Shortest-Path Algorithms

Shortest-Path Algorithms

- The input is a weighted graph
  - associated with each edge (v_i, v_j) is a cost $c_{i,j}$
- The cost of a path is the sum of the weights of the edges, the ***weighted path length***
- The ***unweighted path length*** is only the number of edges on the path

- ***Single-Source Shortest-Path Problem***: Given a weighted graph G = (V, E), and a distinguished starting vertex s, find the shortest weighted path from s to every other vertex in G.
  - *"The shortest path from where I am to everywhere else"*

Shortest-Path Algorithms

- An edge has a weight (cost) associated with it
  - Could have negative weights; really? yes, really!

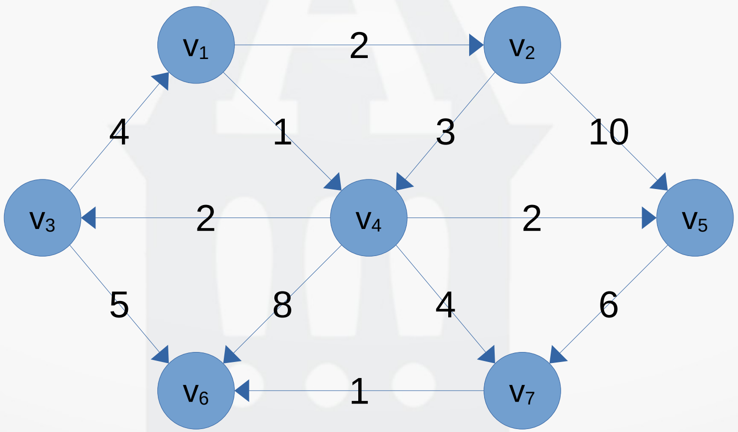
Shortest-Path Algorithms

- An edge has a weight (cost) associated with it
  - Could have negative weights; really? yes, really!

- Positive cost: $ to traverse (gas or electricity)
- Negative cost: $ of profit made from selling something along that path

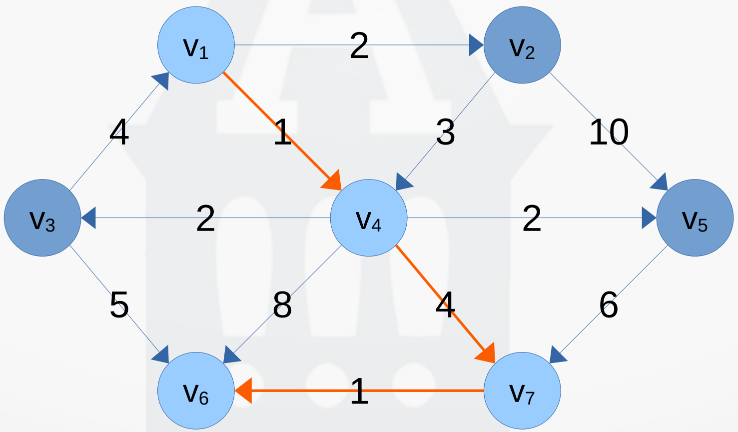
Shortest-Path Algorithms

- Here is our weighted graph



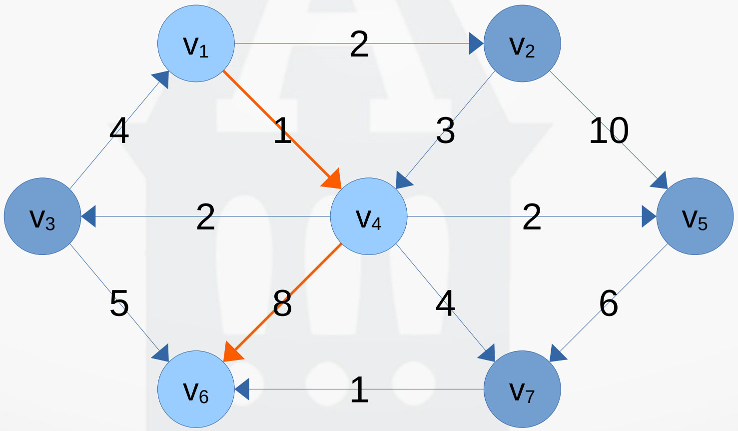
Shortest-Path Algorithms

- The shortest weighted path from V_1 to V_6 has a cost of 6, with 3 edges
  - Path: v_1, v_4, v_7, v_6



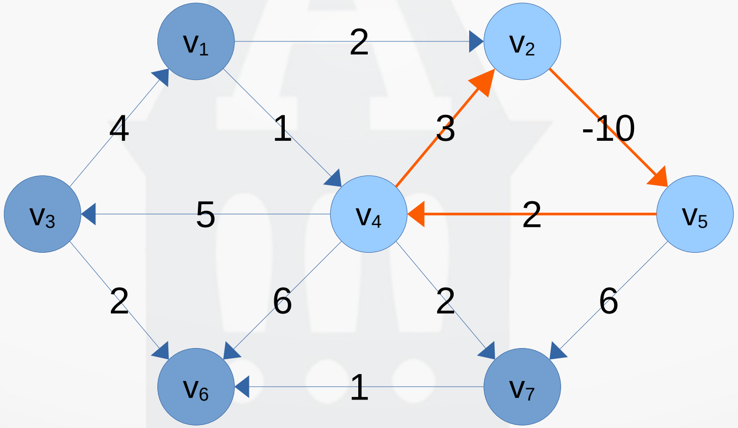
Shortest-Path Algorithms

- The shortest unweighted path has 2 edges
  - Path: v_1, v_4, v_6



Shortest-Path Algorithms – Negative Cost Cycles

- Between v_2 and v_5 there is a negative cost of -10
- The path from v_5 to v_4 has a cost of 2
- But a *shorter* path exists by the following loop: v_5, v_4, v_2, v_5, v_4; cost of -3
  - But could stay in the loop arbitrarily long, continually reducing the cost!



