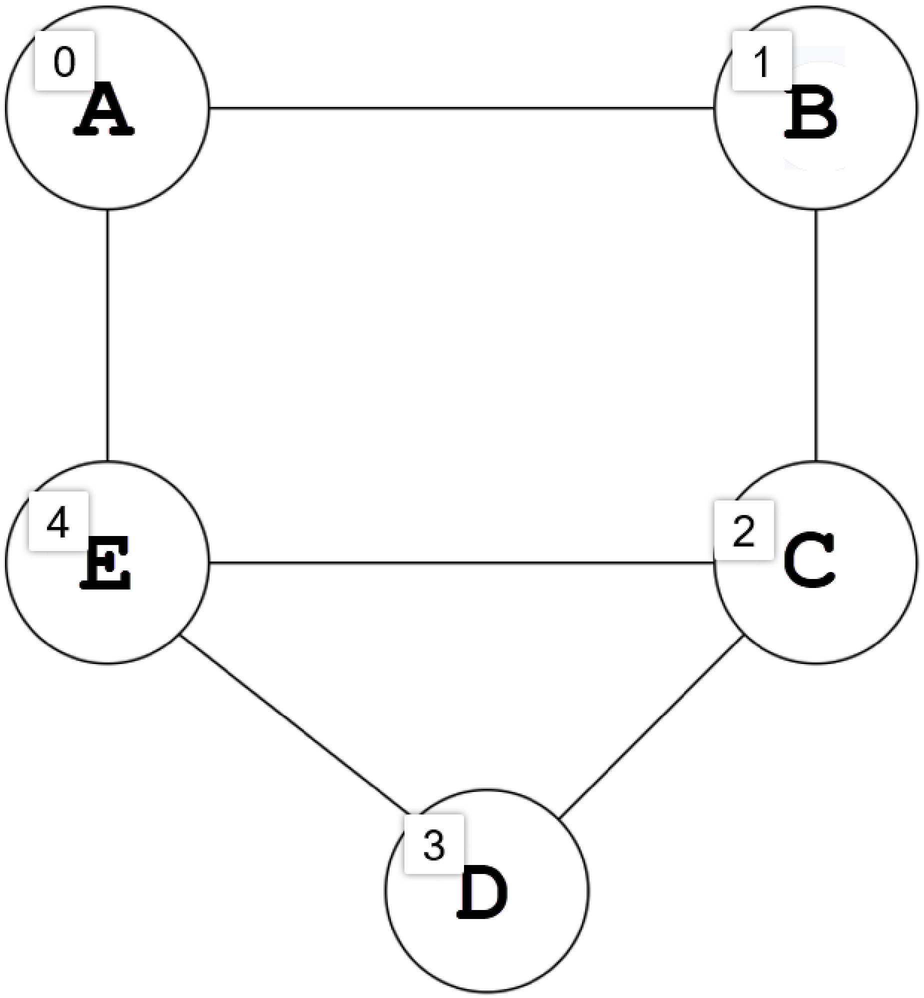
CSE 2320

Week of 07/20/2020

Instructor : Donna French

Breadth-first Search

0	1	4	2	3



Breadth-first Search

How long does breadth-first search take for a graph with vertex set V and edge set E?

$O(V+E)$

What does this mean?

Let's make some assumptions…

Breadth-first Search

Assume that $|E| \geq |V|$

This is true for most graphs – especially for the graphs where we want to use Breadth-first search.

$|V| + |E| \leq |E| + |E| = 2|E|$

So when $|E| \geq |V|$, $O(V + E)$ really means $O(E)$

Asymptotic notation ignores constant factors, so the 2 is dropped.

Breadth-first Search

Now assume that $|E| < |V|$

This is possible…

$|V| + |E| \leq |V| + |V| = 2|V|$

So when $|E| < |V|$, $O(V + E)$ really means $O(V)$

Asymptotic notation ignores constant factors, so the 2 is dropped.

Breadth-first Search

So $|E| \geq |V|$ results in $O(E)$ and $|E| < |V|$ results in $O(V)$

$O(V + E)$

We visited each vertex once – we put the vertices in a queue to keep from visiting them again.

We examine the edges incident on a vertex only when we visit.

Each edge is examined at most twice, once for each of the vertices it is incident on.

