Single Source Shortest Paths with Negative Weights

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 5

Dijkstra's algorithm

- Recall the burning pipeline analogy

Dijkstra's algorithm

- Recall the burning pipeline analogy
- We keep track of the following
  - The vertices that have been burnt
  - The expected burn time of vertices

Dijkstra's algorithm

- Recall the burning pipeline analogy
- We keep track of the following
  - The vertices that have been burnt
  - The expected burn time of vertices
- Initially
  - No vertex is burnt
  - Expected burn time of source vertex is 0
  - Expected burn time of rest is ∞

Initialization (assume source vertex 0)

- $B(i) = \text{False}$, for $0 \leq i < n$
  - $UB = \{k \mid B(k) = \text{False}\}$
- $EBT(i) = \begin{cases} 0, & \text{if } i = 0 \\ \infty, & \text{otherwise} \end{cases}$

Dijkstra's algorithm

- Recall the burning pipeline analogy
- We keep track of the following
  - The vertices that have been burnt
  - The expected burn time of vertices
- Initially
  - No vertex is burnt
  - Expected burn time of source vertex is 0
  - Expected burn time of rest is ∞
- While there are vertices yet to burn
  - Pick unburnt vertex with minimum expected burn time, mark it as burnt
  - Update the expected burn time of its neighbours

Initialization (assume source vertex 0)

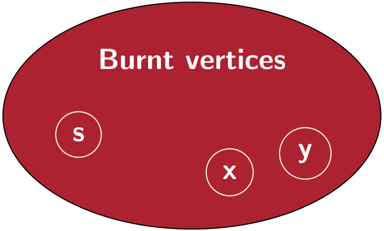
- $B(i) = \text{False}$, for $0 \leq i < n$
  - $UB = \{k \mid B(k) = \text{False}\}$
- $EBT(i) = \begin{cases} 0, & \text{if } i = 0 \\ \infty, & \text{otherwise} \end{cases}$

Update, if $UB \neq \emptyset$

- Let $j \in UB$ such that $EBT(j) \leq EBT(k)$ for all $k \in UB$
- Update $B(j) = \text{True}$, $UB = UB \setminus \{j\}$
- For each $(j, k) \in E$ such that $k \in UB$, $EBT(k) = \min(\ EBT(k),\ EBT(j) + W(j, k)\ )$

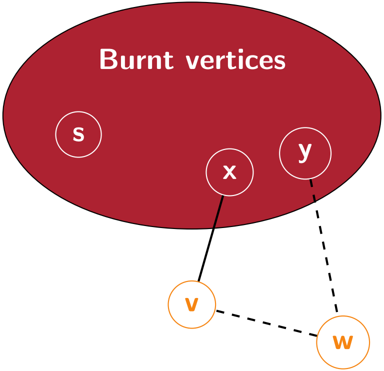
Correctness requires non-negative edge weights

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt



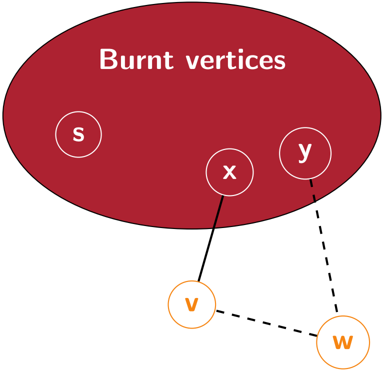
Correctness requires non-negative edge weights

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt
- Next vertex to burn is **v**, via **x**
- Cannot find a shorter path later from **y** to **v** via **w**
  - Burn time of **w** $\geq$ burn time of **v**
  - Edge from **w** to **v** has weight ≥ 0



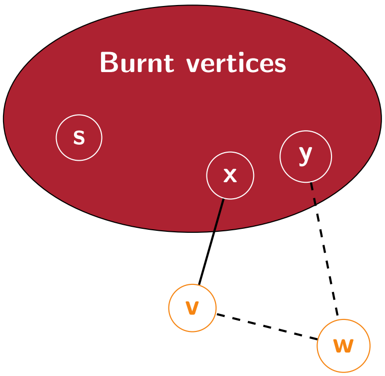
Correctness requires non-negative edge weights

