Grafer, två viktiga algoritmer

Koffman & Wolfgang

- kapitel 10, avsnitt 6

Dijkstras algoritm för att hitta den kortaste vägen

Bredden-först-sökning hittar den kortaste vägen från startnoden till alla andra noder

- om vi antar att längden (eller vikten) för varje båge är lika stor
- dvs, om grafen är *oviktad*

Dijkstras algoritm hittar den kortaste vägen i en *viktad* graf

- funkar för både riktade och oriktade grafer

Dijkstras algoritm

Vi behöver 2 mängder (**S** och **V–S**) och 2 fält (**d** och **p**).

S innehåller de noder som vi har beräknat kortaste vägen för

- vi börjar med att lägga in startnoden s i **S**

V–S innehåller de noder som är kvar att besöka

- vi börjar med att lägga alla andra noder v i **V–S**

d[v] innehåller den kortaste vägen från s till v

- vi börjar med att sätta **d**[v] till w,
  om det finns en båge (s, v) med vikt w
- om det inte finns någon båge (s, v), sätt **d**[v] till ∞

p[v] innehåller föregångaren till v i stigen från s till v

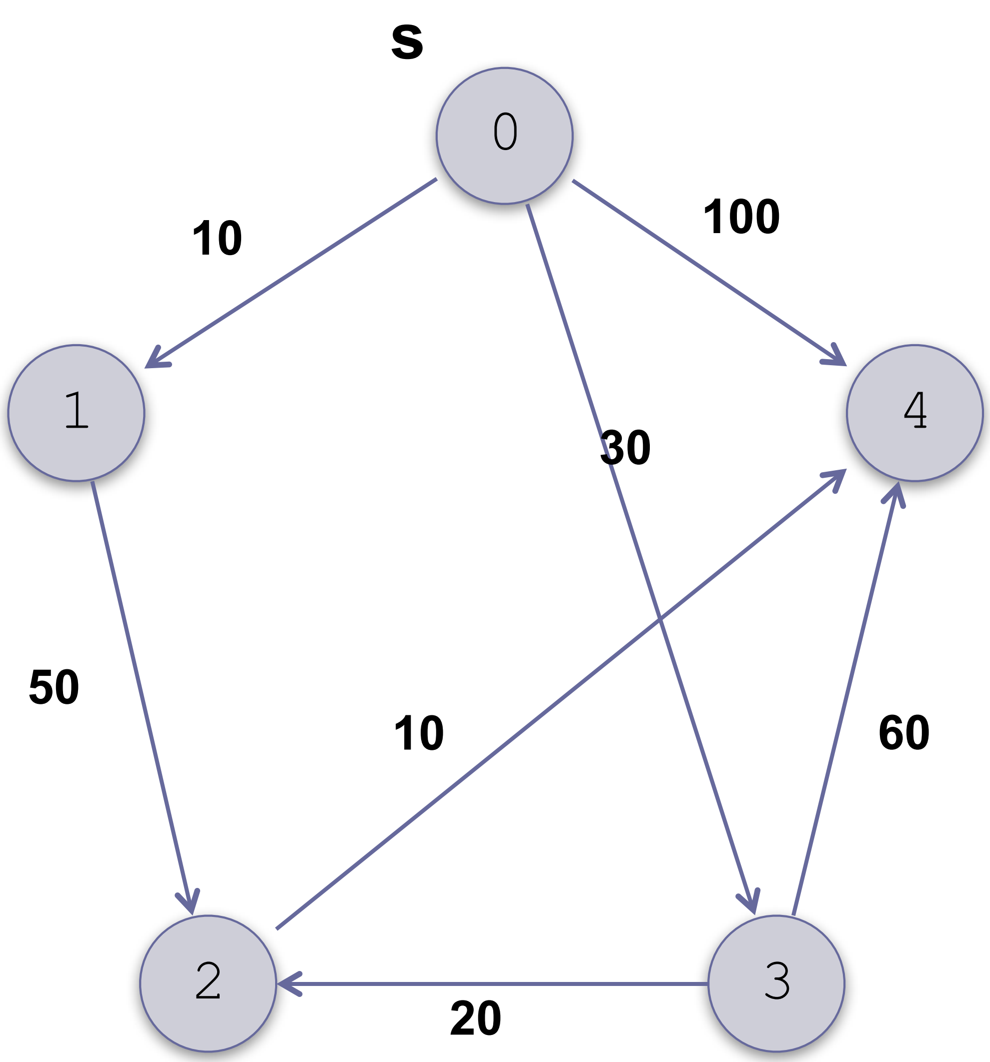
- vi börjar med att sätta **p**[v] till s för varje v i **V–S**

Dijkstra's Algorithm

S = { }

V-S = { }

v	d[v]	p[v]
1		
2		
3		
4		



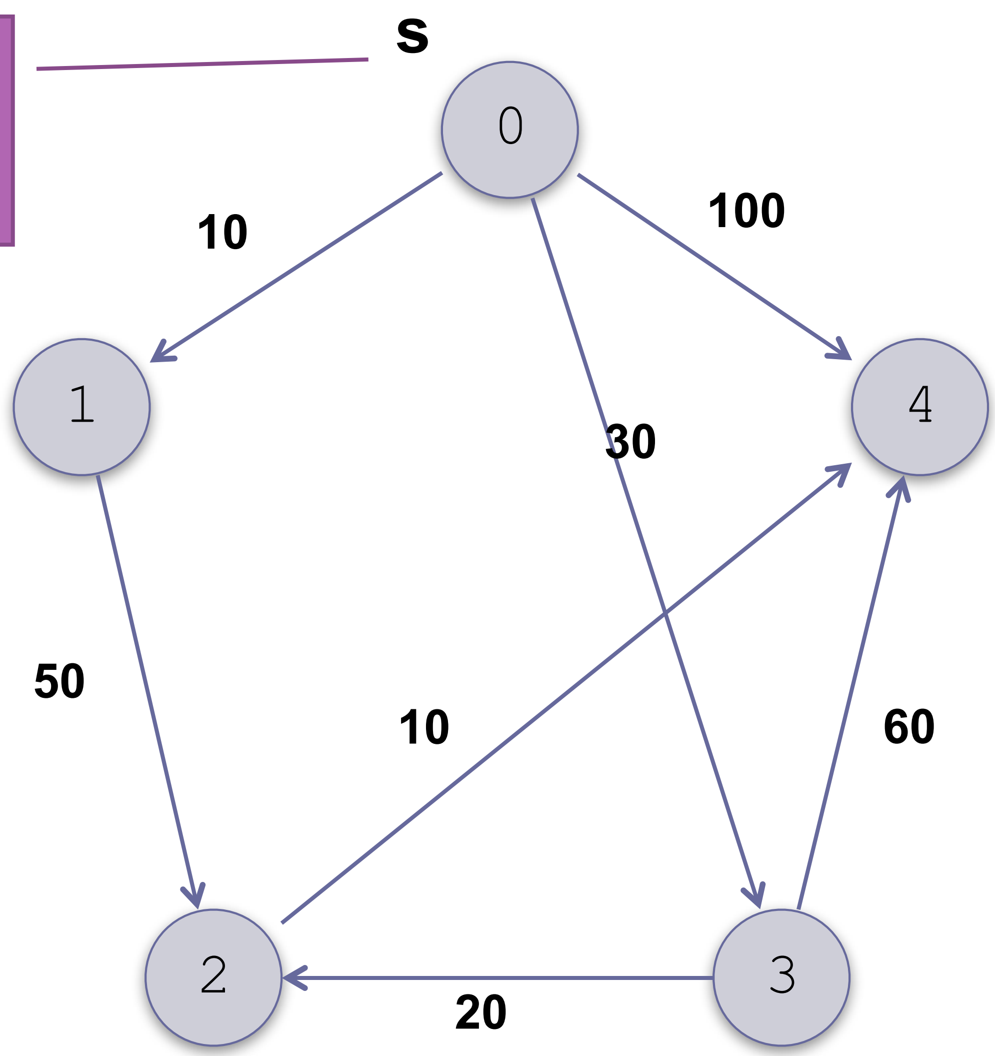
Dijkstra's Algorithm (cont.)

S = { }

V-S = { }

v	d[v]	p[v]
1		
2		
3		
4		

s is the start vertex



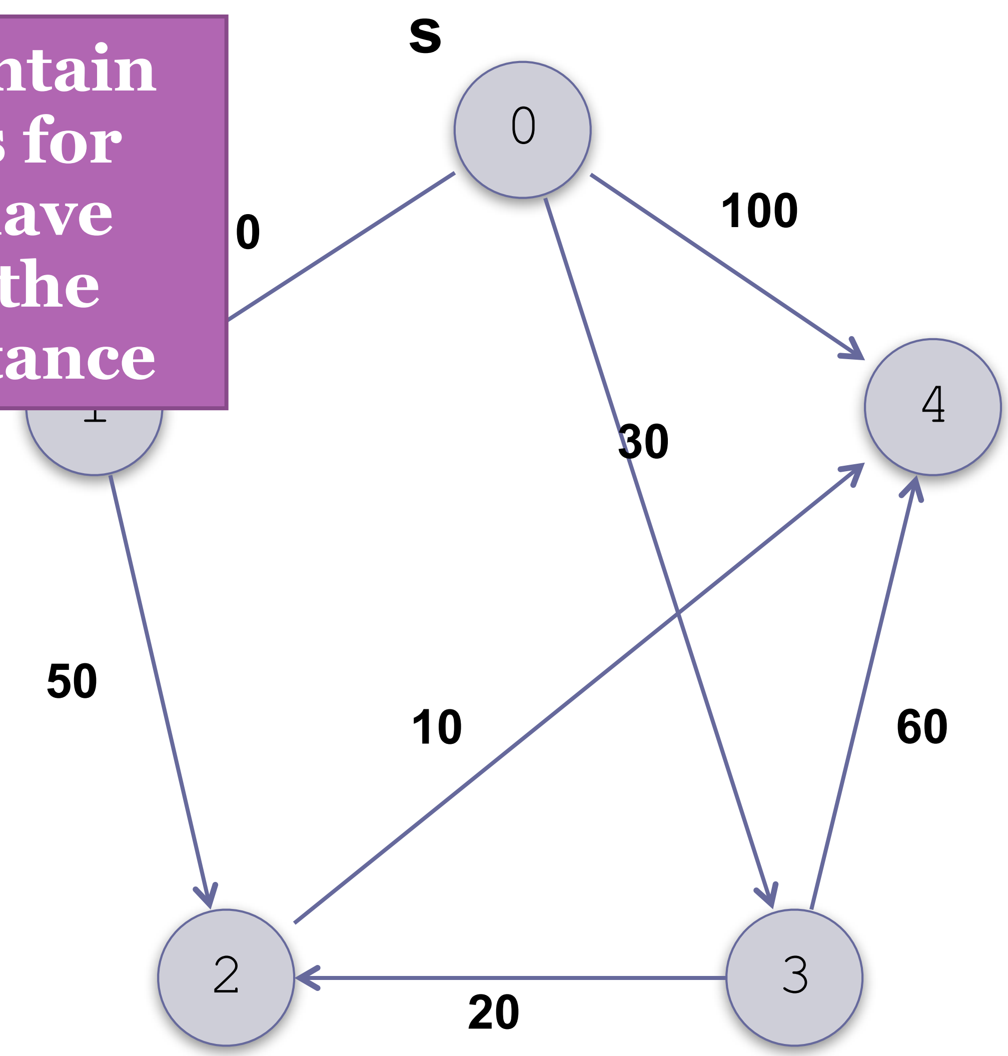
Dijkstra's Algorithm (cont.)

S = { }

V-*S* = { }

Set *S* will contain the vertices for which we have computed the shortest distance

v	d[*v*]	p[*v*]
1		
2		
3		
4		



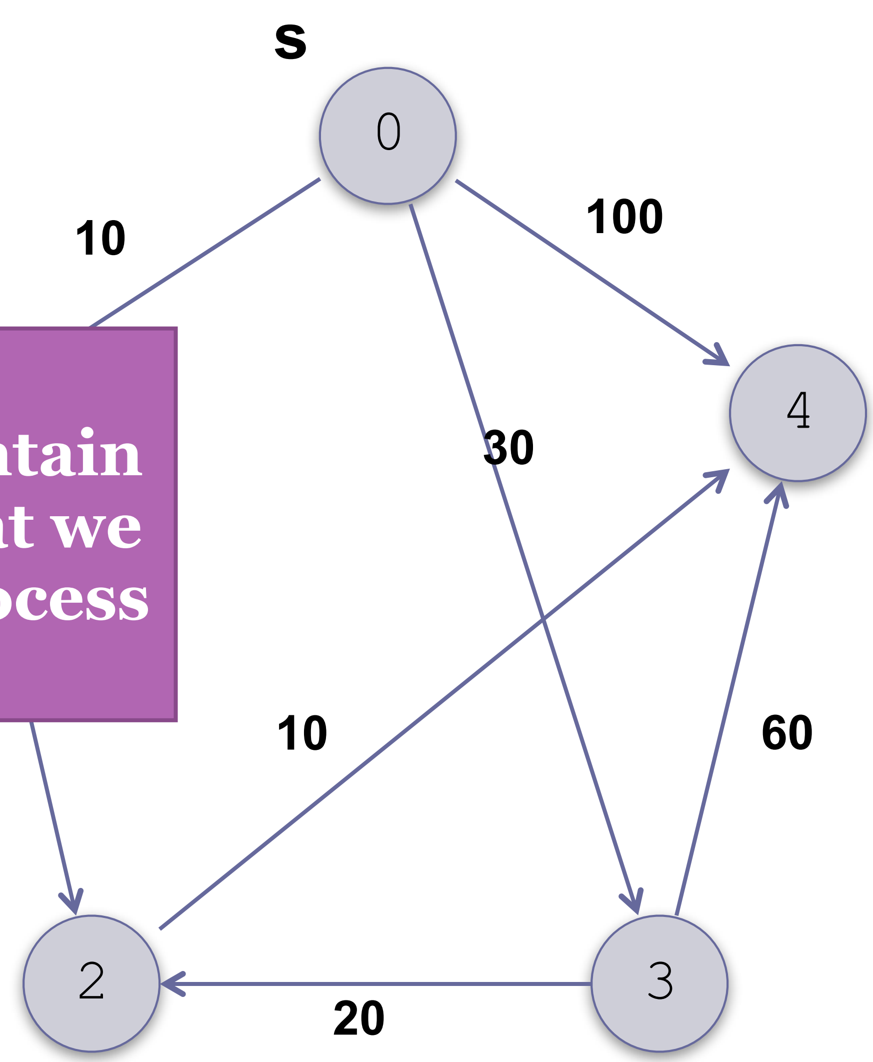
Dijkstra's Algorithm (cont.)

S = {}

V-*S* = { }

Set *V-S* will contain the vertices that we still need to process

v	d[*v*]	p[*v*]
1		
2		
3		
4		



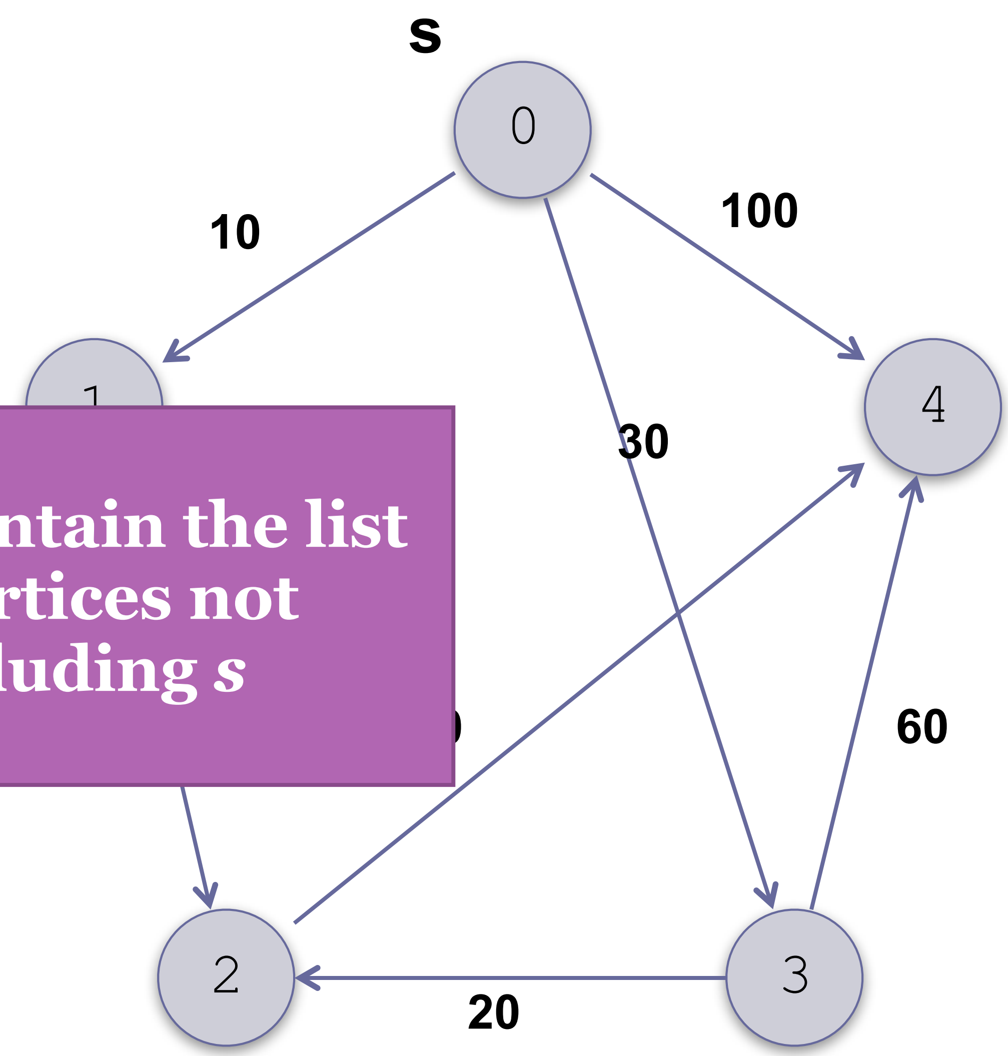
Dijkstra's Algorithm (cont.)

S = { }

V-S = { }

v	d[*v*]	p[*v*]
1		
2		
3		
4		

v **will contain the list of vertices not including *s***



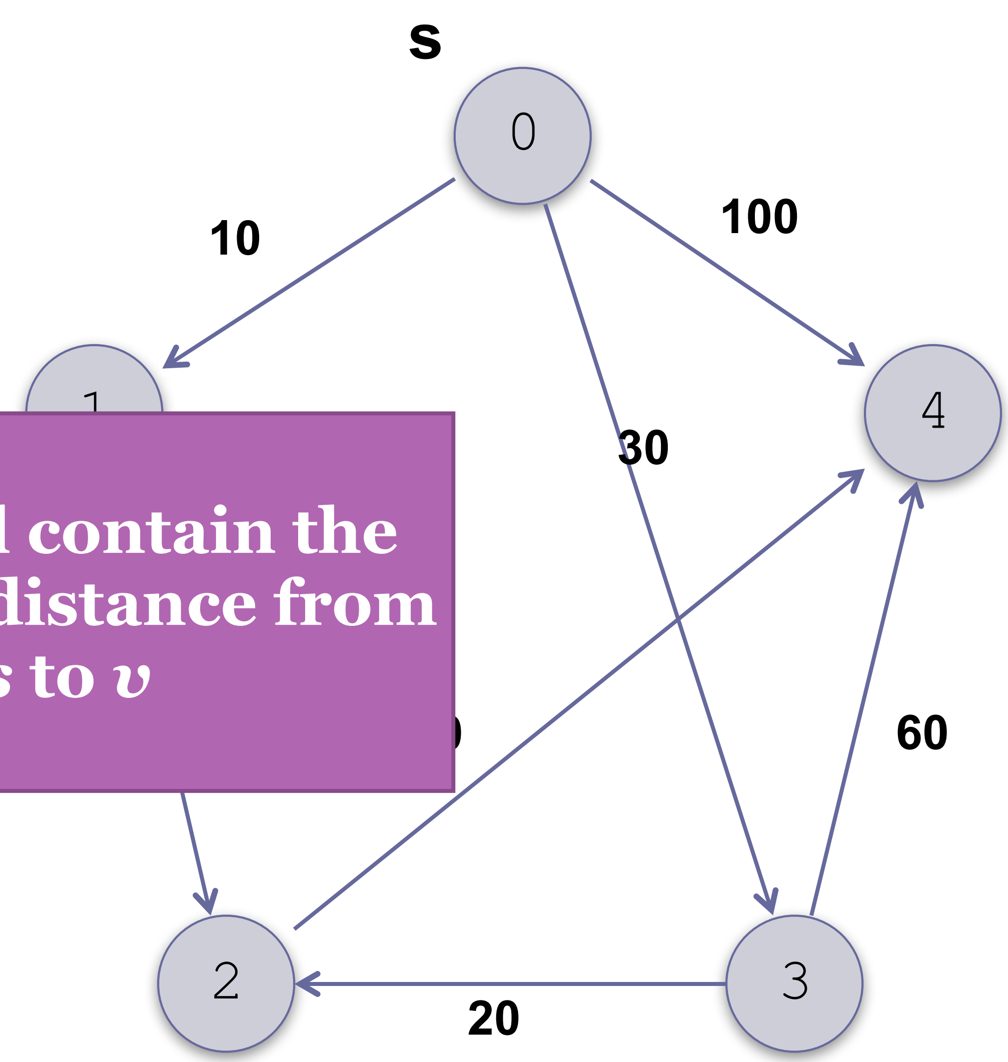
Dijkstra's Algorithm (cont.)

S = { }

V-*S* = { }

v	d[*v*]	p[*v*]
1		
2		
3		
4		

d[*v*] will contain the shortest distance from *s* to *v*



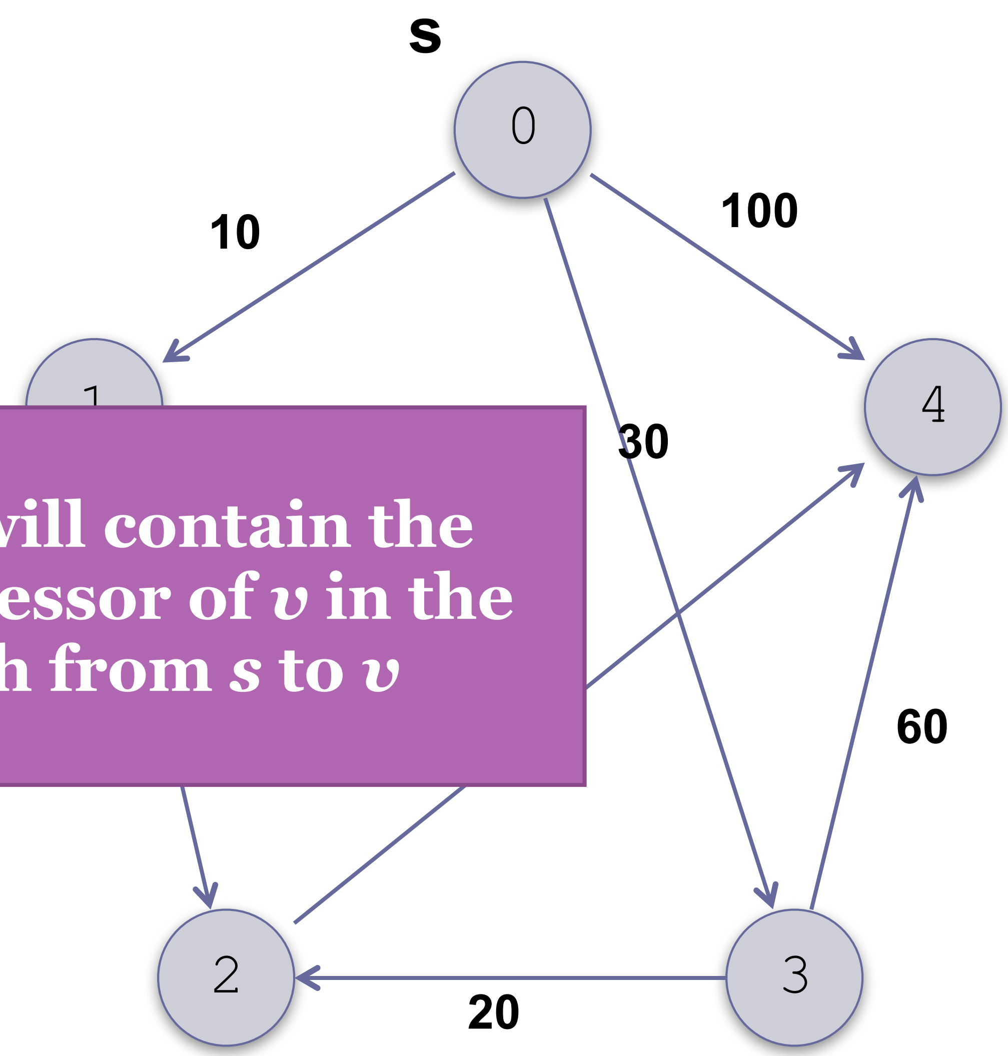
Dijkstra's Algorithm (cont.)

S = { }

V-*S* = { }

v	d[*v*]	p[*v*]
1		
2		
3		
4		

p[*v*] will contain the predecessor of *v* in the path from *s* to *v*

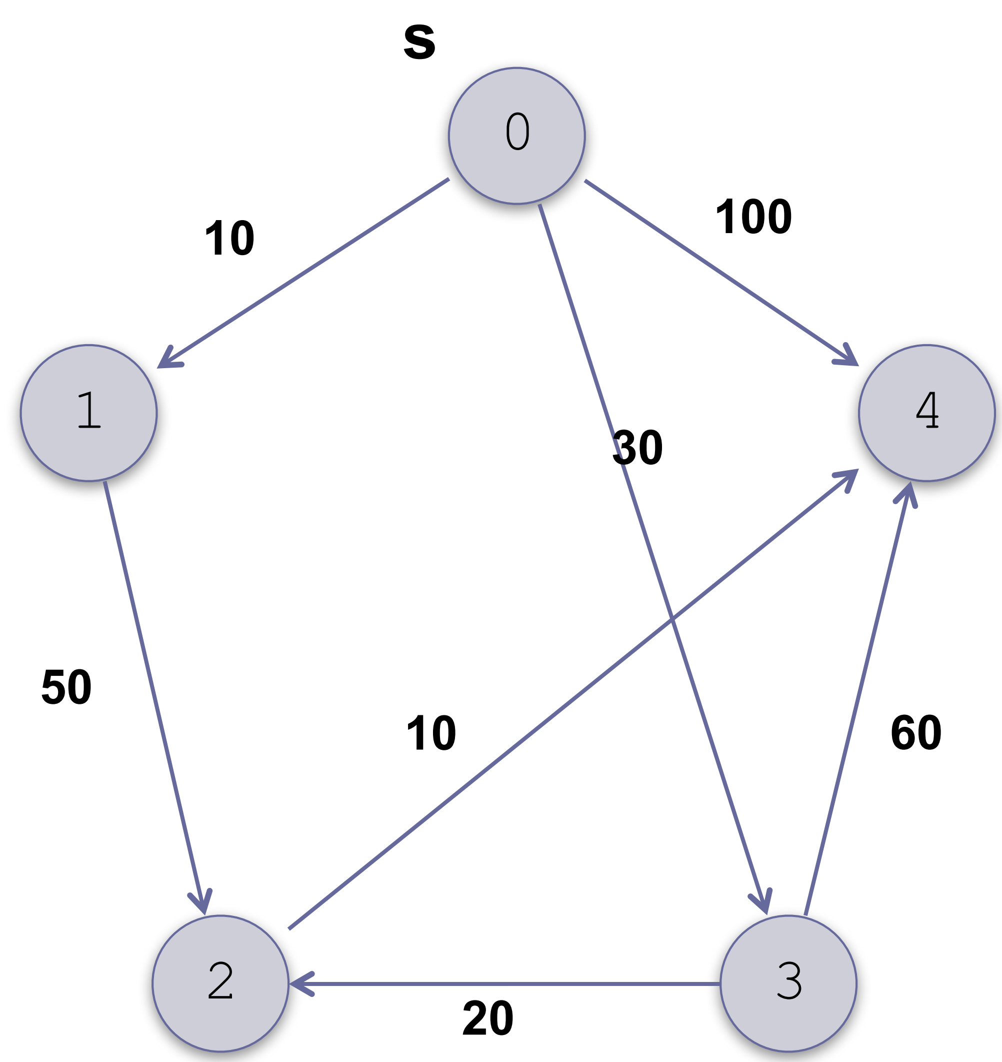


Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

v	d[*v*]	p[*v*]
1		
2		
3		
4		



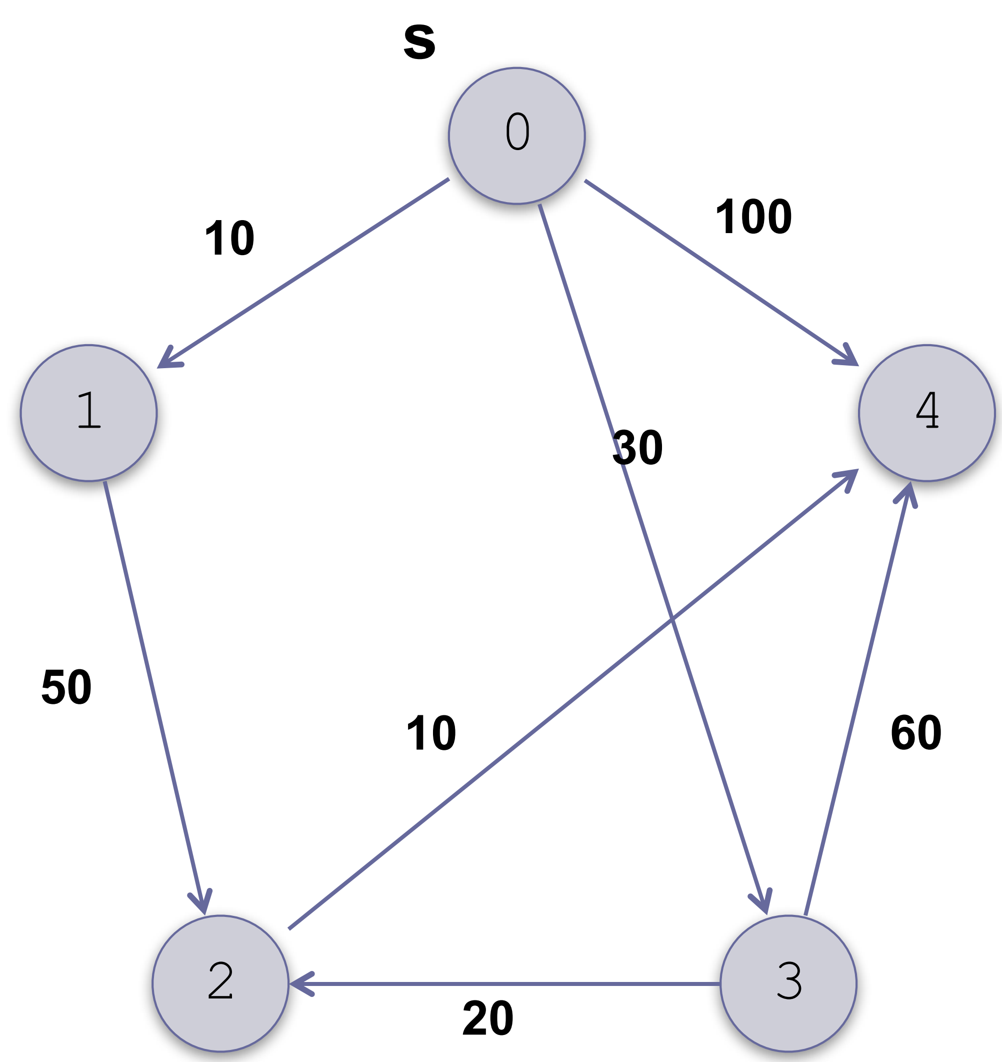
At initialization, the start vertex *s* is placed in *S*, and the remaining vertices into *V-S*

Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

v	d[*v*]	p[*v*]
1		
2		
3		
4		



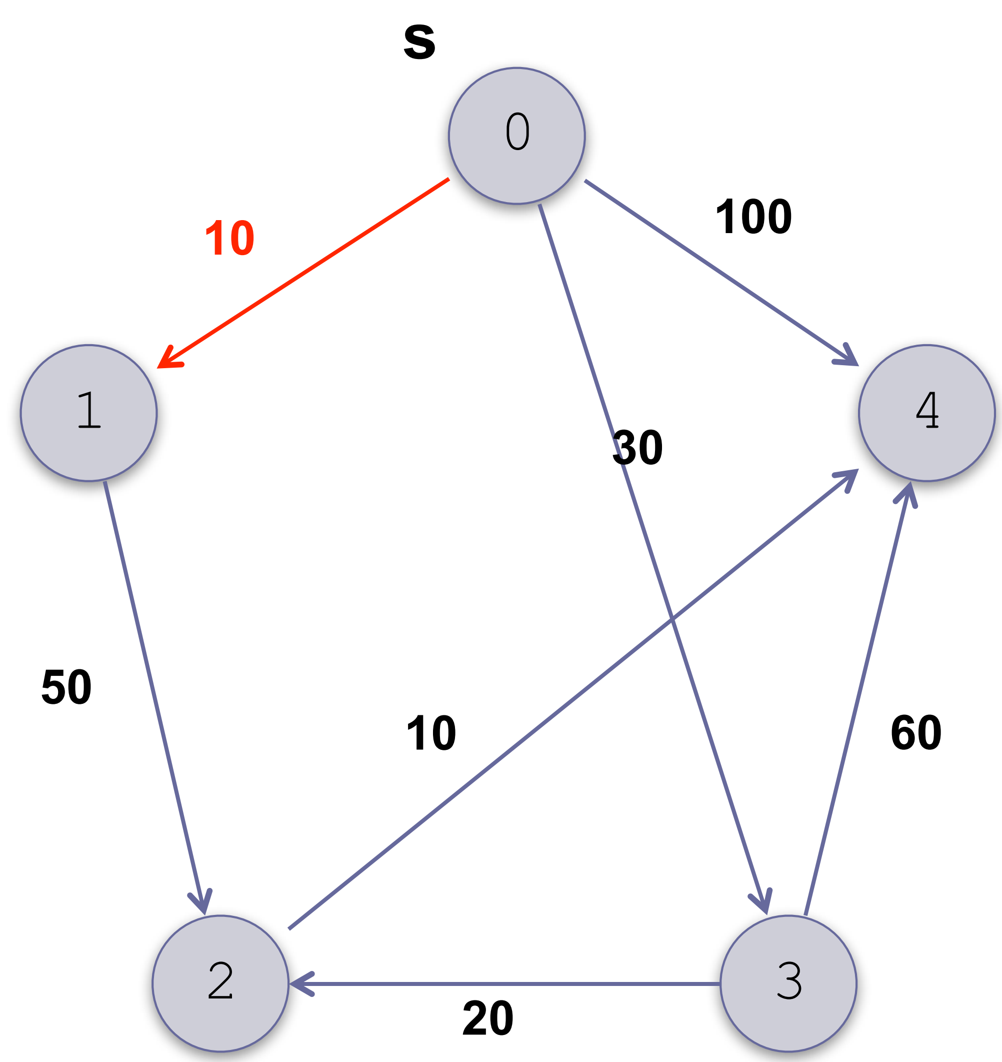
For each *v* in *V-S*, we initialize d by setting d[*v*] equal to the weight of the edge *w*(*s*, *v*) for each vertex *v*, adjacent to *s*

Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

v	d[*v*]	p[*v*]
1	10	
2		
3		
4		

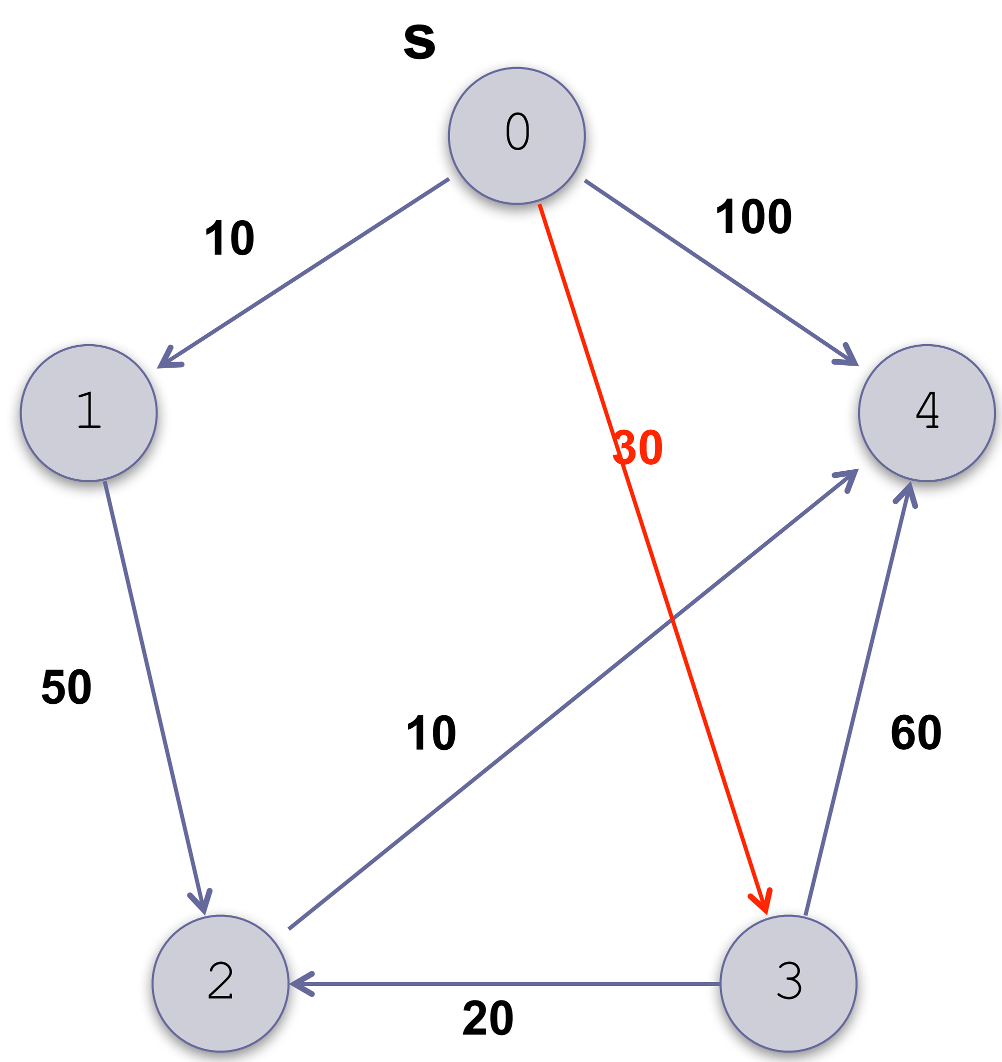


For each *v* in *V-S*, we initialize d by setting d[*v*] equal to the weight of the edge *w*(*s*, *v*) for each vertex *v*, adjacent to *s*

Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }



v	d[*v*]	p[*v*]
1	10	
2		
3	30	
4		

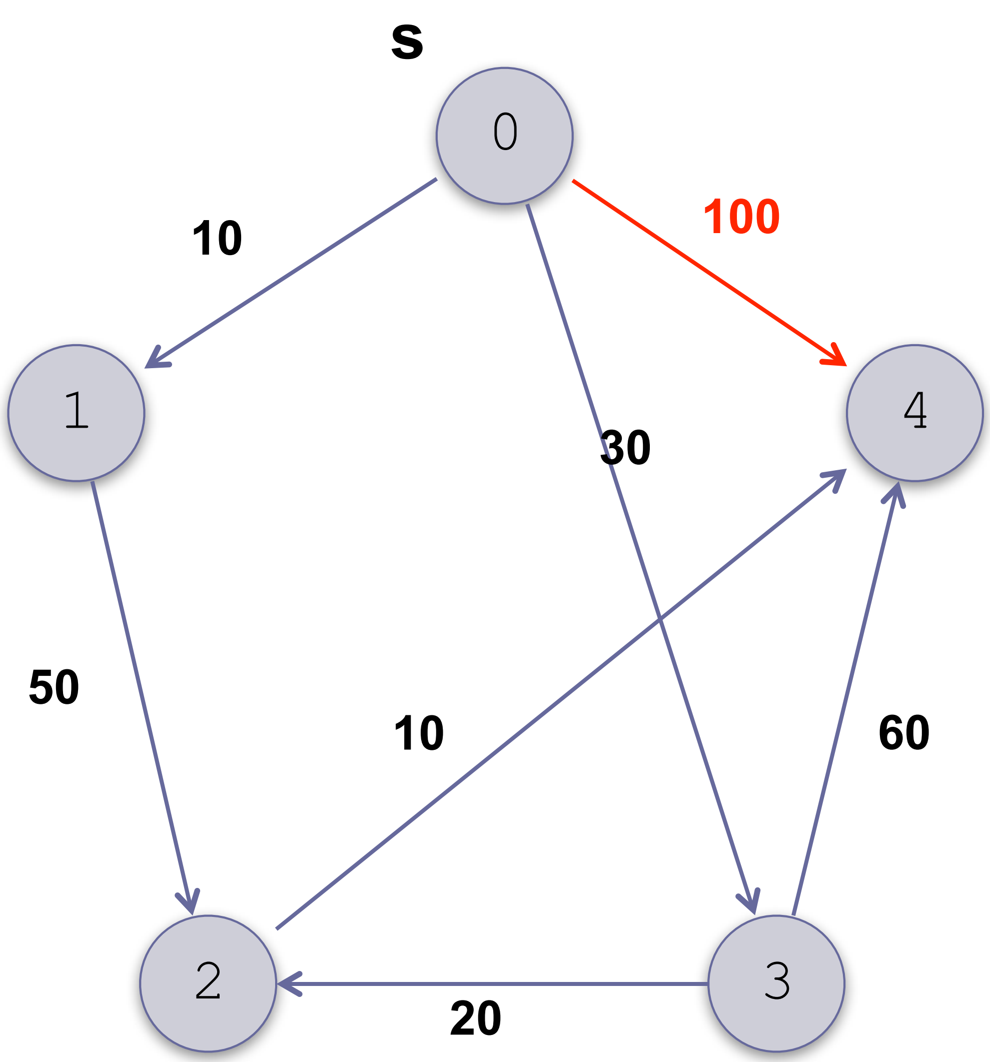
For each *v* in *V-S*, we initialize d by setting d[*v*] equal to the weight of the edge *w*(*s*, *v*) for each vertex *v*, adjacent to *s*

Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

v	d[*v*]	p[*v*]
1	10	
2		
3	30	
4	100	

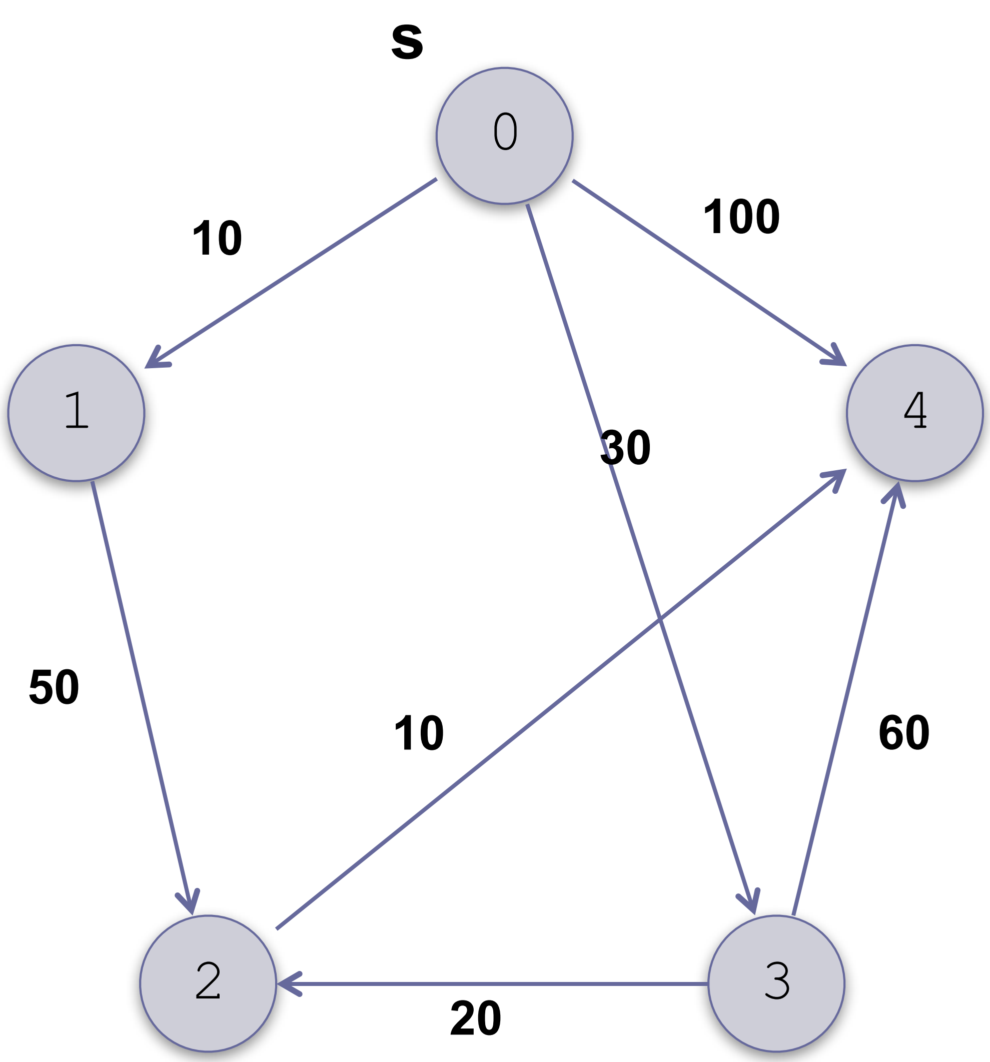


For each *v* in *V-S*, we initialize d by setting d[*v*] equal to the weight of the edge *w*(*s*, *v*) for each vertex *v*, adjacent to *s*

Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }



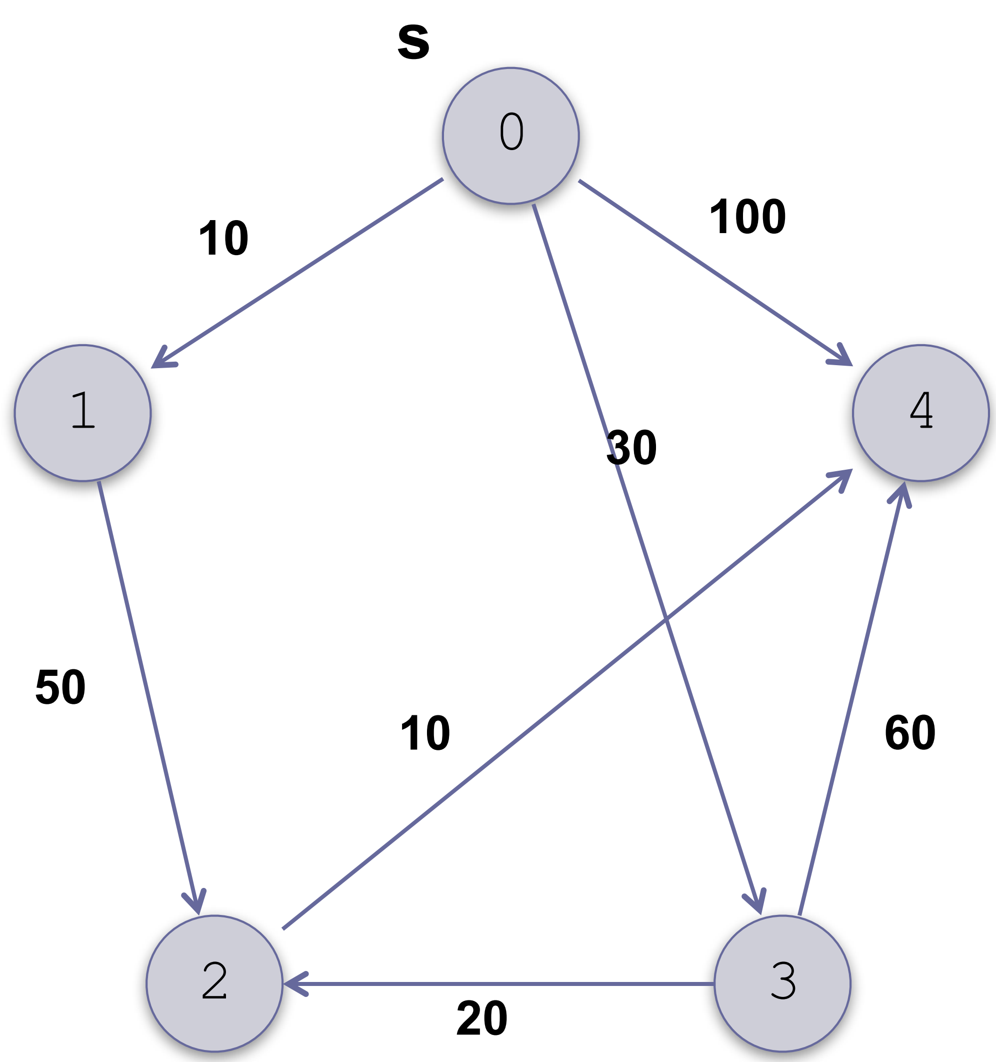
v	d[*v*]	p[*v*]
1	10	
2		
3	30	
4	100	

For each *v* not adjacent to *s*, we set d[*v*] equal to ∞

Dijkstra's Algorithm (cont.)

S = { 0 }

V-*S* = { 1, 2, 3, 4 }



v	d[*v*]	p[*v*]
1	10	
2	∞	
3	30	
4	100	

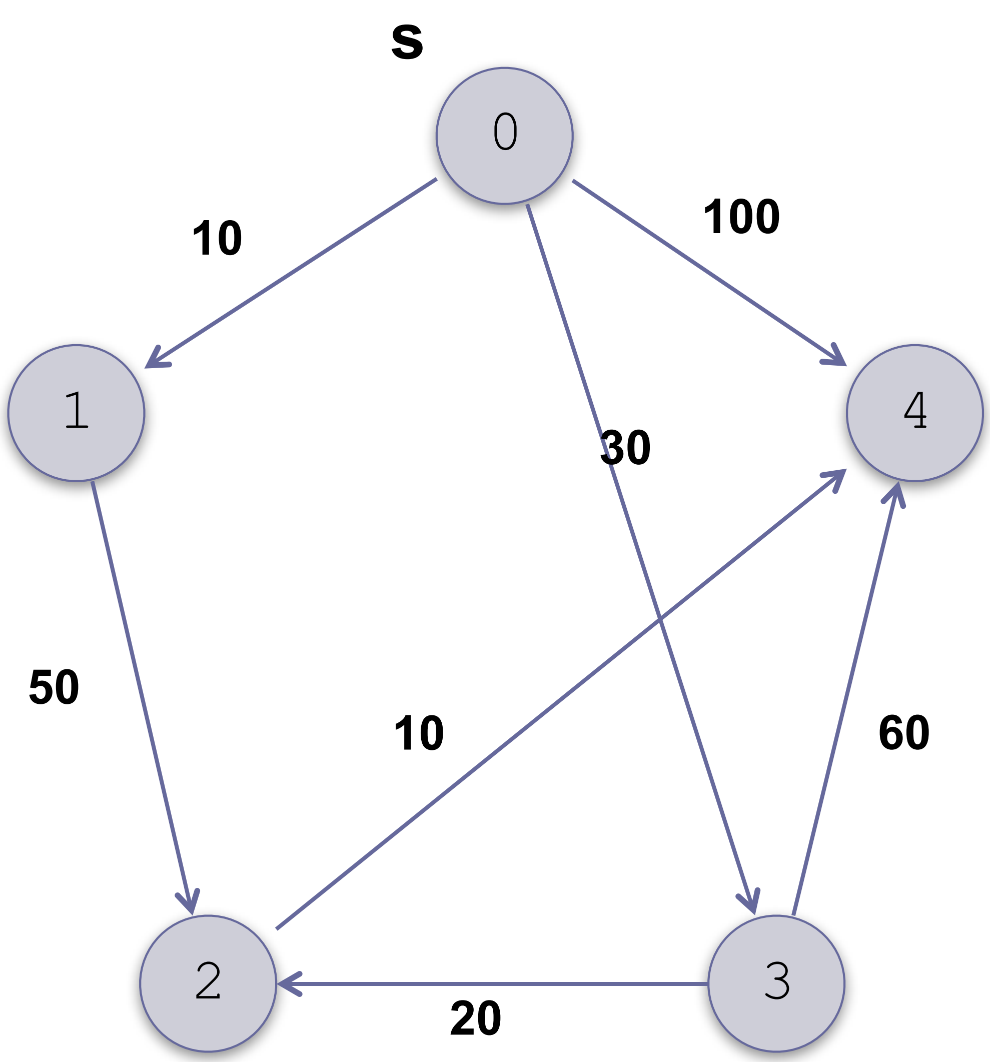
For each *v* not adjacent to *s*, we set d[*v*] equal to ∞

Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

v	d[v]	p[v]
1	10	
2	∞	
3	30	
4	100	

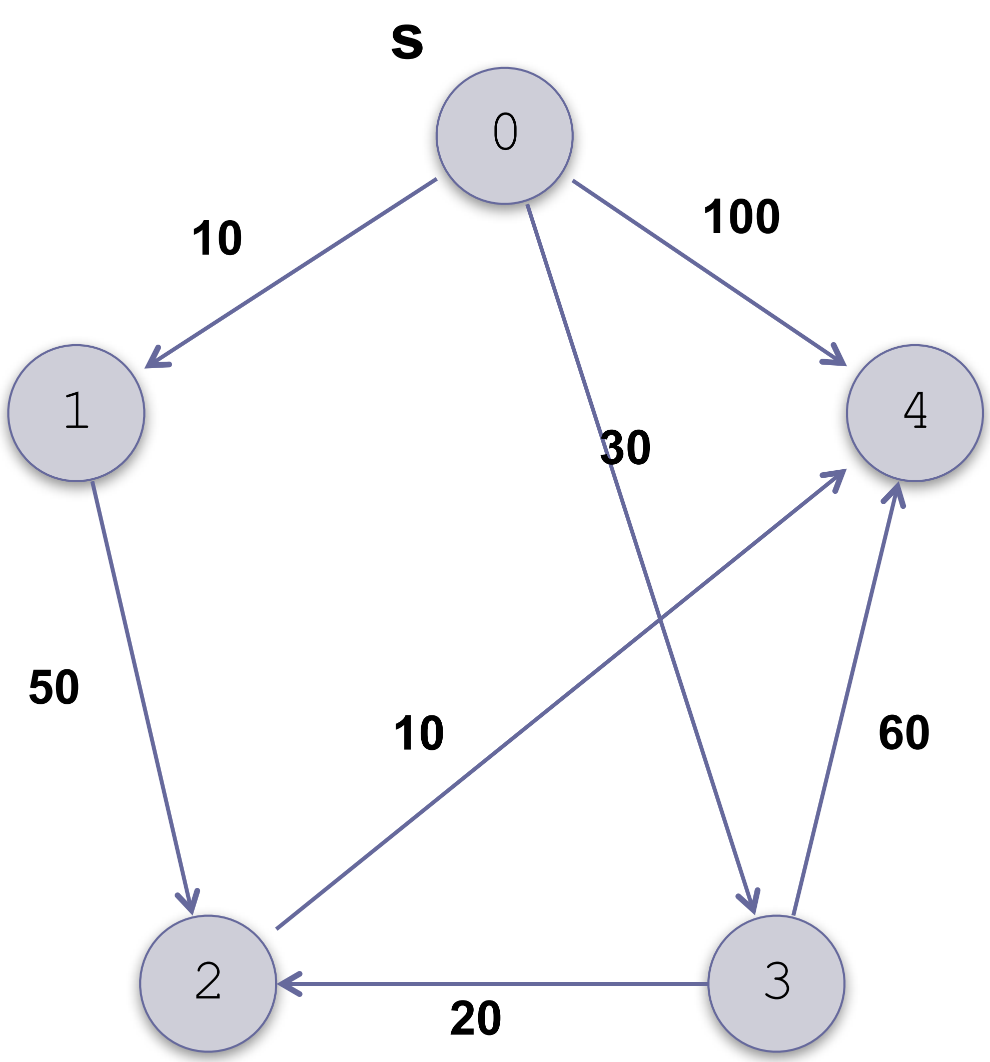


We initialize each p[v] to s

Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }



v	d[v]	p[v]
1	10	0
2	∞	0
3	30	0
4	100	0

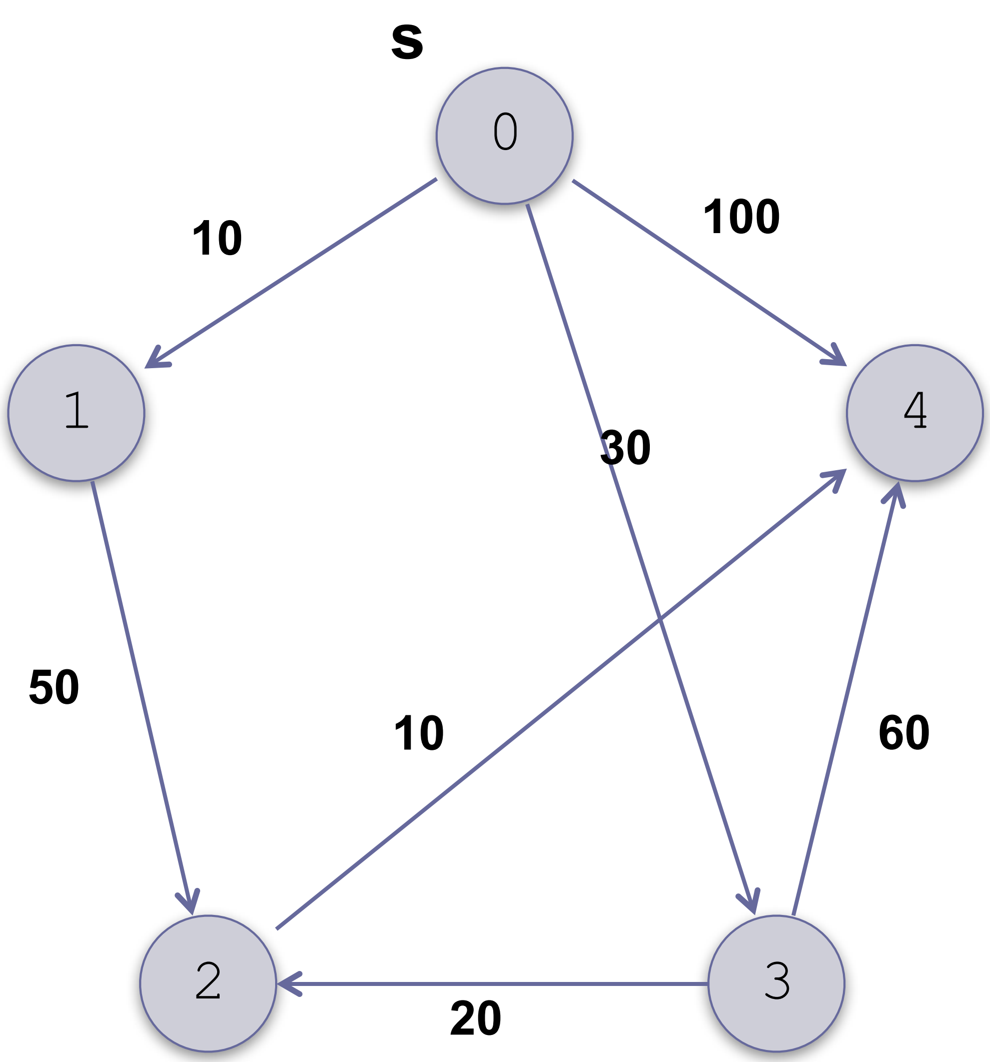
We initialize each p[v] to *s*

Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

v	d[*v*]	p[*v*]
1	10	0
2	∞	0
3	30	0
4	100	0



We now find the vertex *u* in *V-S* that has the smallest value of d[*u*]

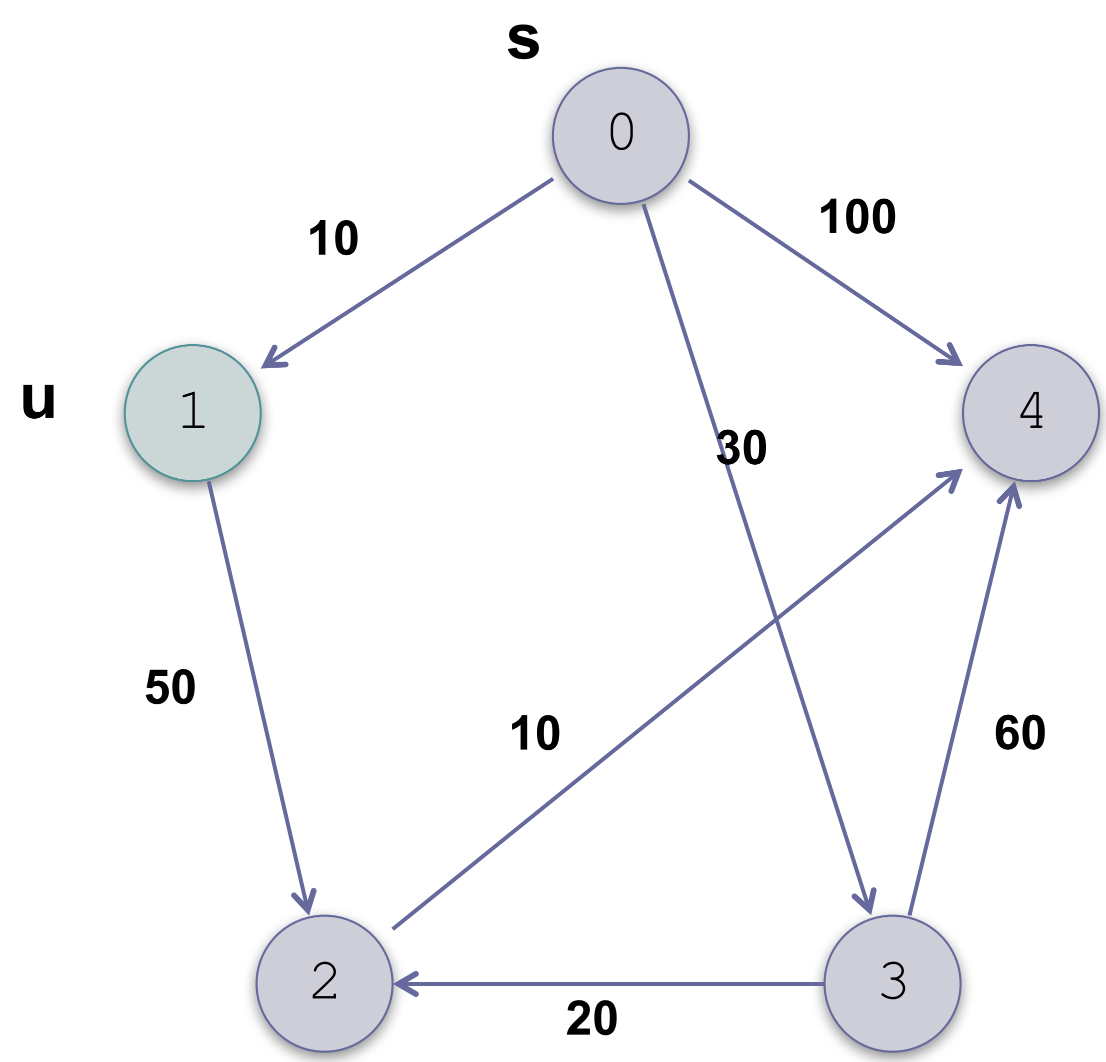
Dijkstra's Algorithm (cont.)