Breadth-first Search

The BFS algorithm is particularly useful for one thing – finding the shortest path on unweighted graphs.

BFS can find the shortest path for an **unweighted** graph.

We will use a different algorithm to find the shortest path on a weighted graph.

Shortest does not mean unique – there can be multiple shortest paths.

Breadth-first Search

Shortest does not mean unique – there can be multiple shortest paths.

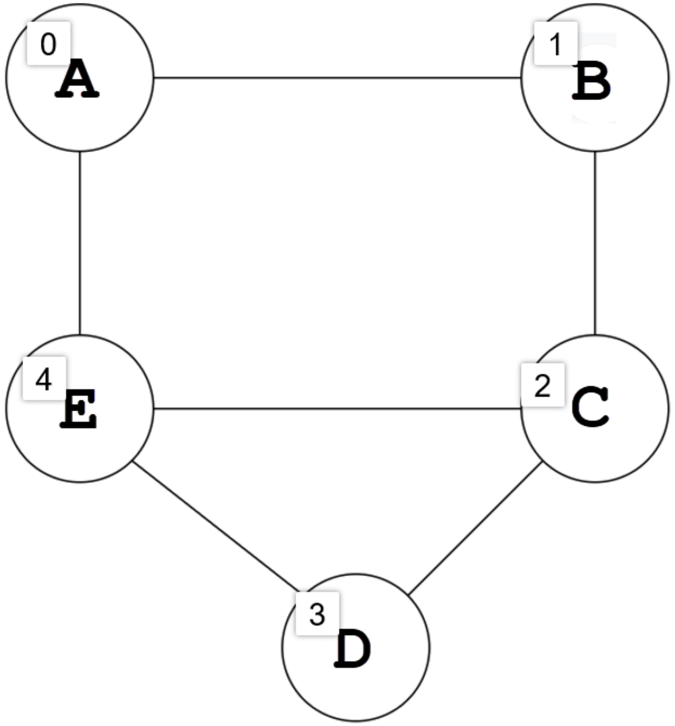
Which shortest path will be chosen will depend on the order of the vertices and how they are processed during the BFS.

Since all edges are equal in an unweighted graph, all shortest paths are valid but only one will be chosen by the code.

Breadth-first Search

The first step is to add more information to our vertex structure.

```
typedef struct
{
    char label;
    int distance;
    int previous;
    int visited;
}
Vertex;
```

