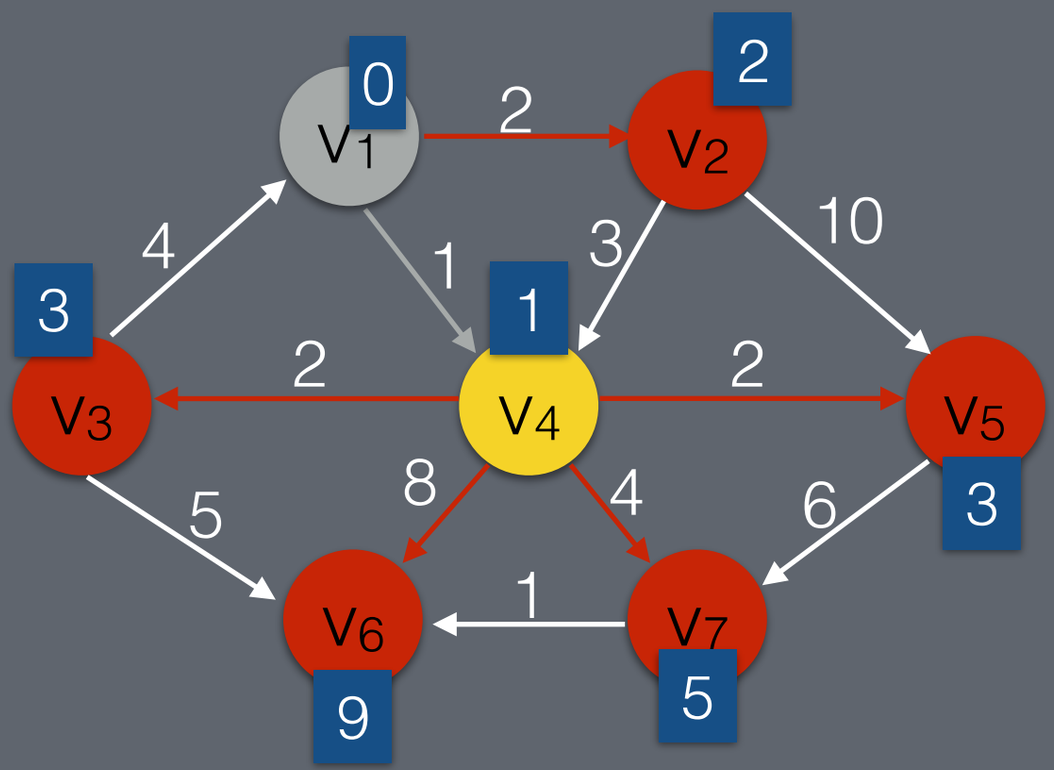



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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

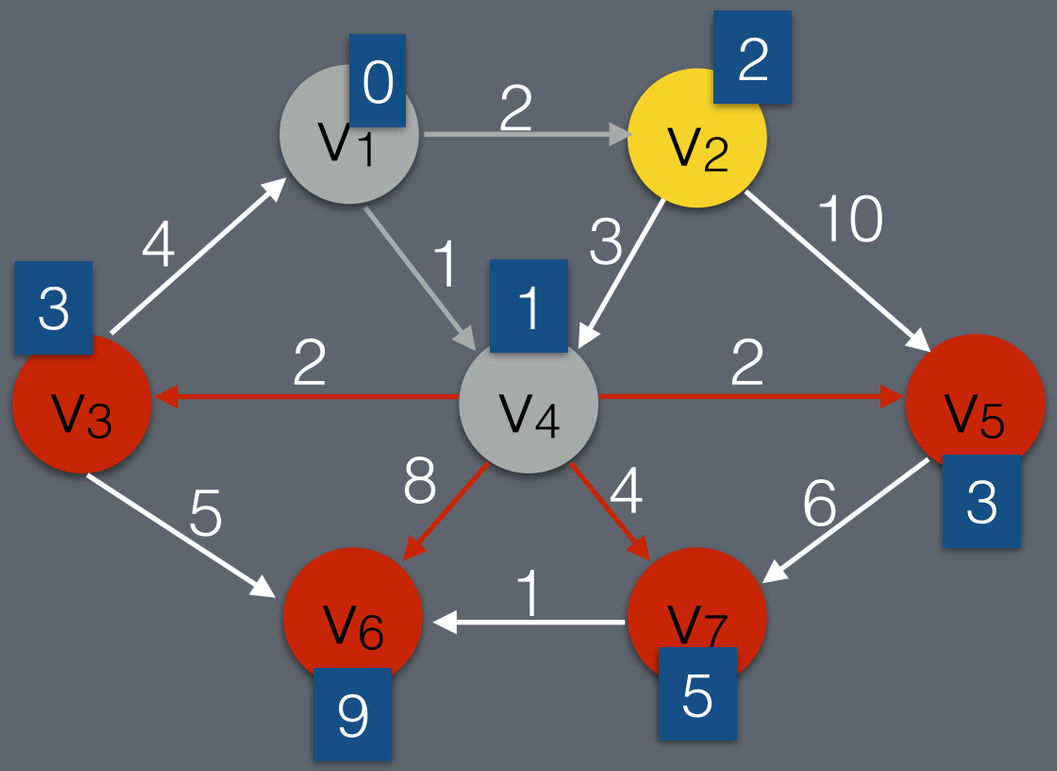



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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