S = { 0 }

V-*S* = { 1, 2, 3, 4 }

u = 1

v	d[*v*]	p[*v*]
1	10	0
2	∞	0
3	30	0
4	100	0



We now find the vertex *u* in *V*-*S* that has the smallest value of d[*u*]

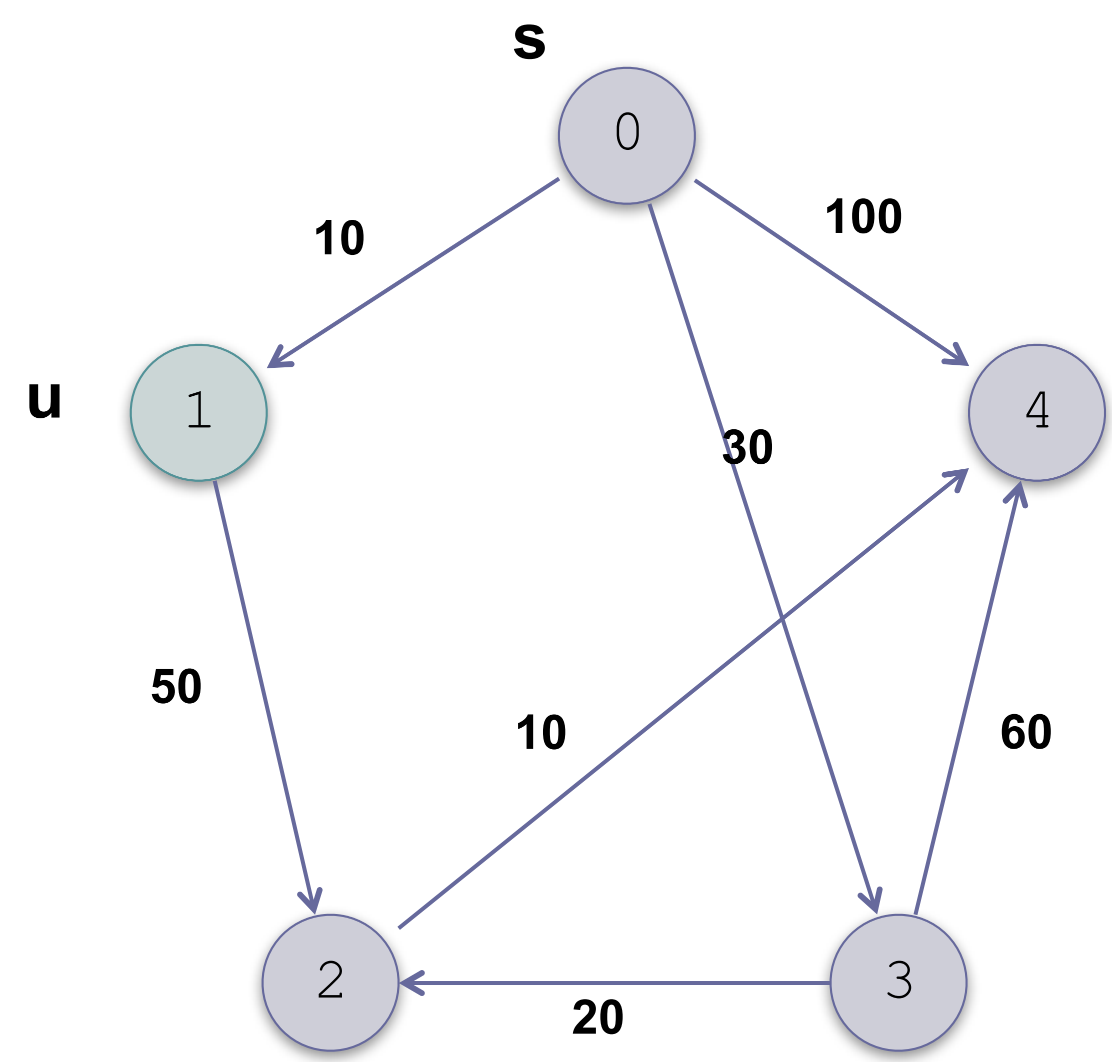
Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

u = 1

v	d[v]	p[v]
1	10	0
2	∞	0
3	30	0
4	100	0



Consider the vertices v that are adjacent to u

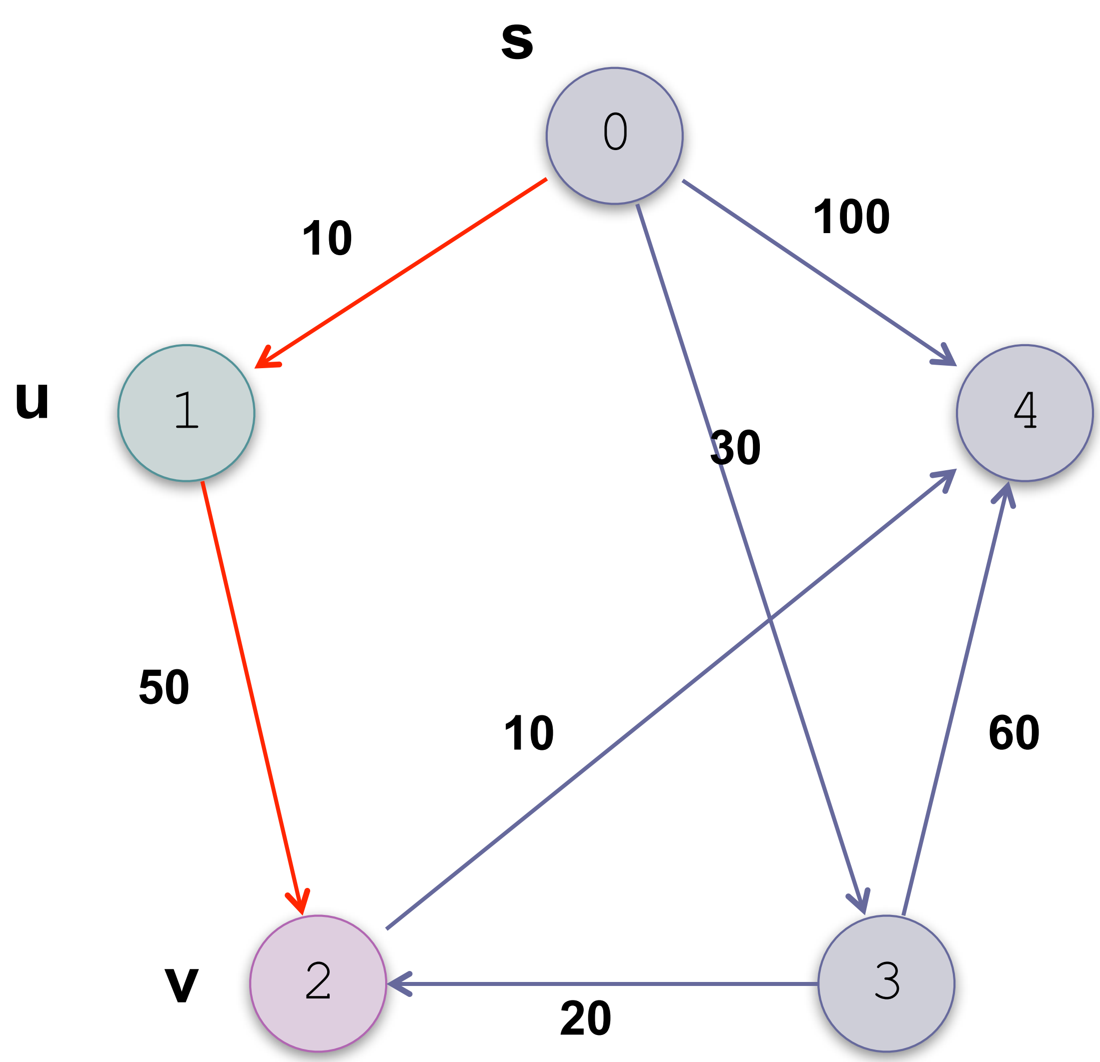
Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

u = 1

v	d[*v*]	p[*v*]
1	10	0
2	∞	0
3	30	0
4	100	0



If the distance from *s* to *u* (d[*u*]) plus the distance from u to v is smaller than d[*v*] we update d[*v*] to that value

Dijkstra's Algorithm (cont.)

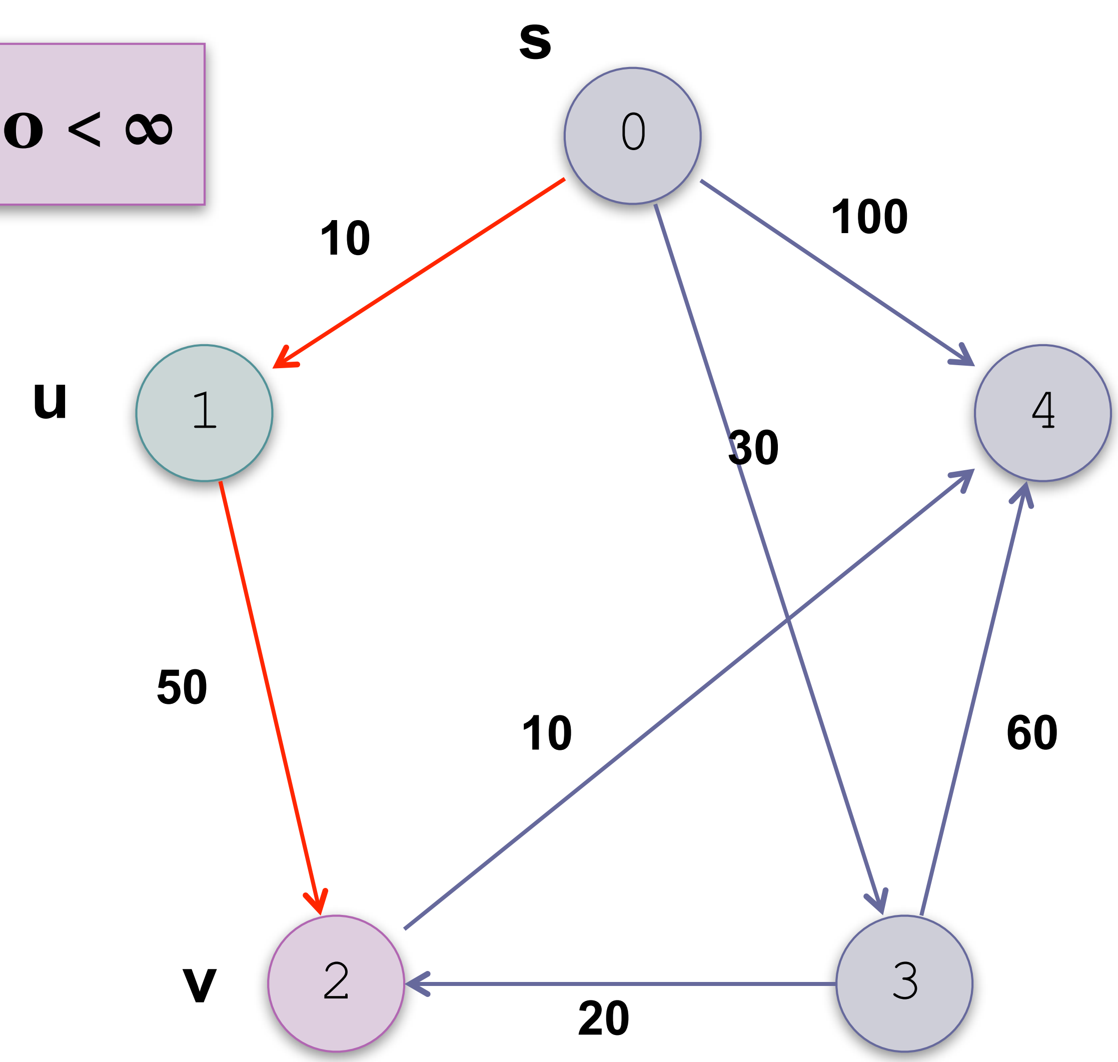
S = { 0 }

V-S = { 1, 2, 3, 4 }

u = 1

v	d[v]	p[v]
1	10	0
2	∞	0
3	30	0
4	100	0

60 < ∞



If the distance from s to u (d[u]) plus the distance from u to v is smaller than d[v] we update d[v] to that value

Dijkstra's Algorithm (cont.)

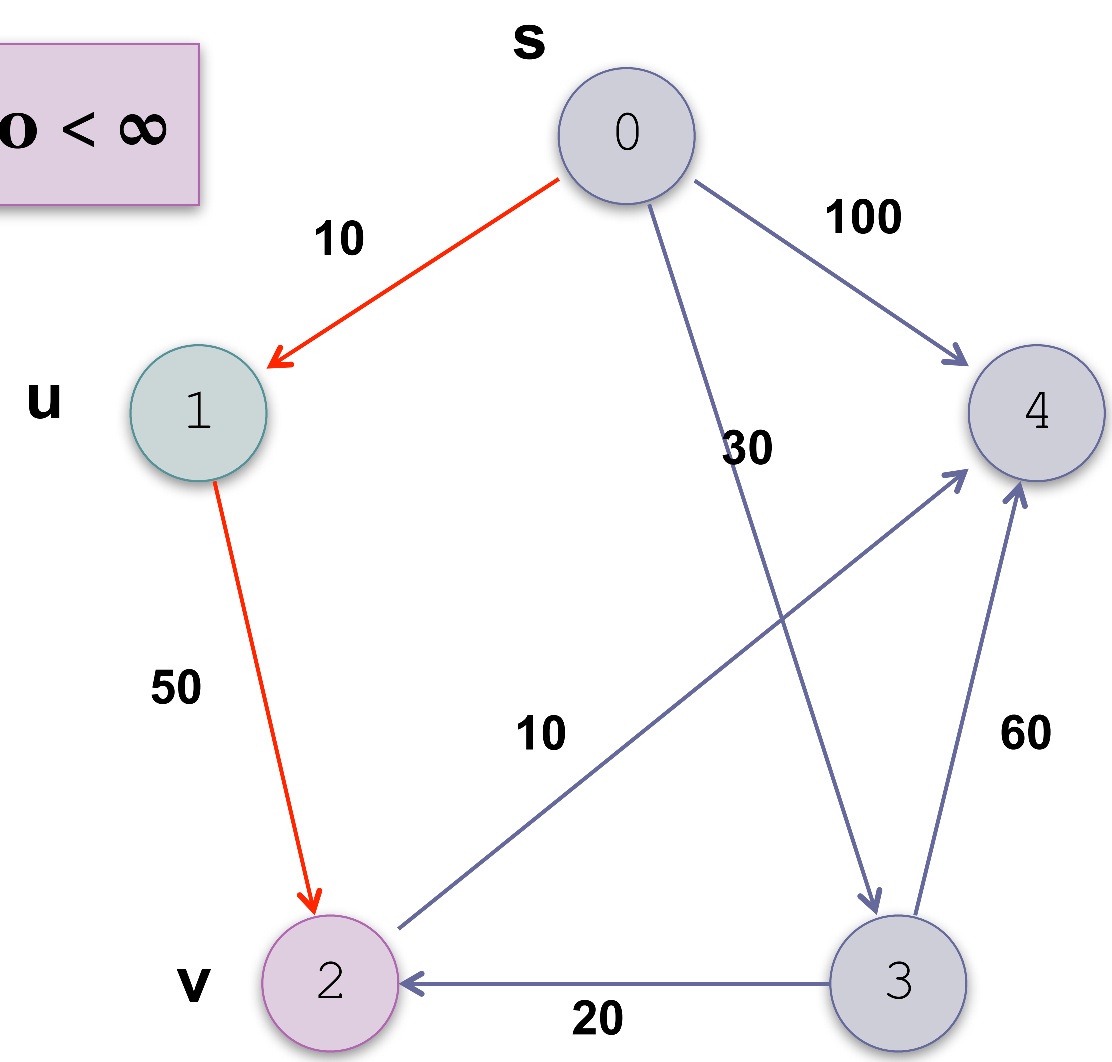
S = { 0 }

V-S = { 1, 2, 3, 4 }

u = 1

v	d[v]	p[v]
1	10	0
2	60	0
3	30	0
4	100	0

60 < ∞



If the distance from s to u (d[u]) plus the distance from u to v is smaller than d[v] we update d[v] to that value

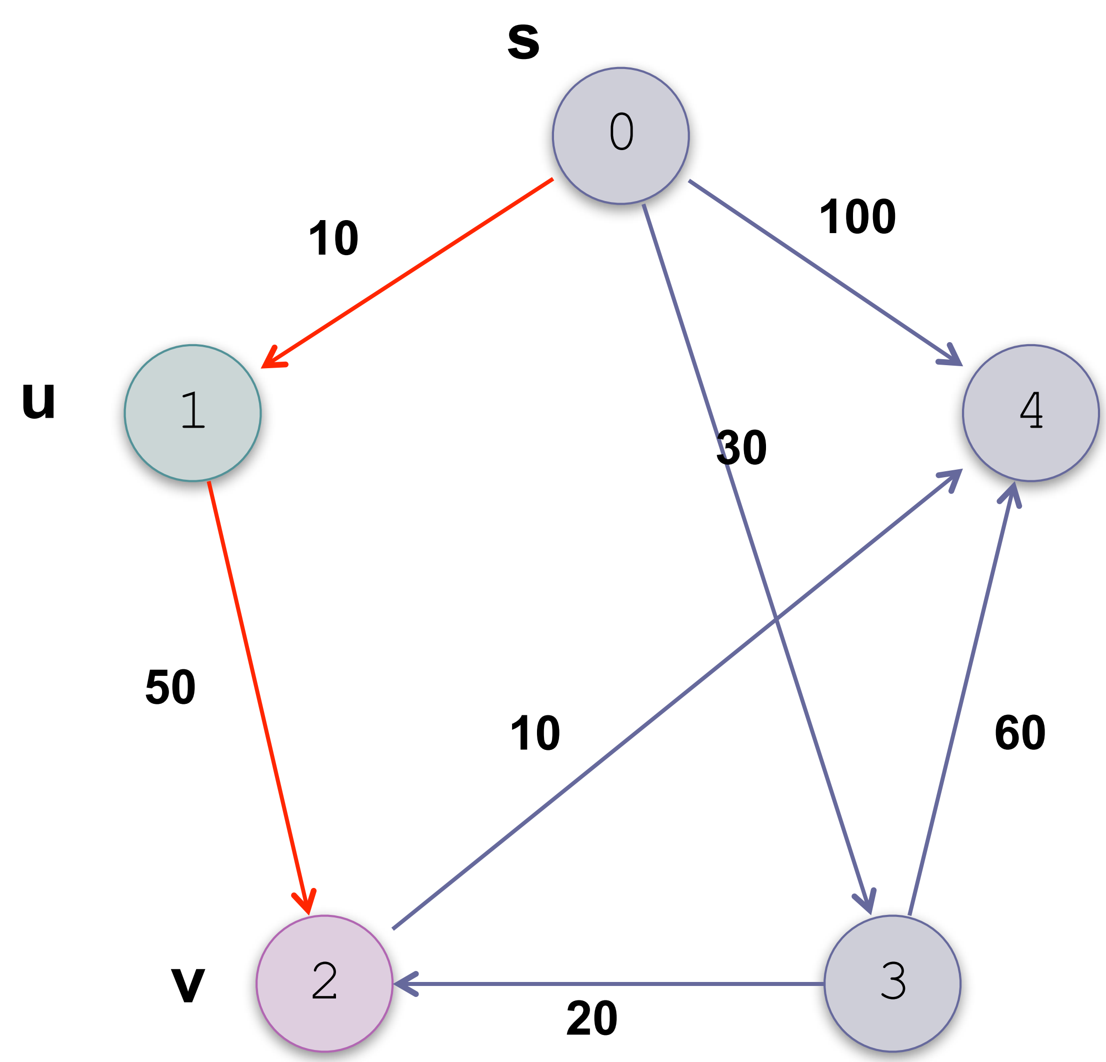
Dijkstra's Algorithm (cont.)

S = { 0 }

V-*S* = { 1, 2, 3, 4 }

u = 1

v	d[*v*]	p[*v*]
1	10	0
2	60	1
3	30	0
4	100	0



and set p[*v*] to *u*

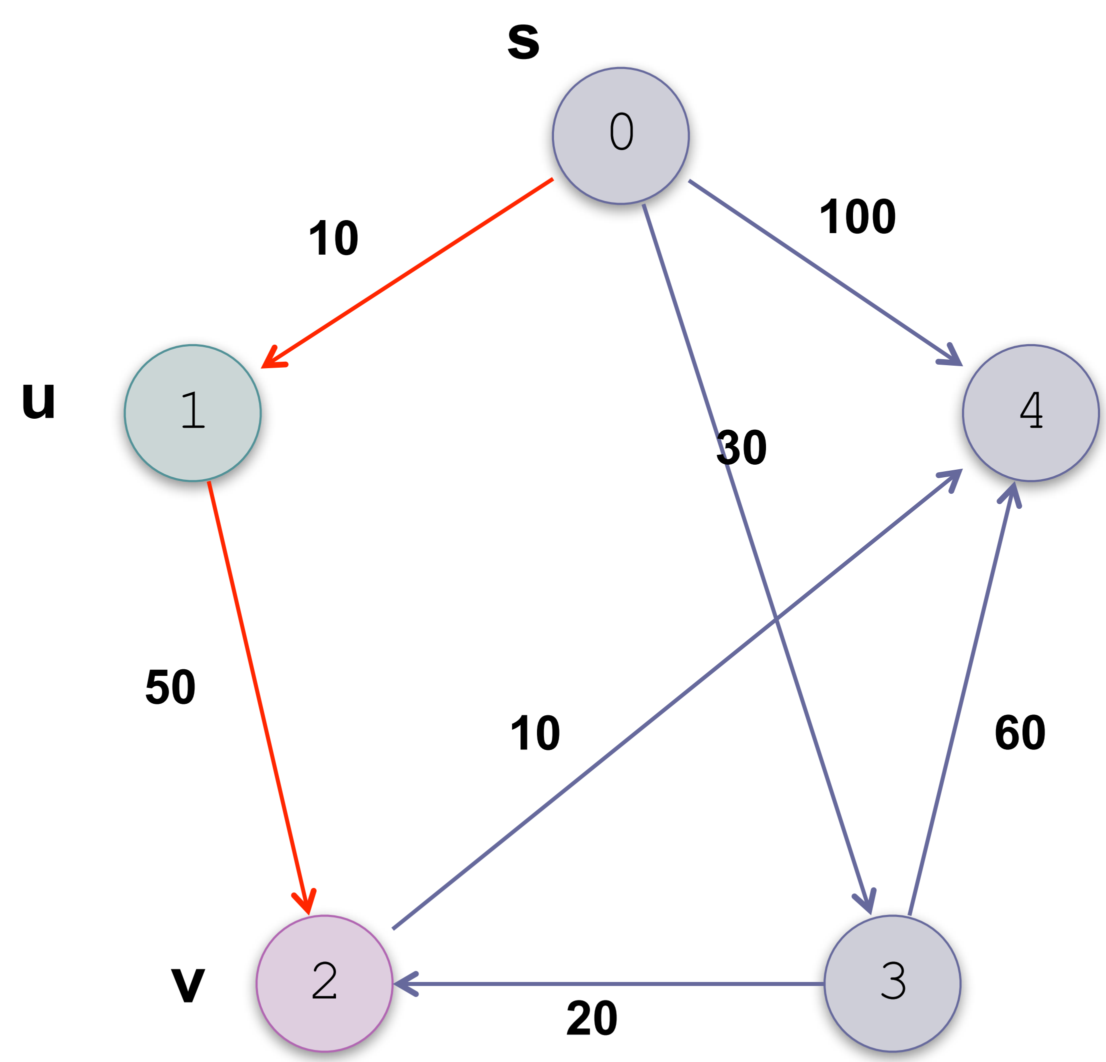
Dijkstra's Algorithm (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4 }

u = 1

v	d[*v*]	p[*v*]
1	10	0
2	60	1
3	30	0
4	100	0



Remove *u* from *V-S* and place it in *S*

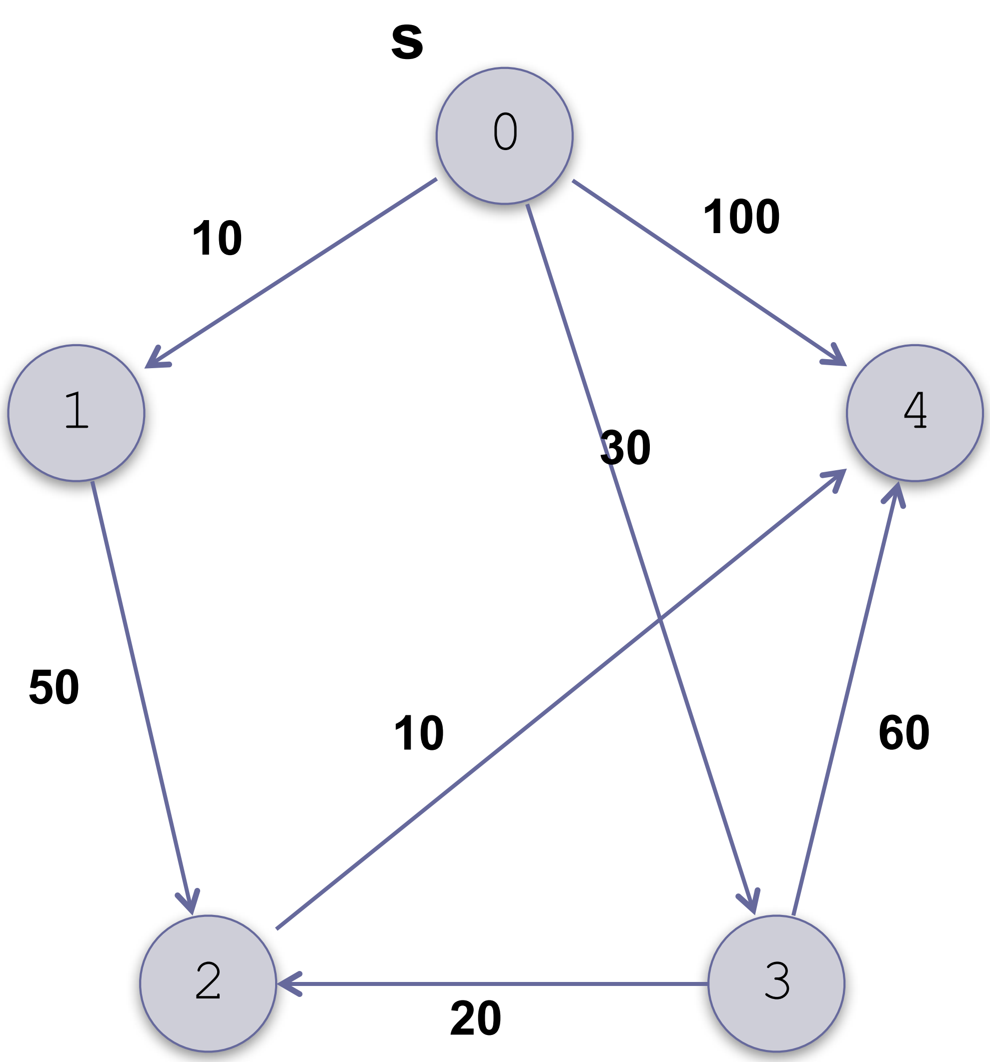
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-*S* = { 2, 3, 4 }

u = 1

v	d[*v*]	p[*v*]
1	10	0
2	60	1
3	30	0
4	100	0



Repeat until *V*-*S* is empty

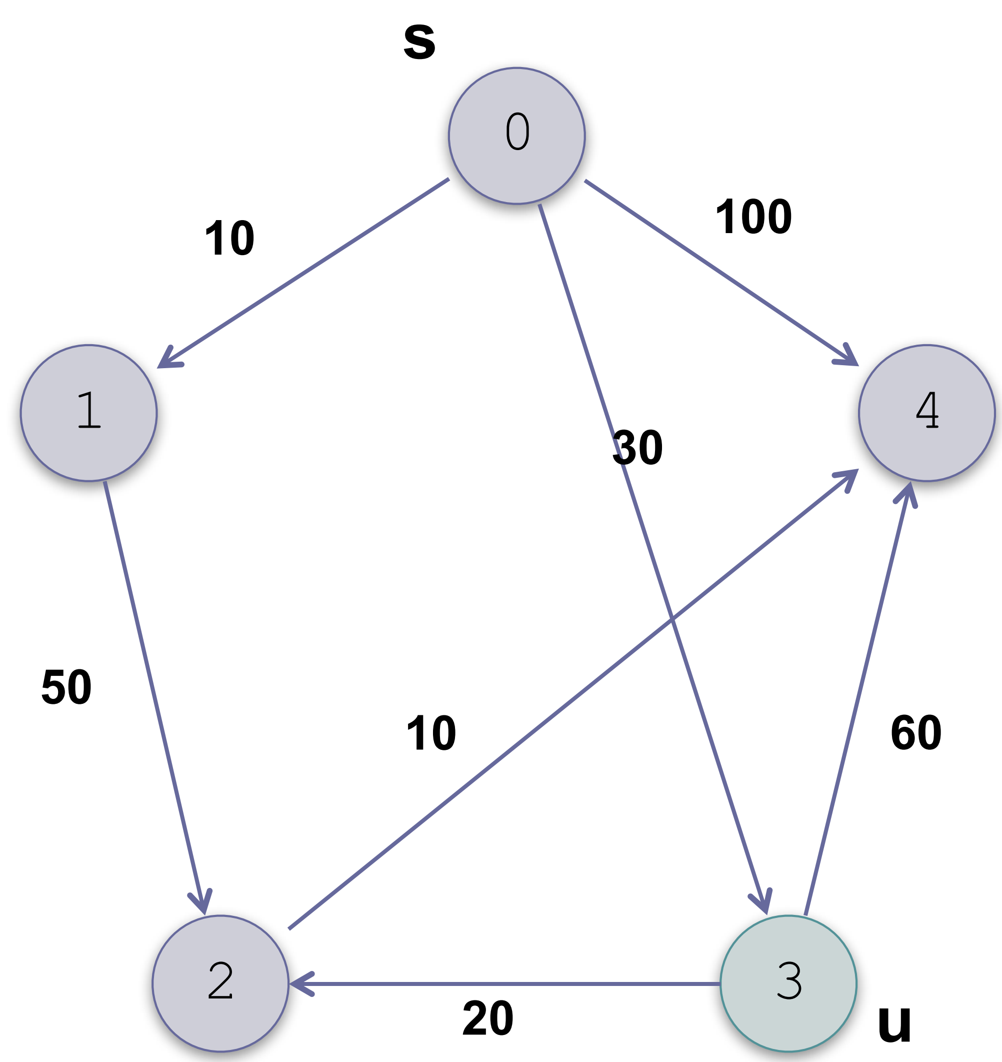
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-*S* = { 2, 3, 4 }

u = 3

v	d[*v*]	p[*v*]
1	10	0
2	60	1
3	30	0
4	100	0



The smallest d[*v*] in *V*-*S* is vertex 3

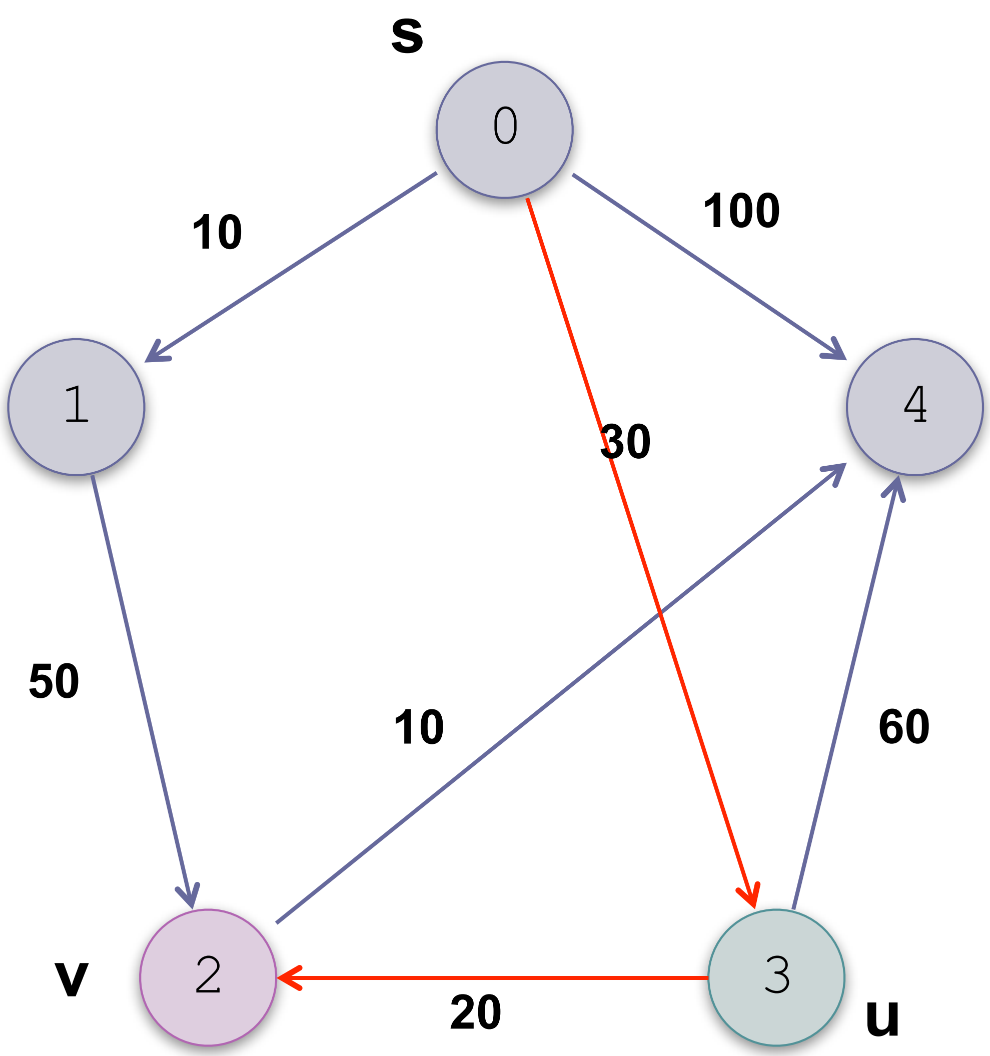
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-*S* = { 2, 3, 4 }

u = 3

v	d[*v*]	p[*v*]
1	10	0
2	60	1
3	30	0
4	100	0



The distance from *s* to *u* plus the distance from *u* to *v* is 50

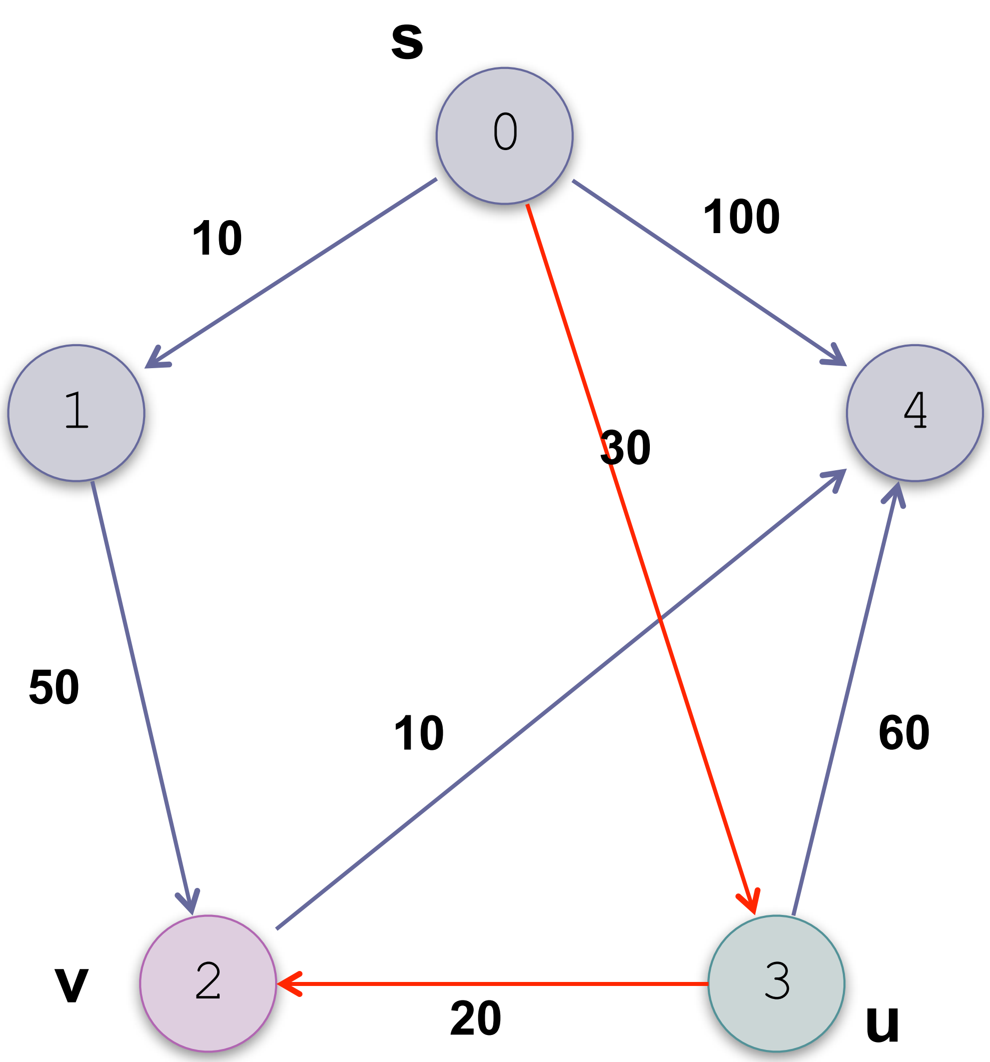
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-*S* = { 2, 3, 4 }

u = 3

v	d[*v*]	p[*v*]
1	10	0
2	60	1
3	30	0
4	100	0



50 < d[2] (which is 60)