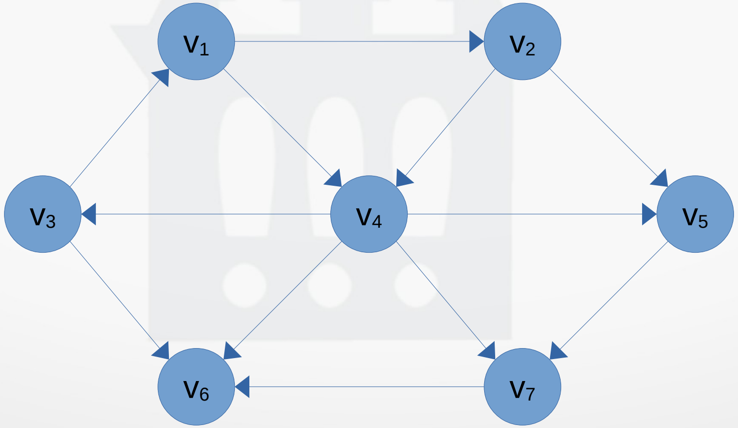
Shortest-Path Algorithms – Problems

- Four problems we'll look at...

1. Unweighted shortest path
2. Weighted shortest path without negative edges
3. Weighted shortest path with negative edges
4. Weighted shortest path of acyclic graphs

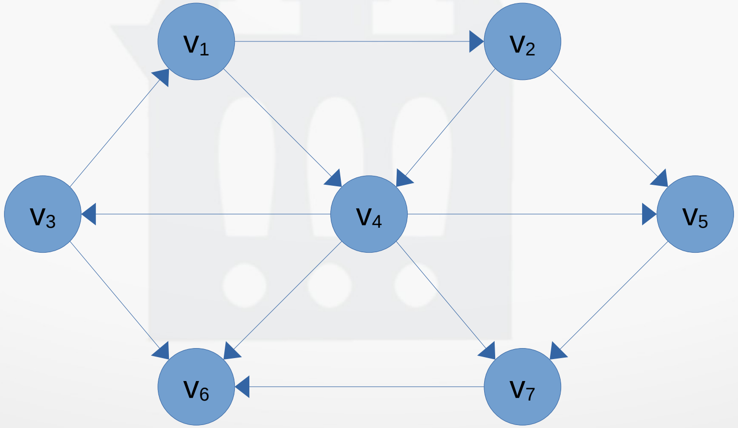
Unweighted Shortest Path

- Using some vertex s, which is an input parameter, find the shortest path from s to all other vertices in a unweighted graph
  - Assume s = v_3
  - For example, multiple ways to get to v_7; how do we do this?



Unweighted Shortest Path

- Using some vertex s, which is an input parameter, find the shortest path from s to all other vertices in a unweighted graph
  - Use a ***breadth-first search***; process nodes by distance, 1, 2, …



Unweighted Shortest Paths – Pseudocode

```
void unweighted(Vertex s) {
    Queue<Vertex> q = new Queue<Vertex>();
    for each Vertex v {
        v.dist = INFINITY;
    }
    s.dist = 0;
    q.enqueue(s);
    while (!q.isEmpty()) {
        Vertex v = q.dequeue();
        for each Vertex w adjacent to v {
            if (w.dist == INFINITY) {  // Hasn't been visited.
                w.dist = v.dist + 1;
                w.predecessor = v;   // track the shortest path back
                q.enqueue(w);
            }
        }
    }
}
```

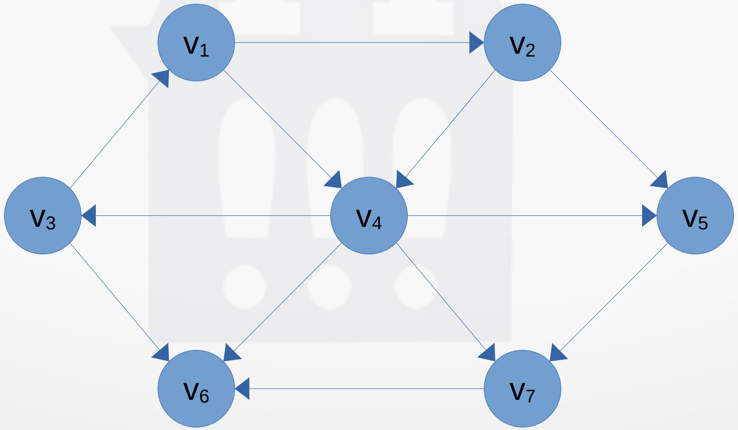
Unweighted Shortest Path – Example

- Set all distances to infinity

queue

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
inf	inf	inf	inf	inf	inf	inf