- Each new shortest path we discover extends an earlier one
- By induction, assume we have found shortest paths to all vertices already burnt
- Next vertex to burn is **v**, via **x**
- Cannot find a shorter path later from **y** to **v** via **w**
  - Burn time of **w** $\geq$ burn time of **v**
  - Edge from **w** to **v** has weight ≥ 0
- This argument breaks down if edge (**w**,**v**) can have negative weight



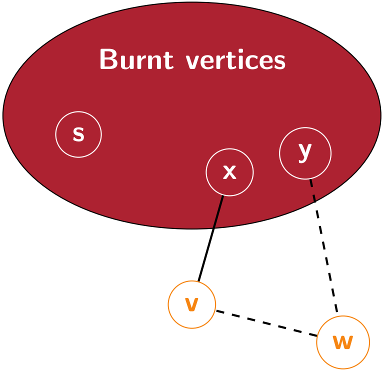
Extending to negative edge weights

- The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt



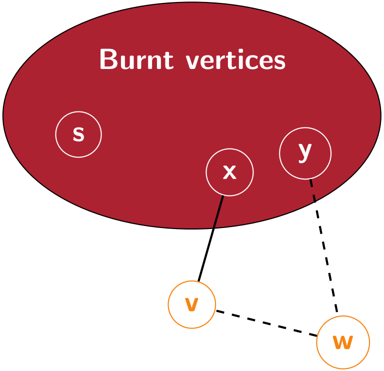
Extending to negative edge weights

- The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt
- What if we allow updates even after a vertex is burnt?



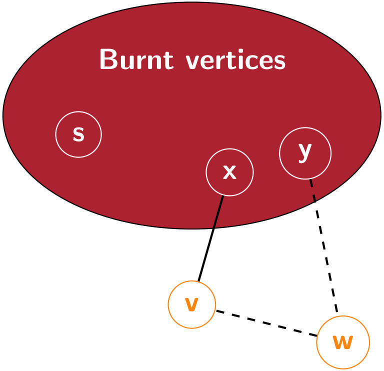
Extending to negative edge weights

- The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt
- What if we allow updates even after a vertex is burnt?
- Recall, negative edge weights are allowed, but no negative cycles



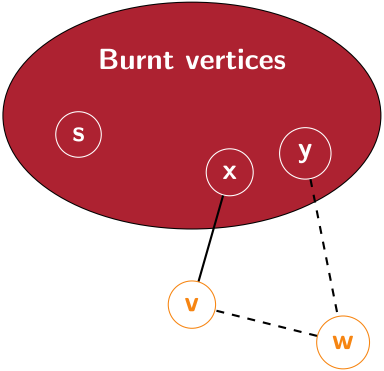
Extending to negative edge weights

- The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt
- What if we allow updates even after a vertex is burnt?
- Recall, negative edge weights are allowed, but no negative cycles
- Going around a cycle can only add to the length



Extending to negative edge weights

- The difficulty with negative edge weights is that we stop updating the burn time once a vertex is burnt
- What if we allow updates even after a vertex is burnt?
- Recall, negative edge weights are allowed, but no negative cycles
- Going around a cycle can only add to the length
- Shortest route to every vertex is a path, no loops



Extending to negative edge weights

- Suppose minimum weight path from 0 to k is

$$0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \cdots \xrightarrow{w_{\ell-1}} j_{\ell-1} \xrightarrow{w_\ell} k$$

  - Need not be minimum in terms of number of edges

Extending to negative edge weights

- Suppose minimum weight path from 0 to k is

  $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \cdots \xrightarrow{w_{\ell-1}} j_{\ell-1} \xrightarrow{w_\ell} k$

  - Need not be minimum in terms of number of edges

- Every prefix of this path must itself be a minimum weight path
  - $0 \xrightarrow{w_1} j_1$
  - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2$
  - ...
  - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \cdots \xrightarrow{w_{\ell-1}} j_{\ell-1}$