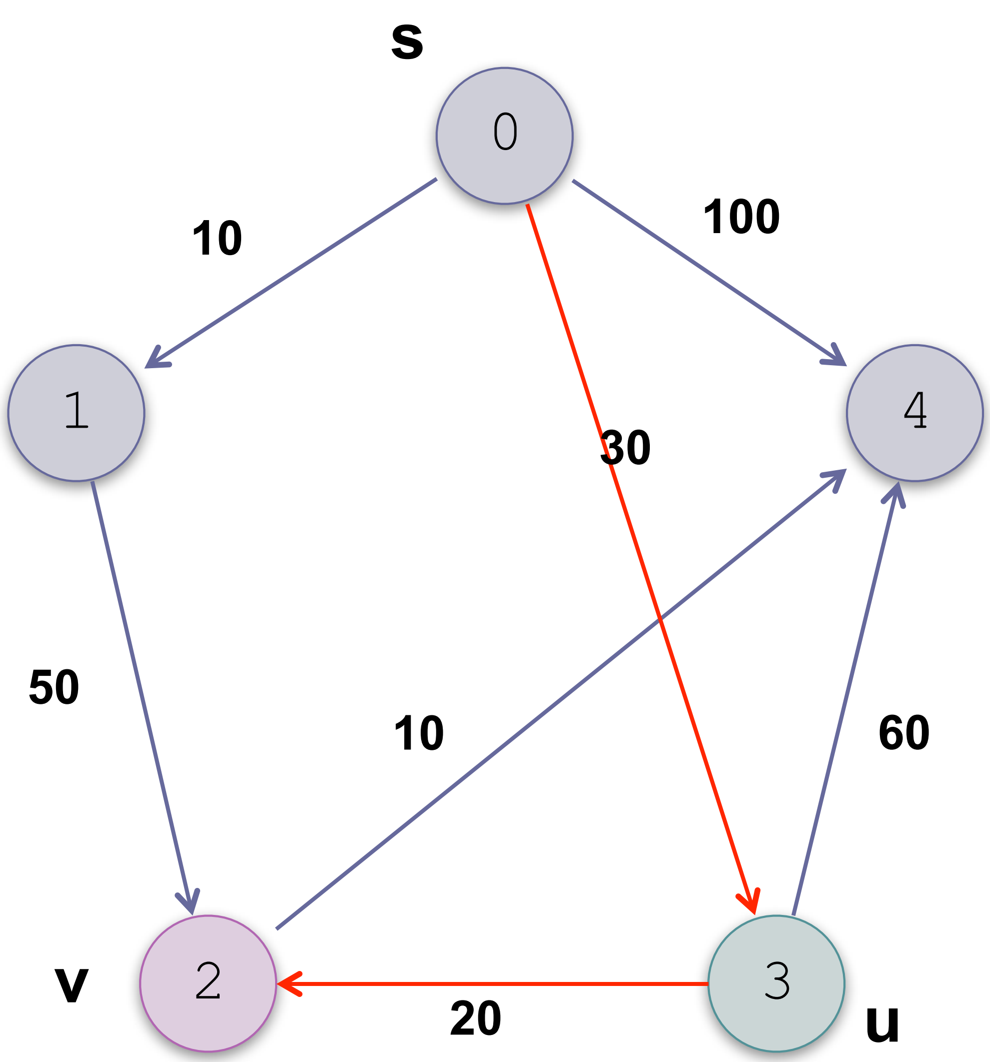
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-*S* = { 2, 3, 4 }

u = 3

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	100	0



Set d[2] to 50 and p[2] to *u*

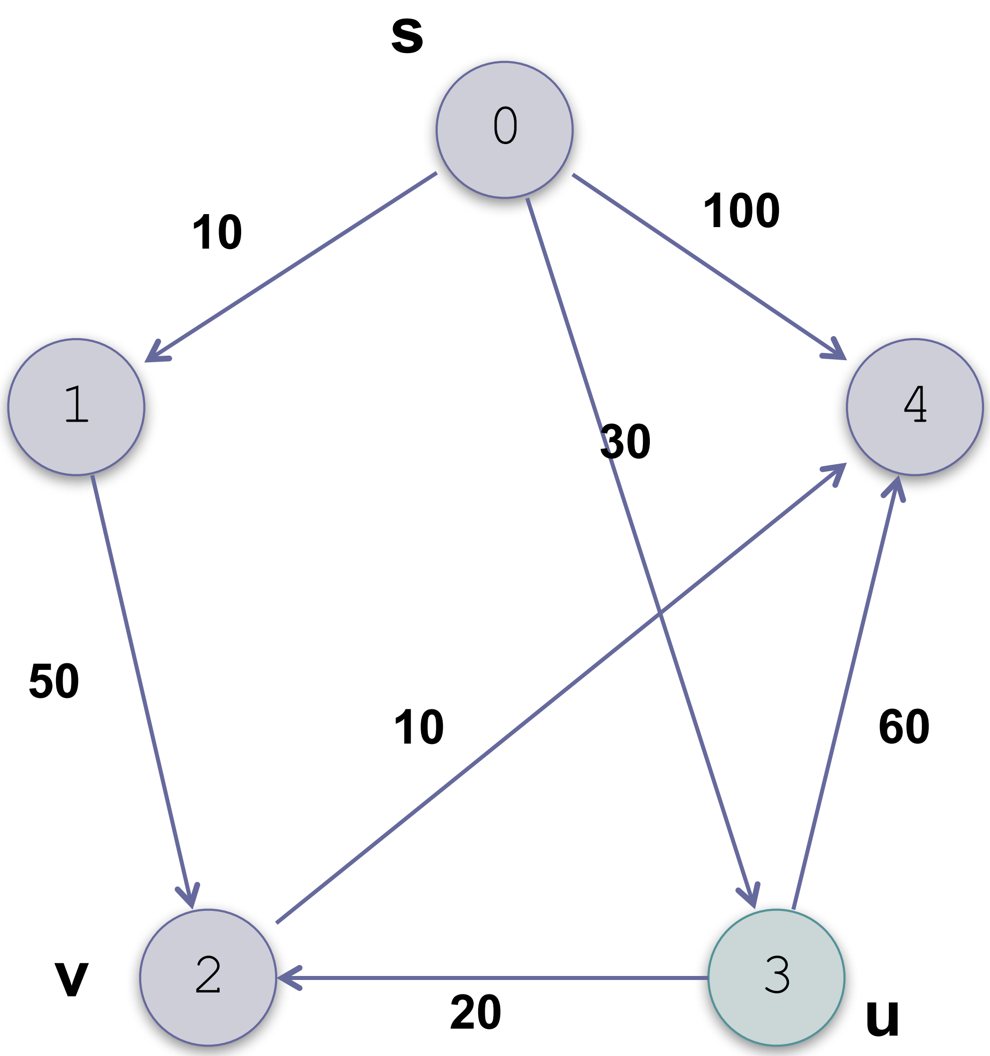
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 3

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	100	0



Continue to the next adjacent vertex

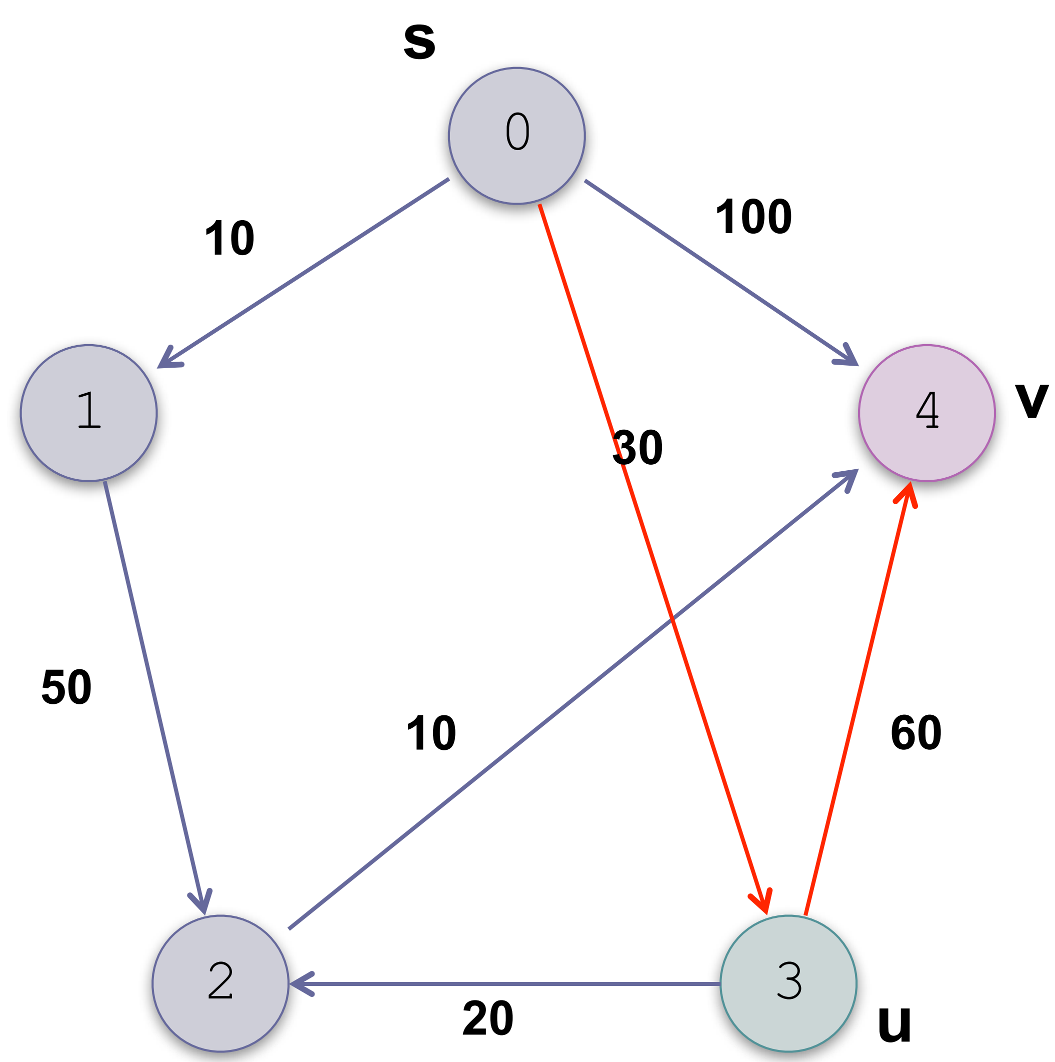
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-*S* = { 2, 3, 4 }

u = 3

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	100	0



Continue to the next adjacent vertex

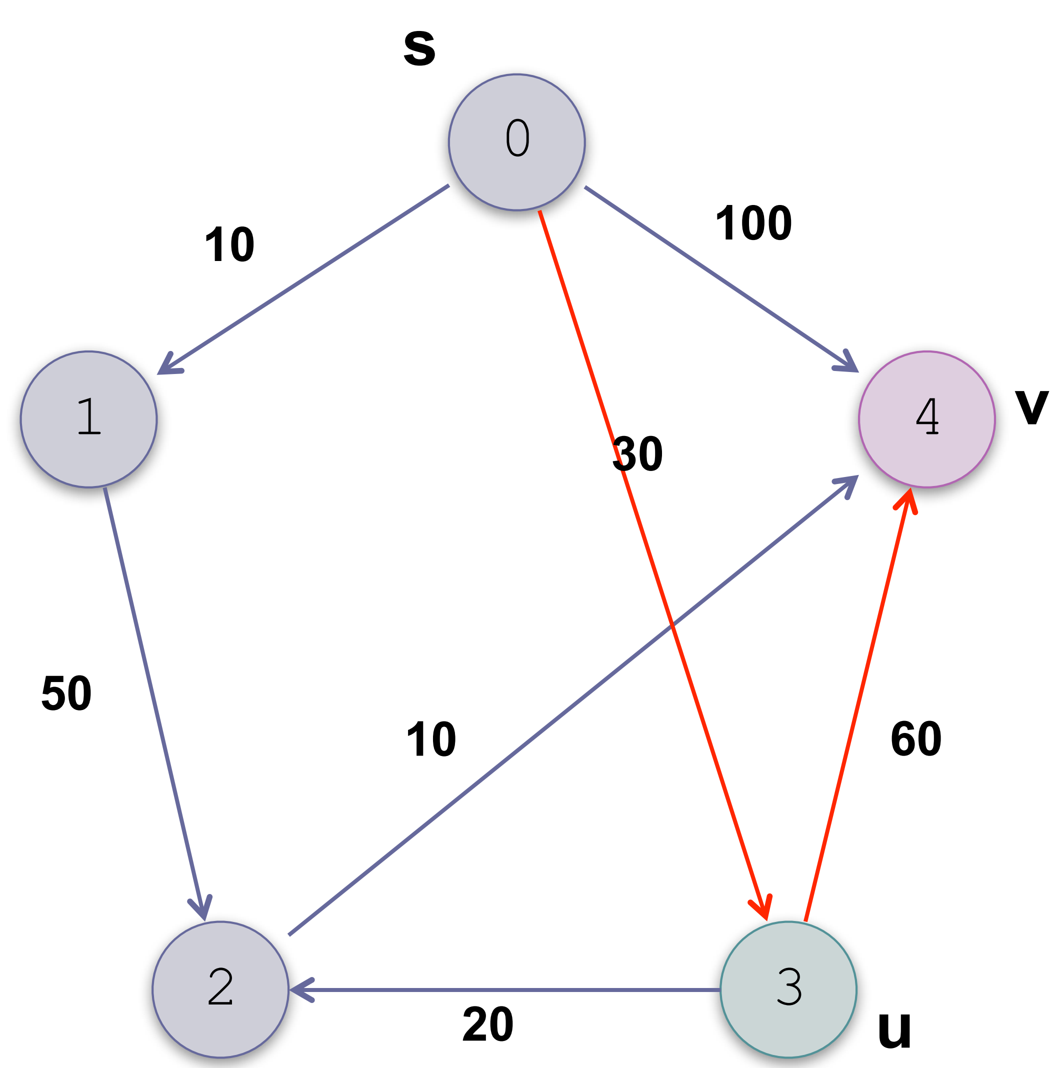
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-*S* = { 2, 3, 4 }

u = 3

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	100	0



90 < 100

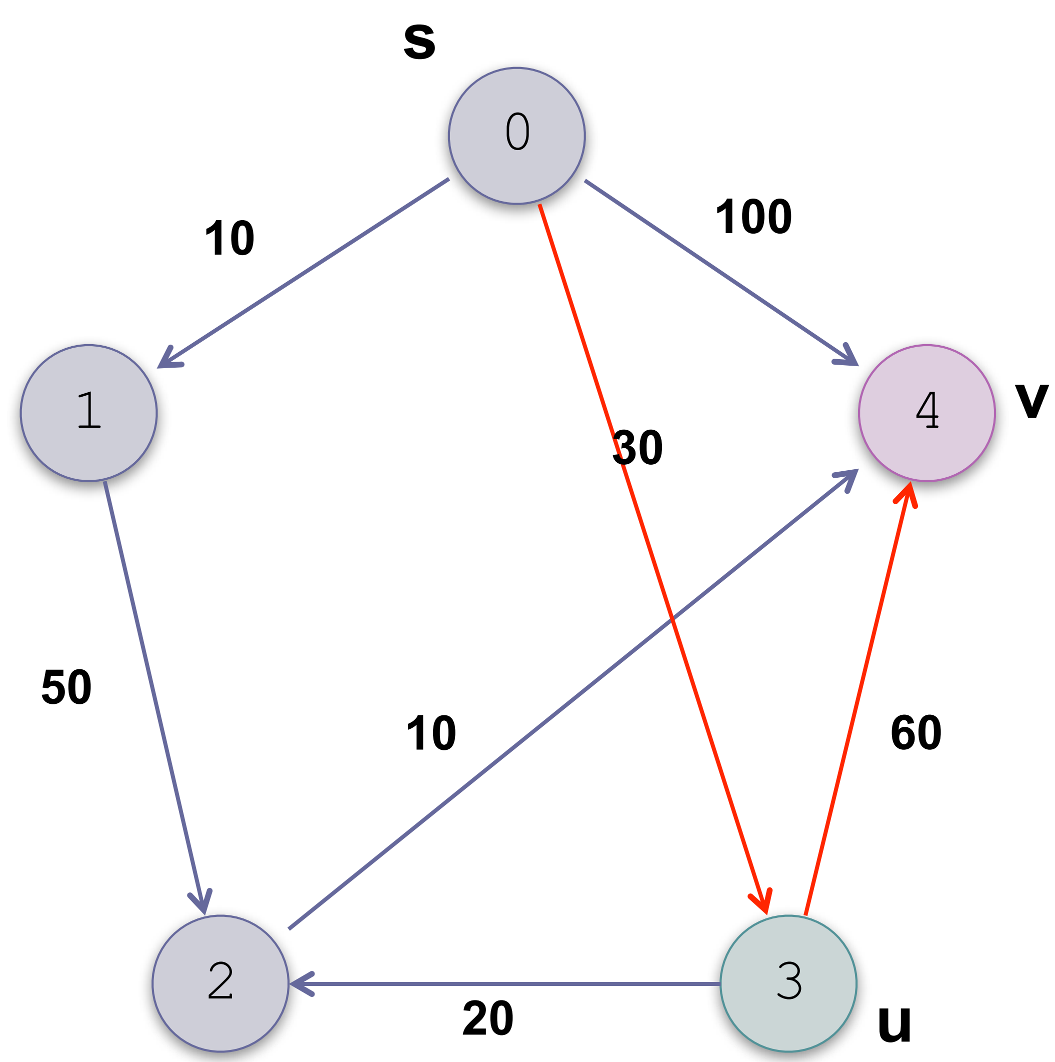
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-S = { 2, 3, 4 }

u = 3

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	90	3



90 < 100

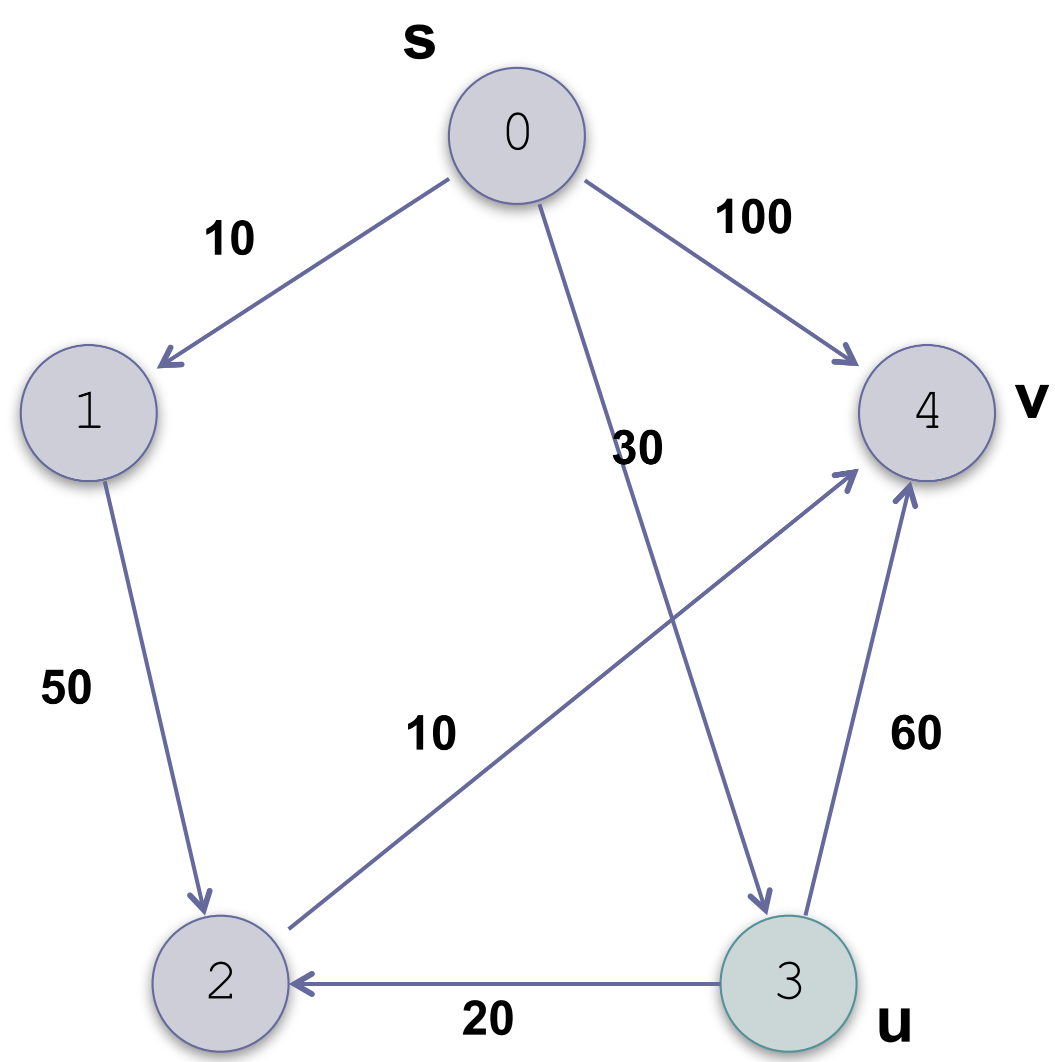
Dijkstra's Algorithm (cont.)

S = { 0, 1 }

V-*S* = { 2, 3, 4 }

u = 3

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	90	3



Move *u* from *V*-*S* to *S*

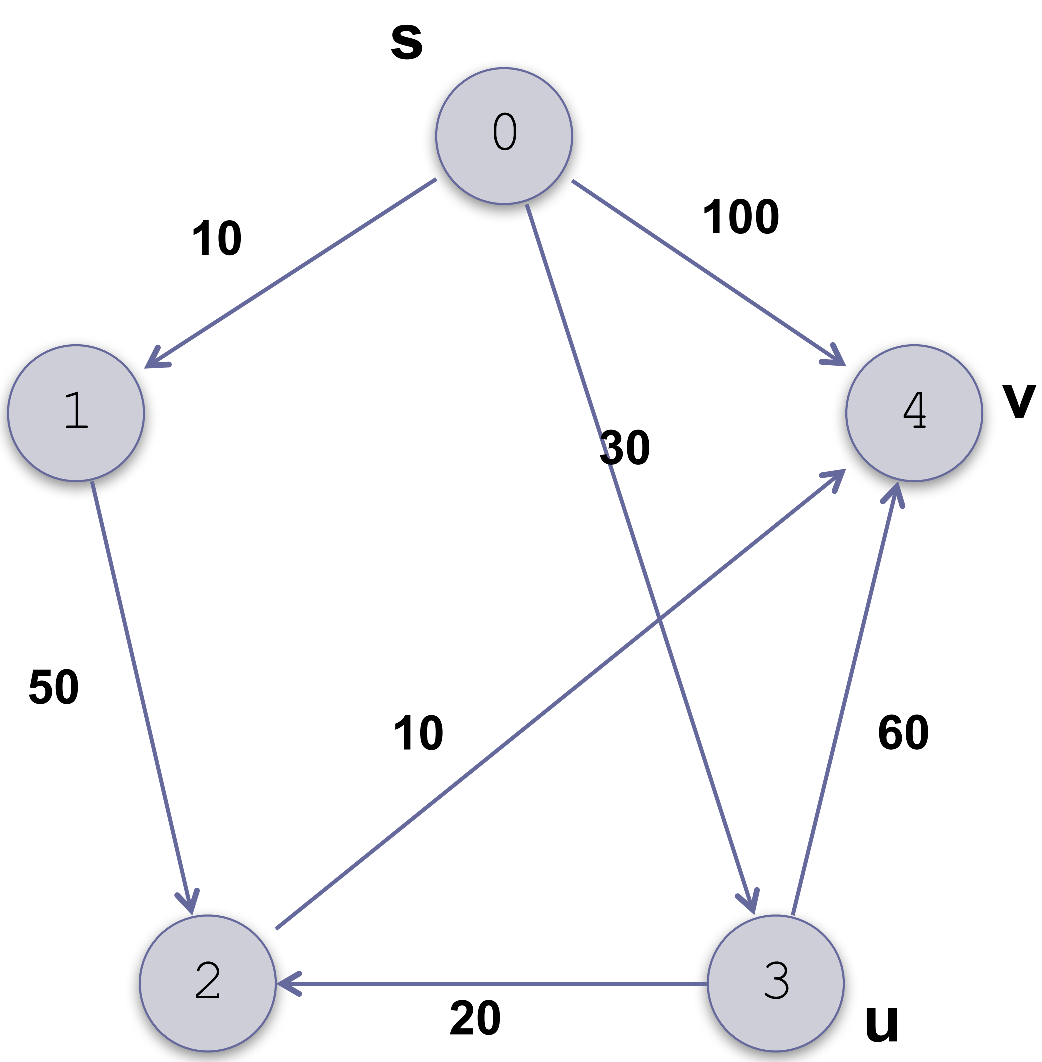
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3 }

V-*S* = { 2, 4 }

u = 3

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	90	3



Move *u* from *V-S* to *S*

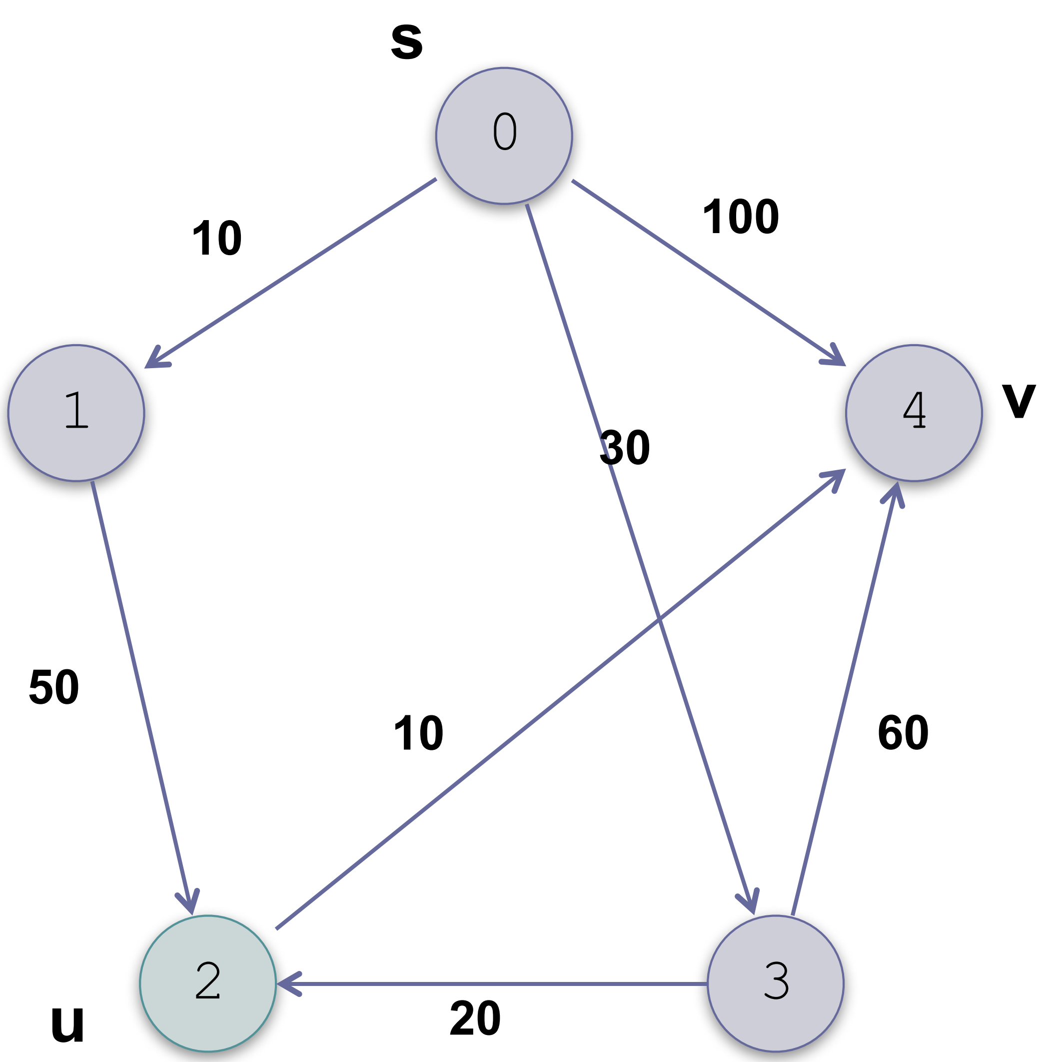
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3 }

V-S = { 2, 4 }

u = 2

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	90	3



Select vertex 2 from *V-S*

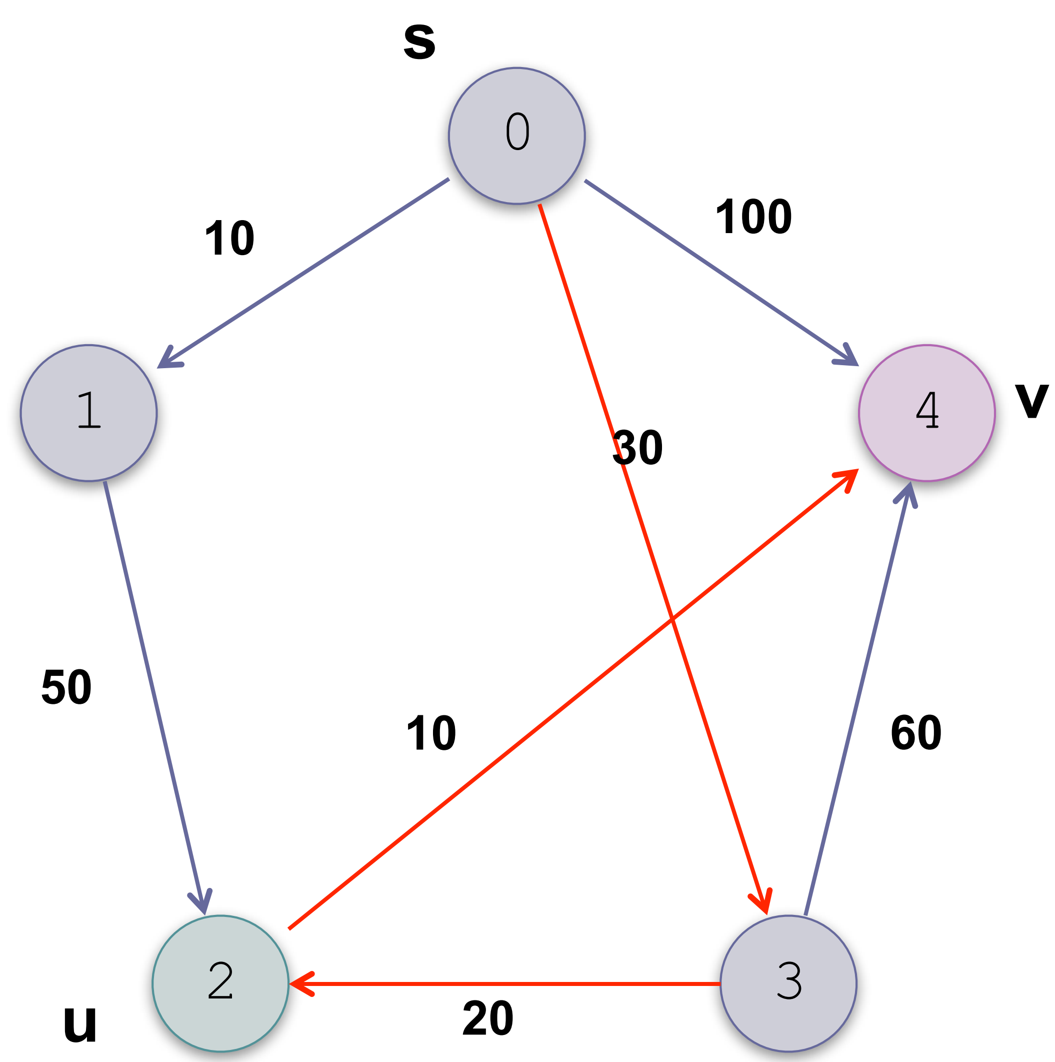
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3 }