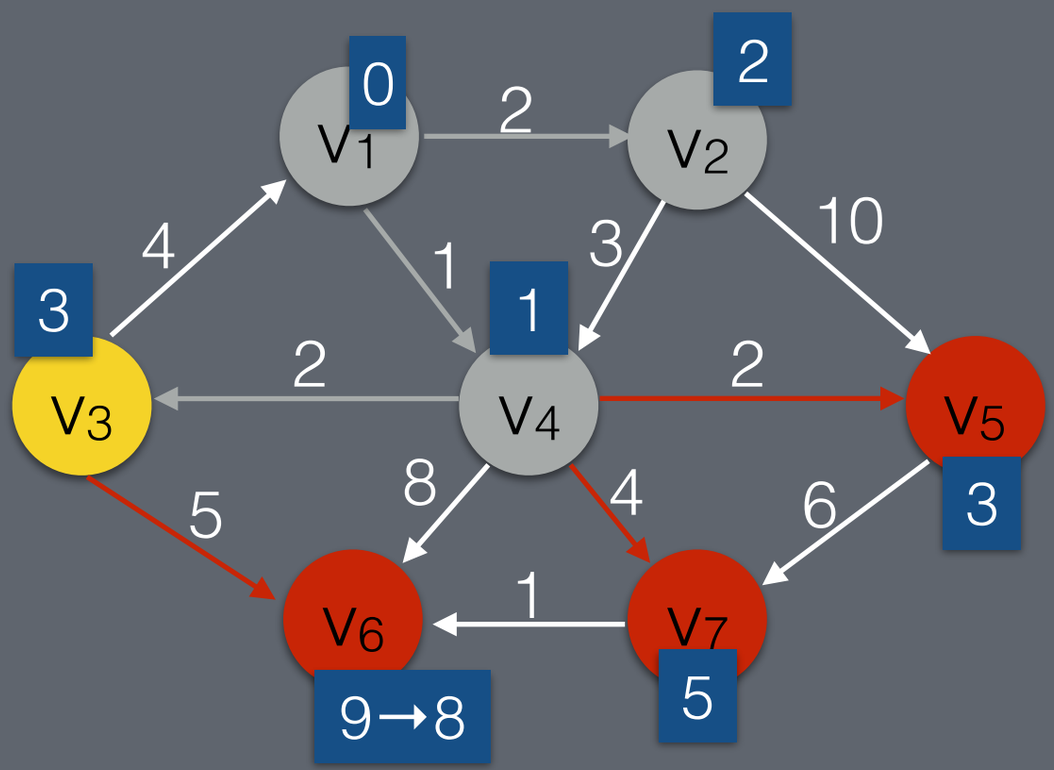



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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