Extending to negative edge weights

- Suppose minimum weight path from 0 to k is

  $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \cdots \xrightarrow{w_{\ell-1}} j_{\ell-1} \xrightarrow{w_\ell} k$

  - Need not be minimum in terms of number of edges

- Every prefix of this path must itself be a minimum weight path
  - $0 \xrightarrow{w_1} j_1$
  - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2$
  - $\ldots$
  - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \cdots \xrightarrow{w_{\ell-1}} j_{\ell-1}$

- Once we discover shortest path to $j_{\ell-1}$, next update will fix shortest path to k

Extending to negative edge weights

- Suppose minimum weight path from 0 to k is

  $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \cdots \xrightarrow{w_{\ell-1}} j_{\ell-1} \xrightarrow{w_\ell} k$

  - Need not be minimum in terms of number of edges

- Every prefix of this path must itself be a minimum weight path
  - $0 \xrightarrow{w_1} j_1$
  - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2$
  - ...
  - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \cdots \xrightarrow{w_{\ell-1}} j_{\ell-1}$

- Once we discover shortest path to $j_{\ell-1}$, next update will fix shortest path to k

- Repeatedly update shortest distance to each vertex based on shortest distance to its neighbours
  - Update cannot push this distance below actual shortest distance

Extending to negative edge weights

- Suppose minimum weight path from 0 to k is

  $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \cdots \xrightarrow{w_{\ell-1}} j_{\ell-1} \xrightarrow{w_\ell} k$

  - Need not be minimum in terms of number of edges

- Every prefix of this path must itself be a minimum weight path
  - $0 \xrightarrow{w_1} j_1$
  - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2$
  - ...
  - $0 \xrightarrow{w_1} j_1 \xrightarrow{w_2} j_2 \xrightarrow{w_3} \cdots \xrightarrow{w_{\ell-1}} j_{\ell-1}$

- Once we discover shortest path to $j_{\ell-1}$, next update will fix shortest path to k

- Repeatedly update shortest distance to each vertex based on shortest distance to its neighbours
  - Update cannot push this distance below actual shortest distance

- After ℓ updates, all shortest paths using $\leq \ell$ edges have stabilized
  - Minimum weight path to any node has at most $n-1$ edges
  - After $n-1$ updates, all shortest paths have stabilized

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

Bellman-Ford Algorithm

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

Repeat $n{-}1$ times

- For each vertex $j \in \{0, 1, \ldots, n{-}1\}$, for each edge $(j, k) \in E$,

  $$D(k) = \min(D(k), D(j) + W(j, k))$$

Bellman-Ford Algorithm

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

Repeat $n{-}1$ times

- For each vertex $j \in \{0, 1, \ldots, n{-}1\}$,
  for each edge $(j, k) \in E$,
  $D(k) = \min(\ D(k),$
  $\qquad D(j) + W(j, k)\ )$

Works for directed and undirected graphs

Bellman-Ford Algorithm

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

Repeat $n{-}1$ times

- For each vertex $j \in \{0, 1, \ldots, n{-}1\}$, for each edge $(j, k) \in E$,
  $D(k) = \min(\ D(k),\ D(j) + W(j, k)\ )$

Works for directed and undirected graphs

```
def bellmanford(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  distance = {}
  for v in range(rows):
    distance[v] = infinity

  distance[s] = 0

  for i in range(rows):
    for u in range(rows):
      for v in range(cols):
        if WMat[u,v,0] == 1:
          distance[v] = min(distance[v],distance[u]
                                        +WMat[u,v,1])
  return(distance)
```

Bellman-Ford Algorithm

Initialization (source vertex 0)

- $D(j)$: minimum distance known so far to vertex j
- $D(j) = \begin{cases} 0, & \text{if } j = 0 \\ \infty, & \text{otherwise} \end{cases}$