Unweighted Shortest Path – Example

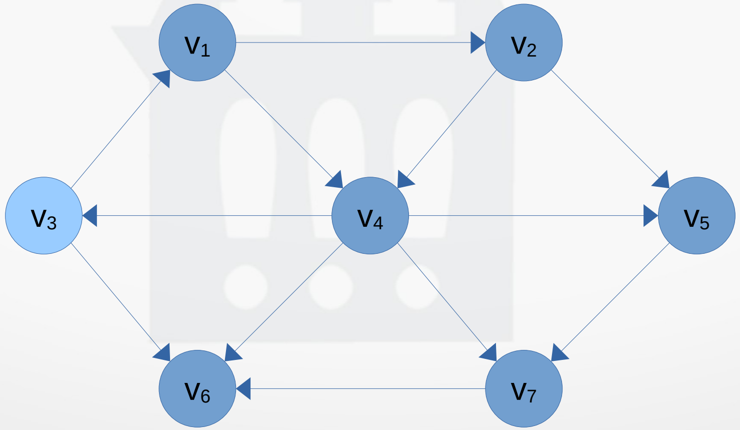
- Start at v_3
  - Set its distance to 0

queue

v_3										

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
inf	inf	0	inf	inf	inf	inf



Unweighted Shortest Path – Example

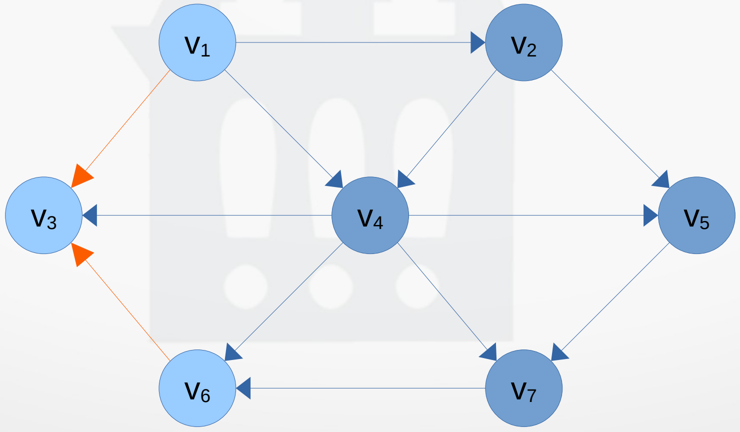
- Dequeue front: v_3
- Add adjacent nodes: v_1, v_6
  - add 1 to distance
  - set v_3 as predecessor

queue

v_1	v_6									

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	inf	0	inf	inf	1	inf



Unweighted Shortest Path – Example

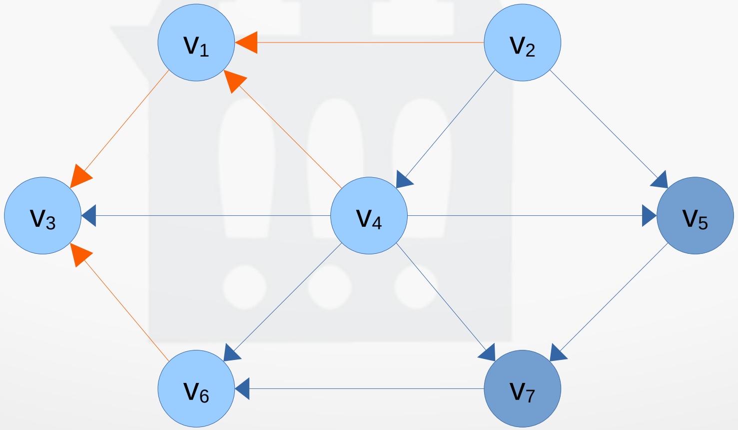
- Dequeue front: v_1
- Add adjacent nodes: v_2, v_4
  - add 1 to distance
  - set v_1 as predecessor

queue

v_6	v_2	v_4								

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	2	0	2	inf	1	inf



Unweighted Shortest Path – Example

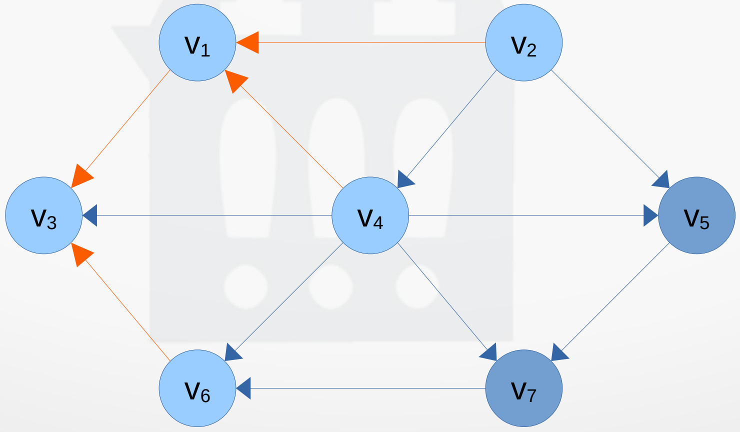
- Dequeue front: v_6
- Add adjacent nodes: none

queue

v_2	v_4									

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	2	0	2	inf	1	inf



Unweighted Shortest Path – Example

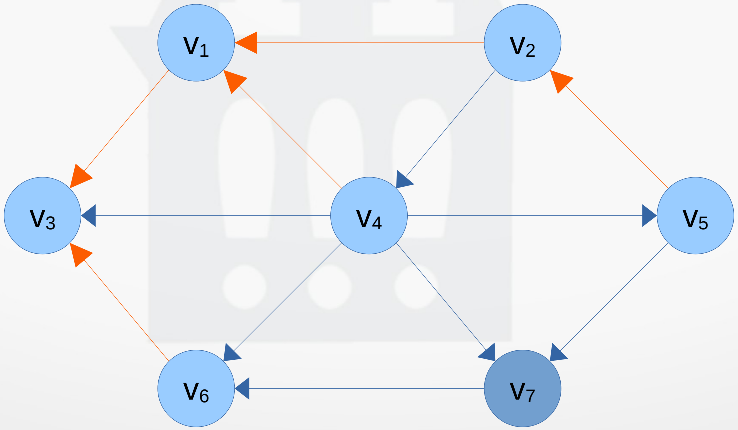
- Dequeue front: v_2
- Add adjacent nodes: v_5
  - add 1 to distance
  - set v_2 as predecessor