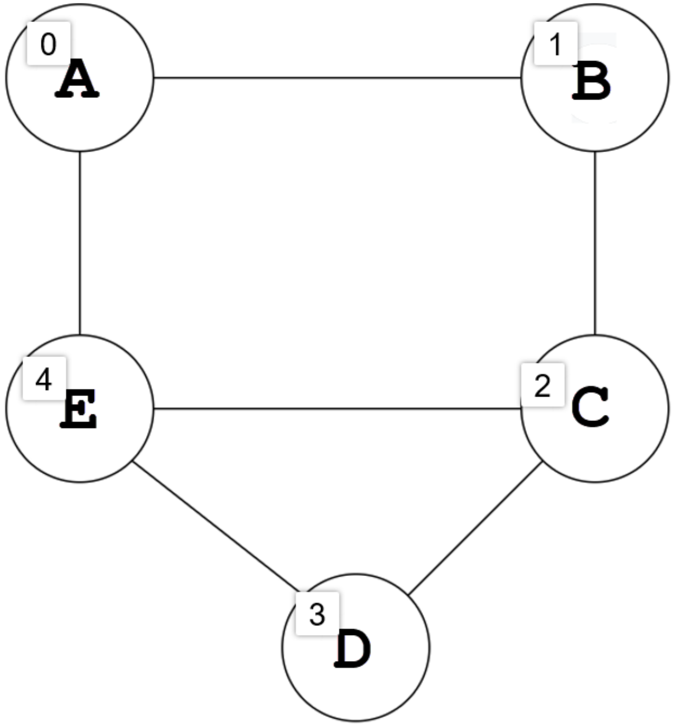


Breadth-first Search

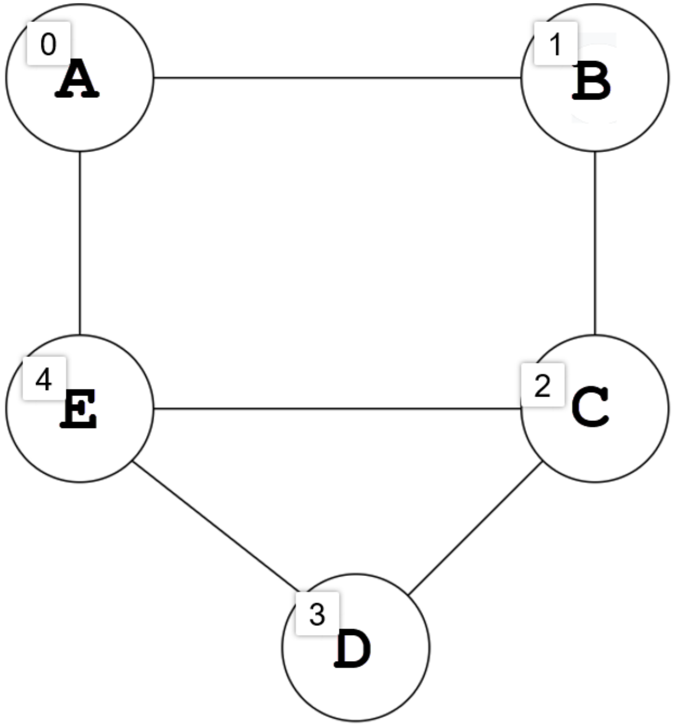
When doing a BFS, we pick one vertex to be our starting point. That vertex is the vertex we put into the vertex array at cell 0.

The new `distance` attribute in our vertex is recording how away from the starting vertex that vertex is.

Vertex B is 1 away from Vertex A (our starting vertex) so `distance` in Vertex B would be set to 1.

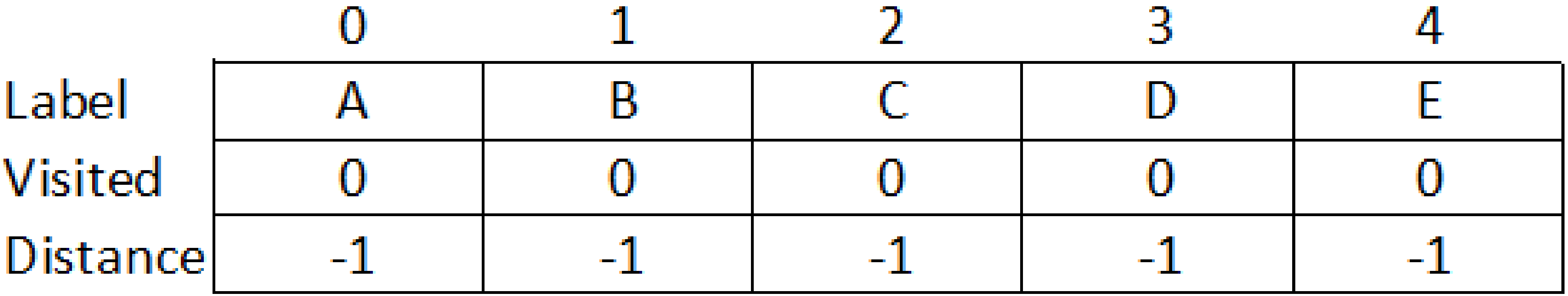


Breadth-first Search



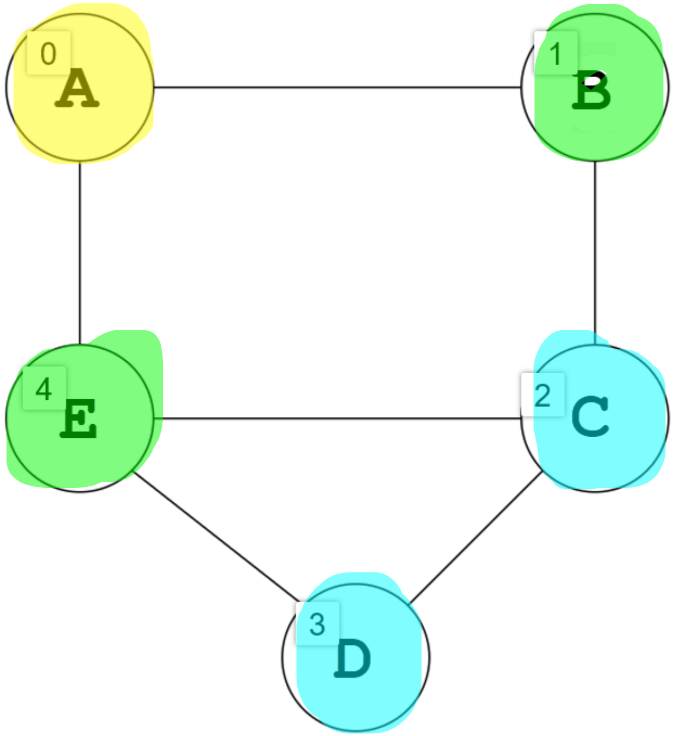
So how would we fill in `distance` in the Vertex Array?