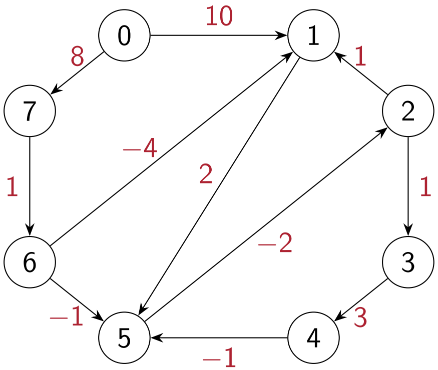
Repeat $n-1$ times

- For each vertex $j \in \{0, 1, \ldots, n-1\}$,

  for each edge $(j, k) \in E$,

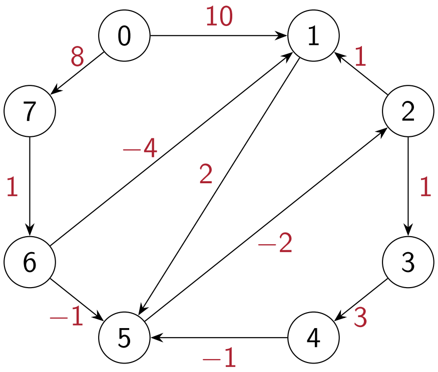
  $D(k) = \min(\ D(k),$
  $\qquad\qquad D(j) + W(j, k)\ )$

Works for directed and undirected graphs

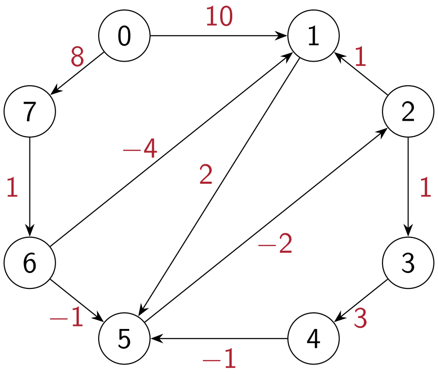


Bellman-Ford Algorithm

v	$D(v)$							
0								
1								
2								
3								
4								
5								
6								
7								



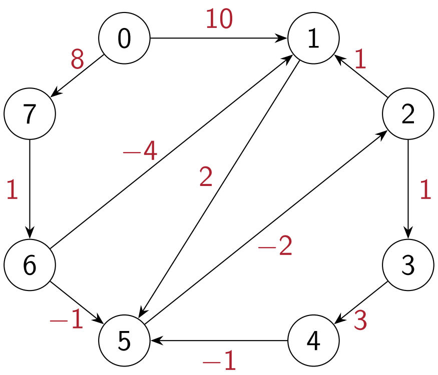
v	$D(v)$							
0	0							
1	∞							
2	∞							
3	∞							
4	∞							
5	∞							
6	∞							
7	∞							



- Initialize $D(0) = 0$

Bellman-Ford Algorithm

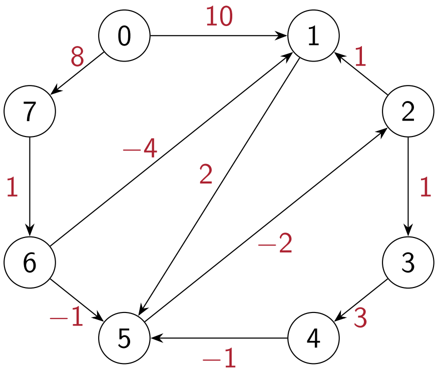
v	$D(v)$							
0	0	0						
1	∞	10						
2	∞	∞						
3	∞	∞						
4	∞	∞						
5	∞	∞						
6	∞	∞						
7	∞	8						



- Initialize $D(0) = 0$
- For each $(j,k) \in E$, update
  $$D(k) = \min(\ D(k),\ D(j) + W(j,k)\ )$$

Bellman-Ford Algorithm

v	$D(v)$							
0	0	0	0					
1	∞	10	10					
2	∞	∞	∞					
3	∞	∞	∞					
4	∞	∞	∞					
5	∞	∞	12					
6	∞	∞	9					
7	∞	8	8					