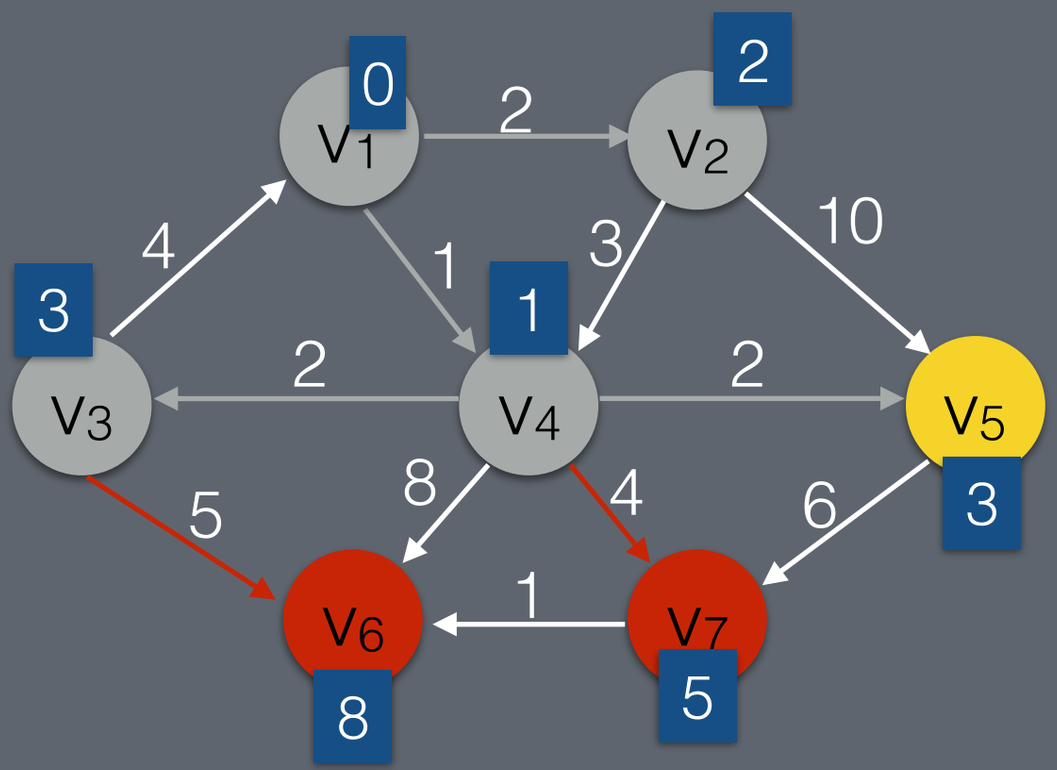



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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

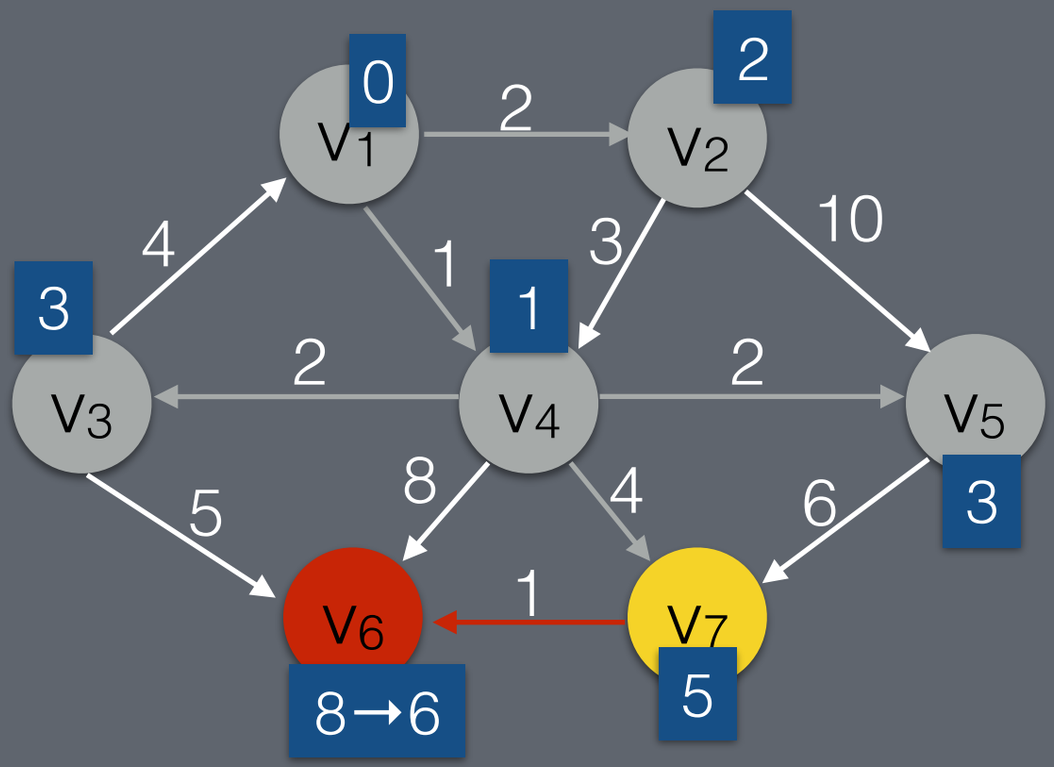



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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