queue

v_4	v_5									

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	2	0	2	3	1	inf



Unweighted Shortest Path – Example

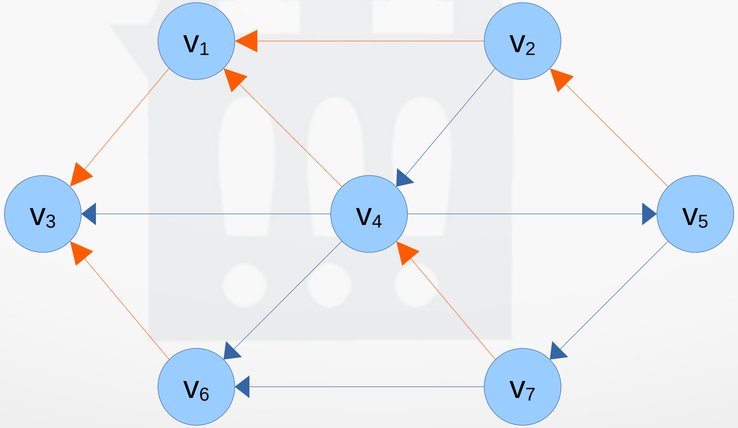
- Dequeue front: v_4
- Add adjacent nodes: v_7
  - add 1 to distance
  - set v_4 as predecessor

queue

v_5	v_7									

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
1	2	0	2	3	1	3



What if the edges have weights?

Weighted Shortest-Path – Dijkstra's Algorithm

- Very famous, don't forget this or you lose your CS card!

- Add the edge weight from the node to its successor to the current distance
- Pull nodes from a priority queue, based on shortest distance

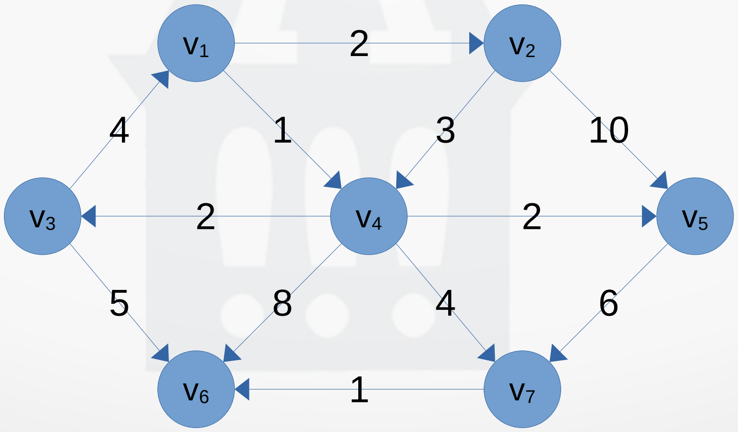
- ***known vertices*** are those for which the shortest path has been determined
- The initial distance d_v is tentative. It is the shortest path length from s to v using only *known vertices*
- It is a *greedy algorithm*: proceeds in stages doing the best at each stage
- Select a vertex v with smallest d_v among all unknown vertices and declare it as known
  - Remainder of the stage consists of updating the values d_w for all edges (v, w)

Dijkstra’s Algorithm – Example

priority queue

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
inf	inf	inf	inf	inf	inf	inf



Dijkstra's Algorithm – Example

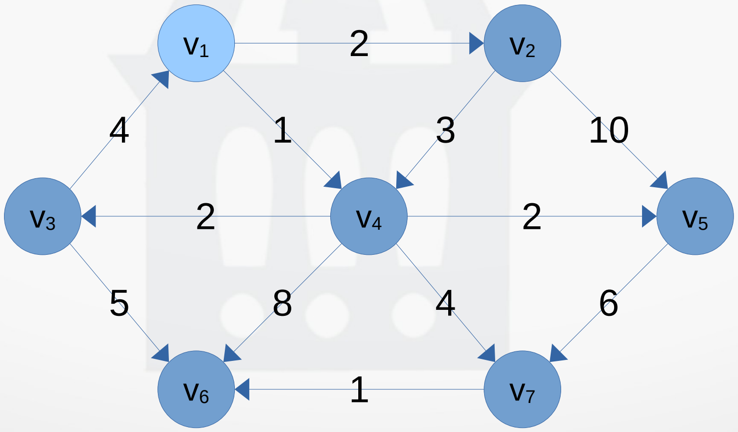
- Start at v_1
  - set it to known
- inspect/add adjacent: v_2, v_4
  - update distance

