Single Source Shortest Path

Dijekstra Algorithm

Single-Source Shortest Paths

Input: directed graph G=(V, E). (m=|E|, n=|V|)

- each edge has non negative length l_e
- source vertex s

Length of path = sum of edge lengths

Output: for each $v \in V$, compute

L(v) := length of a shortest s-v path in G

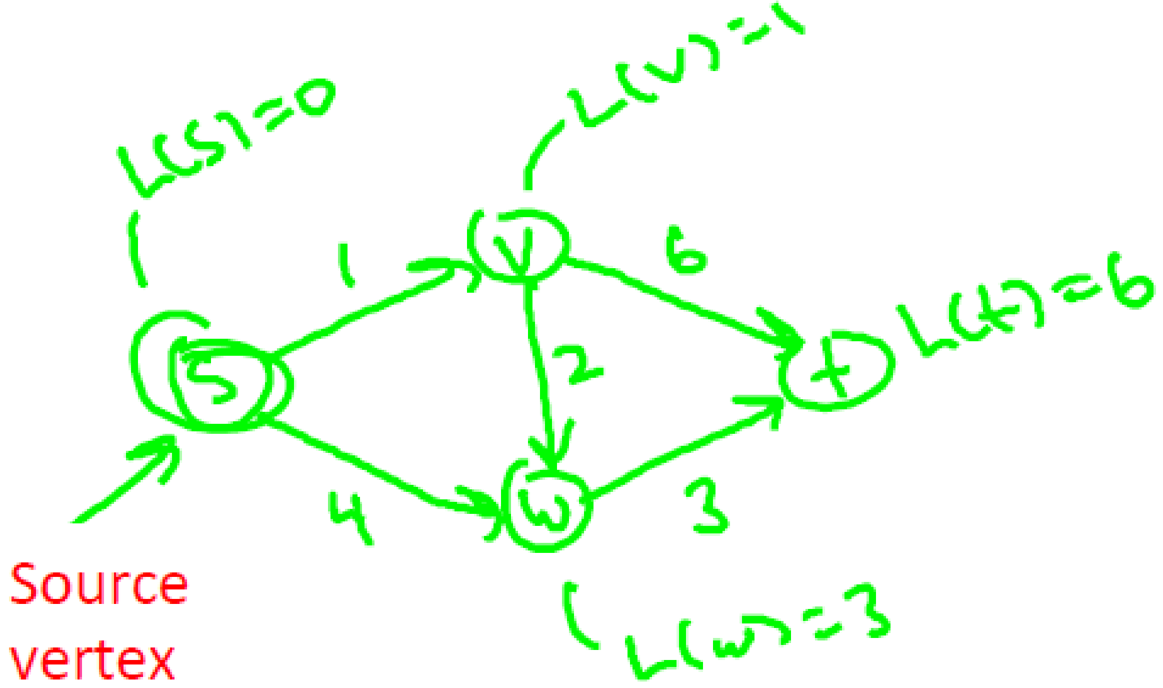
Path length = 6

Assumption:

1. [for convenience] $\forall v \in V, \exists s \Rightarrow v$ path
2. [important] $l_e \geq 0 \;\; \forall e \in E$

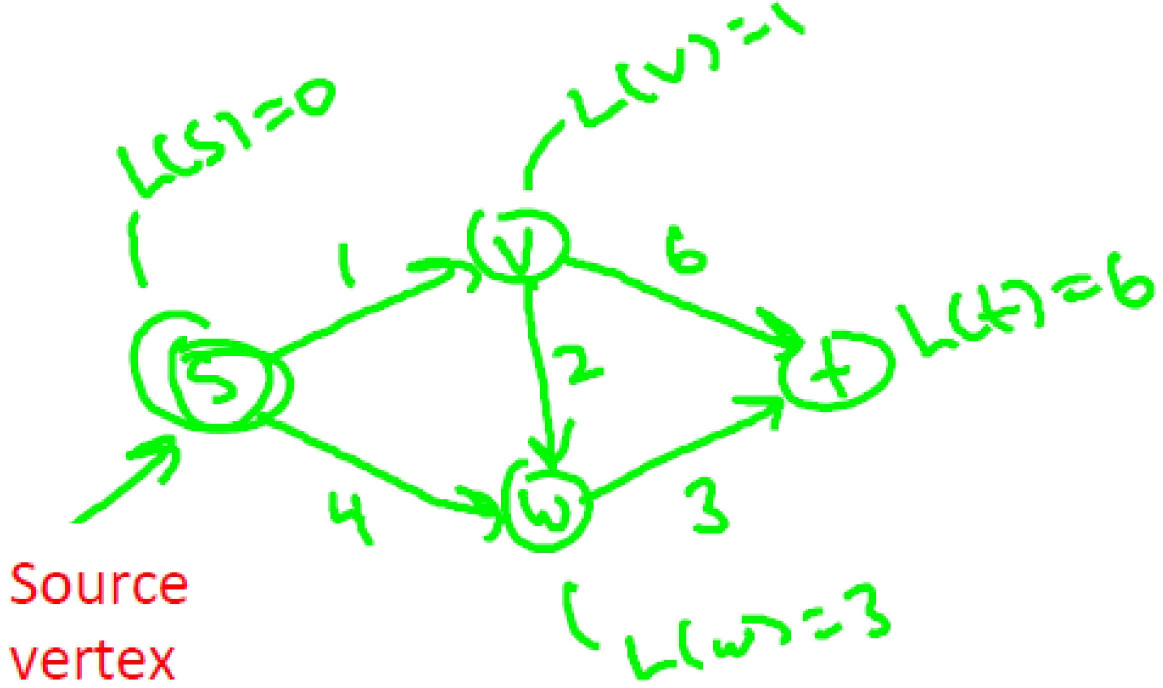
One of the following is the list of shortest---path distances for the nodes s,v,w,t, respectively. Which is it?

a) 0,1,2,3
b) 0,1,4,7
c) 0,1,4,6
d) 0,1,3,6



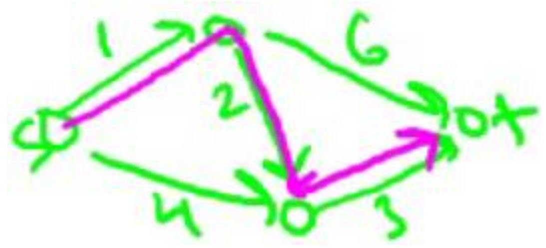
One of the following is the list of shortest---path distances for the nodes s,v,w,t, respectively. Which is it?

a) 0,1,2,3
b) 0,1,4,7
c) 0,1,4,6
d) 0,1,3,6

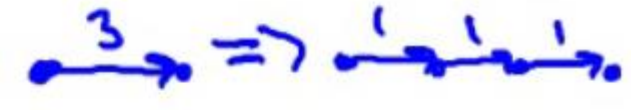


Why Another Shortest---Path Algorithm?

- Question: Doesn't BFS already compute shortest paths in linear time?
- Answer: yes, If l_e = 1 for every edge e.



- Question: why not just replace each edge e by directed path of le unit length edges:



- Answer: blows up graph too much
- Solution: Dijkstra's shortest path algorithm.

Dijkstra's Algorithm

This array only to help explanation!

Initialize:

- X = [s] [vertices processed so far]
- A[s] = 0 [computed shortest path distances]
- B[s] = empty path [computed shortest paths]

Main Loop

- while X≠V:

-need to grow x by one node

Main Loop cont'd:

- among all edges $(v, w) \in E$ with $v \in X, w \notin X$, pick the one that minimizes $A[v] + l_{vw}$

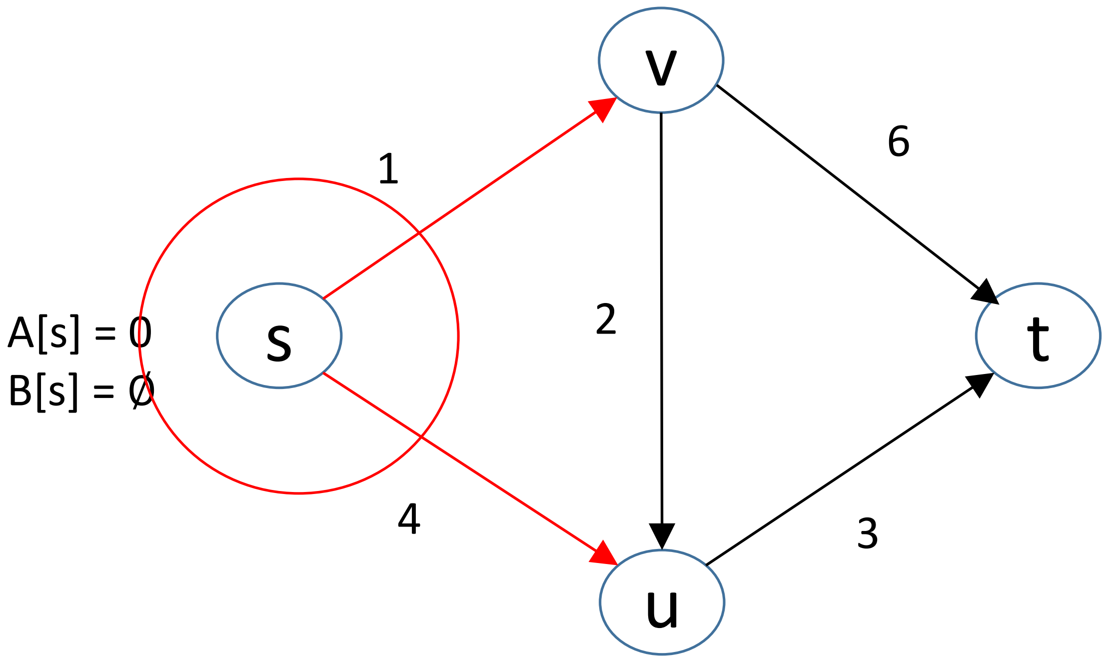
Already computed in earlier iteration

[call it (v*, w*)]