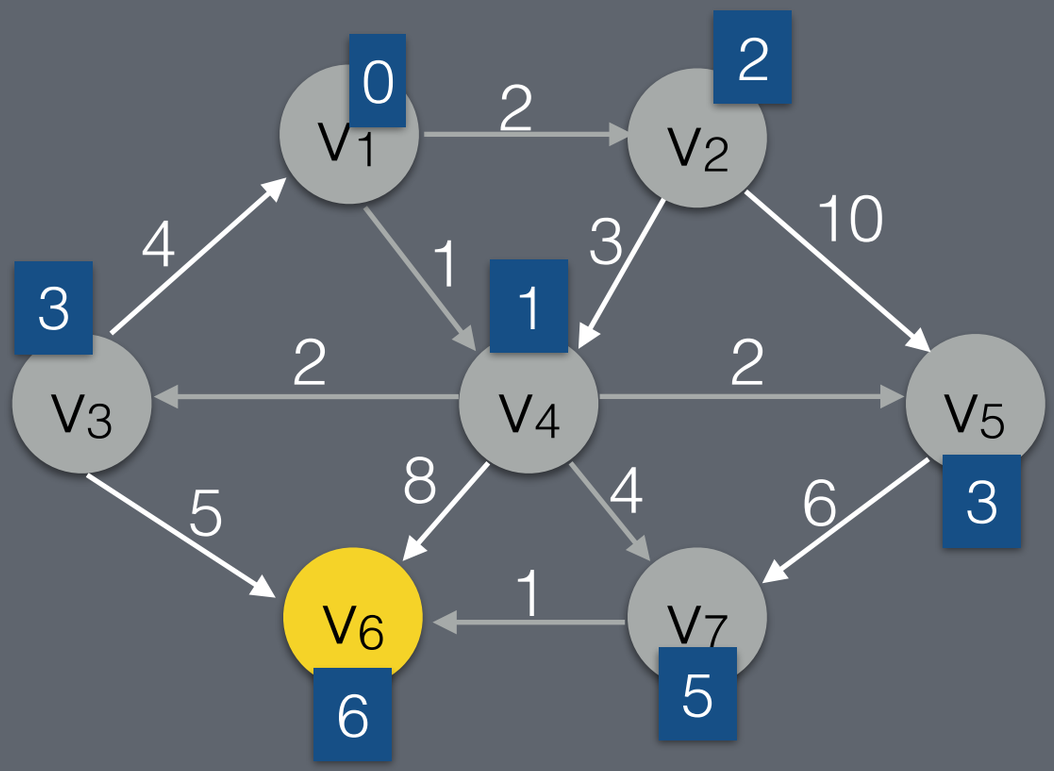



```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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



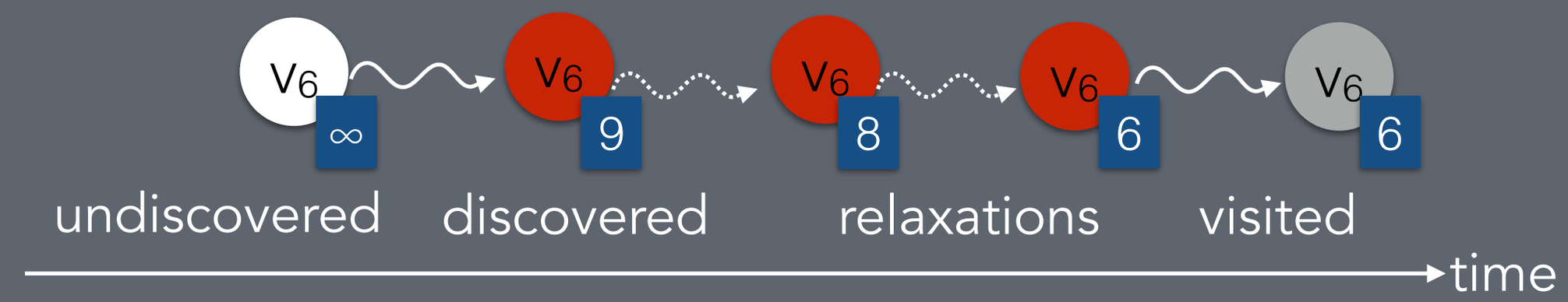

```
for all v:
  v.cost = ∞
  v.visited = false
  v.prev = null
start.cost = 0

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|)