We initialize `distance` to -1 to show a distance has either not be calculated or that there is no edge connecting that vertex to the starting vertex.



	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0
Distance	-1	-1	-1	-1	-1

Breadth-first Search



Notice that the distance of each vertex from the source/starting vertex corresponds to the levels we create when doing a BFS.

A, B, E, C, D

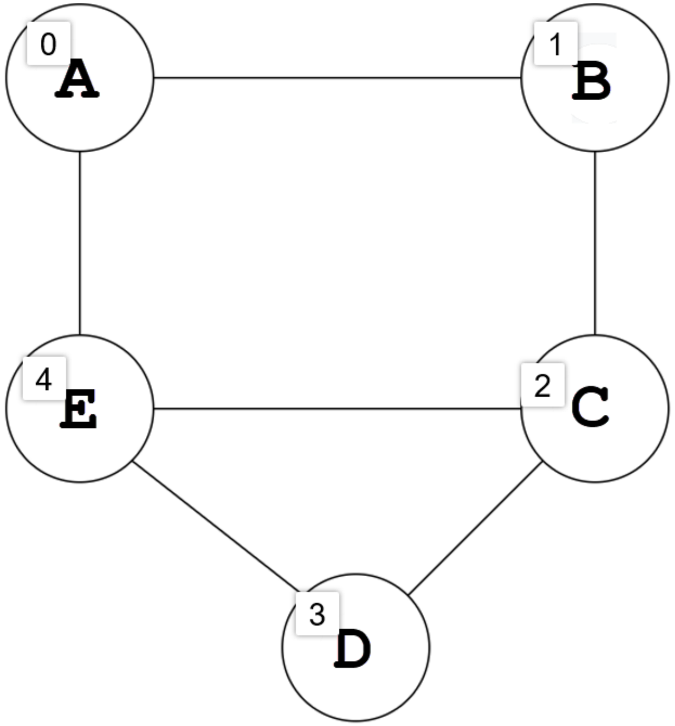
	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0
Distance	0	1	2	2	1

Breadth-first Search

Now we want to fill in our other new attribute, `previous`.

The `previous` member of the `Vertex` struct contains the index of the vertex we just passed through on our way to the current vertex.

For example, to get to vertex B, we started at vertex A so we would put vertex A's index value into vertex B's `previous`.

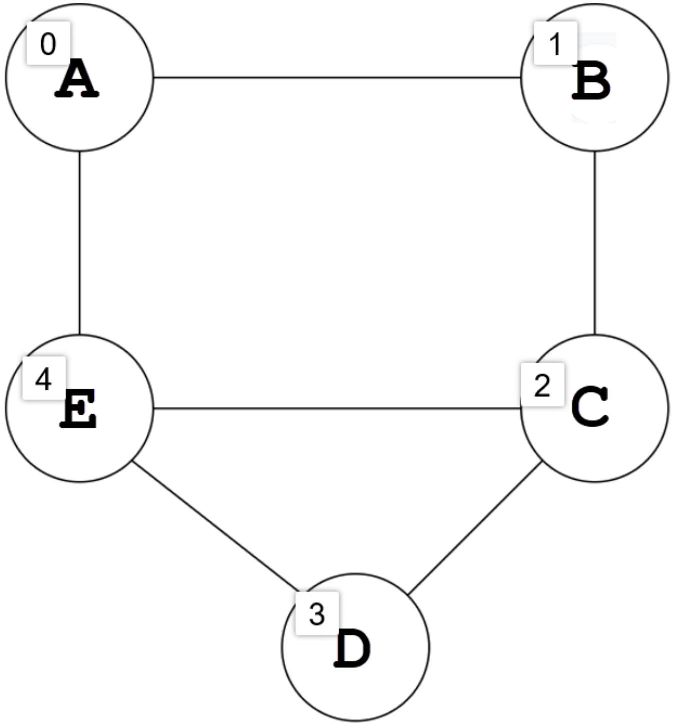


Breadth-first Search

So how would we fill in `previous` in the Vertex Array?