- add w* to X
- set $A[w^*] := A[v^*] + l_{v^*w^*}$
- set $B[w^*] := B[v^*]u(v^*, w^*)$

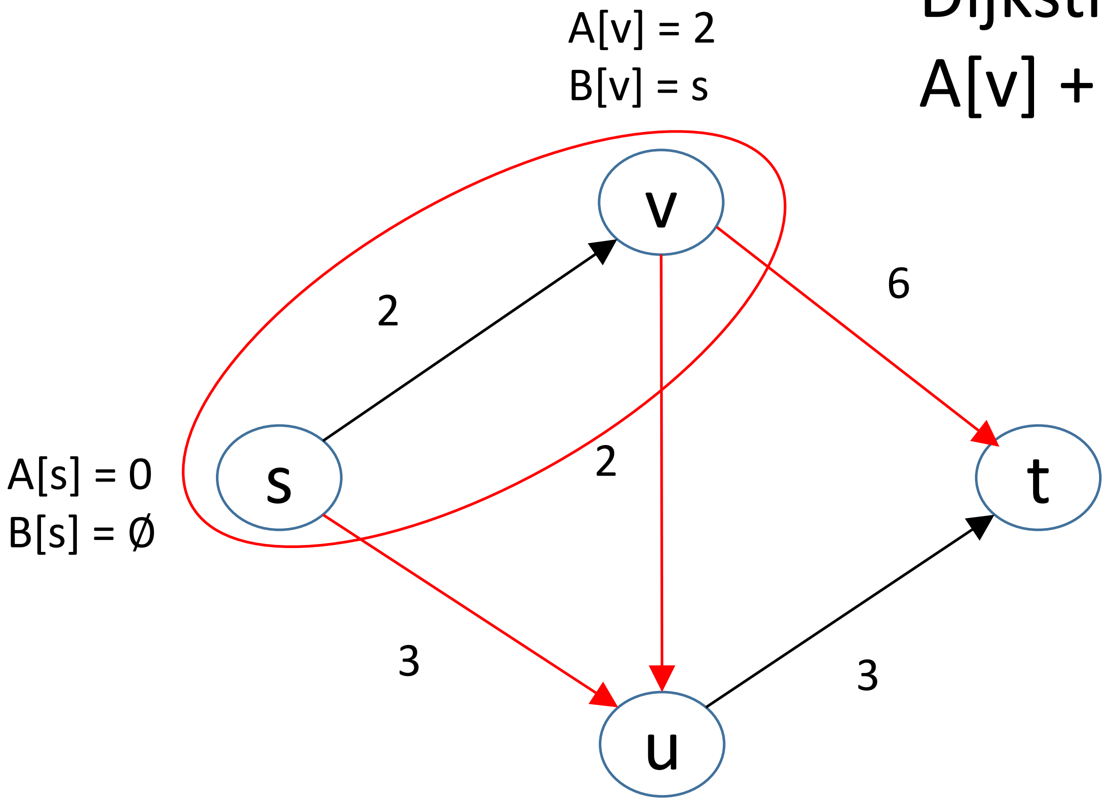
Dijkstra's Algorithm

Dijkstra's Greedy score for vertex u = A[v] + $l_{v,u}$



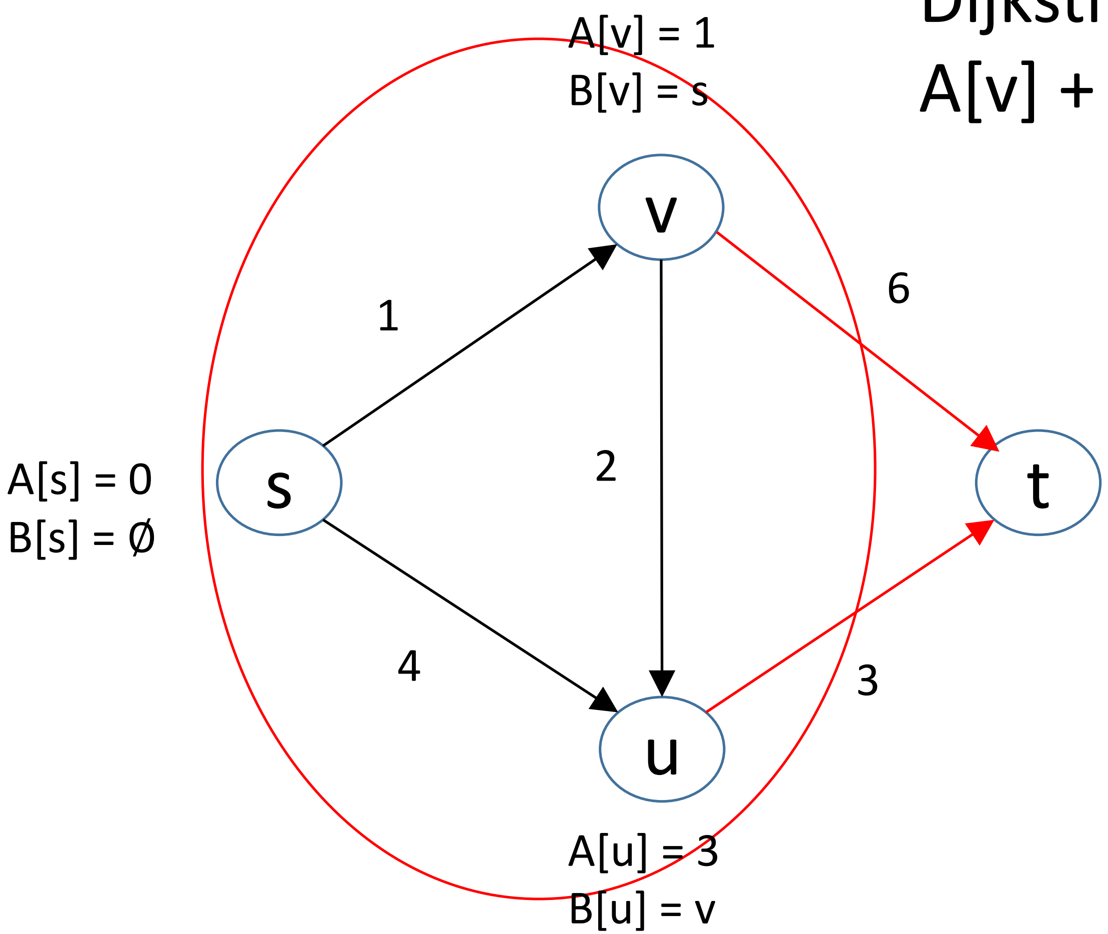
Dijkstra's Algorithm

Dijkstra’s Greedy score for vertex u = $A[v] + l_{v,u}$



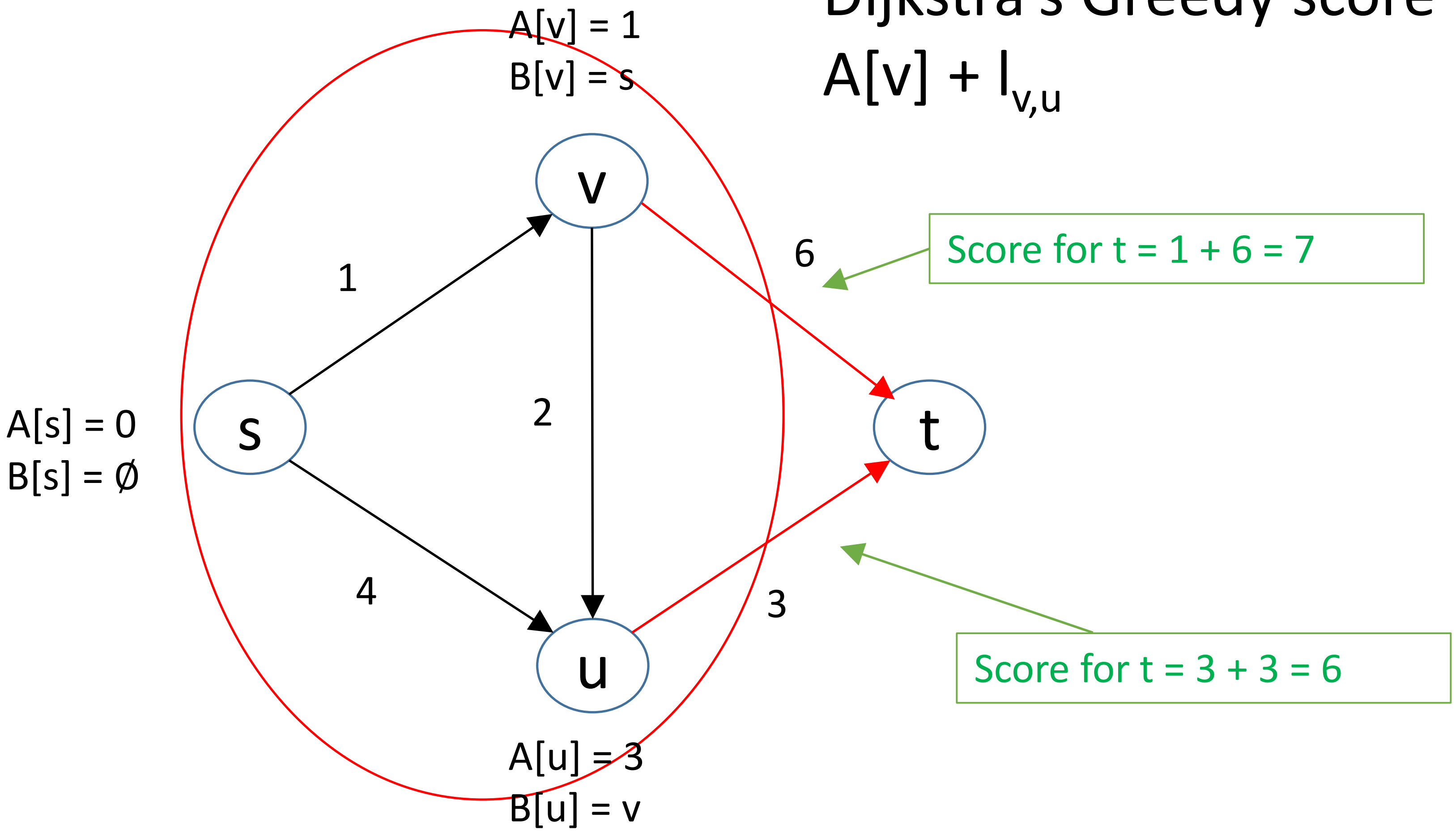
Dijkstra's Algorithm