V-S = { 2, 4 }

u = 2

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	90	3



d[2] + w(2,4) = 50 + 10 = 60

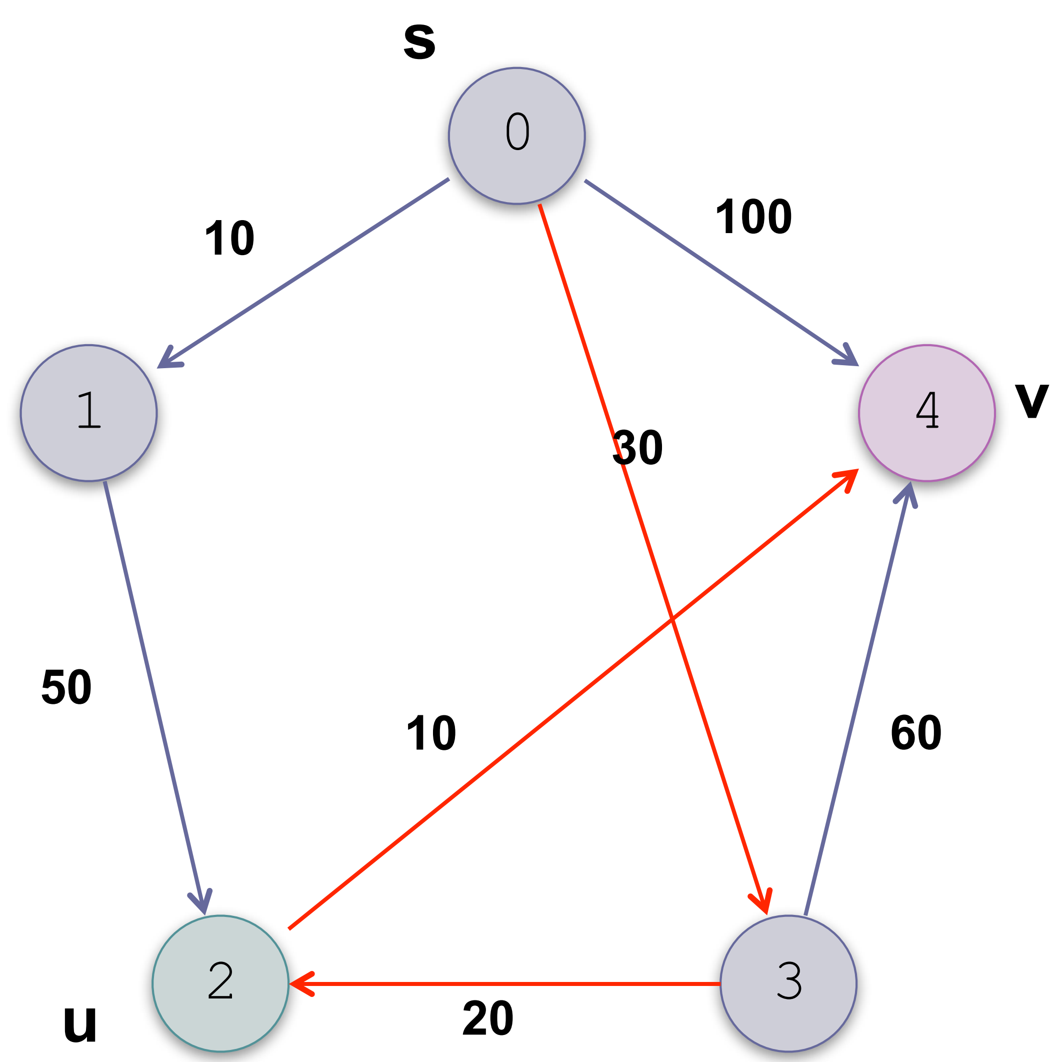
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3 }

V-*S* = { 2, 4 }

u = 2

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	90	3



60 < 90 (d[4])

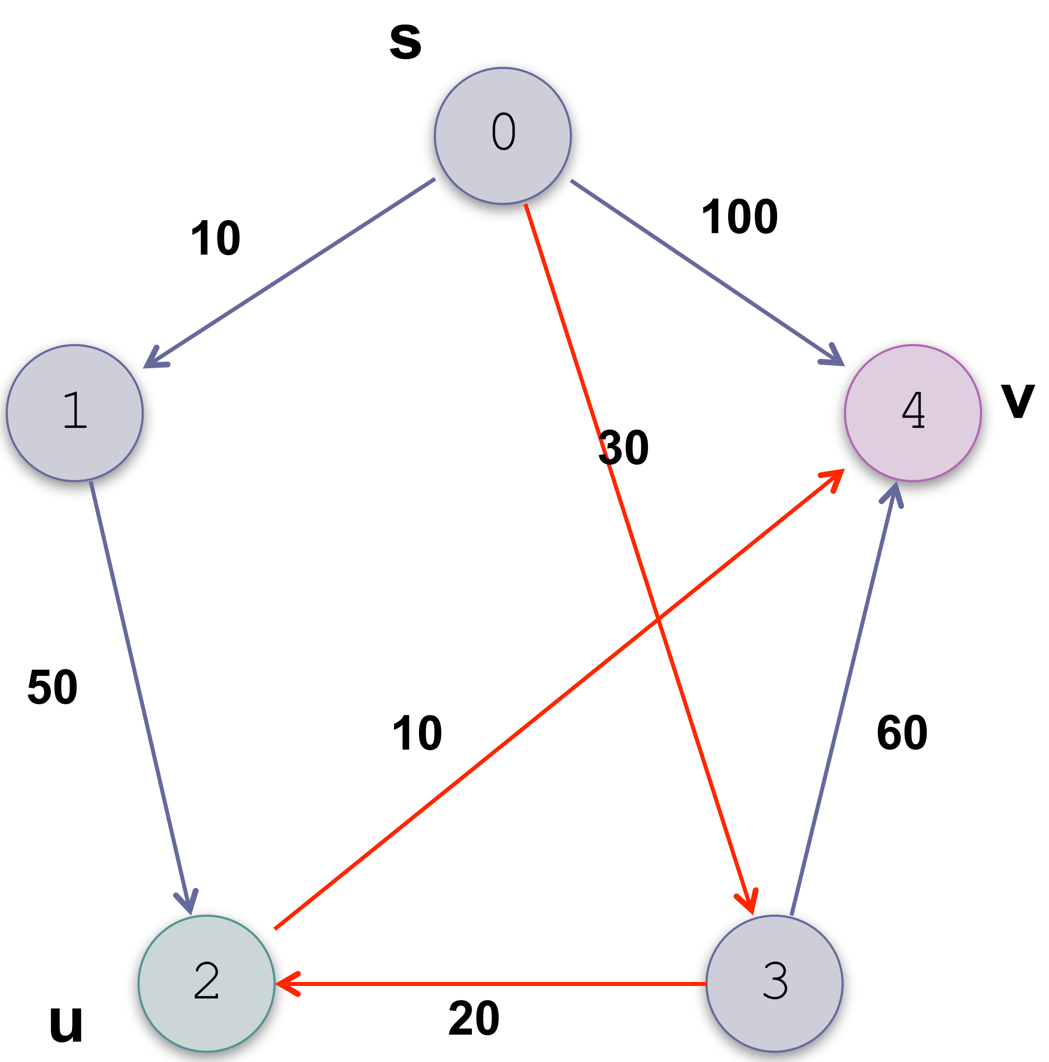
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3 }

V-*S* = { 2, 4 }

u = 2

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	90	3



update d[4] to 60 and p[4] to 2

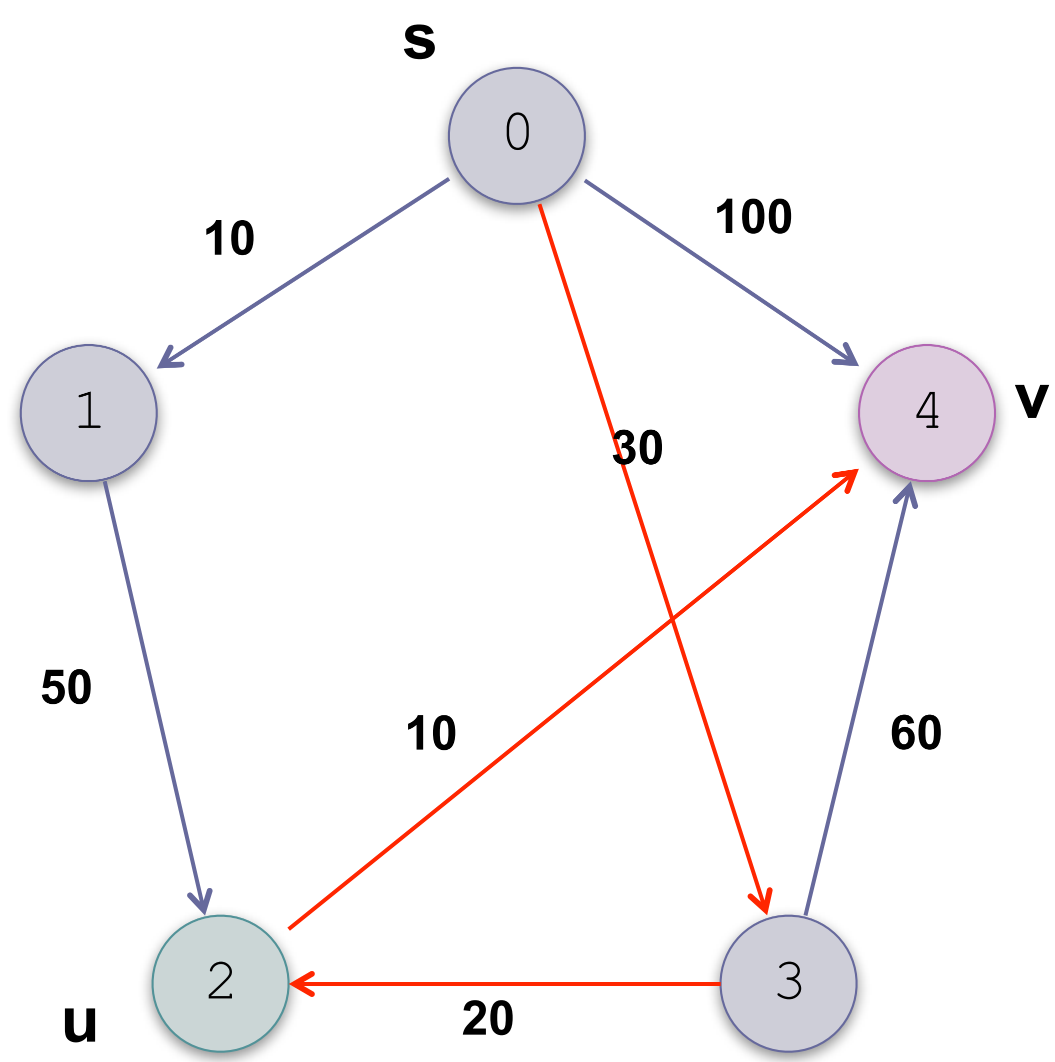
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3 }

V-S = { 2, 4 }

u = 2

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	60	2



update d[4] to 60 and p[4] to 2

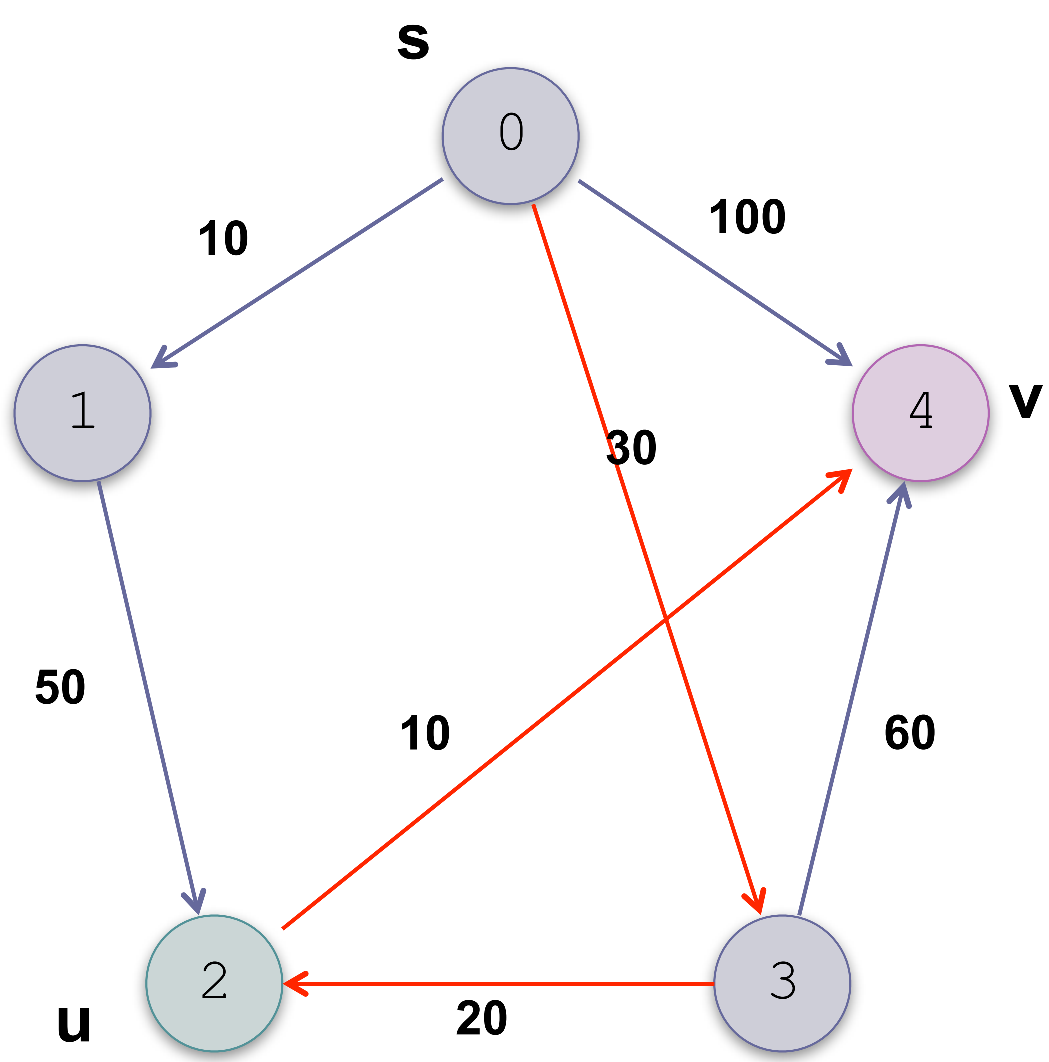
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3 }

V-*S* = { 2, 4 }

u = 2

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	60	2



Remove 2 from *V*-*S*

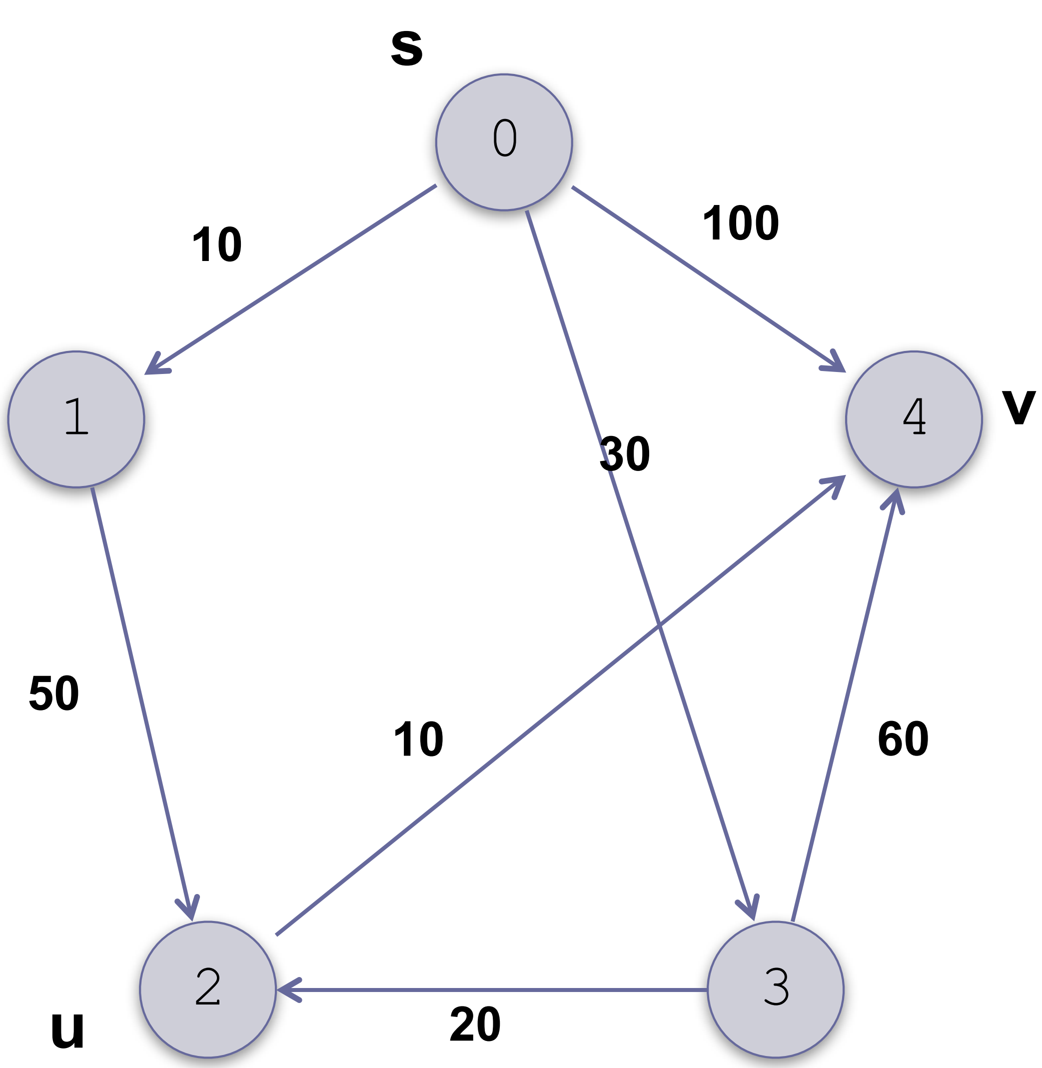
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2 }

V-*S* = { 4 }

u = 2

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	60	2



Remove 2 from *V*-*S*

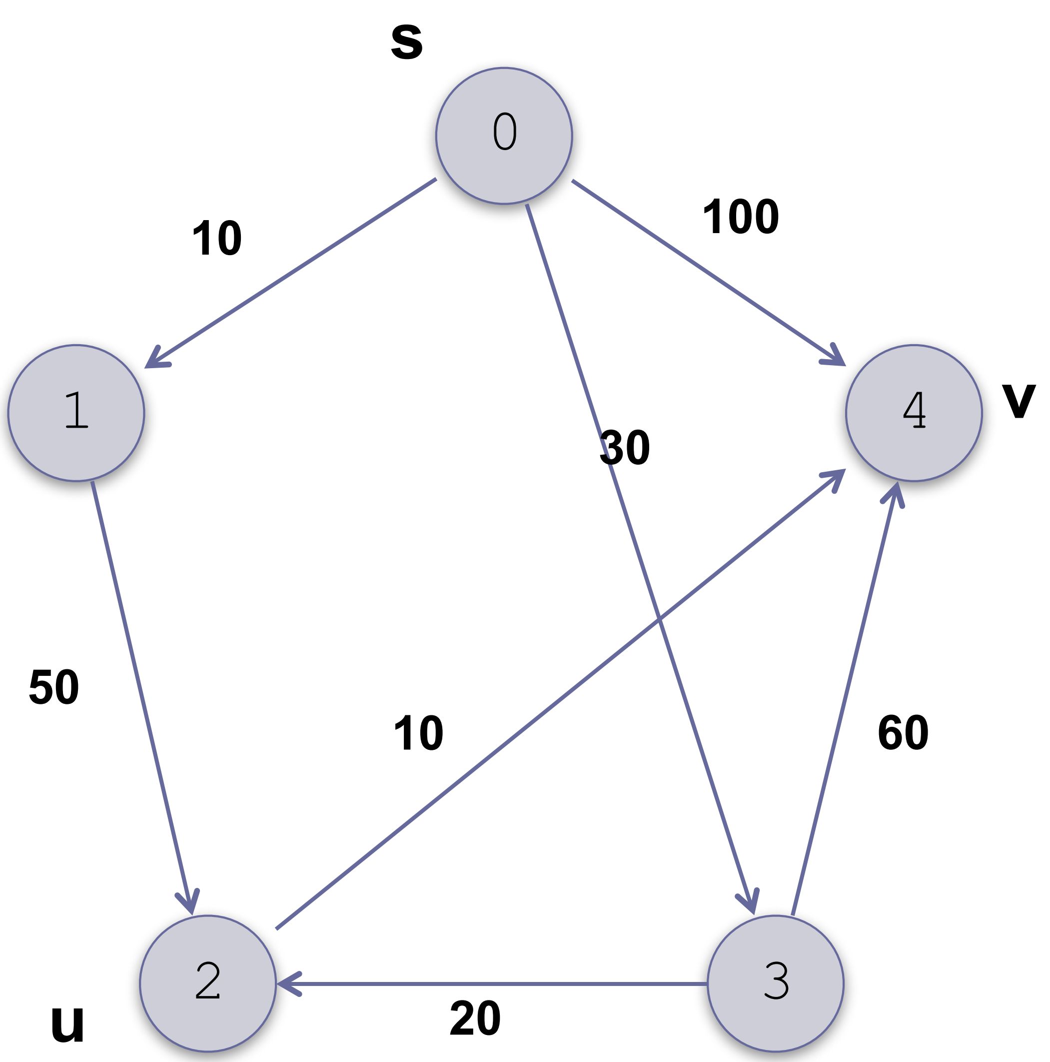
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2 }

V-*S* = { 4 }

u = 2

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	60	2



The final vertex in *V*-*S* is 4

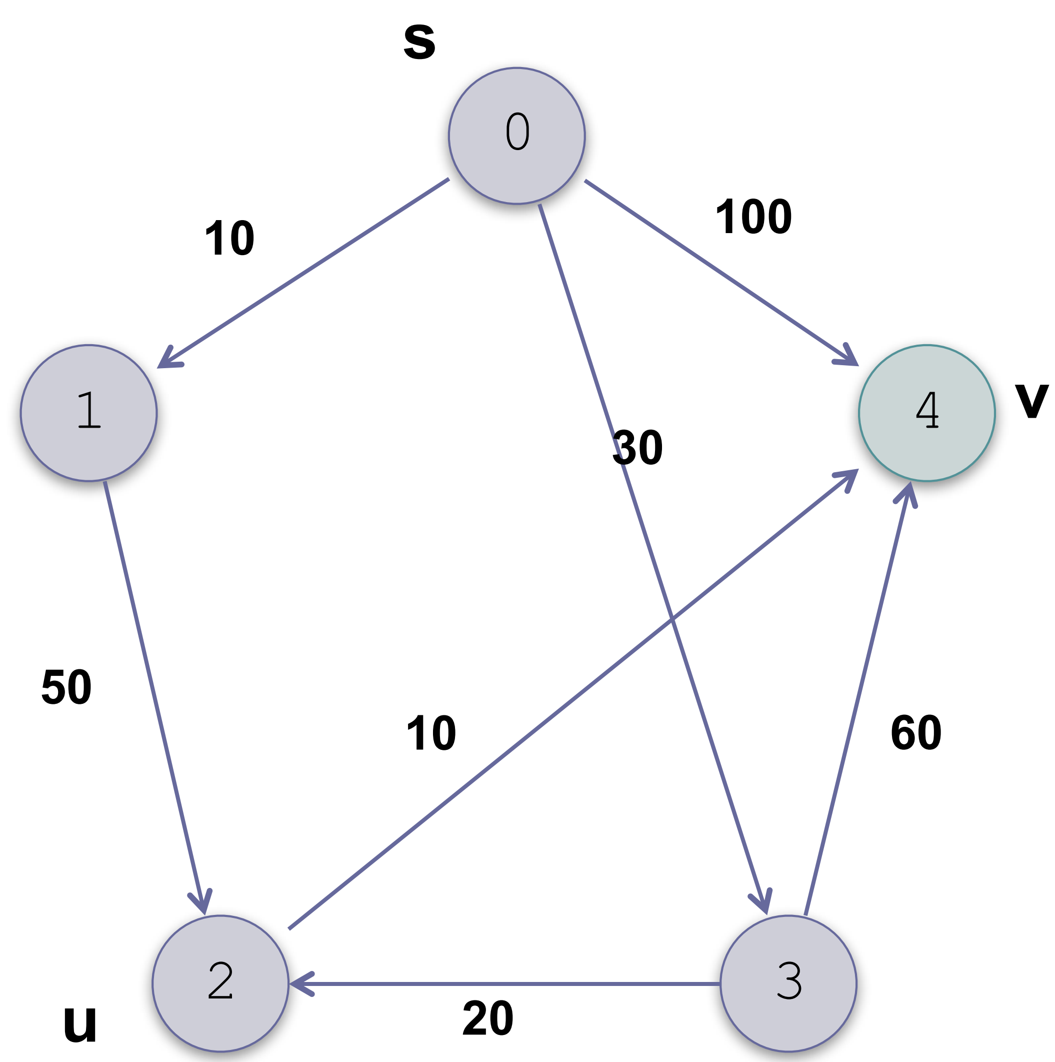
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2 }

V-*S* = { 4 }

u = 2

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	60	2



The final vertex in *V*-*S* is 4

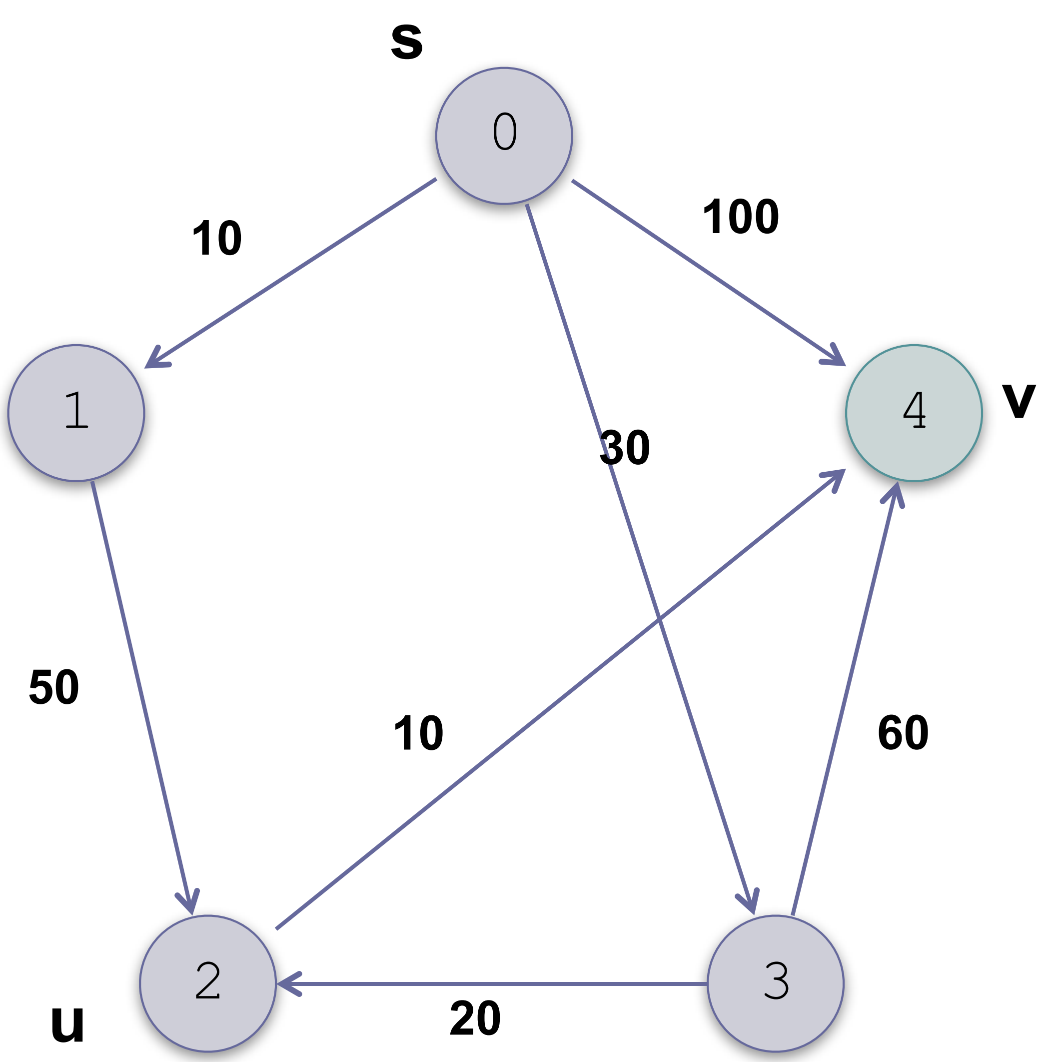
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2 }

V-S = { 4 }

u = 2

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	60	2



4 has no adjacent vertices; we move 4 into *S*

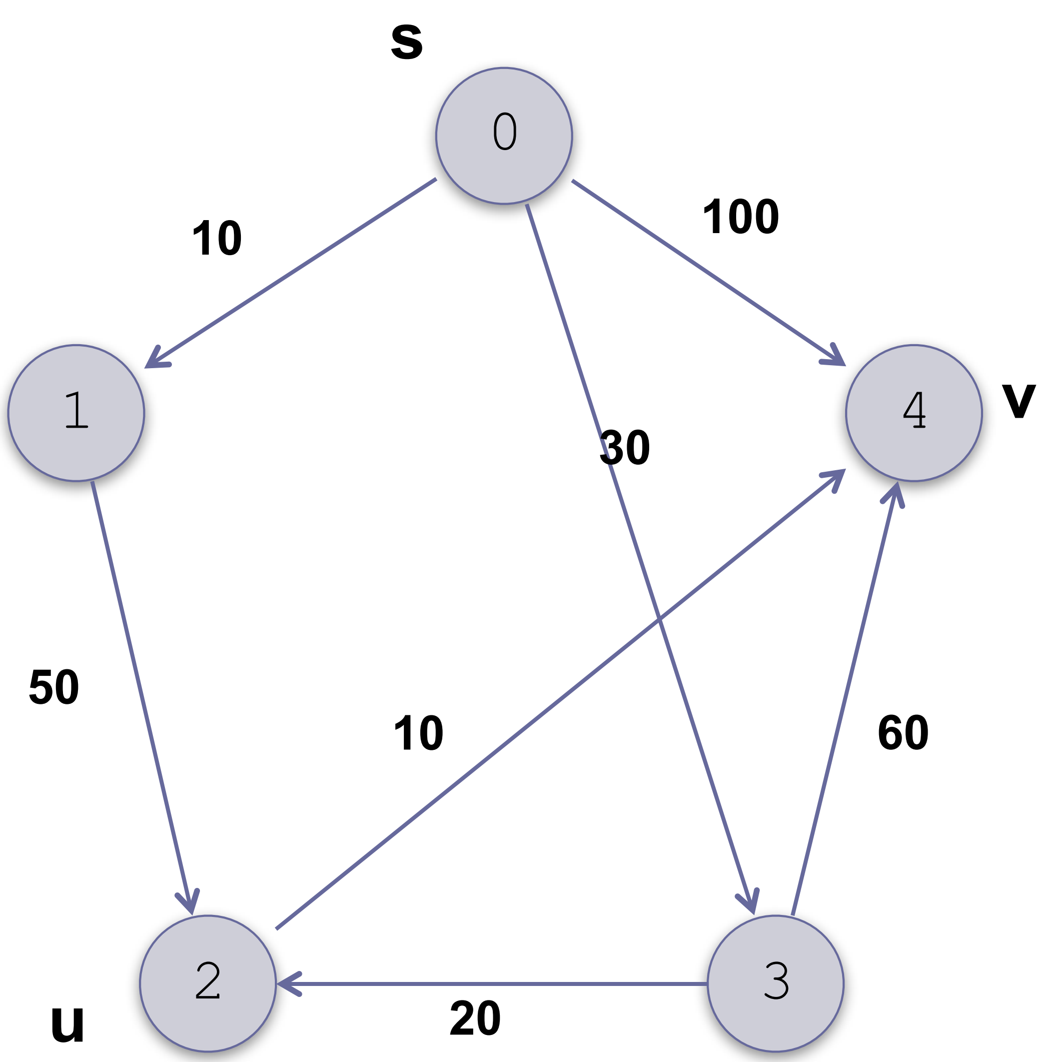
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2, 4 }

V-*S* = { }

u = 2

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	60	2



4 has no adjacent vertices; we move 4 into *S*

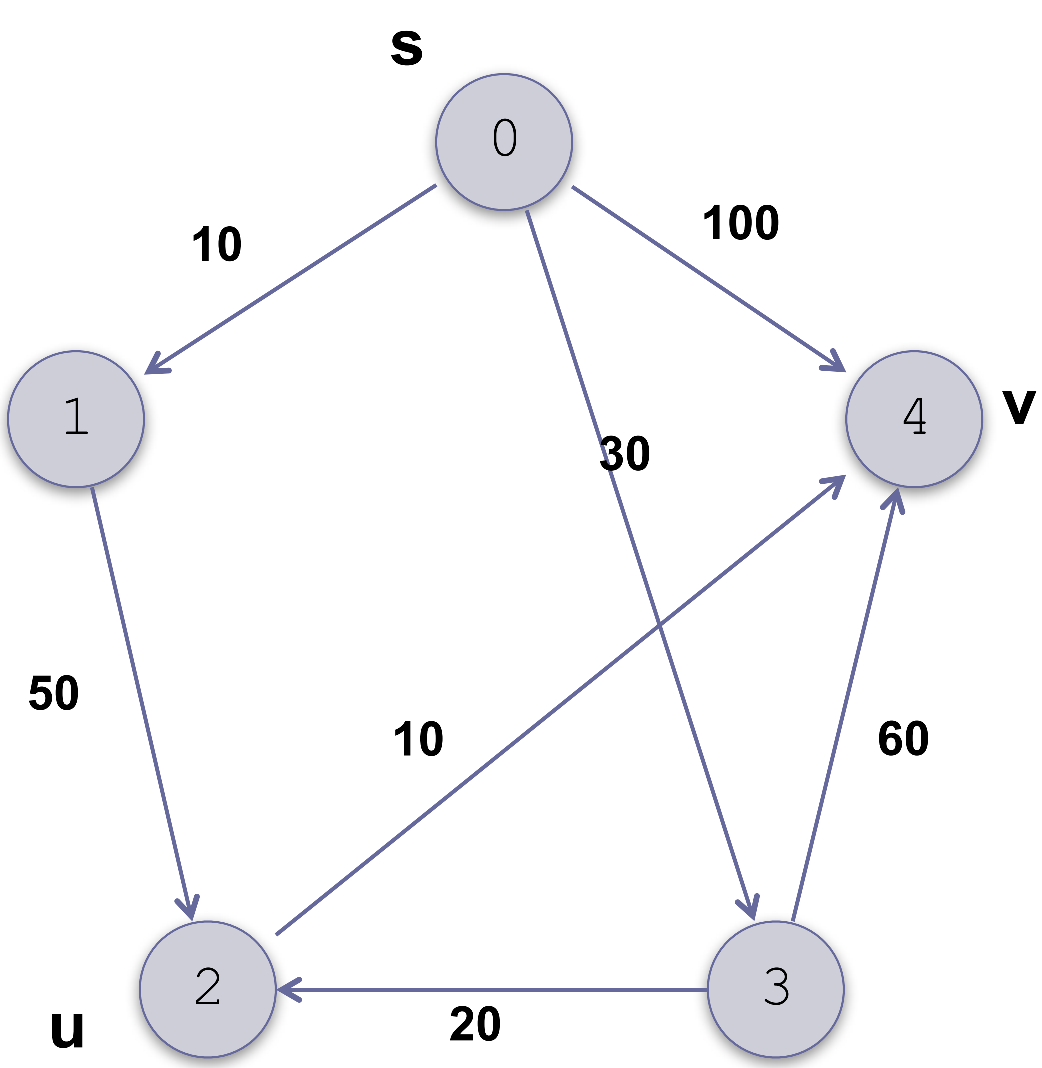
Dijkstra's Algorithm (cont.)