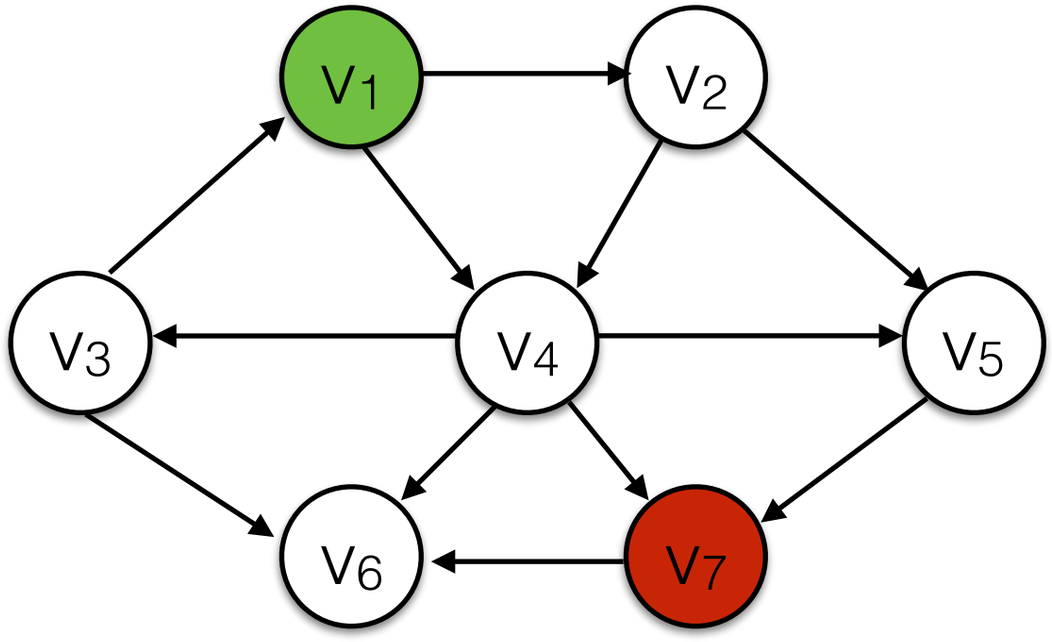
$$O(|E| \log |E|) = O(|E| \log |V|)$$

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

A General View of Graph Search

Goals:

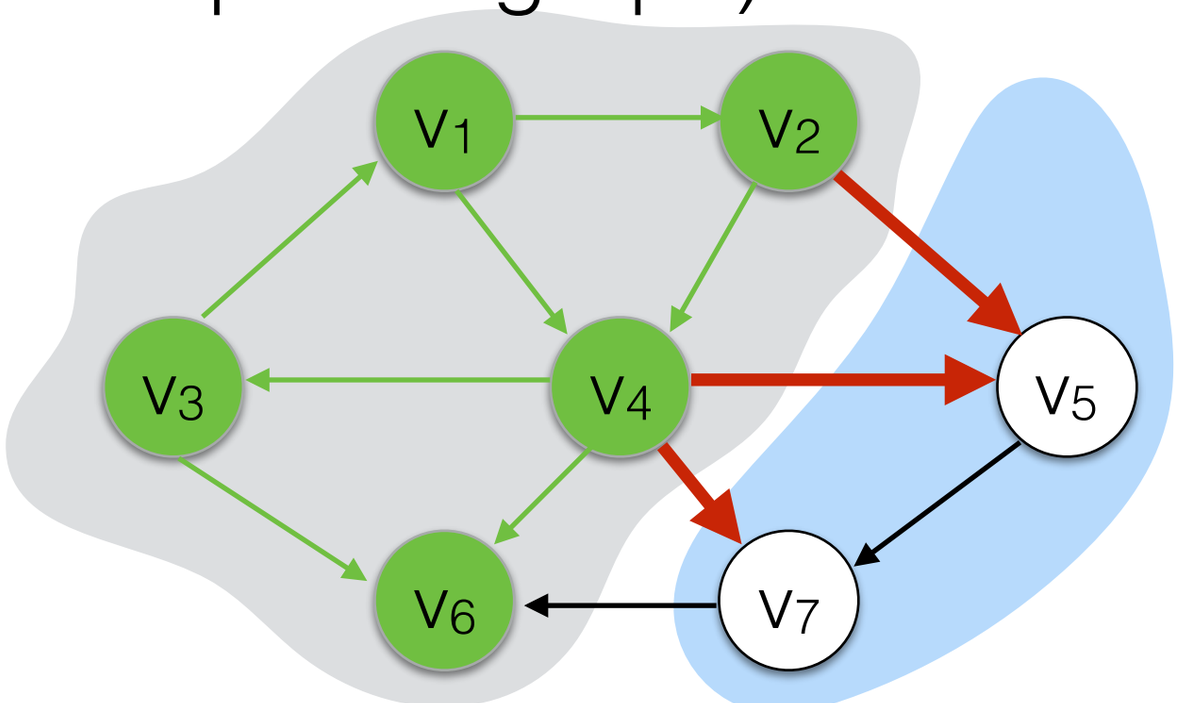
- Explore the graph systematically starting at *s* to
  - Find a vertex *t* / Find a path from *s* to *t*.
  - Find the shortest path from *s* to all vertices.
- ...