Dijkstra’s Greedy score for vertex u = A[v] + $l_{v,u}$



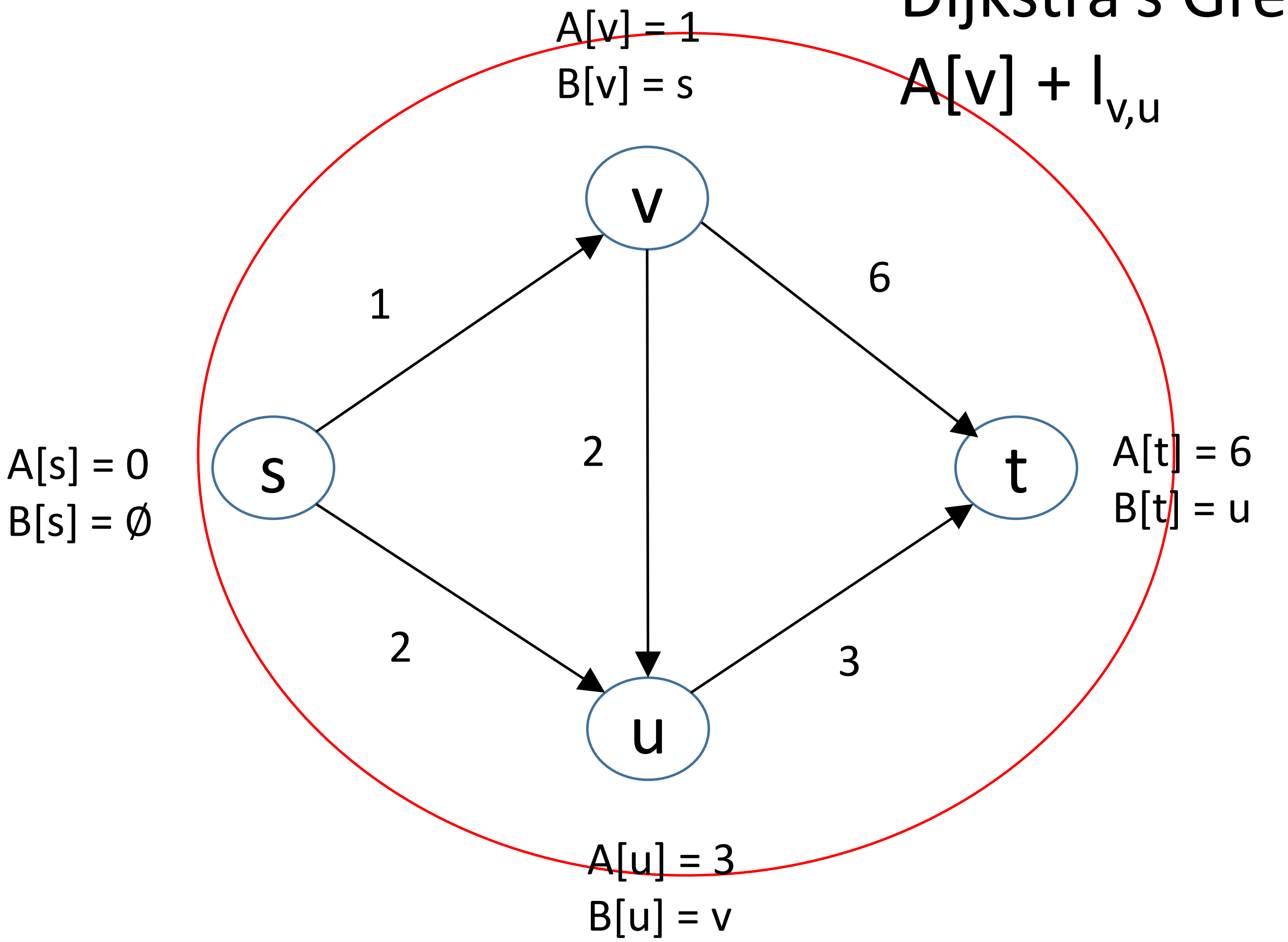
Dijkstra's Algorithm

Dijkstra's Greedy score for vertex u = $A[v] + l_{v,u}$



Dijkstra's Algorithm

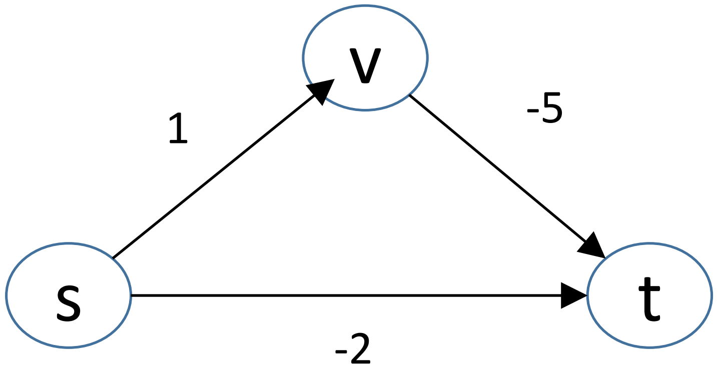
Dijkstra’s Greedy score for vertex u = $A[v] + l_{v,u}$



Negative Edge Lengths

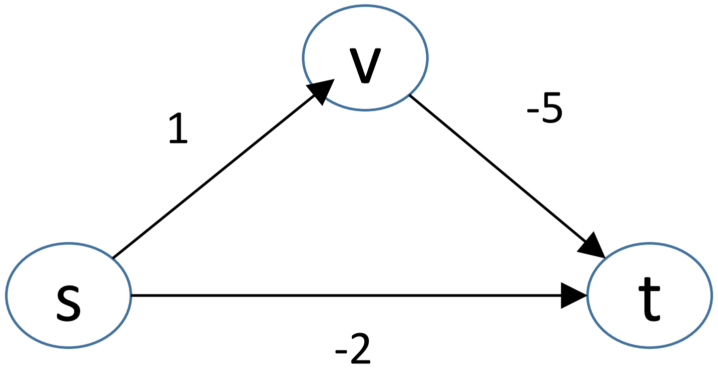
Dijkstra's algorithm incorrect on this graph !