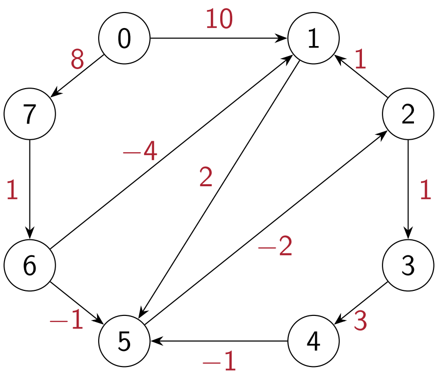


- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update

  $$D(k) = \min(\ D(k),\ D(j) + W(j,k)\ )$$

Bellman-Ford Algorithm

v	$D(v)$							
0	0	0	0	0				
1	∞	10	10	5				
2	∞	∞	∞	10				
3	∞	∞	∞	∞				
4	∞	∞	∞	∞				
5	∞	∞	12	8				
6	∞	∞	9	9				
7	∞	8	8	8				

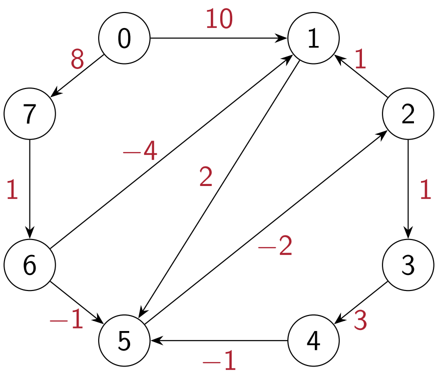


- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update

  $D(k) = \min(\ D(k),$
  $\qquad D(j) + W(j, k)\ )$

Bellman-Ford Algorithm

v	$D(v)$							
0	0	0	0	0	0			
1	∞	10	10	5	5			
2	∞	∞	∞	10	6			
3	∞	∞	∞	∞	11			
4	∞	∞	∞	∞	∞			
5	∞	∞	12	8	7			
6	∞	∞	9	9	9			
7	∞	8	8	8	8			

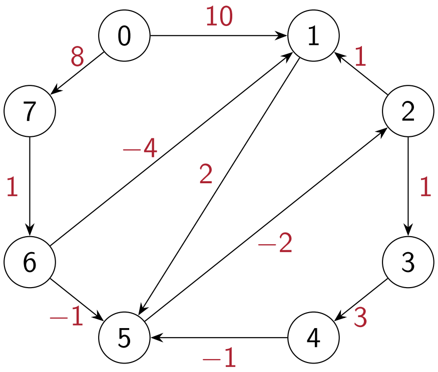


- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update

$$D(k) = \min(\ D(k),\ D(j) + W(j,k)\ )$$

Bellman-Ford Algorithm

v	$D(v)$							
0	0	0	0	0	0	0		
1	∞	10	10	5	5	5		
2	∞	∞	∞	10	6	5		
3	∞	∞	∞	∞	11	7		
4	∞	∞	∞	∞	∞	14		
5	∞	∞	12	8	7	7		
6	∞	∞	9	9	9	9		
7	∞	8	8	8	8	8		