We initialize `previous` to -1 to show that this attribute has not been set or that the vertex is not connected to another vertex.

	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0
Distance	0	1	2	2	1
Previous	-1	-1	-1	-1	-1



Breadth-first Search

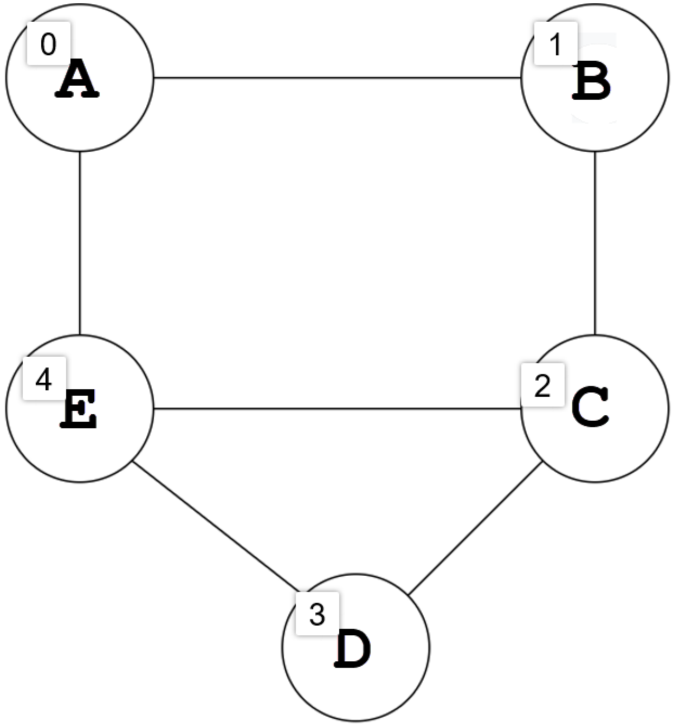
As we do our BFS, we will fill in `previous`.

We start with A.

From A, we go to B and E.

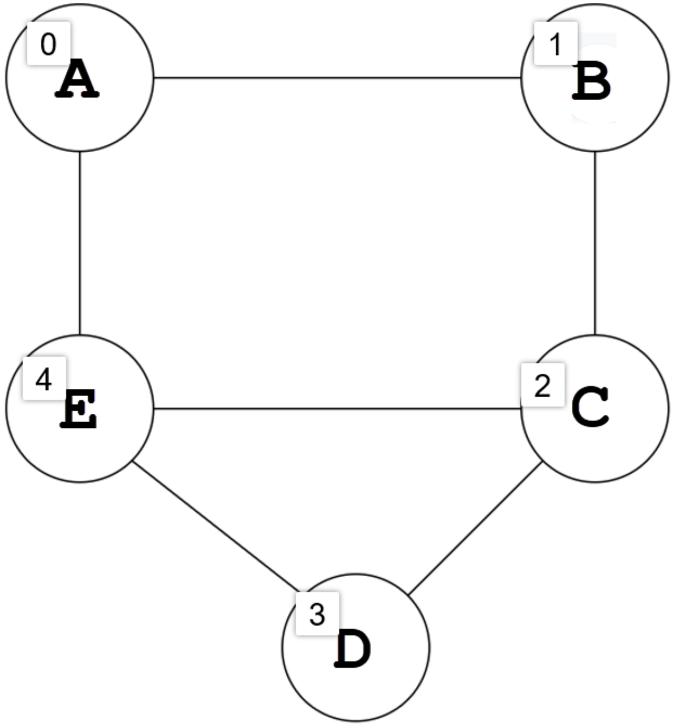
We would set `previous` in B and E to A.