(computes shortest s-t distance to be -2 rather than -4)



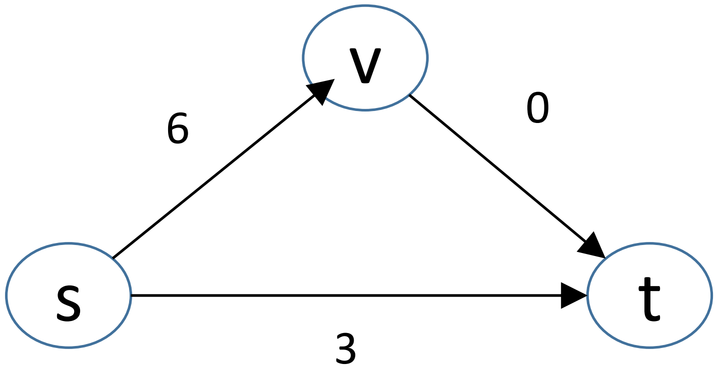
Negative Edge Lengths

Question: why not reduce computing shortest paths with negative edge lengths to the same problem with non negative lengths? (by adding large constant to edge lengths)



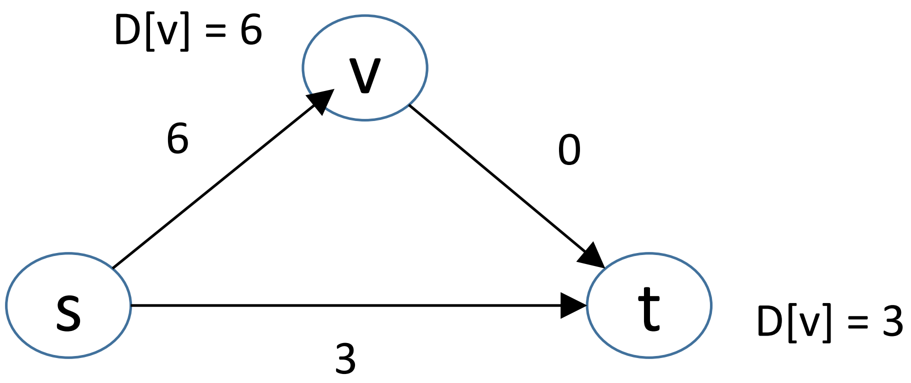
Negative Edge Lengths

Question: why not reduce computing shortest paths with negative edge lengths to the same problem with non negative lengths? (by adding large constant to edge lengths)



Negative Edge Lengths

- Question: why not reduce computing shortest paths with negative edge lengths to the same problem with non negative lengths? (by adding large constant to edge lengths)
- Problem: doesn't preserve shortest paths !
- Total weight added on path from s to t through direct edge is 5
- While total weight added on path from s to t through v is 10.



Dijkstra's Algorithm

This array only to help explanation!

Initialize:

- X = [s] [vertices processed so far]
- A[s] = 0 [computed shortest path distances]
- B[s] = empty path [computed shortest paths]

Main Loop

- while X≠V:

-need to grow x by one node

Main Loop cont'd:

- among all edges $(v, w) \in E$ with $v \in X, w \notin X$, pick the one that minimizes $A[v] + l_{vw}$

  [call it (v*, w*)]

Already computed in earlier iteration

- add w* to X
- set $A[w^*] := A[v^*] + l_{v^*w^*}$
- set $B[w^*] := B[v^*]u(v^*, w^*)$

Dijkstra’s Algorithm

- Which of the following running times seems to best describe a “naive” implementation of Dijkstra’s algorithm?

a) $\theta(m+n)$

b) $\theta(m \log n)$

c) $\theta(n^2)$

d) $\theta(mn)$

Dijkstra’s Algorithm

- Which of the following running times seems to best describe a “naive” implementation of Dijkstra’s algorithm?

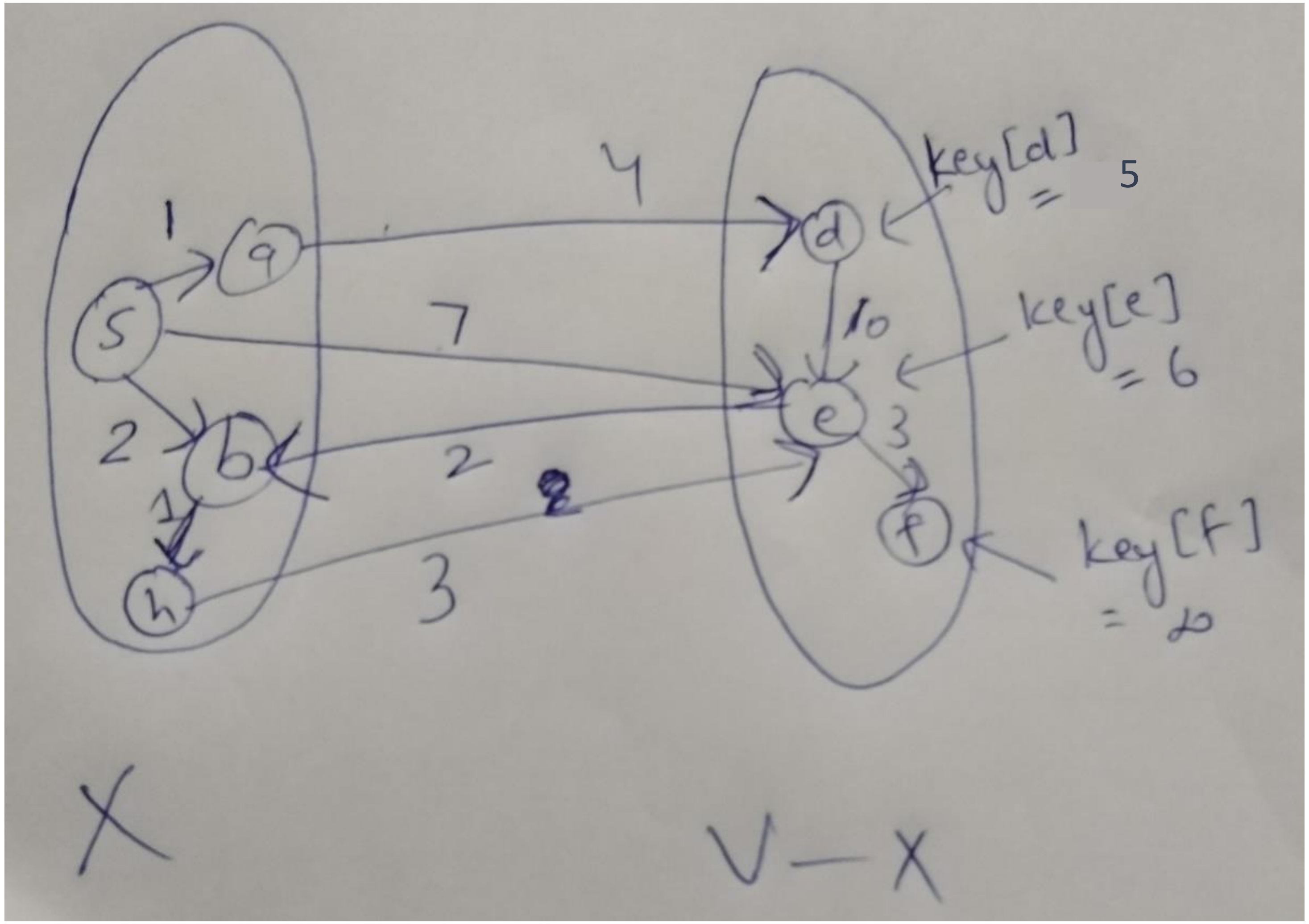
a) $\theta(m+n)$
b) $\theta(m \log n)$
c) $\theta(n^2)$
d) $\theta(mn)$

- (n-1) iterations of while loop
- O(m) work per iteration
- O(1) [work per edge]

CAN WE DO BETTER?