priority queue

v_4	v_2									

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
0	2	inf	1	inf	inf	inf



Dijkstra's Algorithm – Example

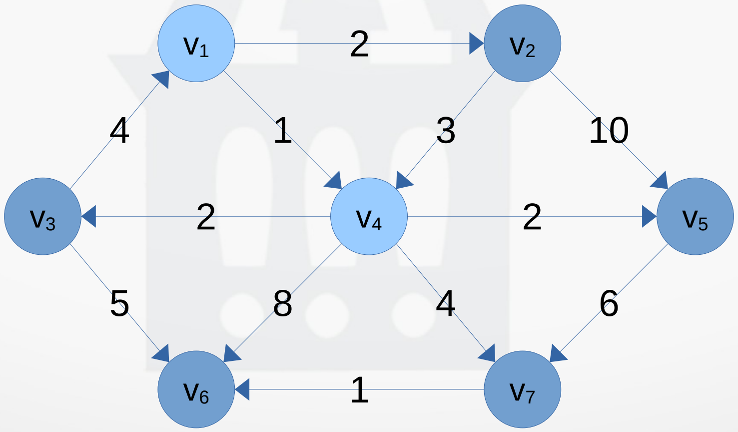
- Dequeue: v_4
  - set it to known
- inspect/add adjacent: V_3, V_5, V_6, V_7
  - update distances

priority queue

V_2	V_3	V_5	V_7	V_6						

distance

V_1	V_2	V_3	V_4	V_5	V_6	V_7
0	2	3	**1**	3	9	5



Dijkstra's Algorithm – Example

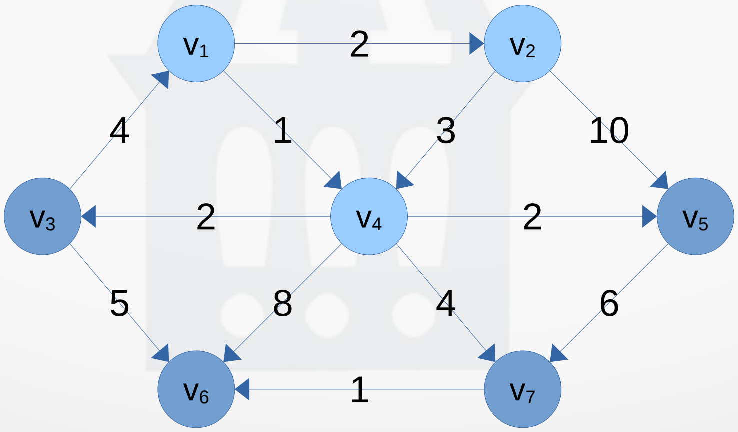
- Dequeue: v_2
  - set it to known
- inspect/add adjacent: v_4, v_5
  - update distance: v_5
    - no change

priority queue

v_3	v_5	v_7	v_6							

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
0	**2**	3	**1**	3	9	5



Dijkstra's Algorithm – Example

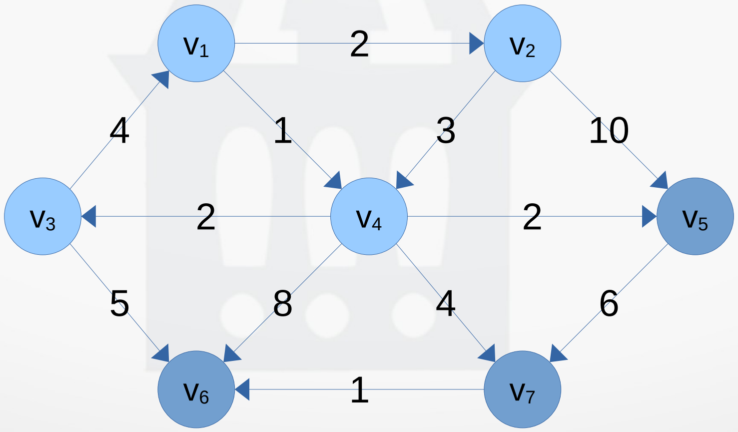
- Dequeue: v_3
  - set it to known
- inspect/add adjacent: v_1, v_6
  - update distance: v_6
    - from 9 to 8

priority queue

v_5	v_7	v_6								

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
0	**2**	**3**	**1**	3	8	5



Dijkstra's Algorithm – Example

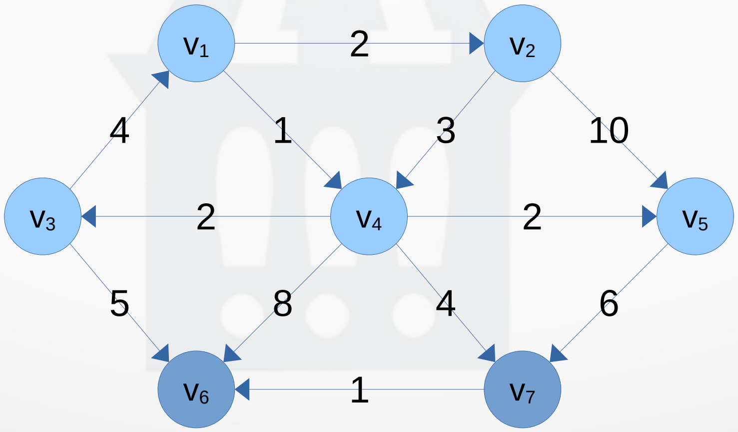
- Dequeue: v_5
  - set it to known
- inspect/add adjacent: v_7
  - update distance: v_7
    - no change

priority queue

v_7	v_6									

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
0	**2**	**3**	**1**	**3**	8	5