S = { 0, 1, 3, 2, 4 }

V-S = { }

u = 2

v	d[*v*]	p[*v*]
1	10	0
2	50	3
3	30	0
4	60	2



We are finished

Dijkstras algoritm

Dijkstra's Algorithm

```
Initialize S with the start vertex, s, and V–S with the remaining vertices.
for all v in V–S
    Set p[v] to s.
    if there is an edge (s, v)
        Set d[v] to w(s, v).
    else
        Set d[v] to ∞.
while V–S is not empty
    for all u in V–S, find the smallest d[u].
    Remove u from V–S and add u to S.
    for all v adjacent to u in V–S
        if d[u] + w(u, v) is less than d[v].
            Set d[v] to d[u] + w(u, v).
            Set p[v] to u.
```

Minimum spanning tree (Minimalt uppspännande träd)

Ett ”spanning tree” är en delmängd av bågarna i en graf, så att:

- alla noder är sammanhängande, och
- det finns inga cykler

Kostnaden för ett spanning tree är summan av bågarnas vikter:

- uppgiften är att hitta det spanning tree som har den minsta kostnaden
- dvs, minimum spanning tree (MST)
- Prims/Jarníks algoritm för MST är väldigt lik Dijkstras algoritm för kortaste vägen
- men algoritmen funkar bara på oriktade grafer!

Prims/Jarniks Algorithm

Noderna delas upp i två mängder:

- **S**, mängen av alla noder som har hamnat i trädet
- **V–S**, de kvarvarande noderna

Som i Dijkstras algoritm har vi två fält:

- **d**[v] innehåller längden av den kortaste bågen från någon nod u i **S** till v (som är i **V–S**)
- **p**[v] innehåller startnoden u för denna båge

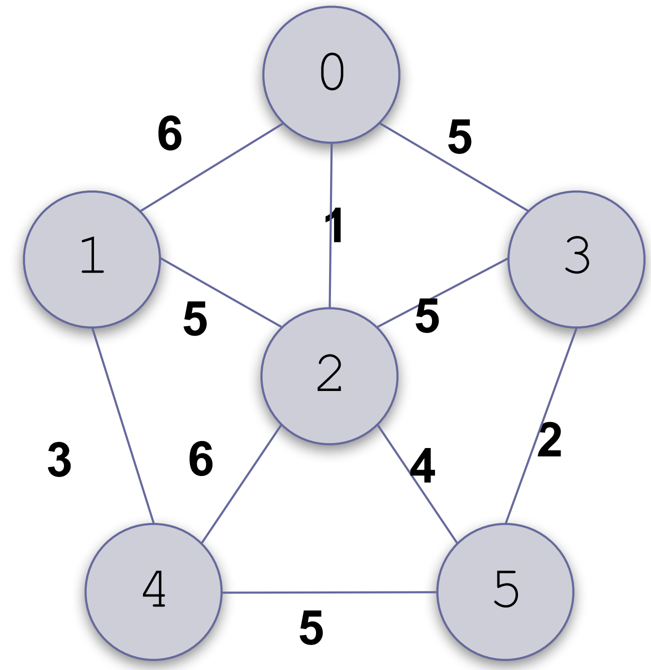
Den enda skillnaden mellan algoritmerna är innehållet i **d**[v]:

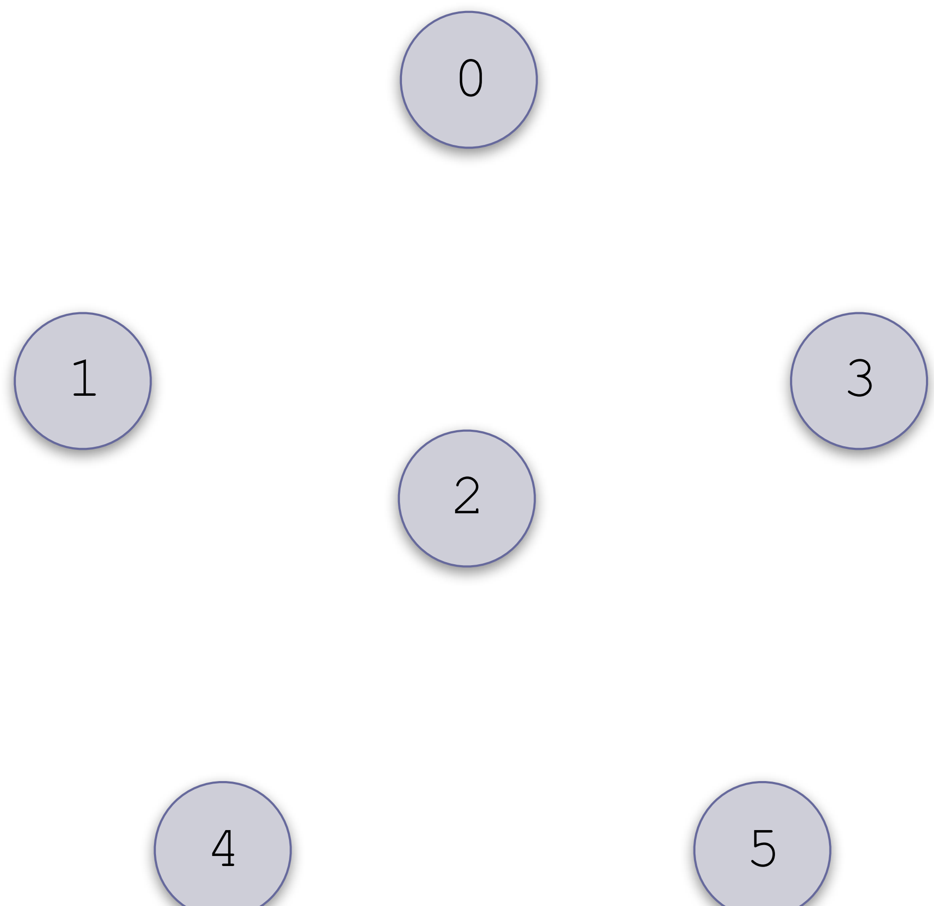
- i Prims/Jarníks algoritm, innehåller **d**[v] endast den senaste bågens längd
- i Dijkstras algoritm, innehåller **d**[v] hela längden från s

Prim's Algorithm Example

S **= { 0 }**

V-*S* **= { 1, 2, 3, 4, 5 }**

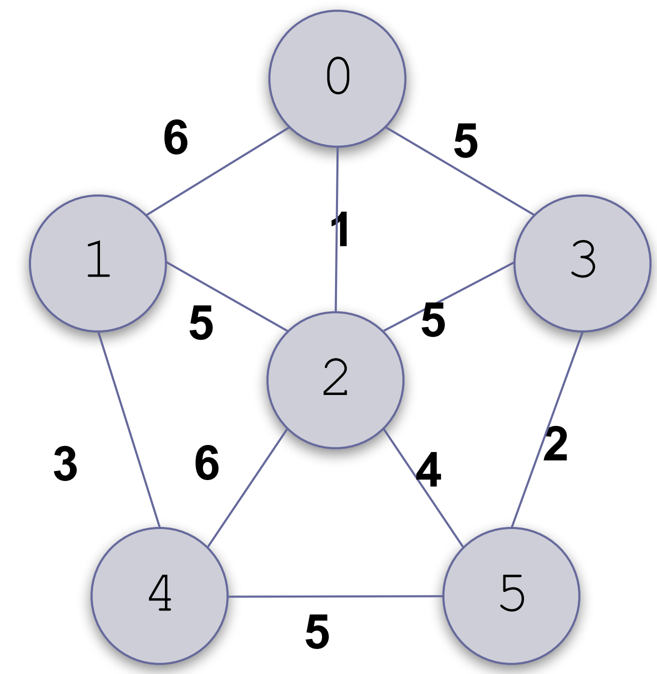


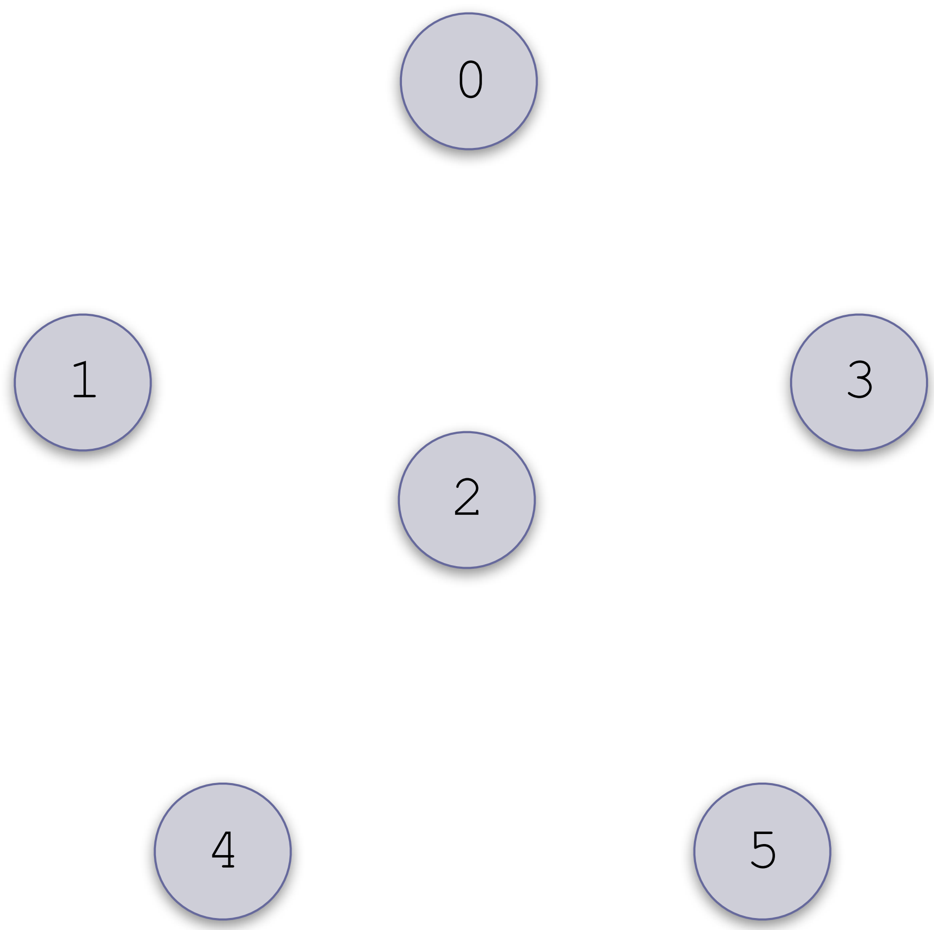


Prim's Algorithm Example (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4, 5 }



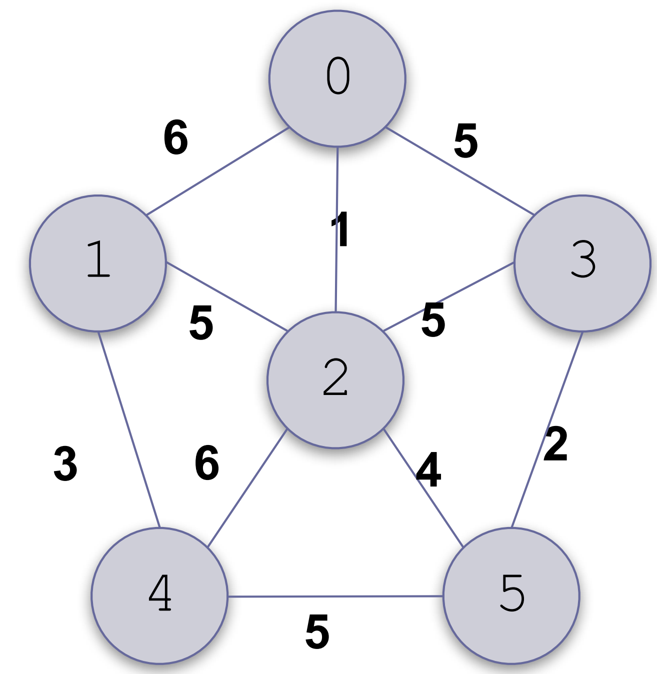


The smallest edge from *u* to *v* where *u* is in *S* and *v* is in *V-S* is the edge (0,2)

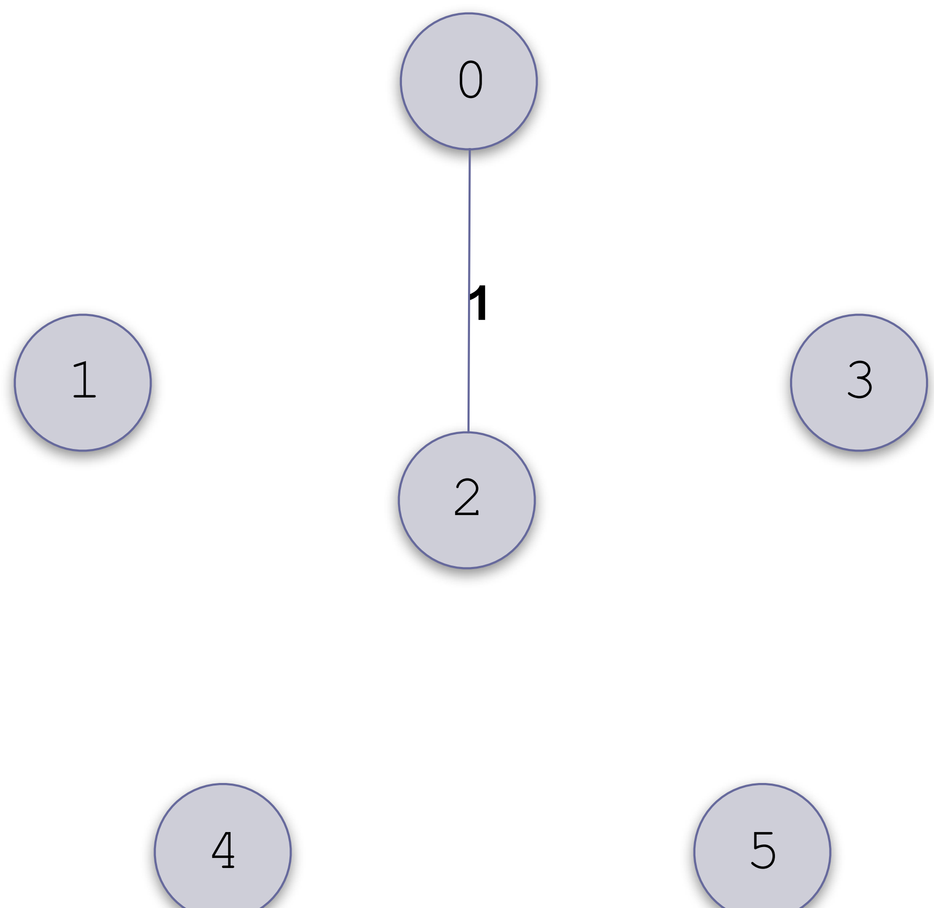
Prim's Algorithm Example (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4, 5 }



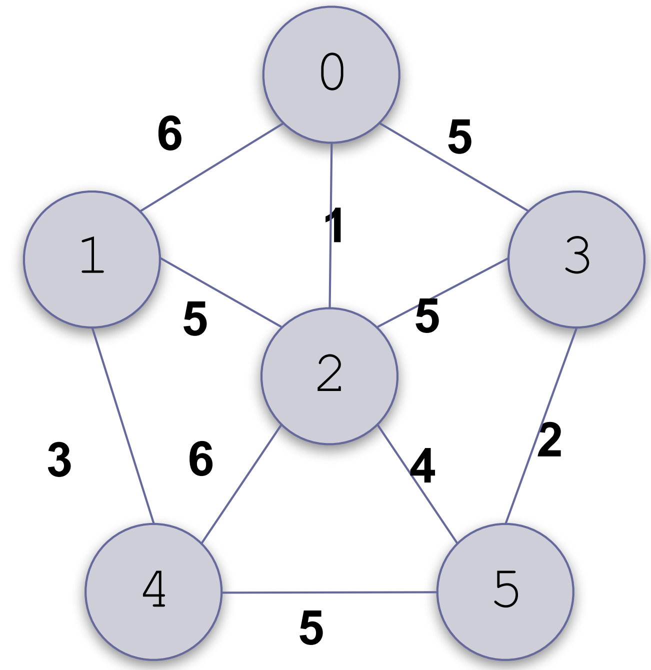
Add this edge to the spanning tree



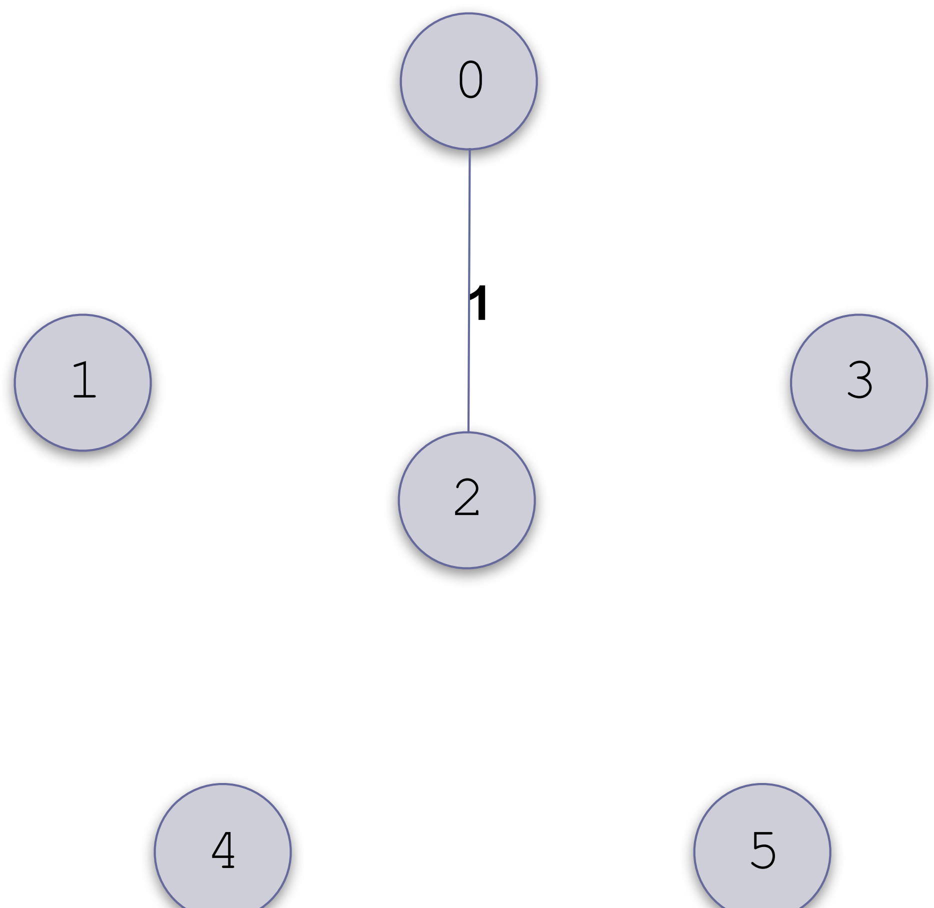
Prim's Algorithm Example (cont.)

S = { 0 }

V-S = { 1, 2, 3, 4, 5 }



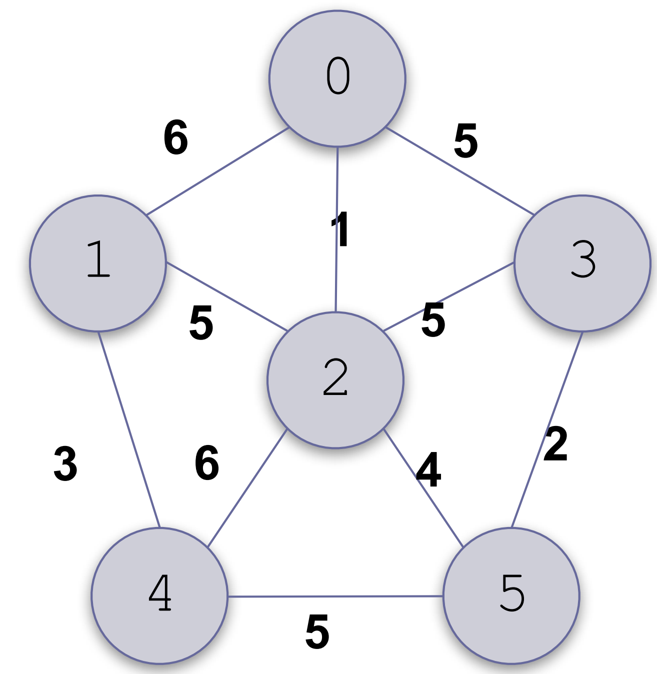
and move 2 to *S*



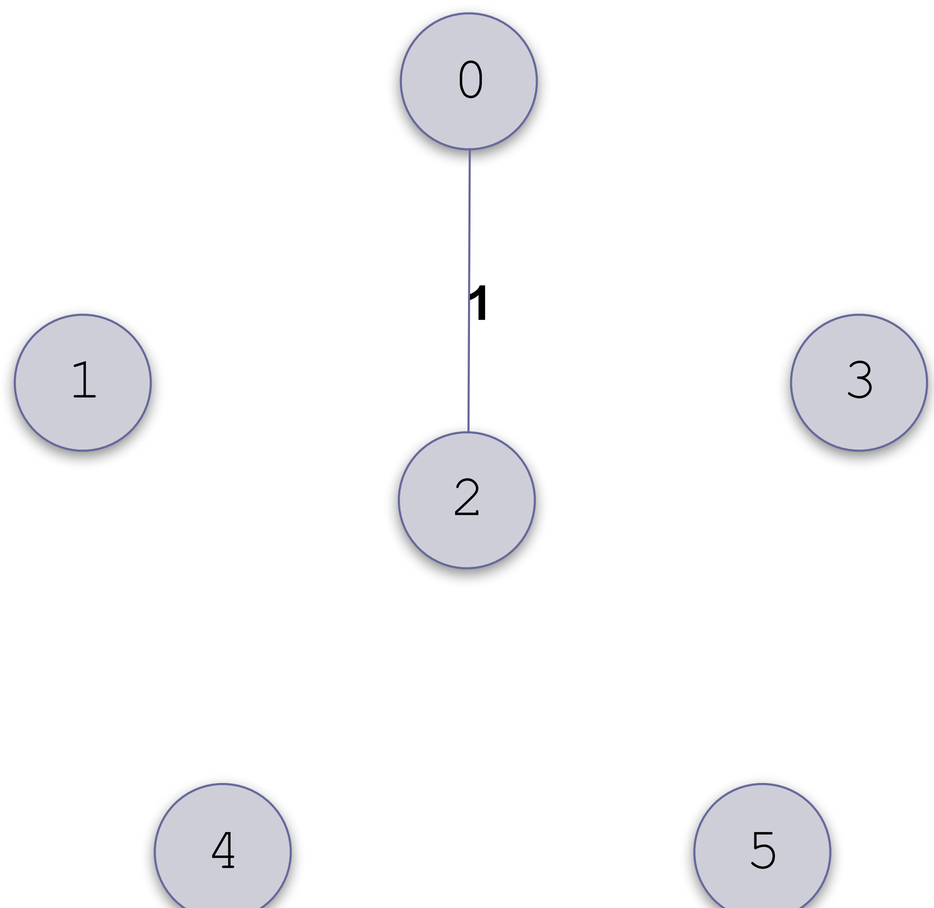
Prim's Algorithm Example (cont.)

S = { 0, 2 }

V-S = { 1, 3, 4, 5 }



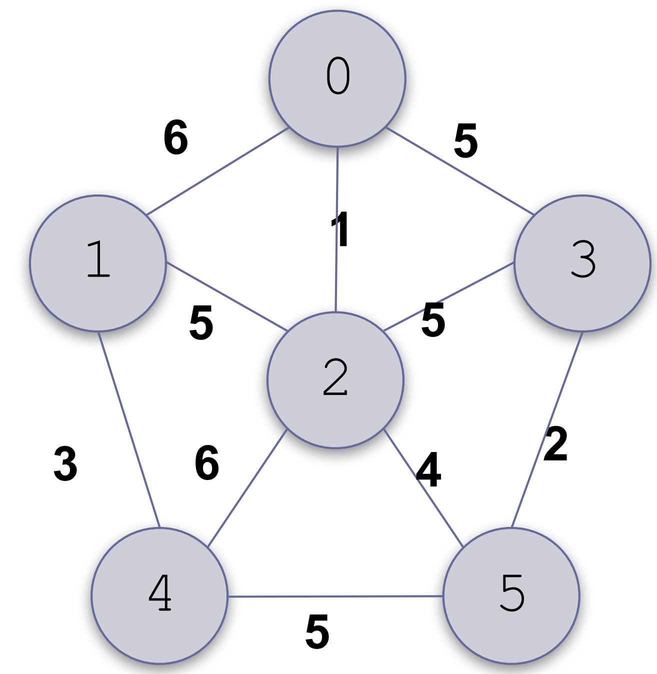
and move 2 to *S*

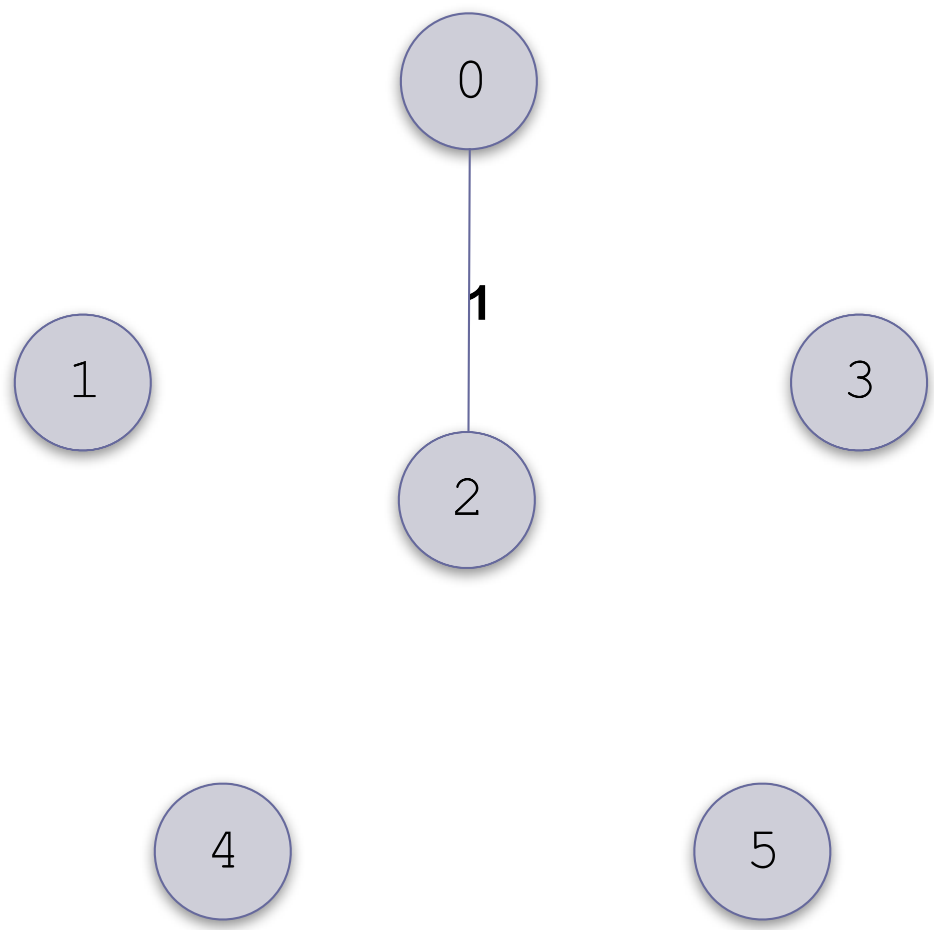


Prim's Algorithm Example (cont.)

S = { 0, 2 }

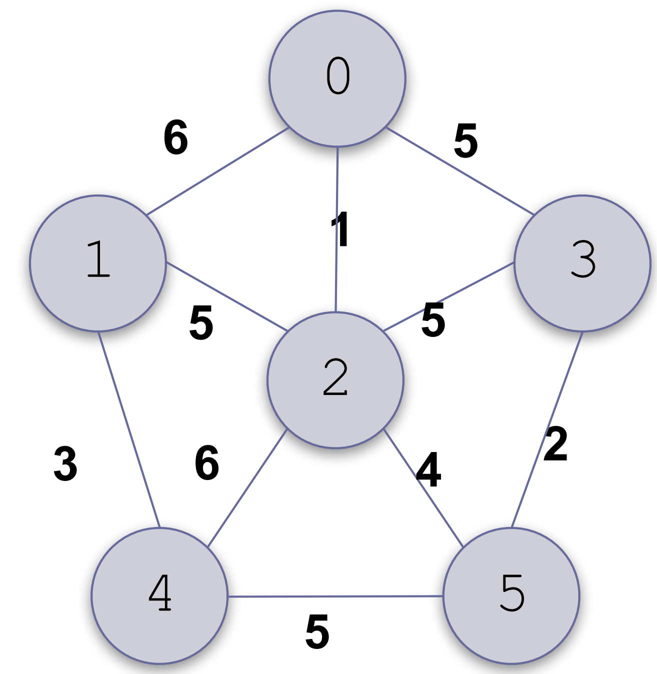
V-S = { 1, 3, 4, 5 }





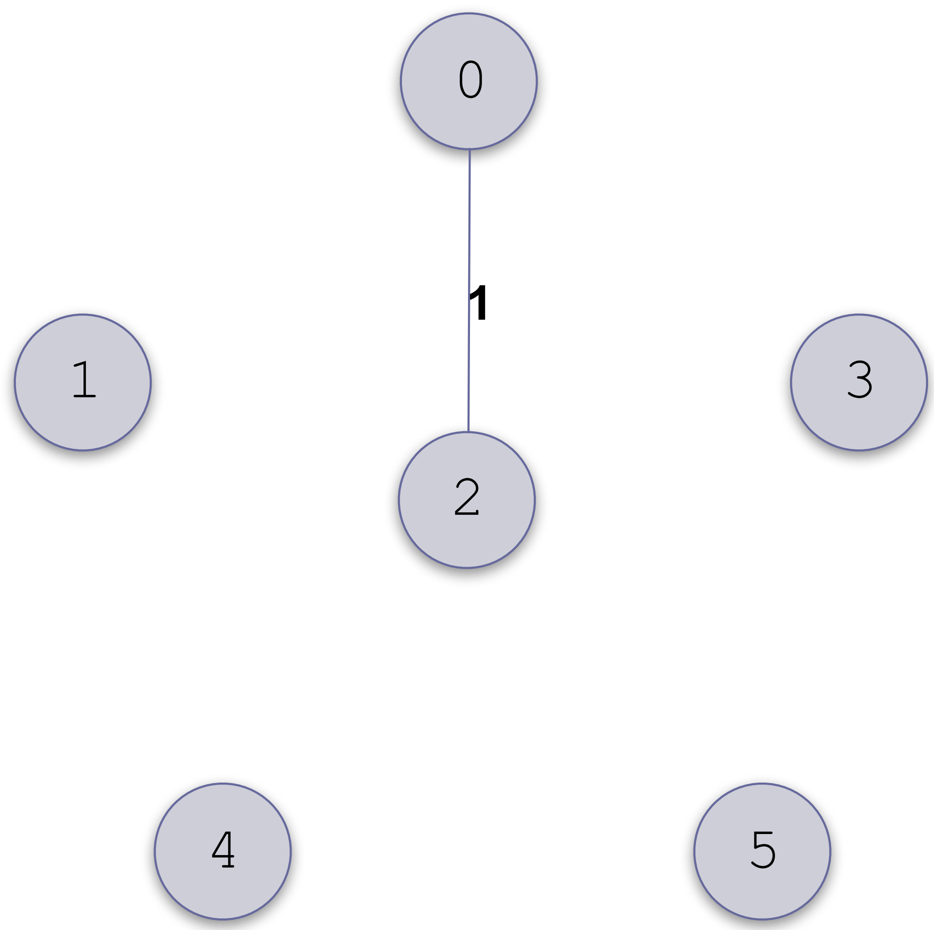
Consider all edges (*u*, *v*) where *u* is in *S* and *v* is in *V-S* (there are 8 possible edges)

Prim's Algorithm Example (cont.)