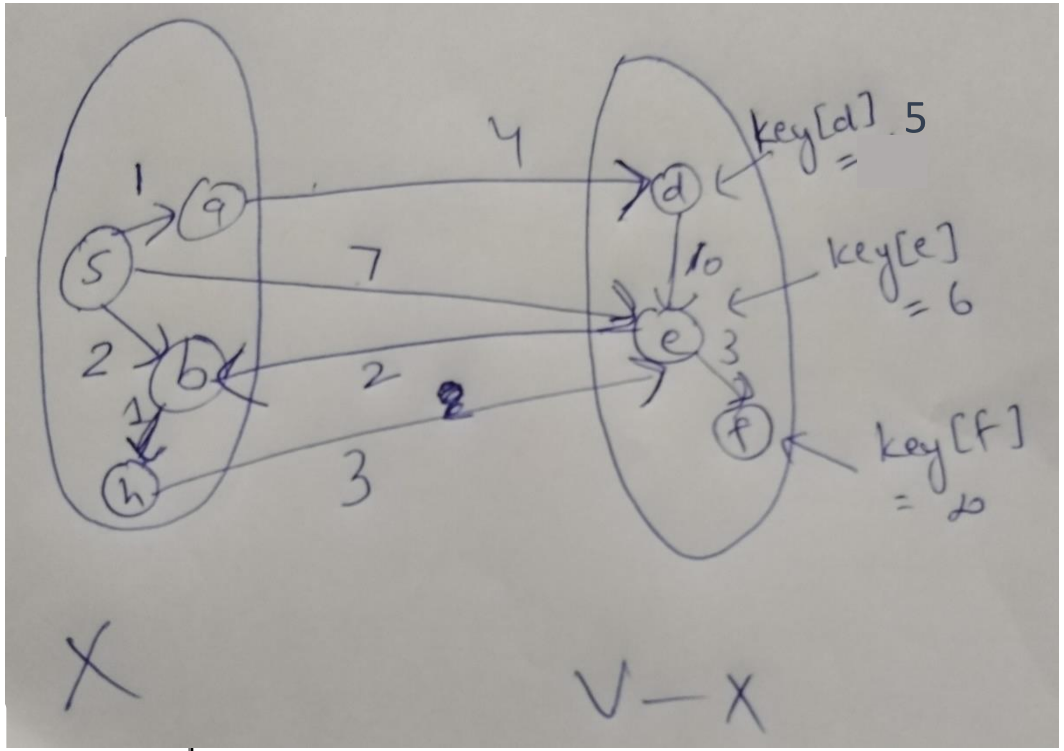
Heap Operations

- perform Insert, Extract-Min, delete in O(log n) time.
- conceptually, a perfectly balanced binary tree
- Heap property: at every node, key <= children's keys
- extract-min by swapping up last leaf, bubbling down
- insert via bubbling up
- Also: will need ability to delete from middle of heap. (bubble up or down as needed)



key[d] = 5
key[e] = 6
key[f] = ∞
X
V−X

Two Invariants



Dijkstra's greedy score of (v, w) : $A[v] + l_{vw}$

Invariant # 1: elements in heap = vertices of V-X.

Invariant #2: for $v \notin X$

Key[v] = smallest Dijstra greedy score of an edge (u, v) in E with u in X

(of $+\infty$ if no such edges exist)

Point: by invariants, Extract-Min yields correct vertex w* to add to X next.

(and we set A[w*] to key[w*])

Maintaining the Invariants

To maintain Invariant #2: [i.e., that $\forall v \notin X$
Key[v] = smallest Dijkstra greedy
score of edge (u,v) with u in X]

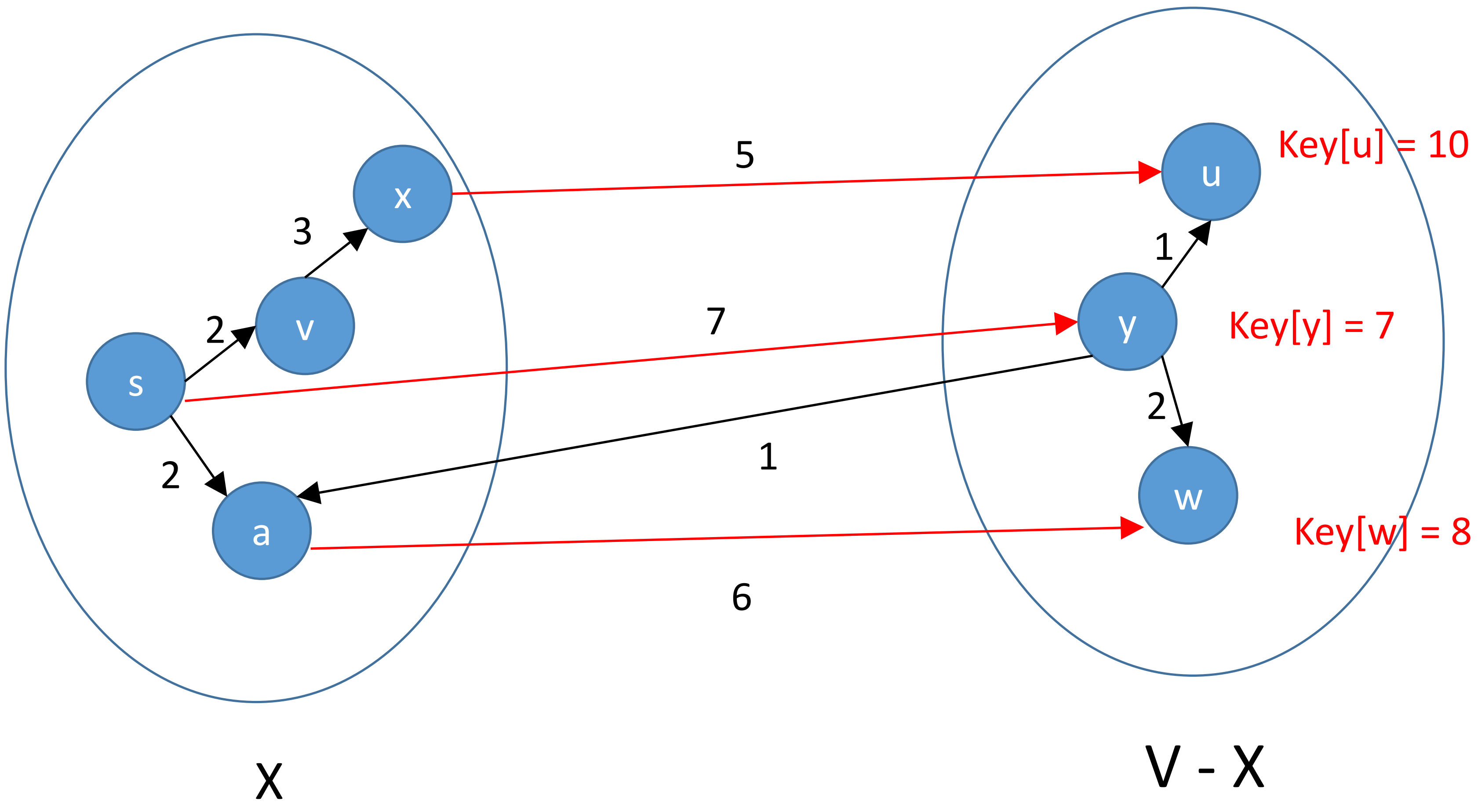
When w extracted from heap (i.e., added to X)

- for each edge (w,v) in E:
  - if v in V-X (i.e., in heap)
    - delete v from heap
    - recompute key[v] = min{key[v] , A[w] + l_{wv} }
    - re-Insert v into heap

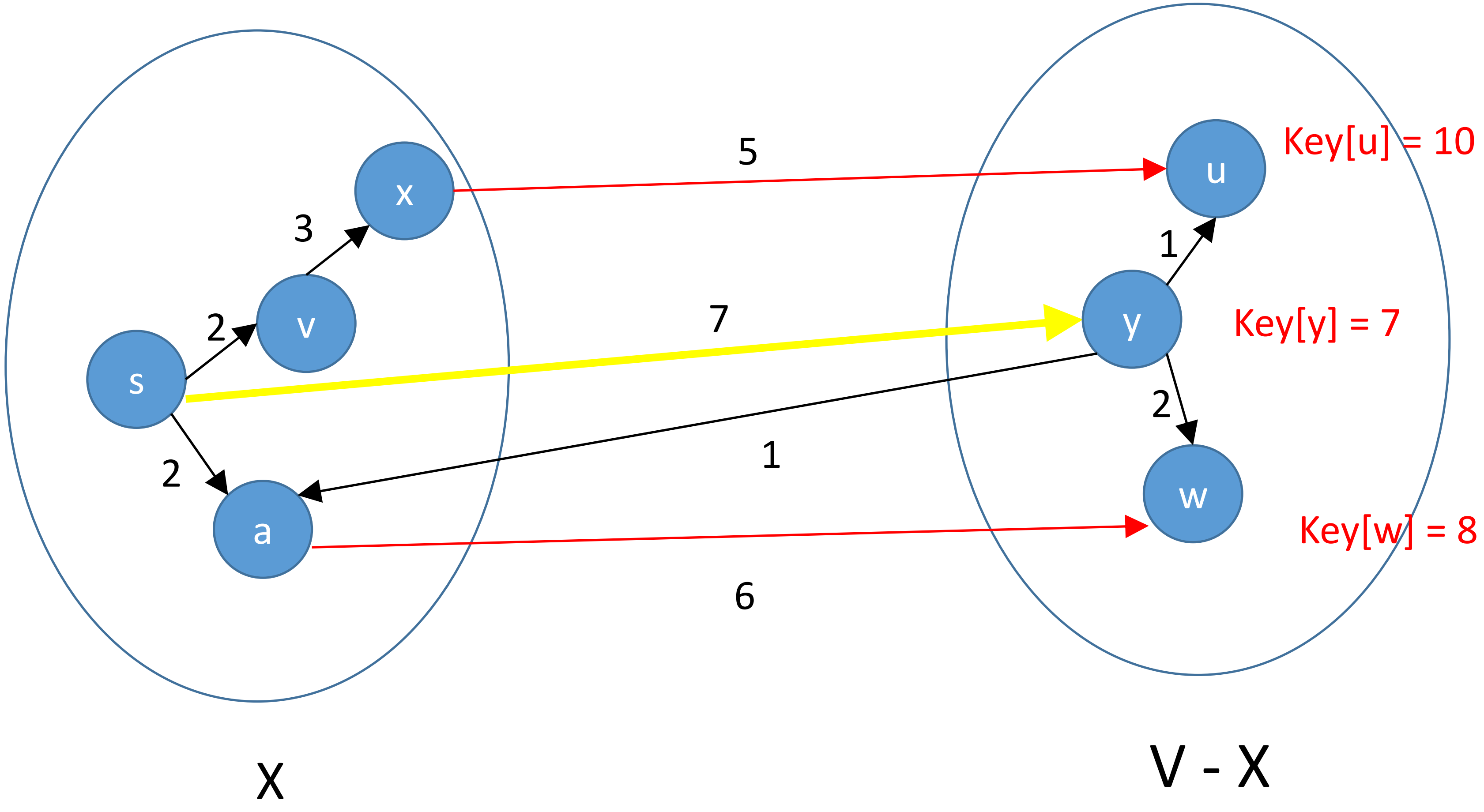
Key update

Greedy score of (w,v)