Dijkstra's Algorithm – Example

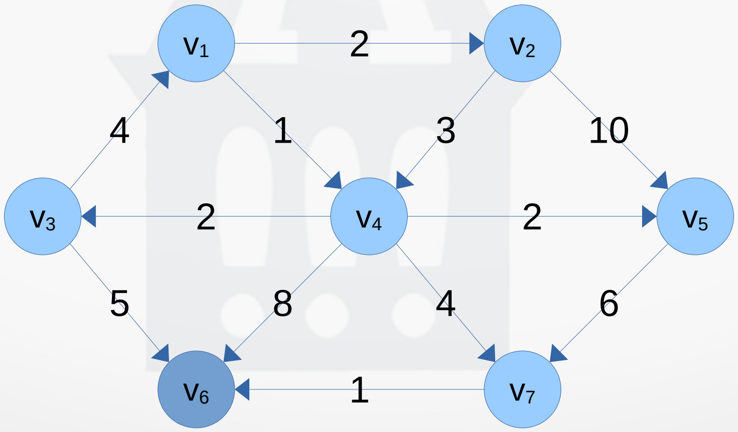
- Dequeue: v_7
  - set it to known
- inspect/add adjacent: v_6
  - update distance: v_6
    - from 8 to 6

priority queue

v_6										

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
0	**2**	**3**	**1**	**3**	6	**5**



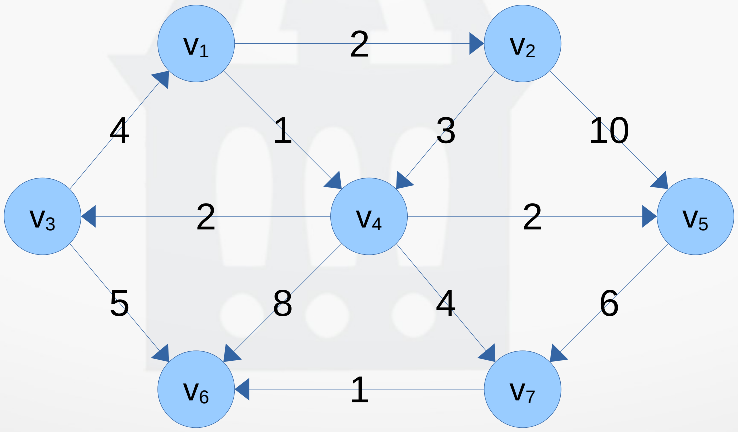
Dijkstra's Algorithm – Example

- Dequeue: v_6
  - set it to known
- no more unknown: done!

priority queue

distance

v_1	v_2	v_3	v_4	v_5	v_6	v_7
0	**2**	**3**	**1**	**3**	**6**	**5**



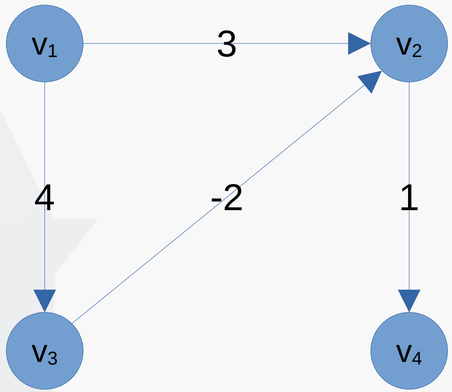
Dijkstra’s – Pseudocode

```
void dijkstra(GraphNode s) {
    for each GraphNode v {
        v.dist = INFINITY;
        v.known = false;
    }
    s.dist = 0;
    while (there is an unknown distance vertex) {
        GraphNode v = unknown vertex with smallest distance
        v.known = true;  // We know we'll never find a better path
        for each GraphNode w adjacent to v {
            if (!w.known) {
                cost = cost of edge from v to w;
                if (v.dist + cost < w.dist) {
                    w.dist =  v.dist + cost;
                    w.path = v.id;
                }
            }
        }
    }
}
```

```
class GraphNode {
    public int id;
    public List adj; // Adjacency list
    public boolean known;
    public int dist;
    public int path;
}
```

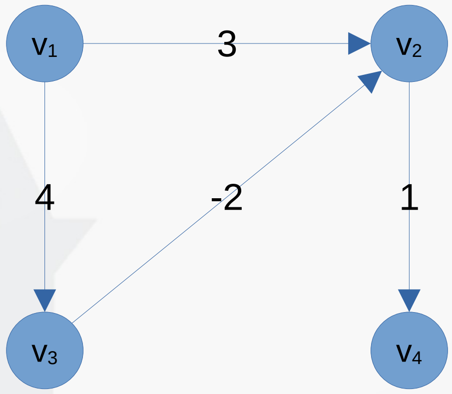
Graphs – Negative Edge Costs

- Dijkstra’s algorithm doesn’t work with negative edge costs
- Why?



Graphs – Negative Edge Costs

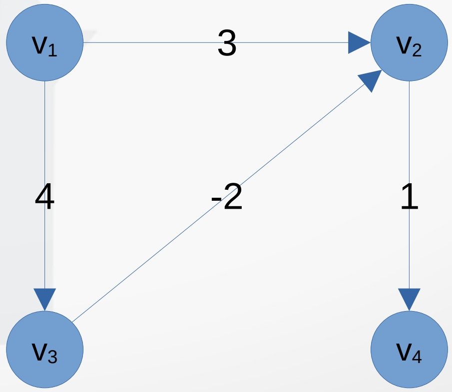
- Dijkstra’s algorithm doesn’t work with negative edge costs
- Why?
  - Start at v_1
  - Go to v_2 in 3: add to queue
  - Go to v_3 in 4: add to queue
  - v_2 is smallest, call it known at 3
  - From v_2 to v_4 in 1: add to queue
  - v_4/v_3 tie, so remove v_3
    - Now, see we can get to v_2 (from v_1) in 2
    - But Dijkstra’s said v_2 was already known!! (oops)



Graphs – Negative Edge Costs

- How to solve?