A General View of Graph Search

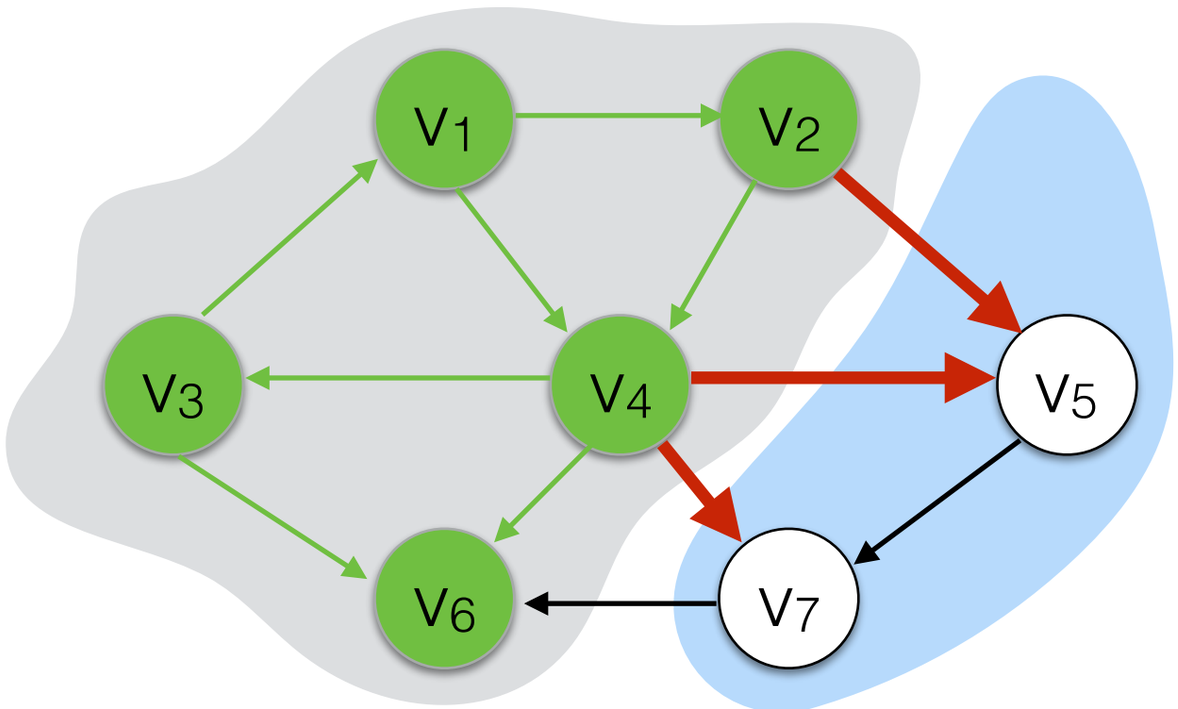
In every step of the search we maintain

- 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)