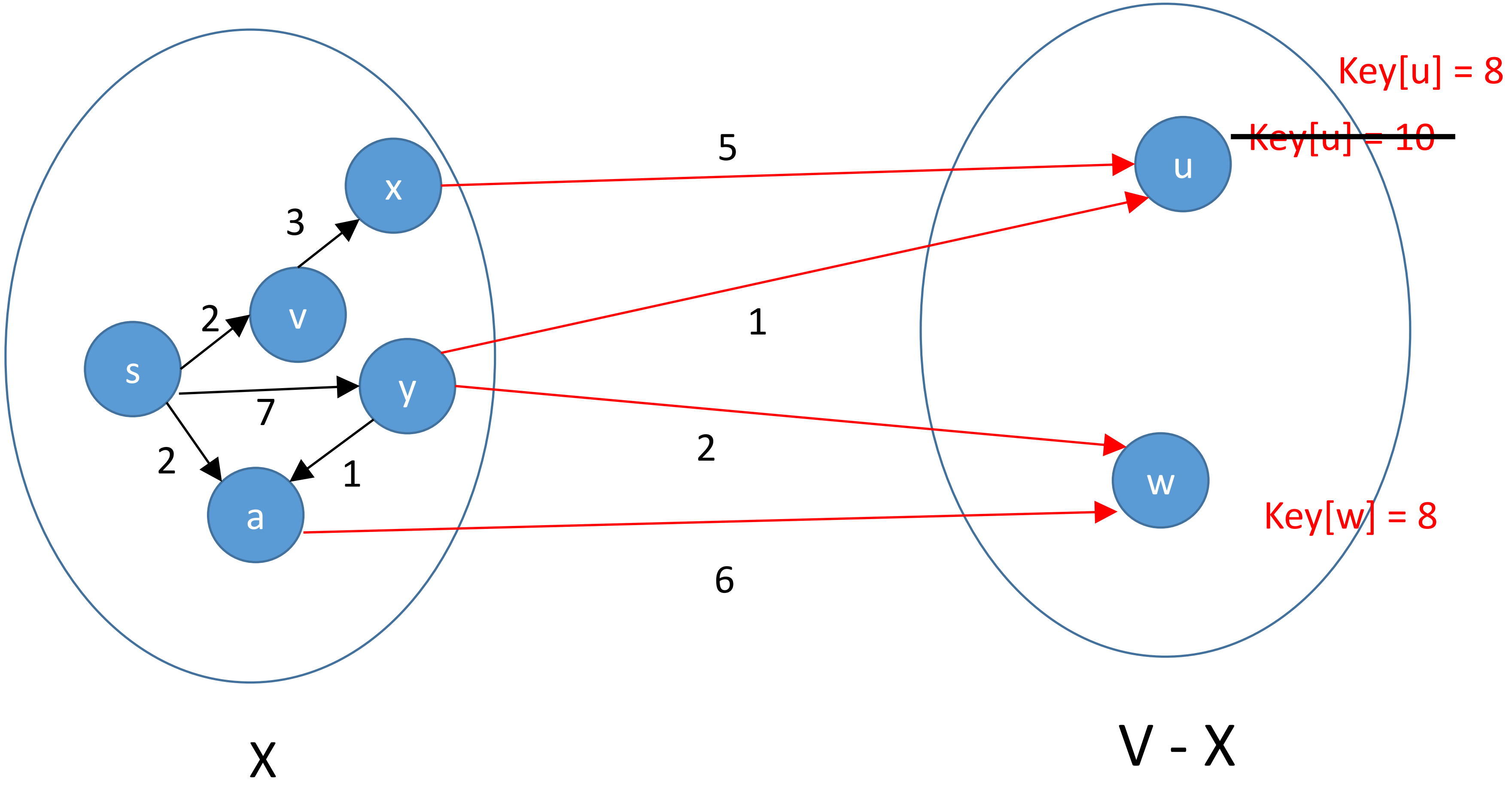
Dijkstra Algorithm



Dijkstra Algorithm



Dijkstra Algorithm



Dijkstra's Algorithm

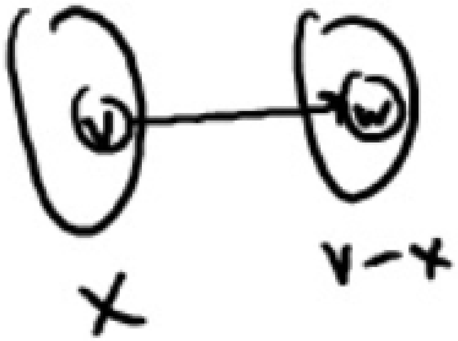
```
Dijkstra(G,w,s)
{                                                   % Initialize
    for (each u ∈ V)
    {
        d[u] = ∞;
        color[u] =white;
    }
    d[s] = 0;
    pred[s] = NIL;
    Q = (queue with all vertices);

    while (Non-Empty(Q))                            % Process all vertices
    {
        u = Extract-Min(Q);                         % Find new vertex
        for (each v ∈ Adj[u])
            if (d[u] + w(u,v) < d[v])               % If estimate improves
            {
                d[v] = d[u] + w(u,v);                 relax
                Decrease-Key(Q, v, d[v]);
                pred[v] = u;
            }
        color[u] =black;
    }
}
```

Running Time Analysis

You check: dominated by heap operations. (O(log(n)) each)

- (n-1) Extract mins
- each edge (v,w) triggers at most one Delete/Insert combo

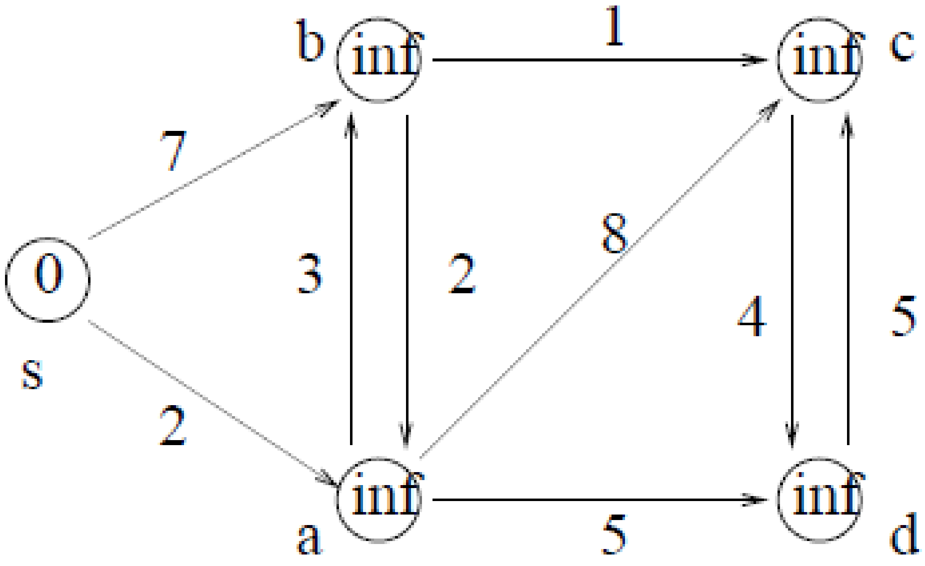
(if v added to X first)



So: # of heap operations in O(n+m) = O(m)

So: running time = O(m log(n)) (like sorting)

Since graph is
weakly connected

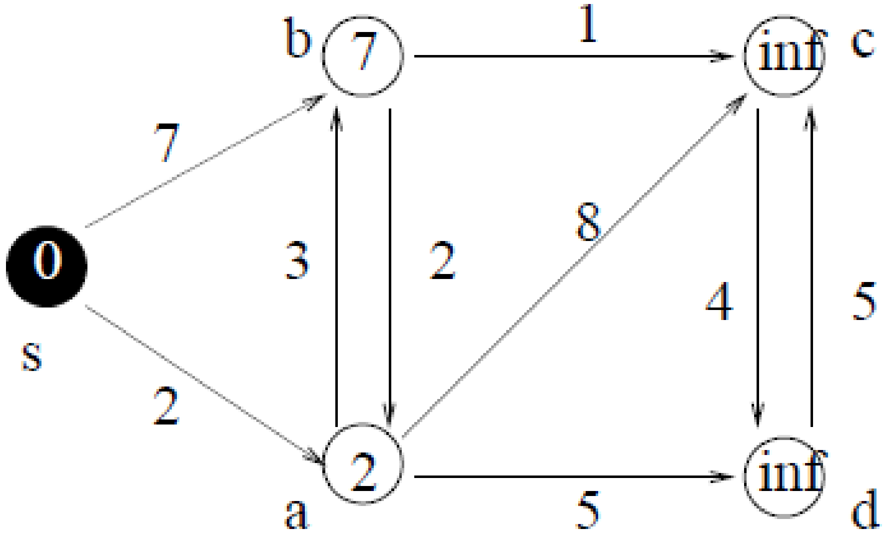


Step 0: Initialization.