S = { 0, 2 }

V-S = { 1, 3, 4, 5 }

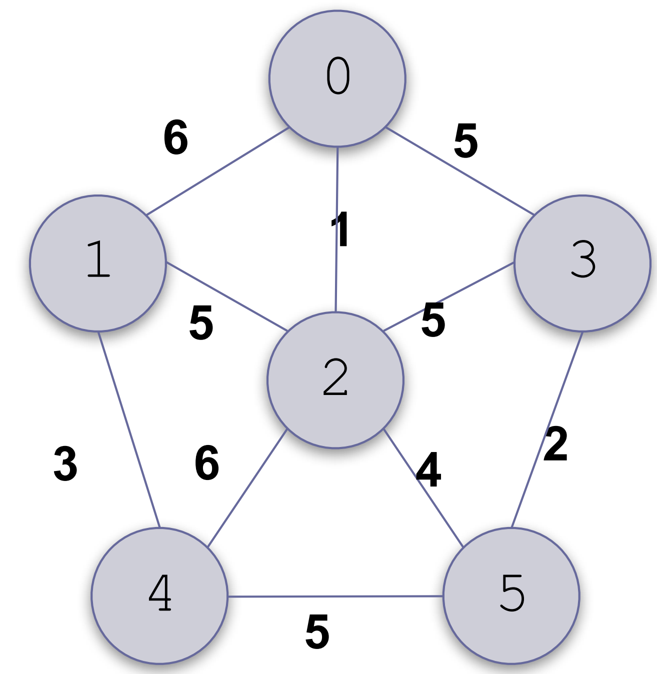


The smallest is (2, 5)

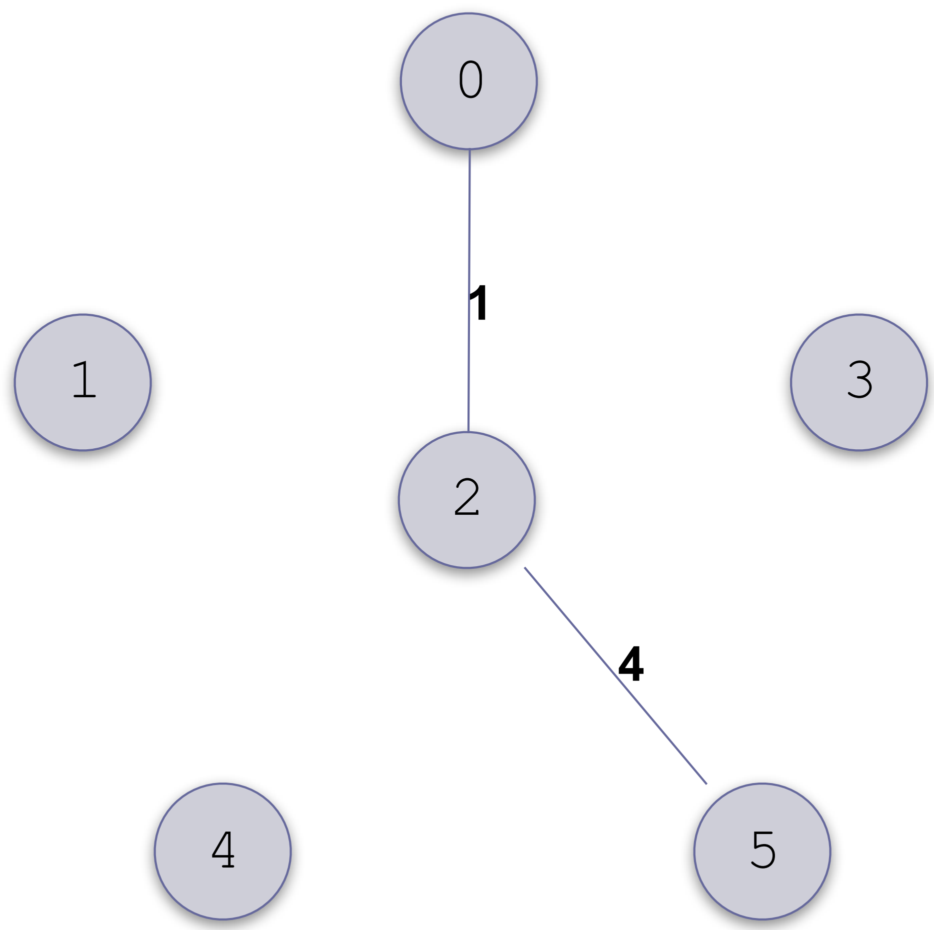
Prim's Algorithm Example (cont.)

S = { 0, 2 }

V-S = { 1, 3, 4, 5 }



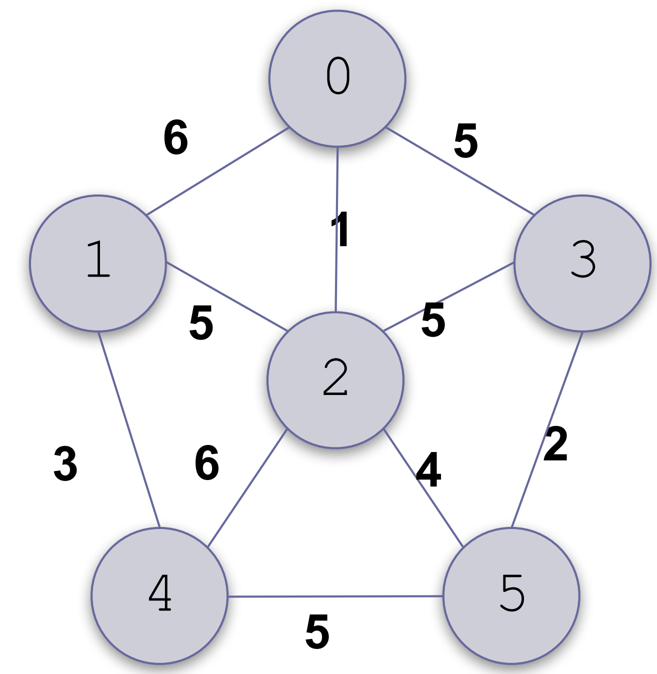
Add (2,5) to the spanning tree



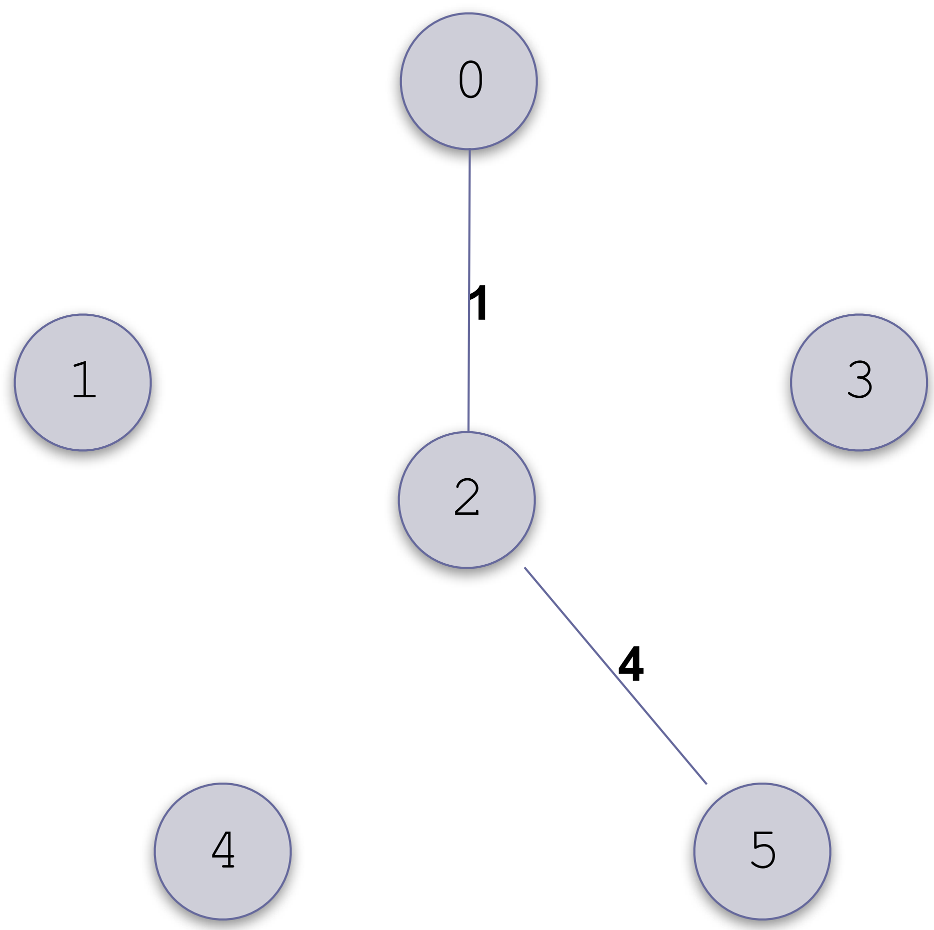
Prim's Algorithm Example (cont.)

S = { 0, 2 }

V-S = { 1, 3, 4, 5 }



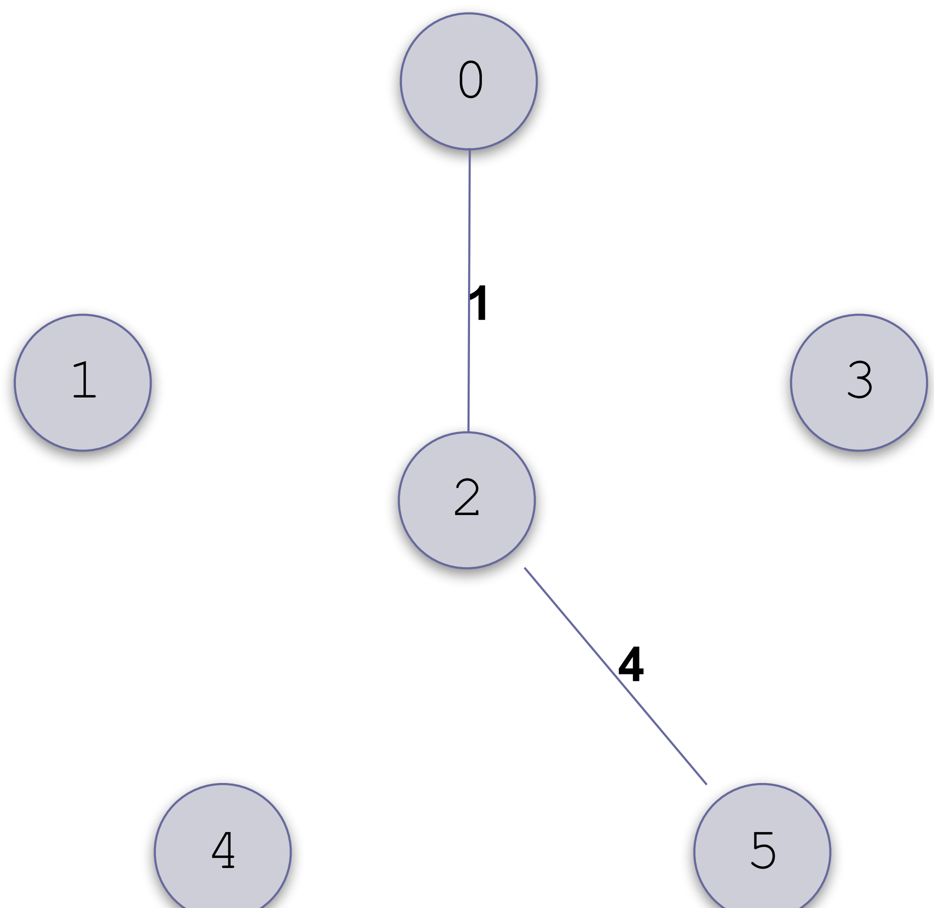
Move 5 from *V-S* to *S*

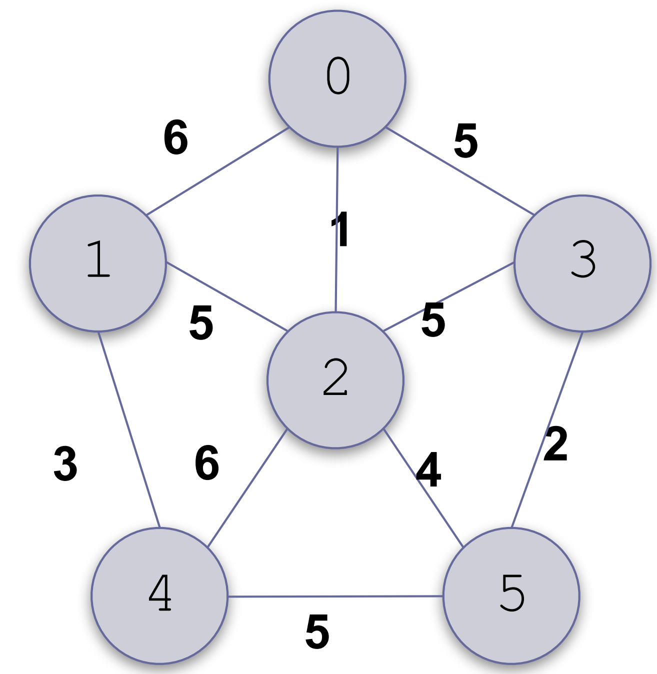


Prim's Algorithm Example (cont.)

S = { 0, 2, 5 }

V-S = { 1, 3, 4 }



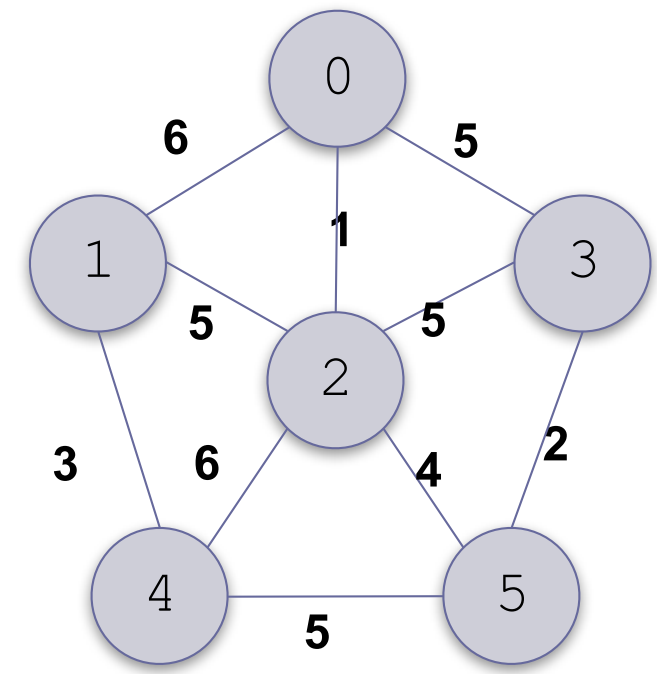


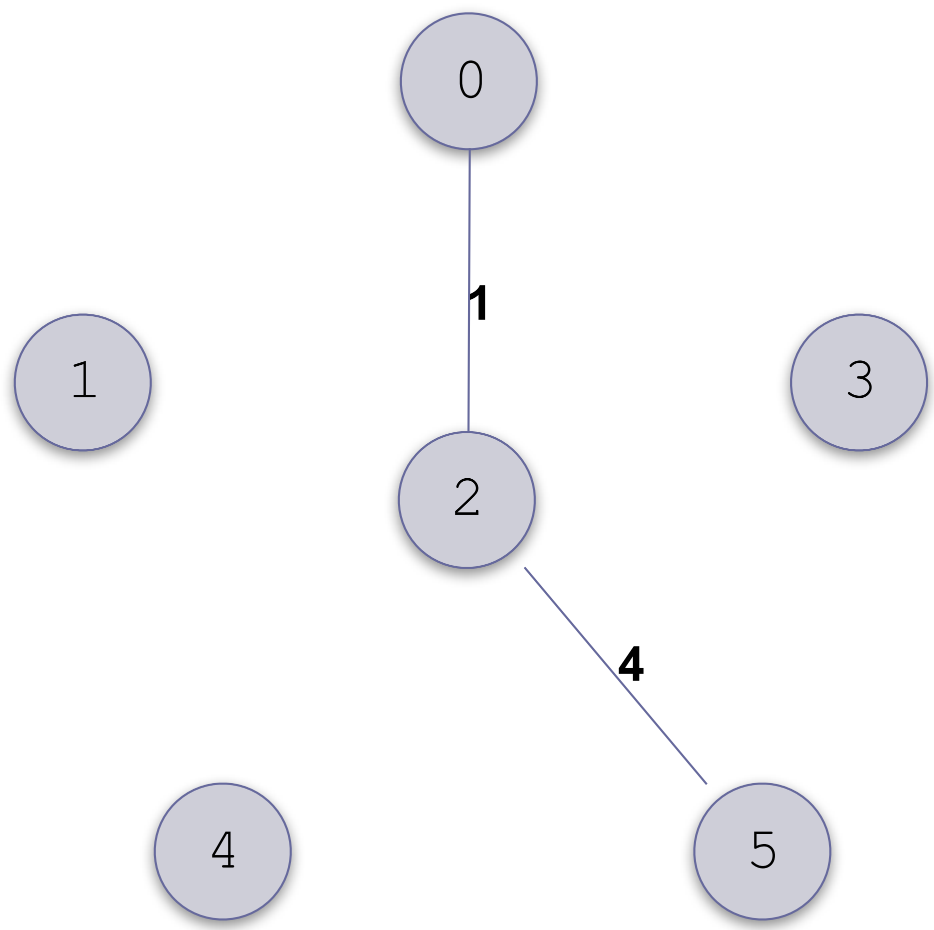
Move 5 from *V-S* to *S*

Prim's Algorithm Example (cont.)

S = { 0, 2, 5 }

V-S = { 1, 3, 4 }



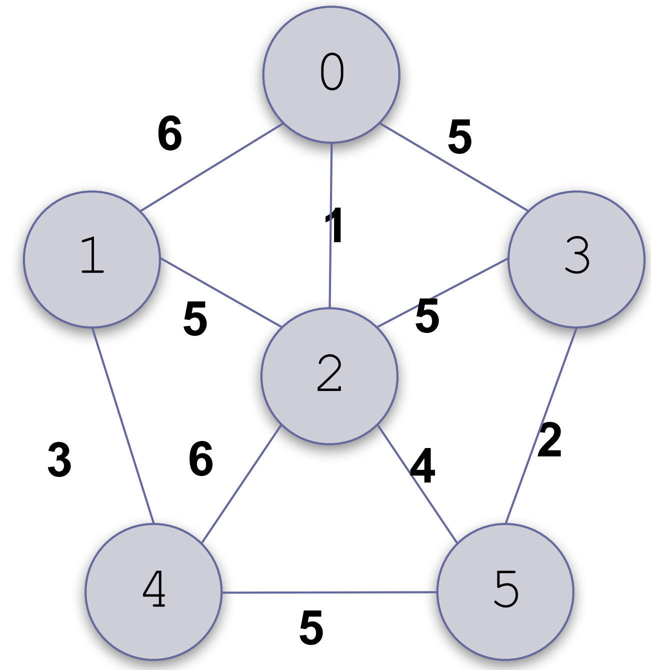


Find the smallest edge (*u*, *v*) where *u* is in *S* and *v* is in *V-S*

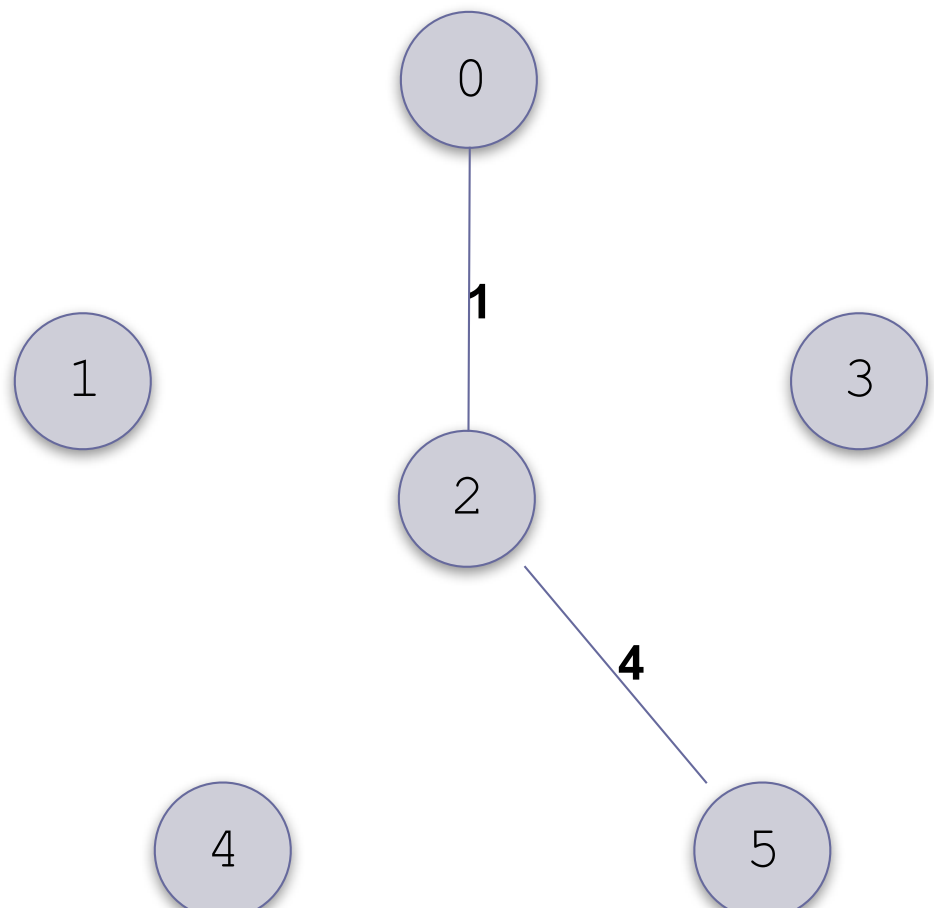
Prim's Algorithm Example (cont.)

S = { 0, 2, 5 }

V-S = { 1, 3, 4 }



The smallest edge is (5, 3)

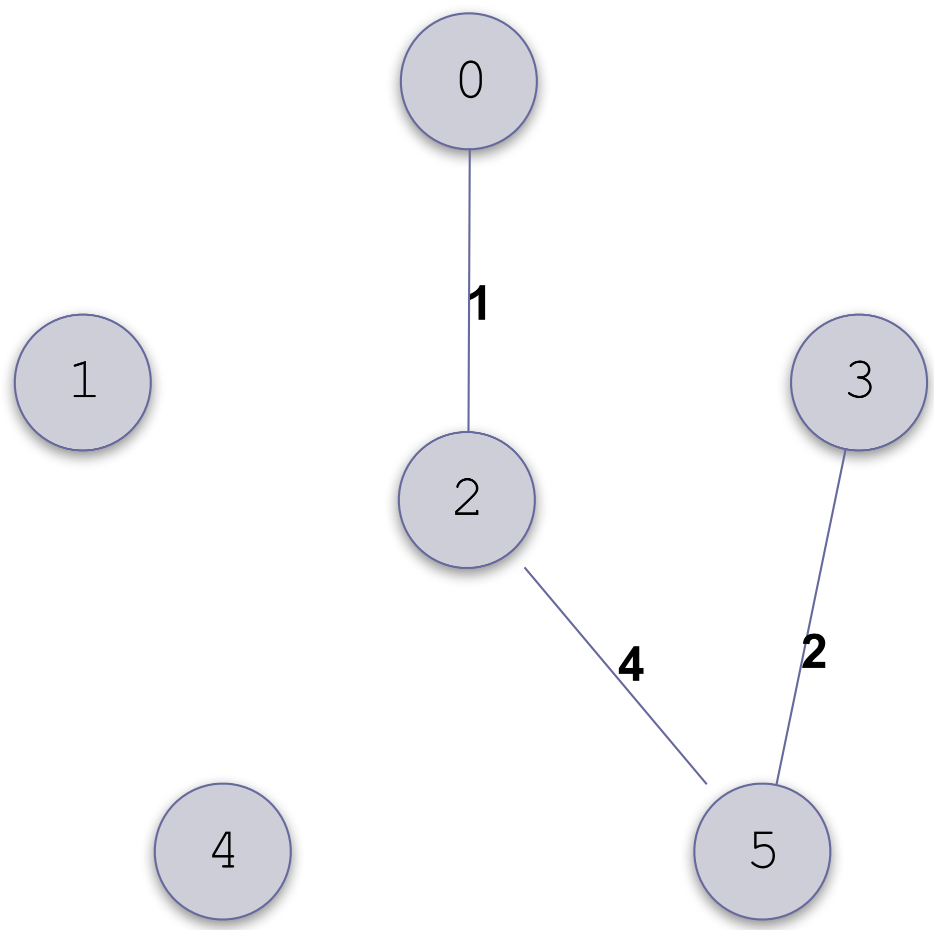


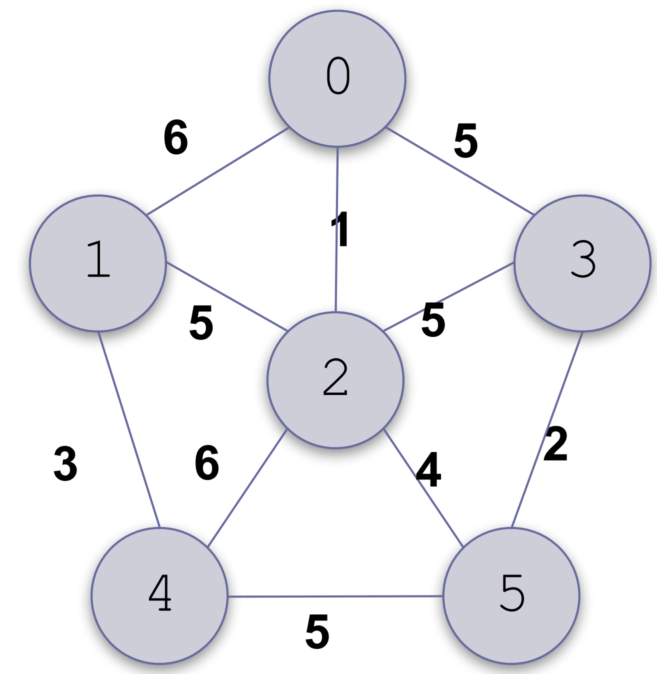
Prim's Algorithm Example (cont.)

S = { 0, 2, 5 }

V-S = { 1, 3, 4 }

The smallest edge is (5, 3)



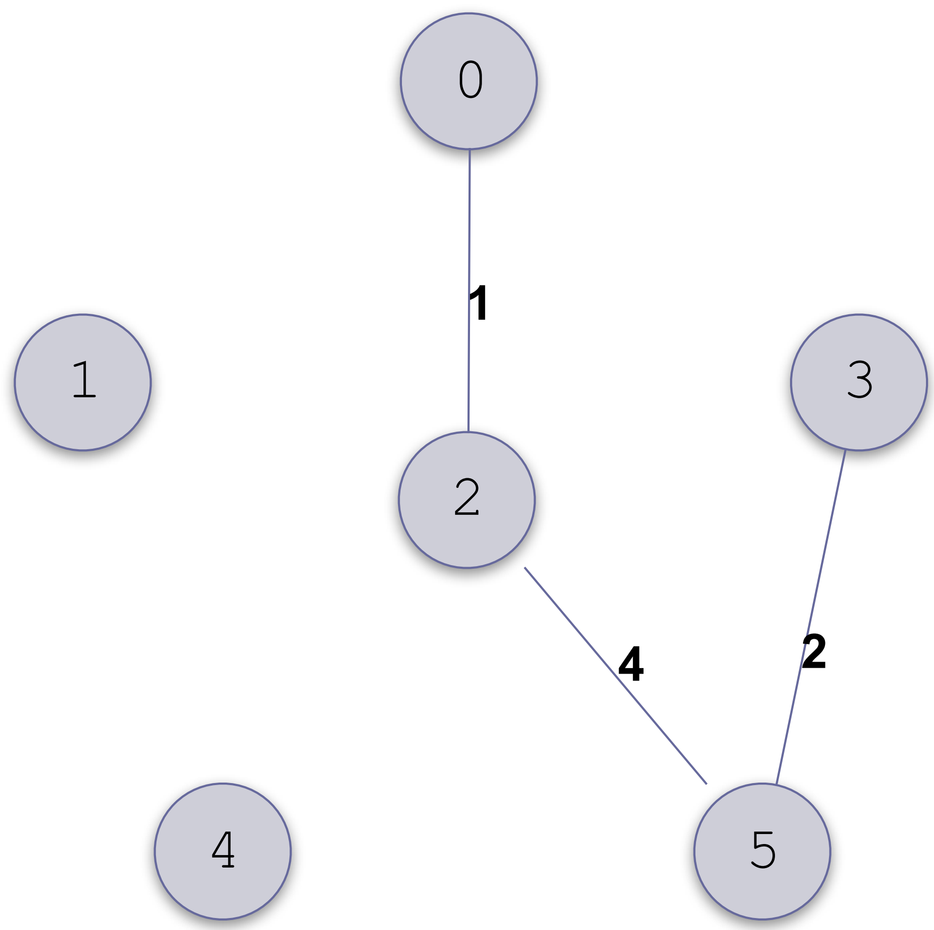


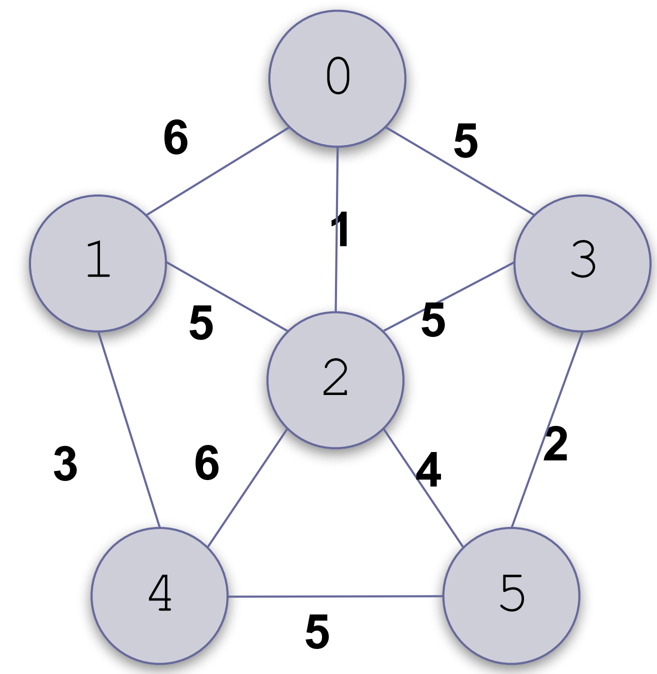
Prim's Algorithm Example (cont.)