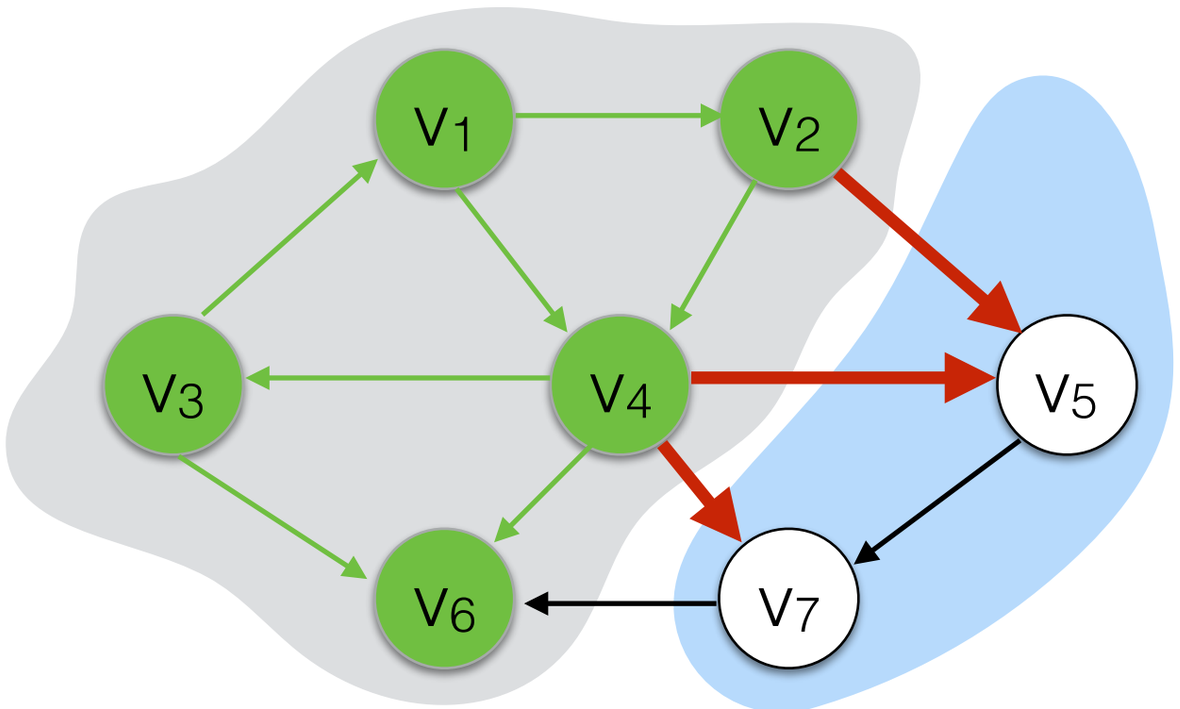
A General View of Graph Search



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:

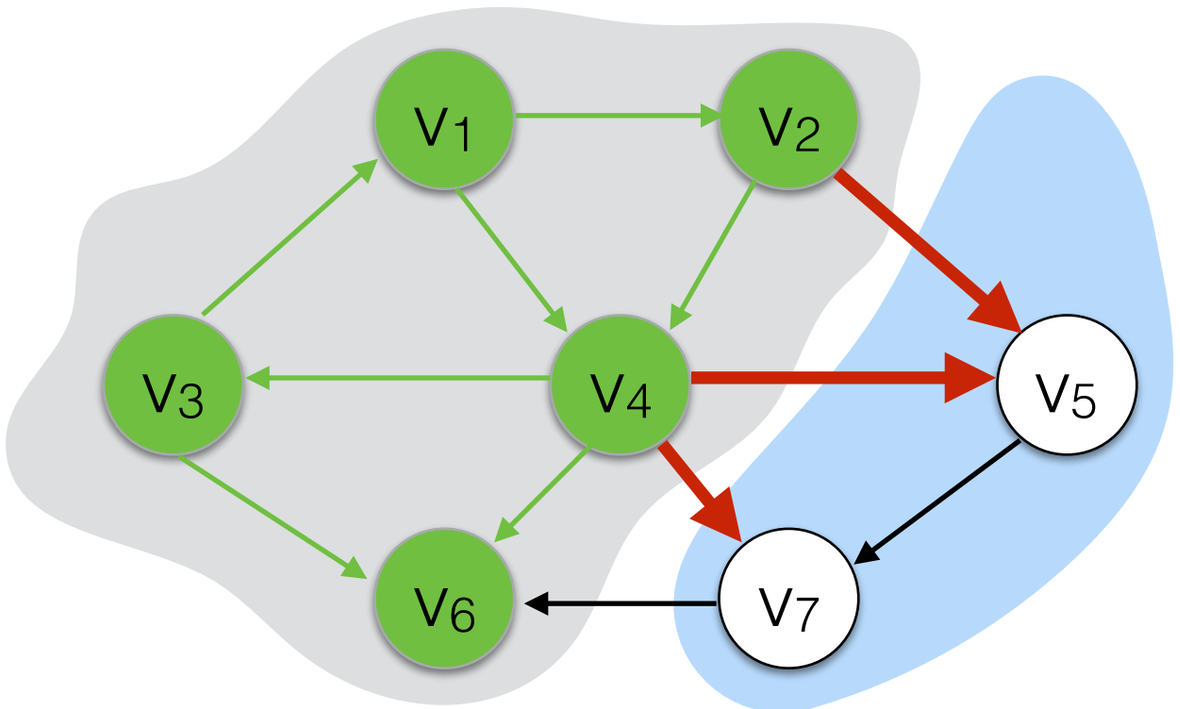


Agenda: (v2,v5), (v4,v5), (v4,v7)

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.