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞
$pred[v]$	nil	nil	nil	nil	nil
$color[v]$	W	W	W	W	W

Priority Queue:

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞

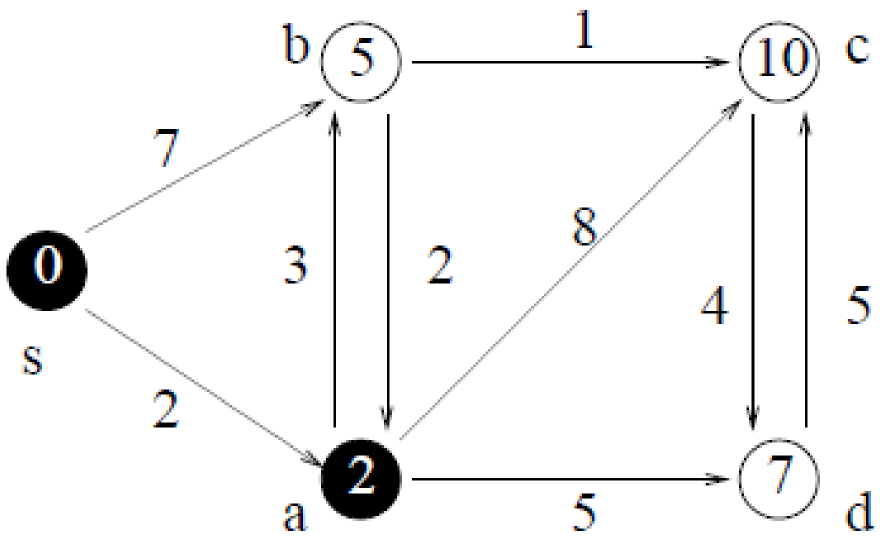


Step 1: As $Adj[s] = \{a, b\}$, work on a and b and update information.

v	s	a	b	c	d
$d[v]$	0	2	7	∞	∞
$pred[v]$	nil	s	s	nil	nil
$color[v]$	B	W	W	W	W

Priority Queue:

v	a	b	c	d
$d[v]$	2	7	∞	∞

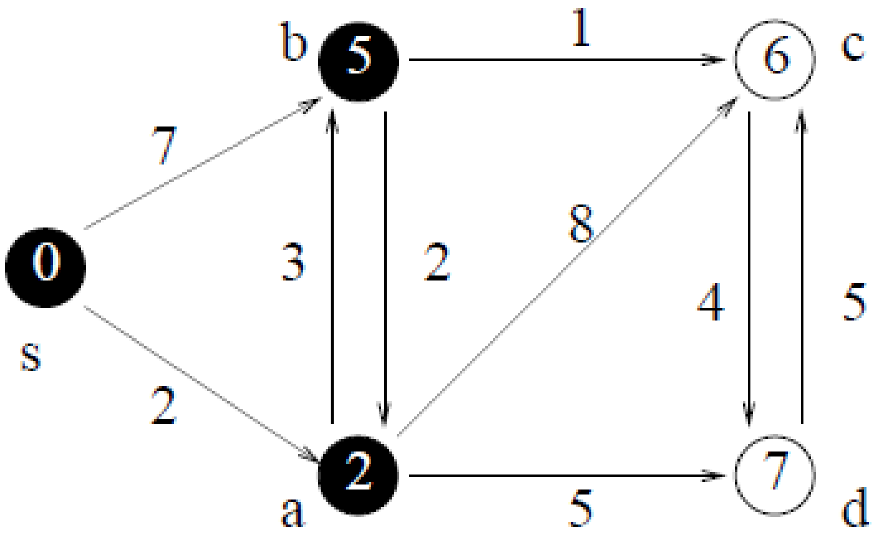


Step 2: After Step 1, a has the minimum key in the priority queue. As $Adj[a] = \{b, c, d\}$, work on b, c, d and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	10	7
$pred[v]$	nil	s	a	a	a
$color[v]$	B	B	W	W	W

Priority Queue:

v	b	c	d
$d[v]$	5	10	7

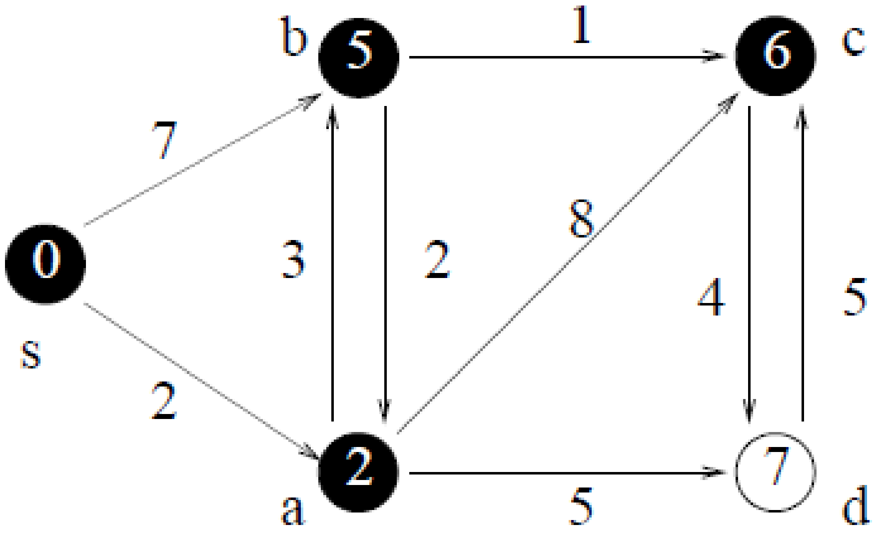


Step 3: After Step 2, b has the minimum key in the priority queue. As $Adj[b] = \{a, c\}$, work on a, c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	W	W

Priority Queue:

v	c	d
$d[v]$	6	7

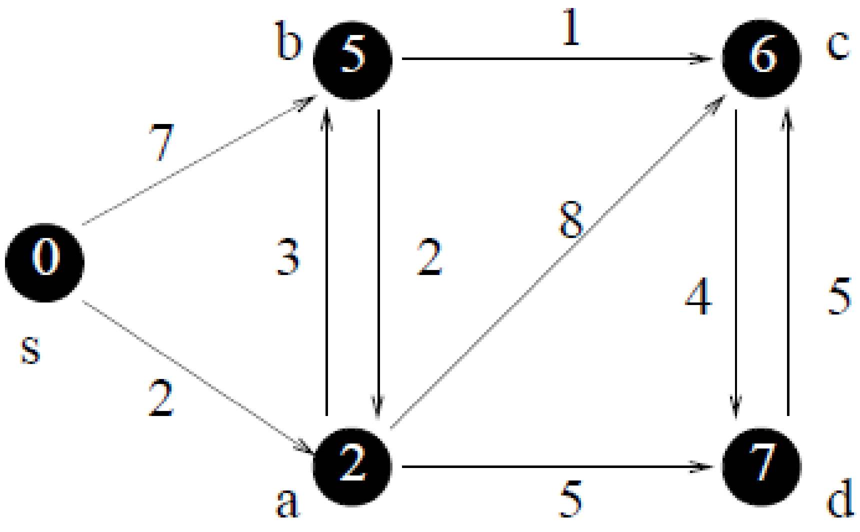


Step 4: After Step 3, c has the minimum key in the priority queue. As $Adj[c] = \{d\}$, work on d and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	W

Priority Queue:

v	d
$d[v]$	7



Step 5: After Step 4, d has the minimum key in the priority queue. As $Adj[d] = \{c\}$, work on c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	B