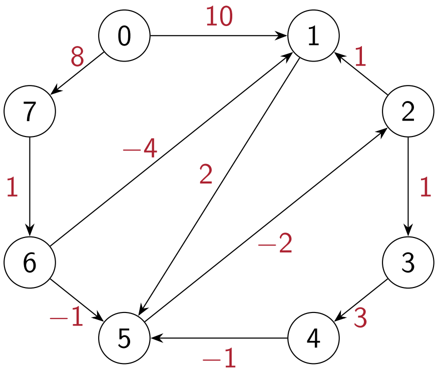


- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update

  $D(k) = \min(\ D(k),$
  $\qquad D(j) + W(j, k)\ )$

Bellman-Ford Algorithm

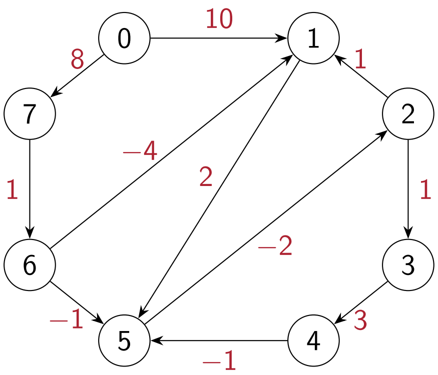
v	$D(v)$							
0	0	0	0	0	0	0	0	
1	∞	10	10	5	5	5	5	
2	∞	∞	∞	10	6	5	5	
3	∞	∞	∞	∞	11	7	6	
4	∞	∞	∞	∞	∞	14	10	
5	∞	∞	12	8	7	7	7	
6	∞	∞	9	9	9	9	9	
7	∞	8	8	8	8	8	8	



- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update

$$D(k) = \min(\ D(k),\ D(j) + W(j,k)\ )$$

v	$D(v)$							
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8

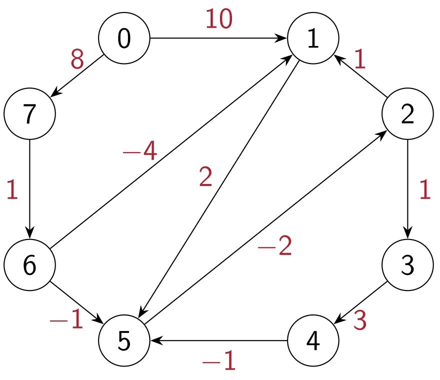


- Initialize $D(0) = 0$
- For each $(j, k) \in E$, update

$$D(k) = \min(\ D(k),\ D(j) + W(j,k)\ )$$

Bellman-Ford Algorithm

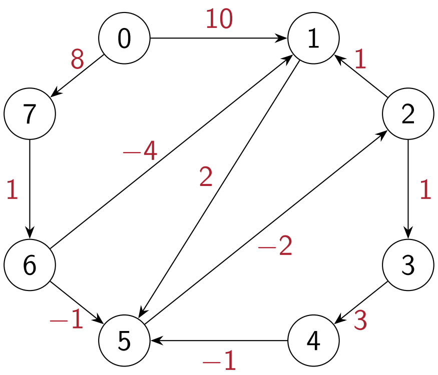
v	$D(v)$							
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8



- What if there was a negative cycle?

Bellman-Ford Algorithm

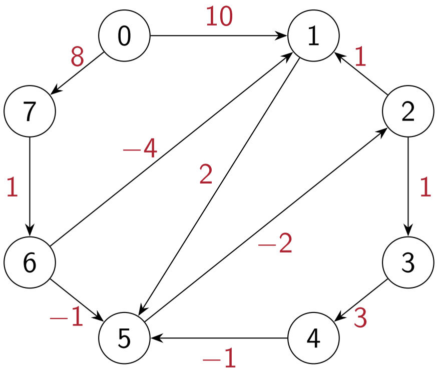
v	$D(v)$							
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8



- What if there was a negative cycle?
- Distance would continue to decrease

Bellman-Ford Algorithm

v	$D(v)$							
0	0	0	0	0	0	0	0	0
1	∞	10	10	5	5	5	5	5
2	∞	∞	∞	10	6	5	5	5
3	∞	∞	∞	∞	11	7	6	6
4	∞	∞	∞	∞	∞	14	10	9
5	∞	∞	12	8	7	7	7	7
6	∞	∞	9	9	9	9	9	9
7	∞	8	8	8	8	8	8	8



- What if there was a negative cycle?
- Distance would continue to decrease
- Check if update n reduces any $D(v)$

Complexity

- Initialing `infinity` takes $O(n^2)$ time

```
def bellmanford(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  distance = {}
  for v in range(rows):
    distance[v] = infinity

  distance[s] = 0

  for i in range(rows):
    for u in range(rows):
      for v in range(cols):
        if WMat[u,v,0] == 1:
          distance[v] = min(distance[v],distance[u]
                                        +WMat[u,v,1])
  return(distance)
```