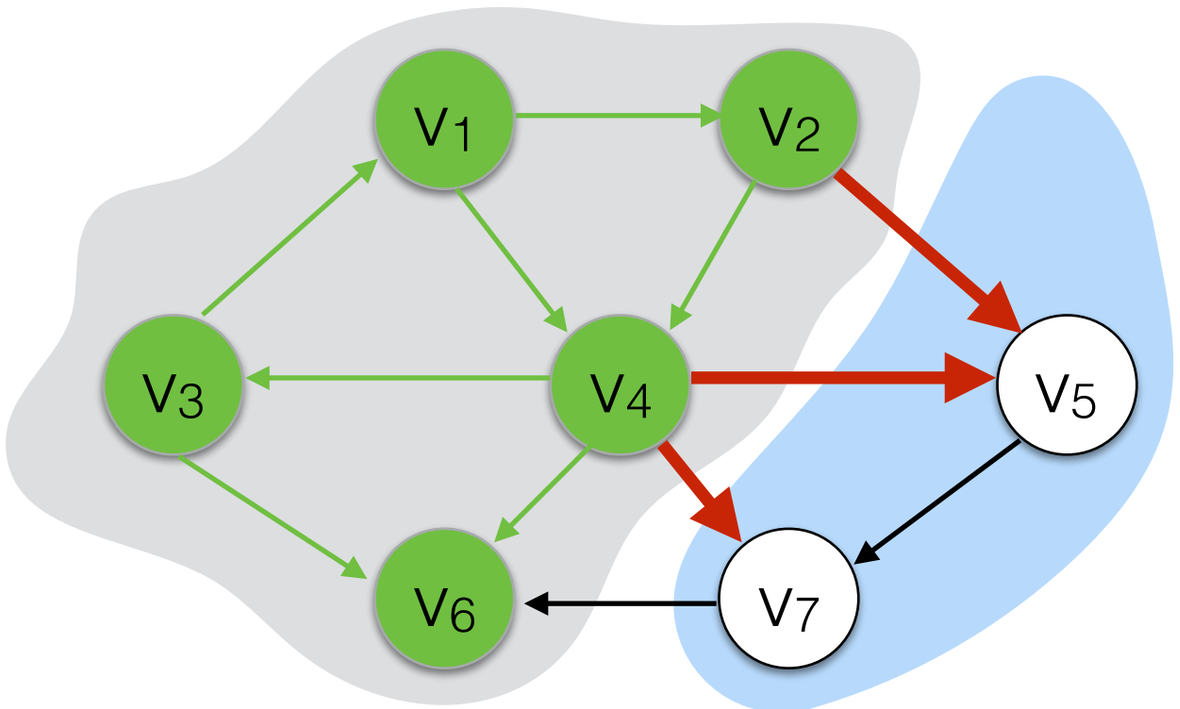


Agenda: (v2,v5), (v4,v5), (v4,v7)

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.

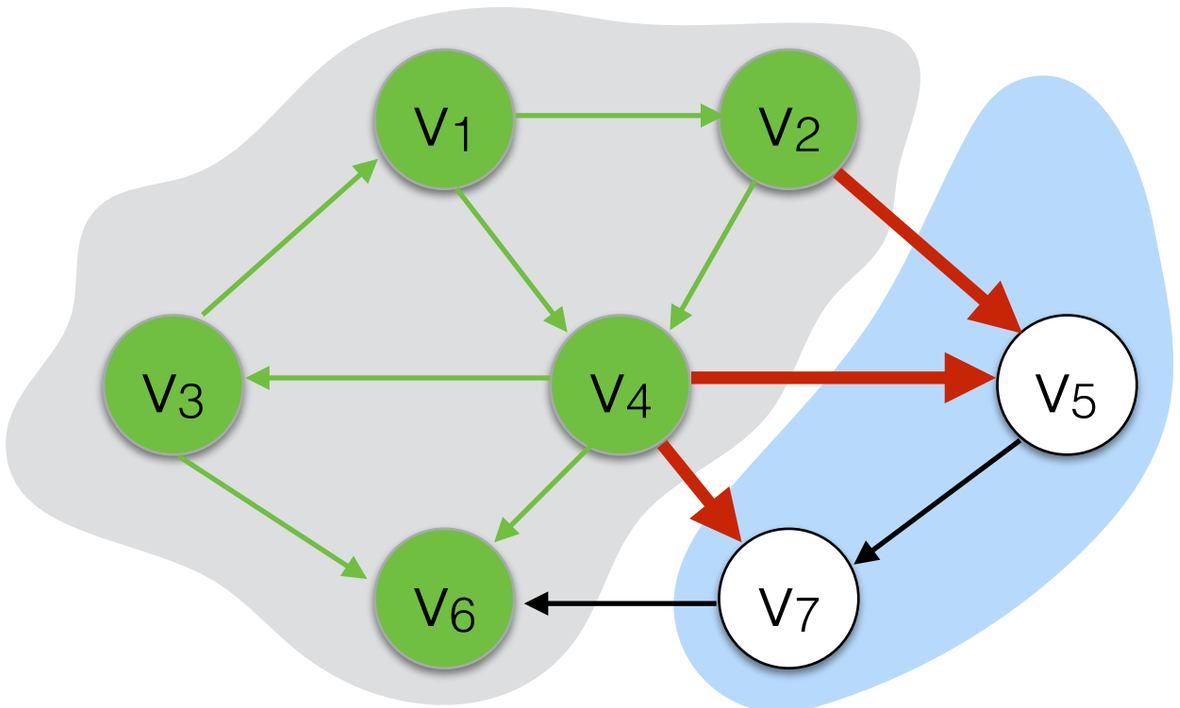


Agenda: (v2,v5), (v4,v5), (v4,v7)

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)