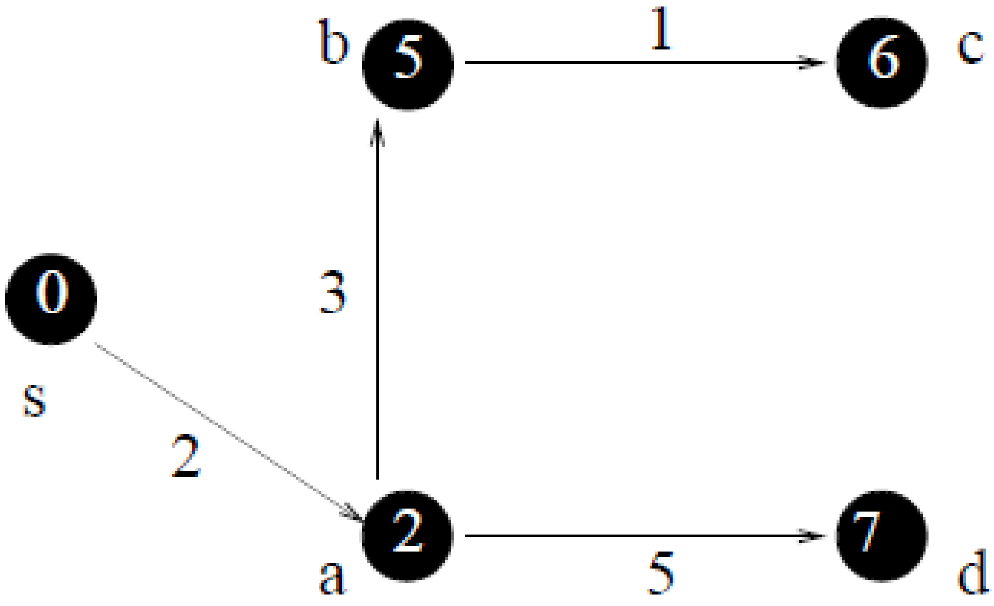
Priority Queue: $Q = \emptyset$.

We are done.

Shortest Path Tree: $T = (V, A)$, where

$$A = \{(pred[v], v) | v \in V \setminus \{s\}\}.$$

The array $pred[v]$ is used to build the tree.



Example:

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a

Correctness Claim

Theorem [Dijkstra] For every directed graph with nonnegative edge lengths, Dijkstra's algorithm correctly computes all shortest-path distances.

$$[i.e., \ A[v] = L(v) \ \forall v \in V]$$

what algorithm computes

True shortest distance from s to v

Proof: by induction on the number of iterations.

Base Case: $A[s] = L[s] = 0$ (correct)

Proof

<u>Inductive Step</u>:

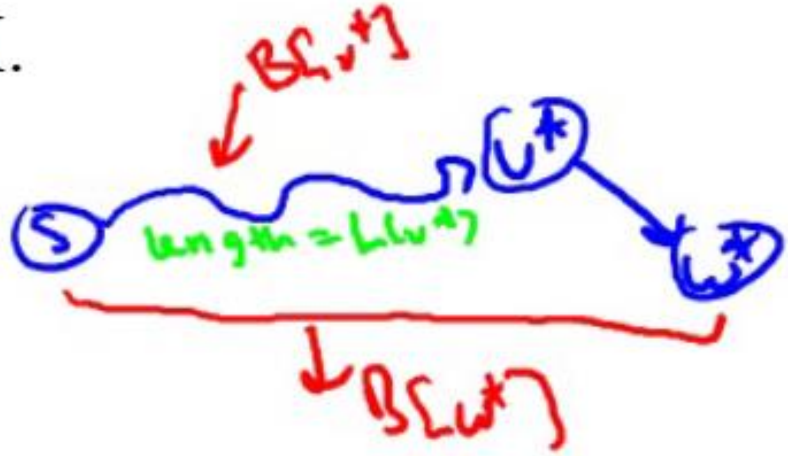
<u>Inductive Hypothesis</u>: all previous iterations correct (i.e., $A[v] = L(v)$ and $B[v]$ is a true shortest s-v path in G, for all v already in X).

<u>In current iteration</u>: in X not in X

We pick an edge (v^*, w^*) and we add w^* to X.

We set $B[w^*] = B[v^*] \cup (v^*, w^*)$

has length $L(v^*) + l_{v^*w^*}$ has length $L(v^*)$



L(v*) by I.H

<u>Also</u>: $A[w^*] = A[v^*] + l_{v^*w^*} = L(v^*) + l_{v^*w^*}$

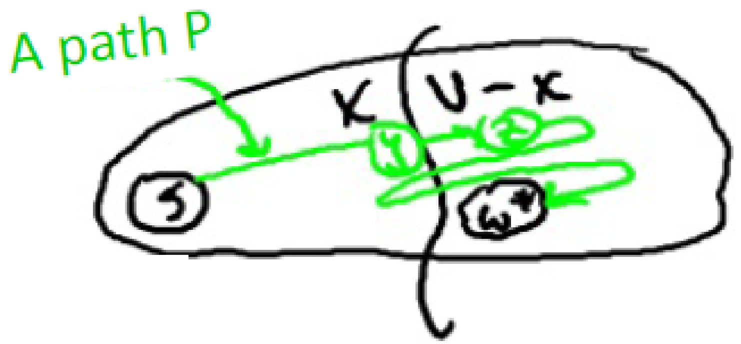
Proof (con’d)

Upshot: in current iteration, we set:

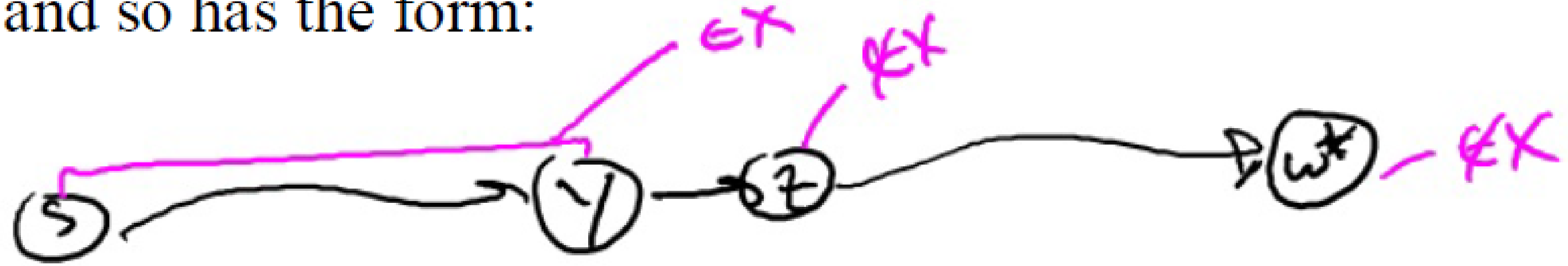
1. $A[w^*] = L(v^*) + l_{v^*w^*}$
2. $B[w^*]$ = an s -> w* path with length $(L(v^*) + l_{v^*w^*})$

To finish proof: need to show that *every* s-w* path has length >= $L(v^*) + l_{v^*w^*}$ (if so, our path is the shortest!)

So: Let P= any s->w* path. Must “cross the frontier”:

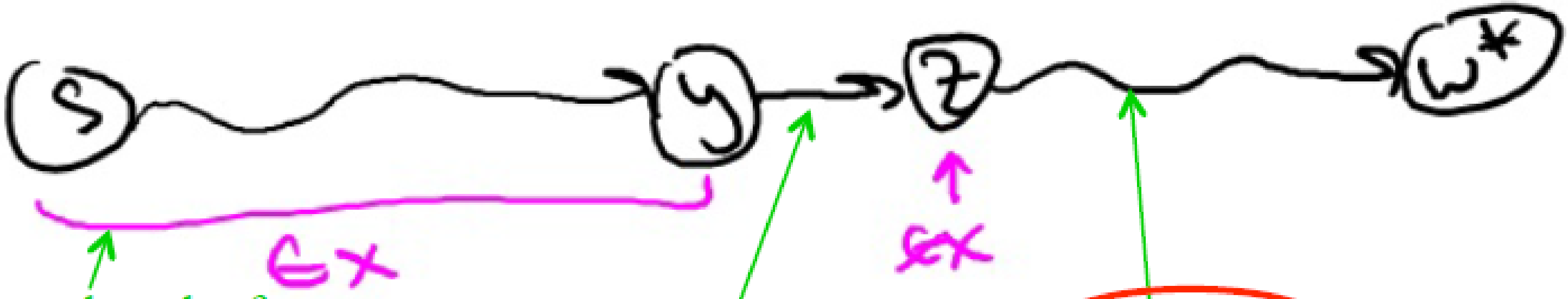


and so has the form:



Proof (con'd)

So: every s->w* path P has to have the form



>= length of shortest s-y path = L(y) = A[y]

By inductive hypothesis (since $y \in X$)

Length = l_{yz}

Length >=0

(Since no negative edges !)

($y \in X$
$z \notin X$)

Total length of path P: at least A[y] + C_{yz}

length of our path !

-> by Dijkstra's greedy criterion $A[v^*] + l_{v^*w^*} \leq A[y] + l_{yz} \leq$ length of P