Complexity

- Initialing `infinity` takes $O(n^2)$ time
- The outer update loop runs $O(n)$ times

```
def bellmanford(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  distance = {}
  for v in range(rows):
    distance[v] = infinity

  distance[s] = 0

  for i in range(rows):
    for u in range(rows):
      for v in range(cols):
        if WMat[u,v,0] == 1:
          distance[v] = min(distance[v],distance[u]
                                        +WMat[u,v,1])
  return(distance)
```

Complexity

- Initialing `infinity` takes $O(n^2)$ time
- The outer update loop runs $O(n)$ times
- In each iteration, we have to examine every edge in the graph
  - This take $O(n^2)$ for an adjacency matrix

```
def bellmanford(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  distance = {}
  for v in range(rows):
    distance[v] = infinity

  distance[s] = 0

  for i in range(rows):
    for u in range(rows):
      for v in range(cols):
        if WMat[u,v,0] == 1:
          distance[v] = min(distance[v],distance[u]
                                        +WMat[u,v,1])
  return(distance)
```

Complexity

- Initialing `infinity` takes $O(n^2)$ time
- The outer update loop runs $O(n)$ times
- In each iteration, we have to examine every edge in the graph
  - This take $O(n^2)$ for an adjacency matrix
- Overall, $O(n^3)$

```
def bellmanford(WMat,s):
  (rows,cols,x) = WMat.shape
  infinity = np.max(WMat)*rows+1
  distance = {}
  for v in range(rows):
    distance[v] = infinity

  distance[s] = 0

  for i in range(rows):
    for u in range(rows):
      for v in range(cols):
        if WMat[u,v,0] == 1:
          distance[v] = min(distance[v],distance[u]
                                        +WMat[u,v,1])
  return(distance)
```

Complexity

- Initialing `infinity` takes $O(n^2)$ time
- The outer update loop runs $O(n)$ times
- In each iteration, we have to examine every edge in the graph
  - This take $O(n^2)$ for an adjacency matrix
- Overall, $O(n^3)$
- If we shift to adjacency lists
  - Initializing `infinity` is $O(m)$
  - Scanning all edges in each update iteration is $O(m)$

```
def bellmanfordlist(WList,s):
  infinity = 1 + len(WList.keys())*
              max([d for u in WList.keys()
                      for (v,d) in WList[u]])
  distance = {}
  for v in WList.keys():
    distance[v] = infinity

  distance[s] = 0

  for i in WList.keys():
    for u in WList.keys():
      for (v,d) in WList[u]:
        distance[v] = min(distance[v],distance[u] + d)
  return(distance)
```

Complexity

- Initialing `infinity` takes $O(n^2)$ time
- The outer update loop runs $O(n)$ times
- In each iteration, we have to examine every edge in the graph
  - This take $O(n^2)$ for an adjacency matrix
- Overall, $O(n^3)$
- If we shift to adjacency lists
  - Initializing `infinity` is $O(m)$
  - Scanning all edges in each update iteration is $O(m)$
- Now, overall $O(mn)$

```
def bellmanfordlist(WList,s):
  infinity = 1 + len(WList.keys())*
               max([d for u in WList.keys()
                       for (v,d) in WList[u]])
  distance = {}
  for v in WList.keys():
    distance[v] = infinity

  distance[s] = 0

  for i in WList.keys():
    for u in WList.keys():
      for (v,d) in WList[u]:
        distance[v] = min(distance[v],distance[u] + d)
  return(distance)
```

Summary

- Dijkstra's algorithm assumes non-negative edge weights
  - Final distance is frozen each time a vertex "burns"
  - Should not encounter a shorter route discovered later
- Without negative cycles, every shortest route is a path
- Every prefix of a shortest path is also a shortest path
- Iteratively find shortest paths of length $1, 2, \ldots, n-1$
- Update distance to each vertex with every iteration — Bellman-Ford algorithm
- $O(n^3)$ time with adjacency matrix, $O(mn)$ time with adjacency list
- If Bellman-Ford algorithm does not converge after $n - 1$ iterations, there is a negative cycle