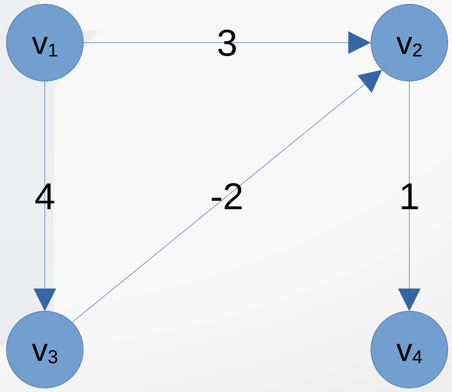
- Add a large constant to all edges? In our example, add 3, to keep everything positive
  - Why doesn't this work?



Graphs – Negative Edge Costs

- How to solve?

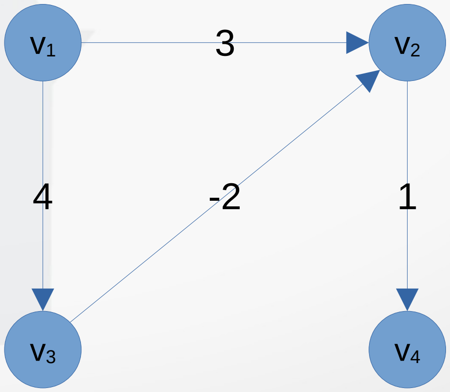
- Add a large constant to all edges? In our example, add 3, to keep everything positive
  - Why doesn't this work?
  - Not obvious, but the cost of a path is now: (# of edges in path) * (added constant)
  - That is biased against paths with more edges



Graphs – Negative Edge Costs

- How to solve?

- Modify Dijkstra's algorithm
  - Forget about the concept of a known vertex
  - Use a regular queue, don't need/want a priority queue anymore (breadth-first search now)
  - Place starting vertex on the queue
  - Examine length to all successors
    - For any successors whose distance has decreased, change their distance and place that node on the queue again
    - If a node has been checked |V| times, don't add it back in the queue...prevents infinite loops due to negative edge costs
  - Repeat until no more nodes on the queue

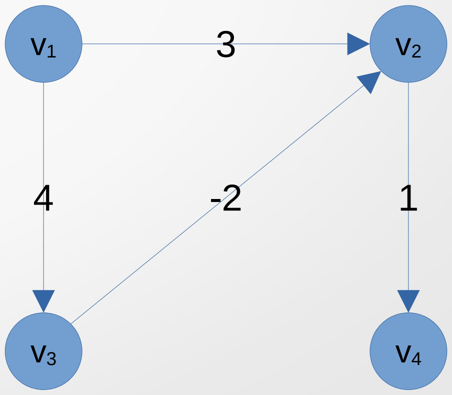


Modified Dijkstra's Algorithm – Negative Edge Costs

FIFO queue

distance

v_1	v_2	v_3	v_4
inf	inf	inf	inf



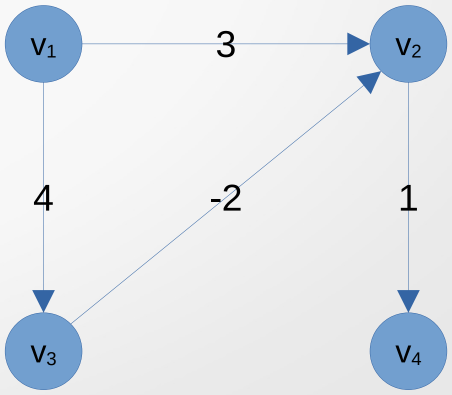
Modified Dijkstra's Algorithm – Negative Edge Costs

FIFO queue

v_1										

distance

v_1	v_2	v_3	v_4
0	inf	inf	inf



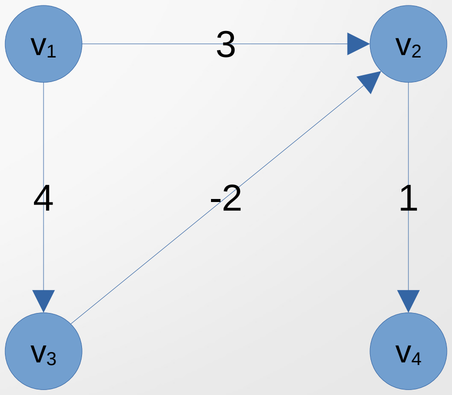
Modified Dijkstra's Algorithm – Negative Edge Costs

FIFO queue

v_2	v_3									

distance

v_1	v_2	v_3	v_4
0	3	4	inf



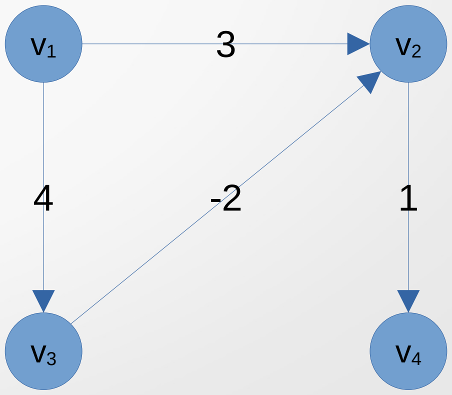
Modified Dijkstra's Algorithm – Negative Edge Costs