S = **{ 0, 2, 5 }**

V-S = **{ 1, 3, 4 }**

Move 3 to *S*

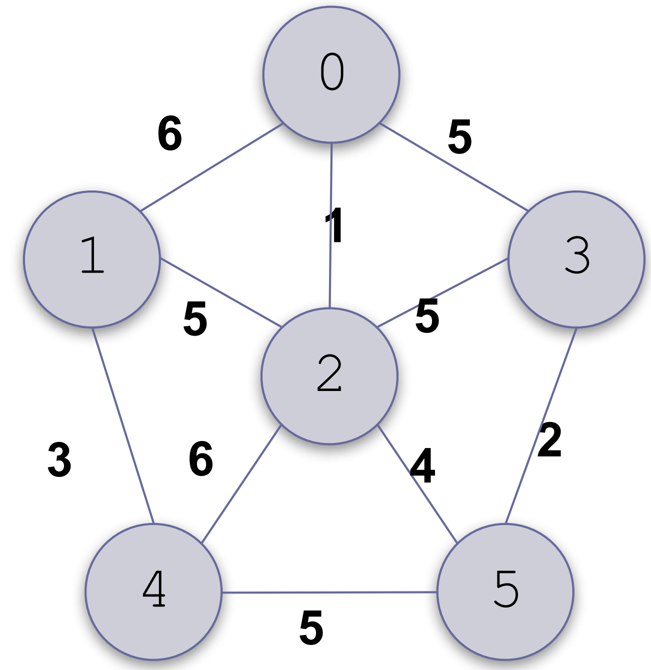




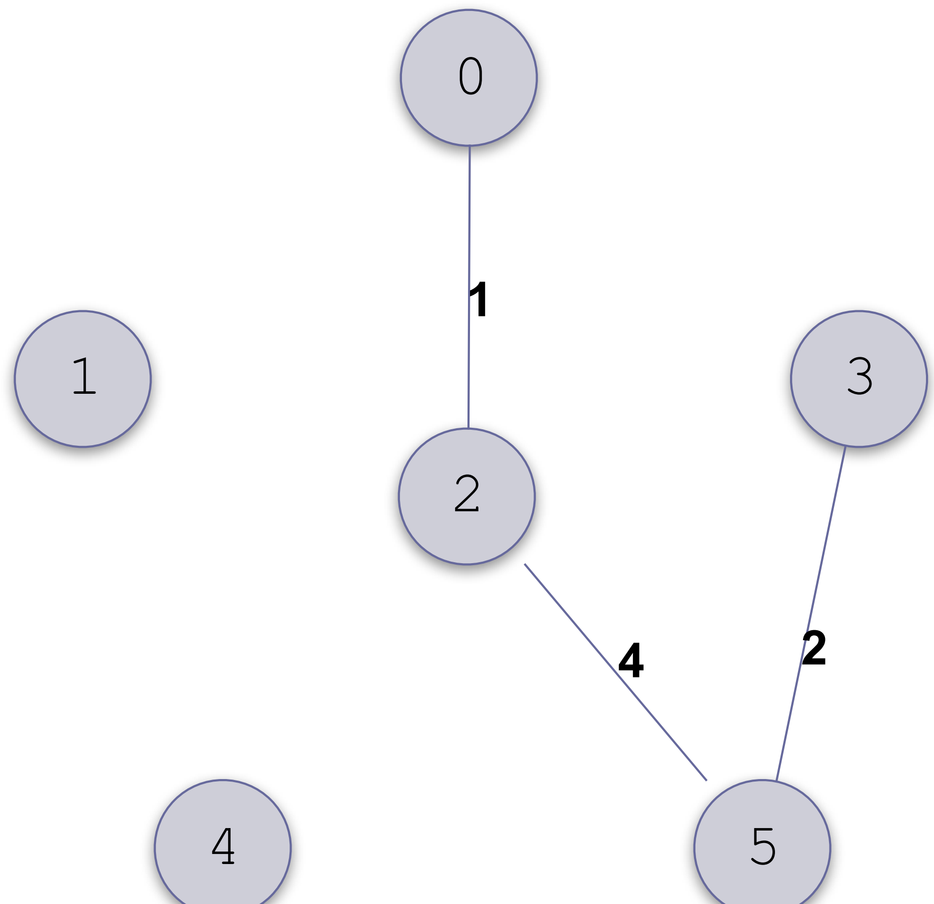
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3 }

V-S = { 1, 4 }



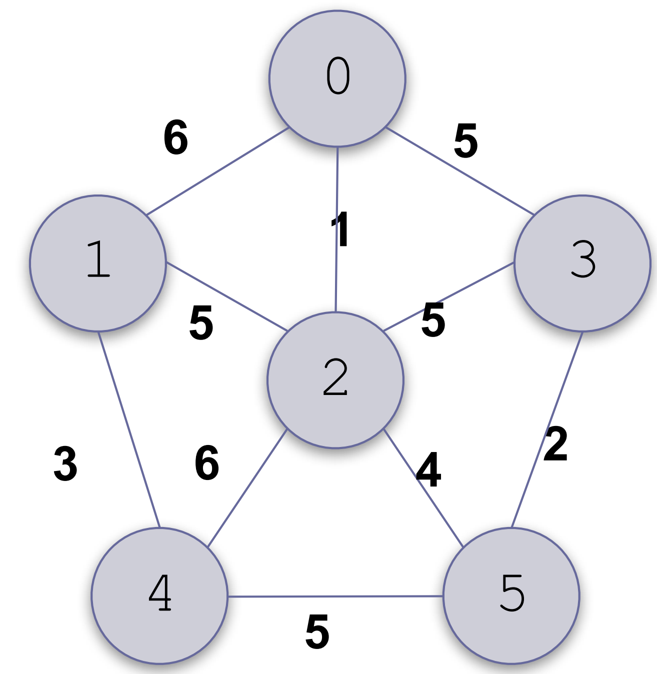
Move 3 to *S*



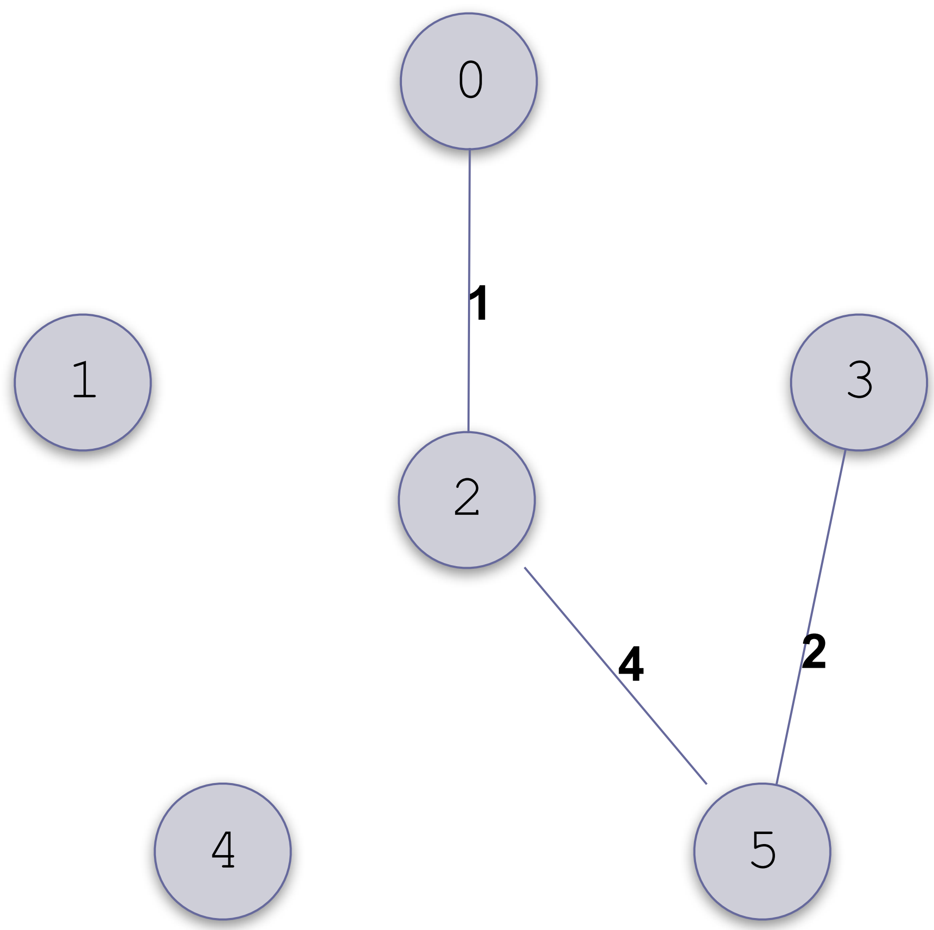
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3 }

V-S = { 1, 4 }



The next smallest edge is (2, 1)

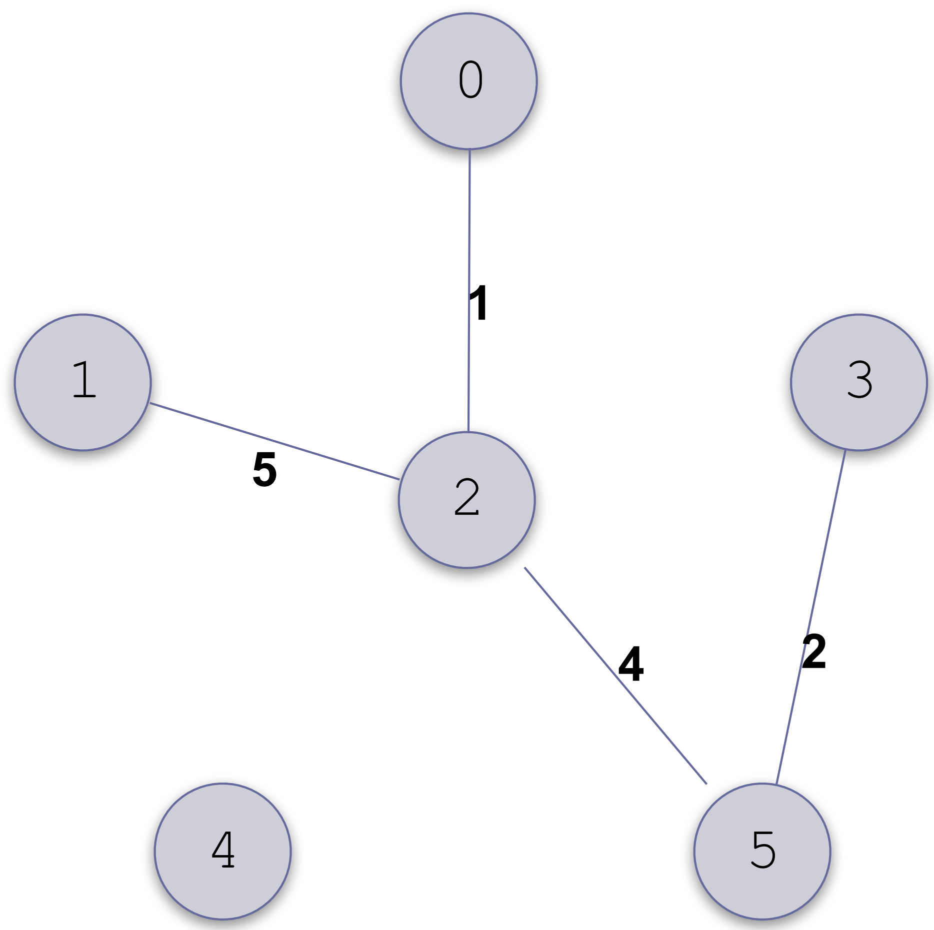


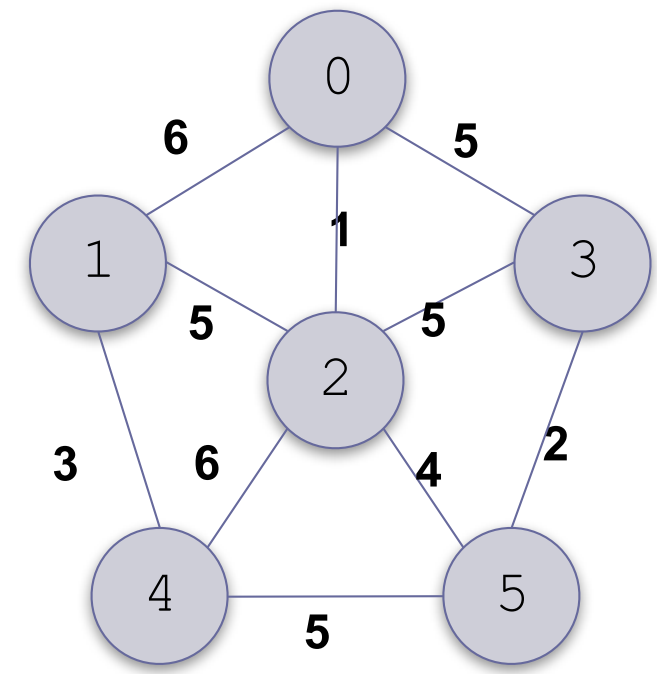
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3 }

V-S = { 1, 4 }

The next smallest edge is (2, 1)

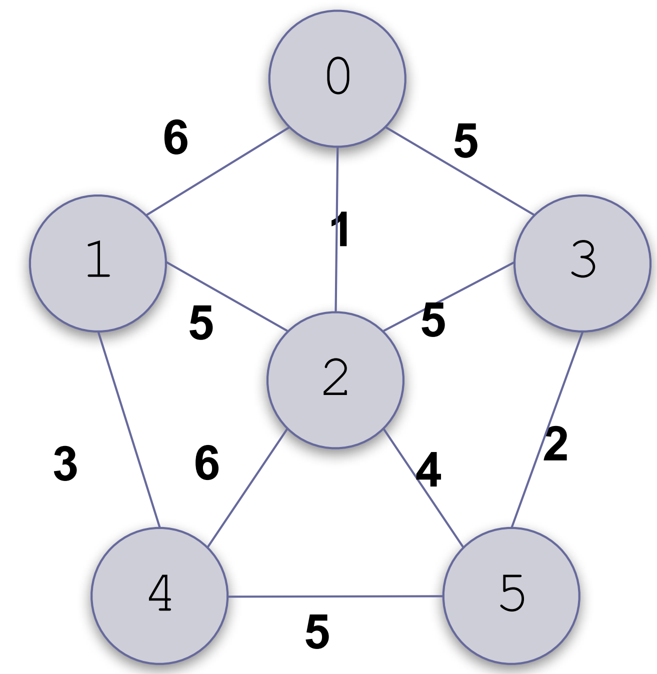




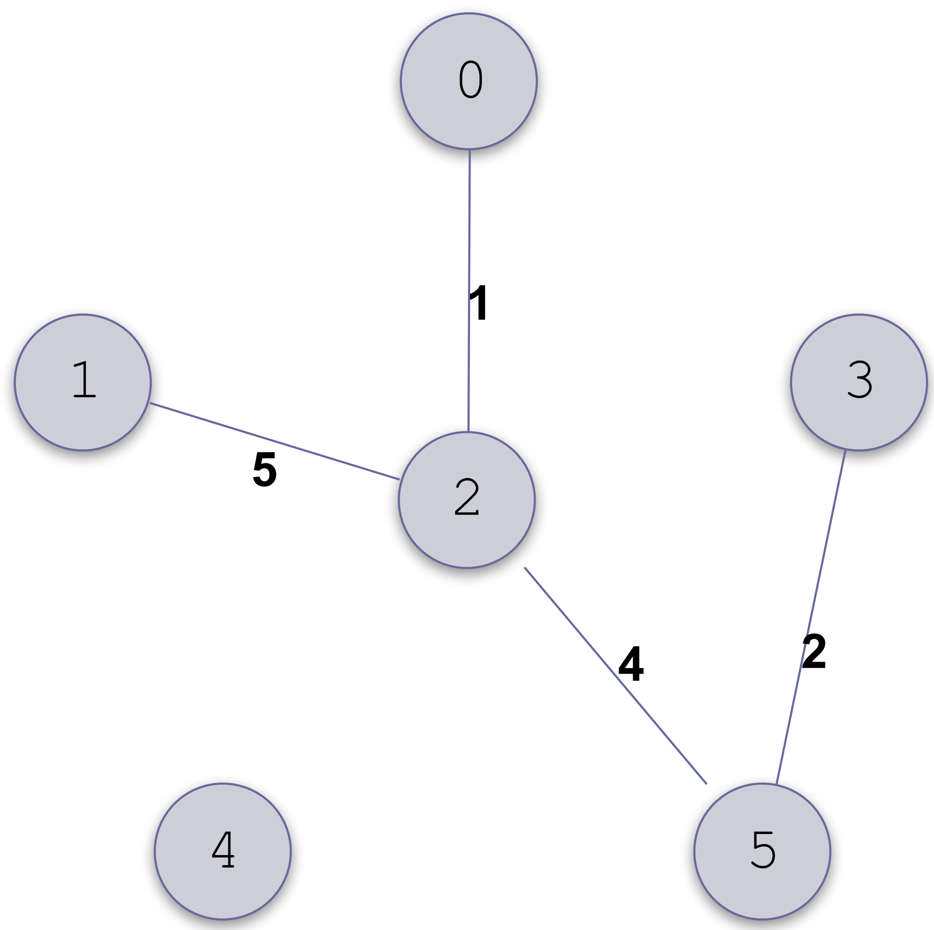
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3 }

V-S = { 1, 4 }



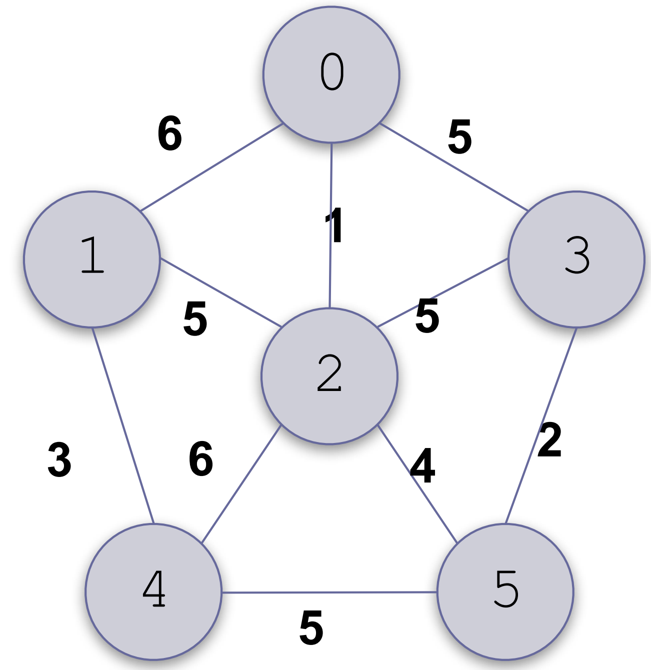
Move 1 to *S*



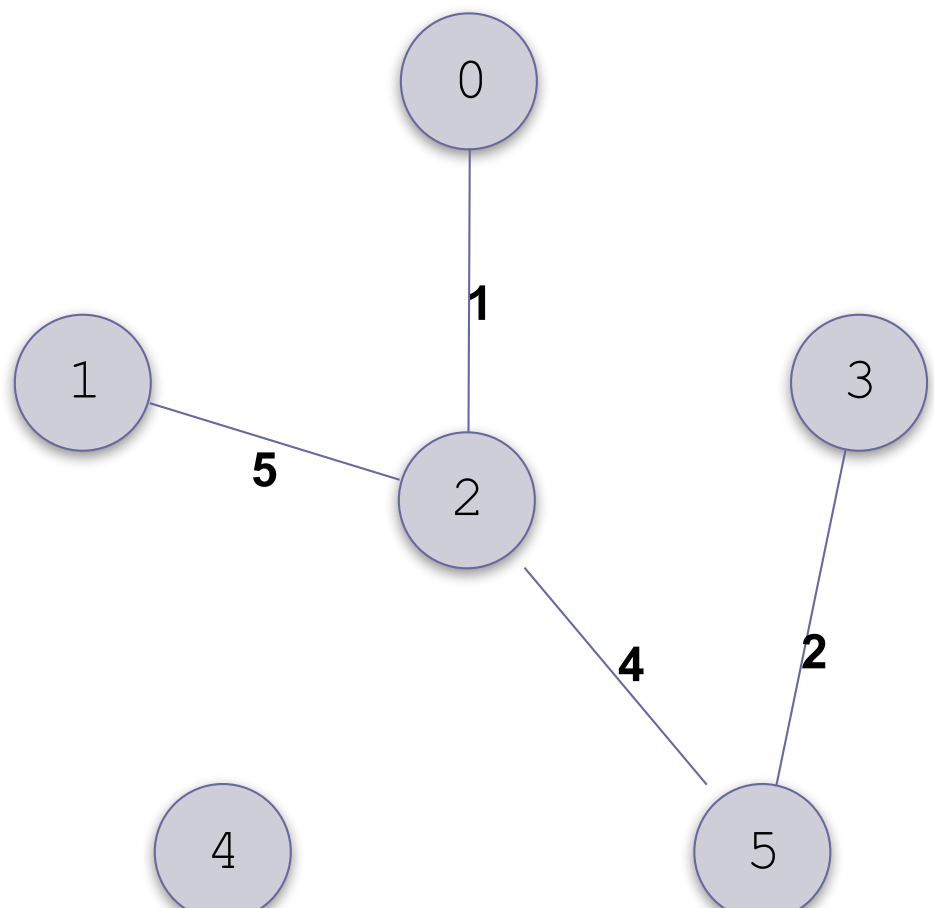
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3, 1 }

V-S = { 4 }



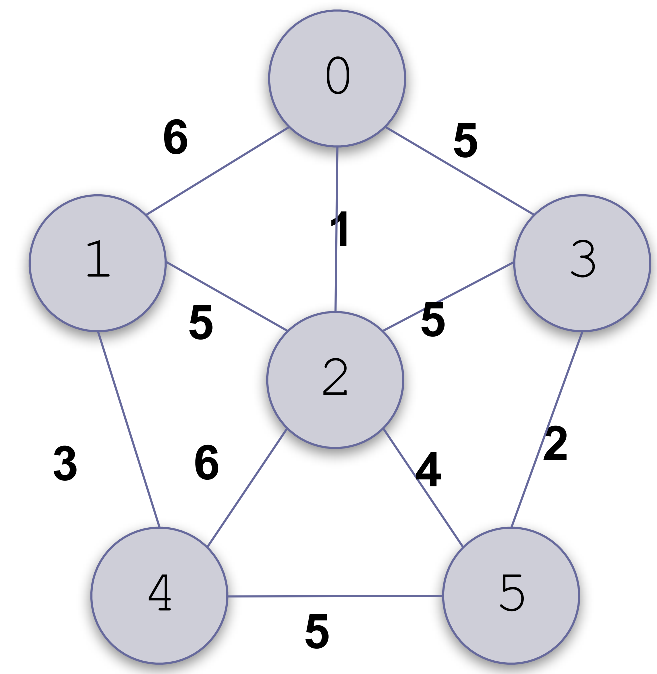
Move 1 to S



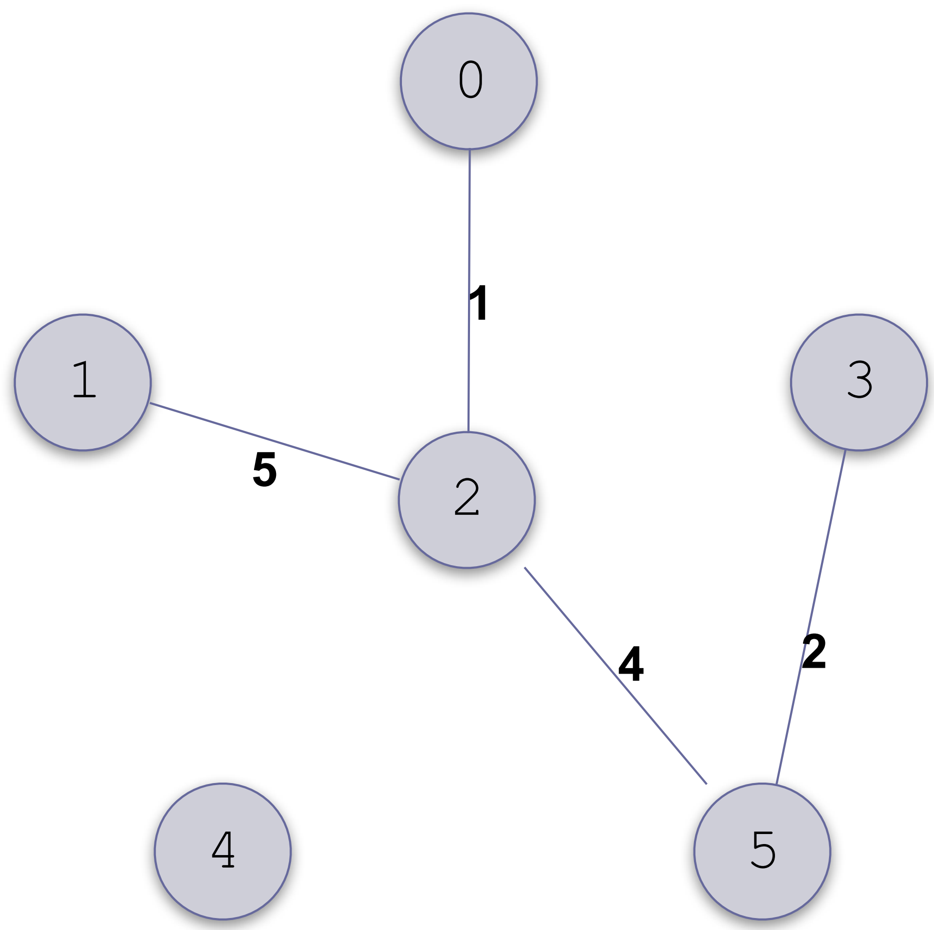
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3, 1 }

V-*S* = { 4 }



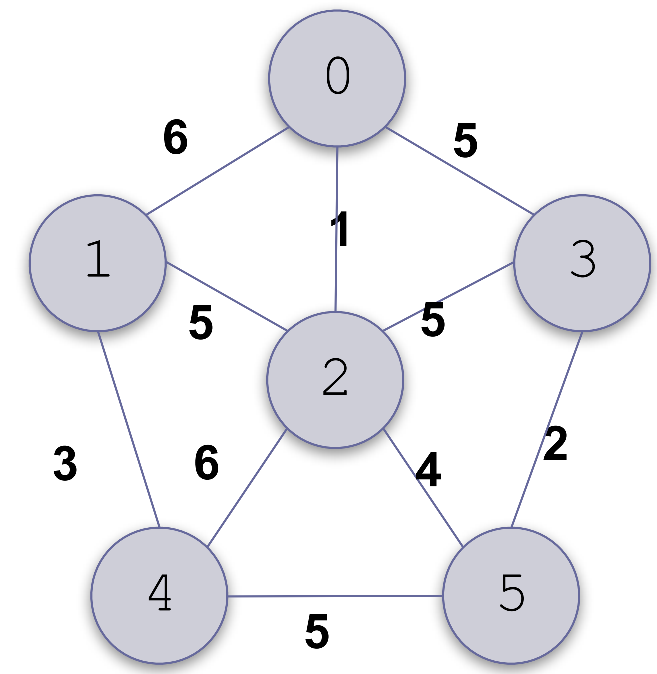
The smallest edge to 4 is (1, 4)



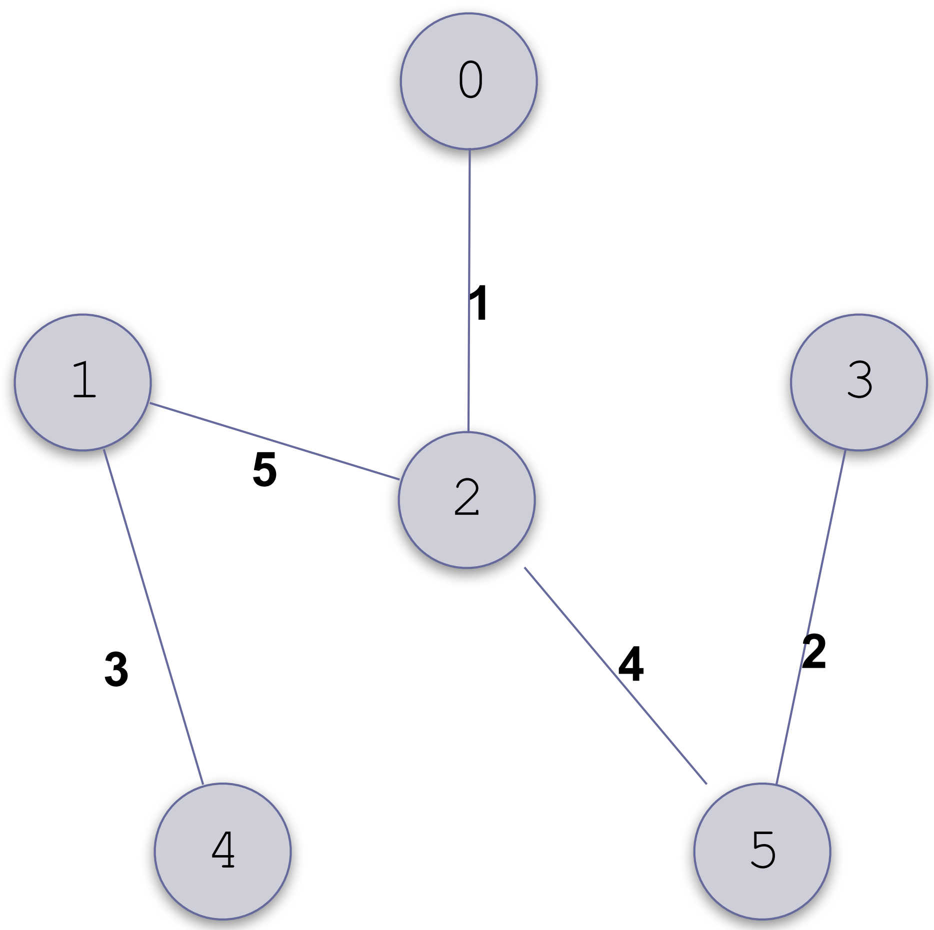
Prim's Algorithm Example (cont.)

S = { 0, 2, 5, 3, 1 }

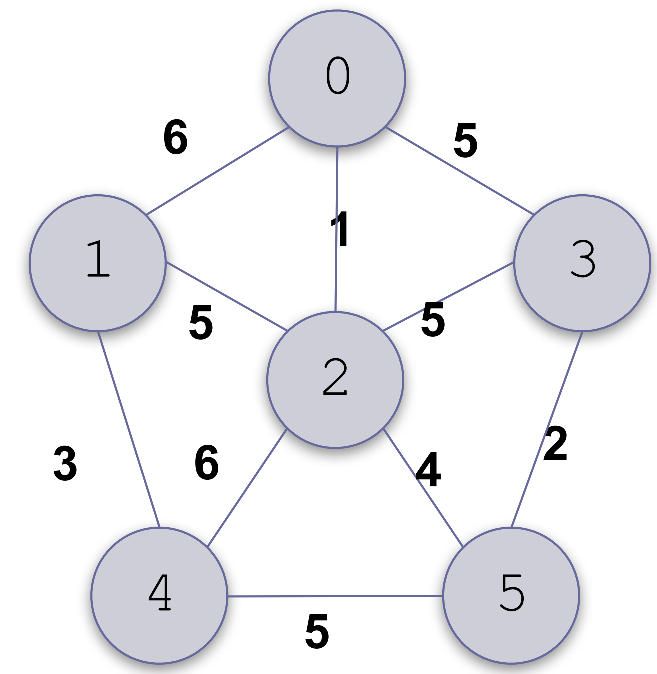
V-S = { 4 }



The smallest edge to 4 is (1, 4)



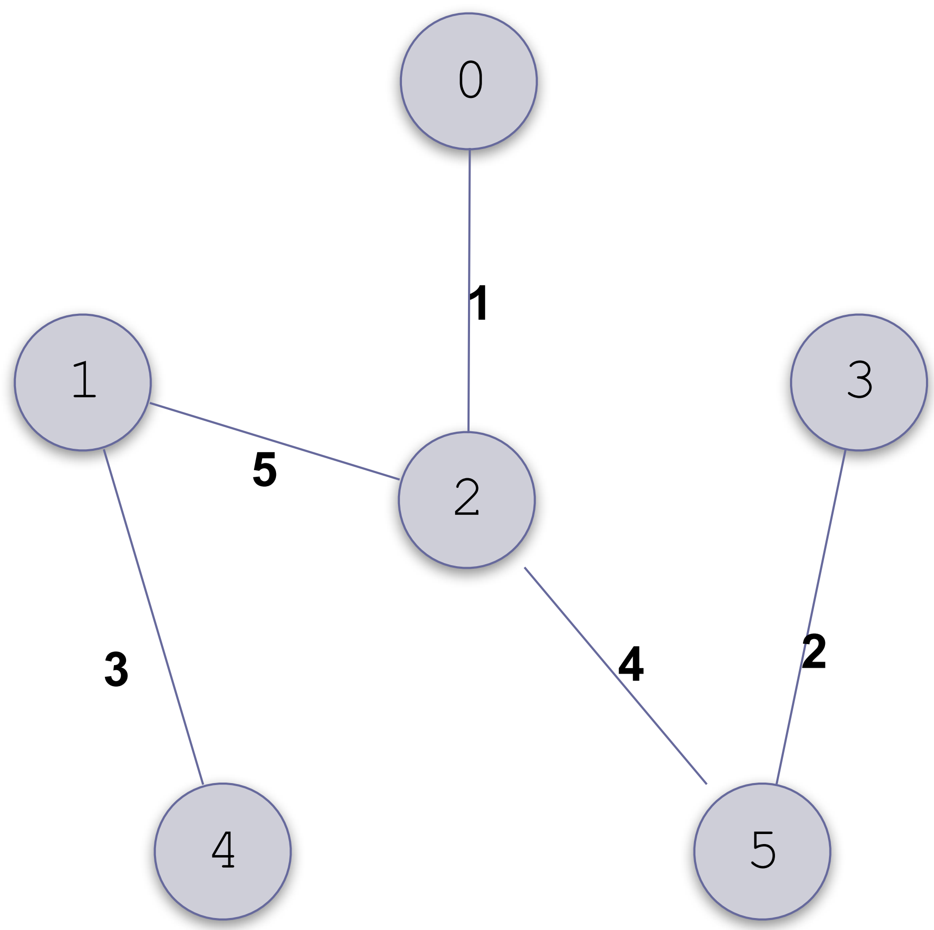
Prim's Algorithm Example (cont.)



S = { 0, 2, 5, 3, 1, 4 }

V-S = { }

The spanning tree is complete



Dijkstras algoritm

Dijkstra's Algorithm

```
1.    Initialize S with the start vertex, s, and V–S with the remaining vertices.
2.    for all v in V–S
3.        Set p[v] to s.
4.        if there is an edge (s, v)
5.            Set d[v] to w(s, v).
          else
6.            Set d[v] to ∞.
7.    while V–S is not empty
8.        for all u in V–S, find the smallest d[u].
9.        Remove u from V–S and add u to S.

10.       for all v adjacent to u in V–S
11.           if d[u] + w(u, v) is less than d[v].
12.               Set d[v] to d[u] + w(u, v).
13.               Set p[v] to u.
```

Prims algoritm

Prim's Algorithm

```
1.  Initialize S with the start vertex, s, and V–S with the remaining vertices.
2.  for all v in V–S
3.      Set p[v] to s.
4.      if there is an edge (s, v)
5.          Set d[v] to w(s, v).
        else
6.          Set d[v] to ∞.
7.  while V–S is not empty
8.      for all u in V–S, find the smallest d[u].
9.      Remove u from V–S and add u to S.
        Insert the edge (u, p[u]) into the spanning tree.
10.     for all v adjacent to u in V–S
11.         if d[u] + w(u, v) is less than d[v].
12.             Set d[v] to w(u, v).
13.             Set p[v] to u.
```

Analys av algoritmerna

Steg 1 kräver O(|V|) steg

Loopen i steg 2–6 kräver O(|V|) steg

Loopen i steg 7–12 gås igenom O(|V|) gånger

- varje gång letar steg 8 och 9 igenom **V–S**, som har storlek O(|V|)
- alltså kräver steg 7–12 $O(|V|^2)$ steg

Prims/Jarníks algoritm är alltså $O(|V|^2)$

- samma resonemang gäller för Dijkstra
- om man använder en prioritetskö för att representera **V–S**, och representerar grafen som en adjacency list så kan man få komplexitet O(|E| + |V| log |V|)