FIFO queue

v_3	v_4									

distance

v_1	v_2	v_3	v_4
0	3	4	4



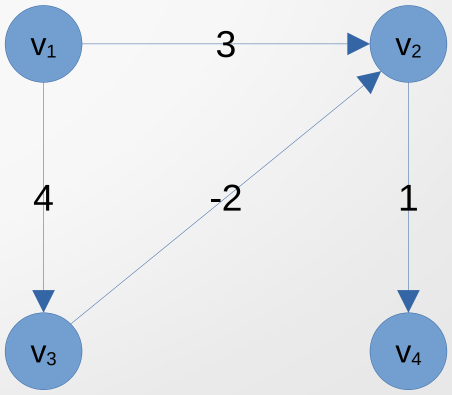
Modified Dijkstra's Algorithm – Negative Edge Costs

FIFO queue

v_4	v_2									

distance

v_1	v_2	v_3	v_4
0	2	4	4



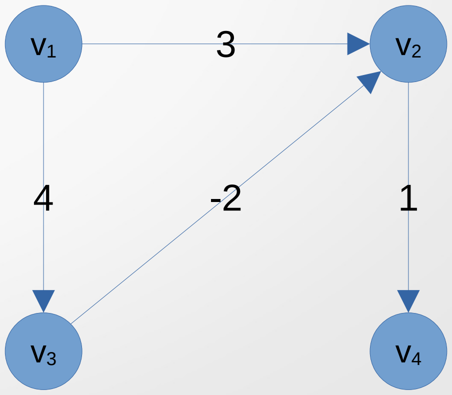
Modified Dijkstra's Algorithm – Negative Edge Costs

FIFO queue

v_2										

distance

v_1	v_2	v_3	v_4
0	2	4	4



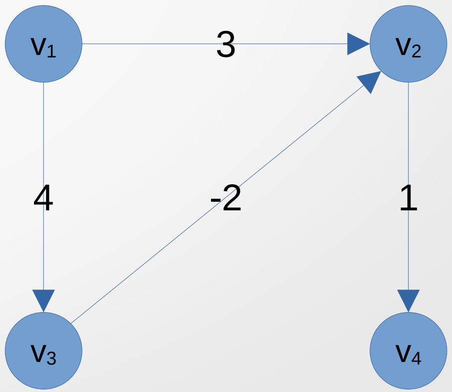
Modified Dijkstra's Algorithm – Negative Edge Costs

FIFO queue

v_4										

distance

v_1	v_2	v_3	v_4
0	2	4	**3**

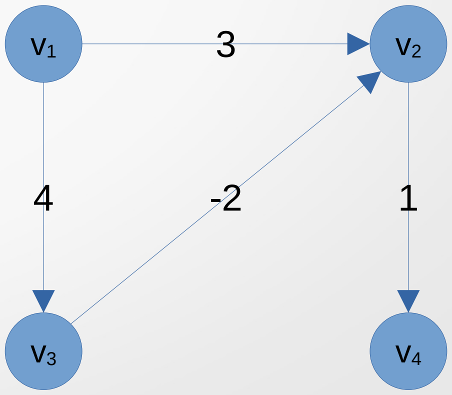


Modified Dijkstra’s Algorithm – Negative Edge Costs

FIFO queue

distance

v_1	v_2	v_3	v_4
0	2	4	3



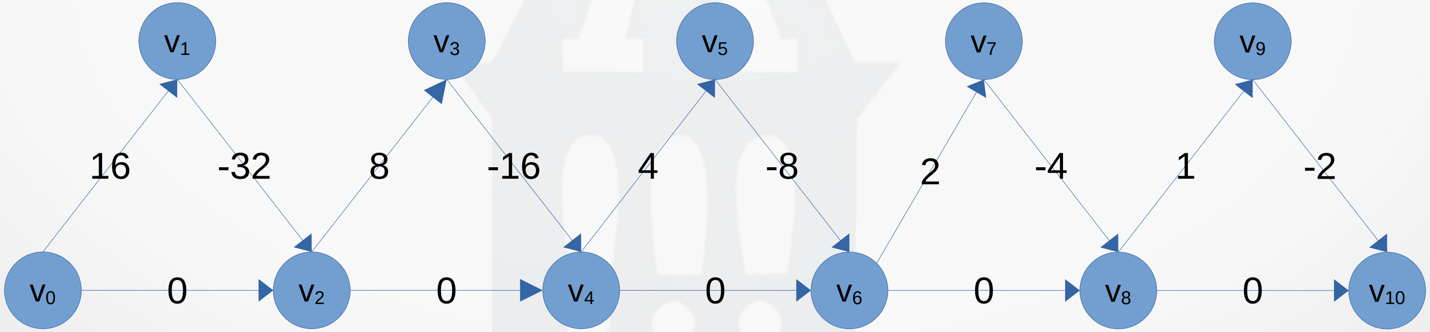
Acyclic Graphs – Negative Edge Costs

- What if the graph is known to be acyclic?

- Consider the nodes in topological order; as all predecessors are final
- Running time = $O(|V| + |E|)$
- This works because when a vertex is selected, its distance can no longer be lowered, because by topological ordering rule, it has no incoming edges emanating from unknown nodes

Acyclic Graphs – Negative Edge Costs

- Trace this following (acyclic) graph for…
  - Modified Dijkstra's algorithm
  - Topological ordering



Shortest Path

- What if we need all pairs shortest path?

- Could use single-source shortest path repeatedly, starting over with each vertex
- Dijkstra’s shortest path is O(E + V log(V))
- Doing it n times is $O(n^3 \log(n))$ (after multiplying out and re-arranging terms)