	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0
Distance	0	1	2	2	1
Previous	-1	-1	-1	-1	-1



Breadth-first Search

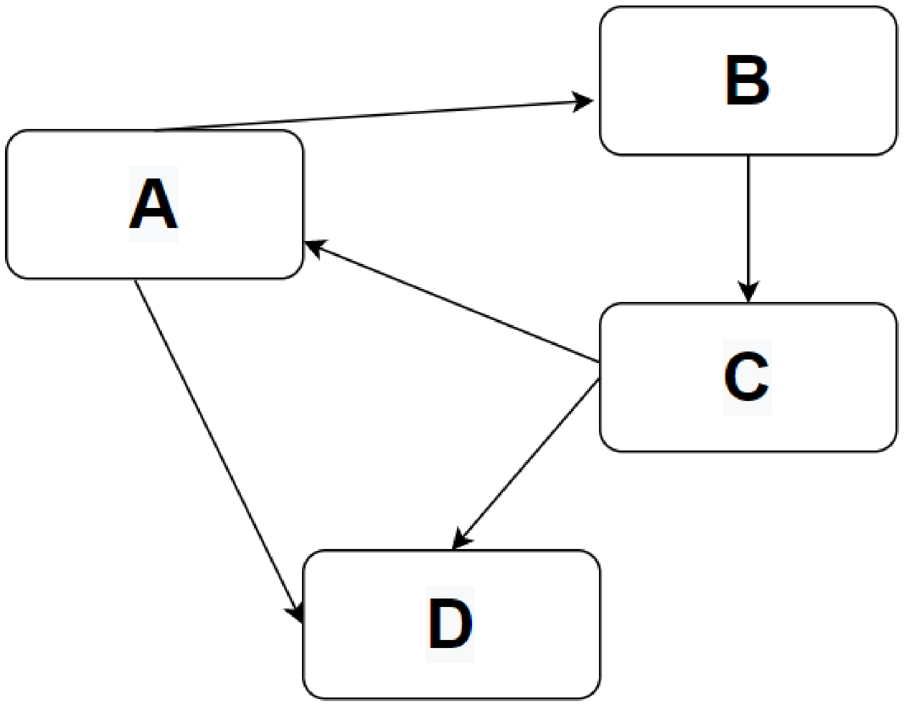
	0	1	2	3	4
Label	A	B	C	D	E
Visited	0	0	0	0	0
Distance	-1	1	2	2	1
Previous	-1	0	1	4	0



Breadth-first Search

	0	1	2	3
Label	A	B	C	D
Visited	0	0	0	0
Distance	-1	-1	-1	-1
Previous	-1	-1	-1	-1

	0	1	2	3
Label	A	B	C	D
Visited	1	1	1	1
Distance	0	1	2	1
Previous	-1	0	1	0



Breadth-first Search

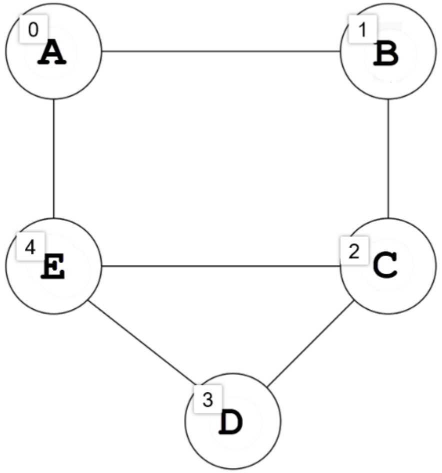
```
void BreadthFirstSearch(Vertex *VertexArray[], int VertexCount, int AdjMatrix[][MAX])
{
      int tail = -1;
      int head = -1;
      int i = 0;
      int queueItemCount = 0;
      int queue[MAX] = {};
      int CurrentVertexIndex = 0;

      VertexArray[0]->visited = 1;
      VertexArray[0]->previous = -1;
      VertexArray[0]->distance = 0;
```

Breadth-first Search

```
while(queueItemCount)
{
    CurrentVertexIndex = dequeue(queue, &head, &tail);
    queueItemCount--;

    for (i = 0; i < VertexCount; i++)
    {
        if (AdjMatrix[CurrentVertexIndex][i] == 1)   /* Found a neighbor */
        {
            if (VertexArray[i]->visited == 0)  // have we visited already?
            {
                enqueue(queue, &head, &tail, i);
                queueItemCount++;
                VertexArray[i]->visited = 1;
                VertexArray[i]->distance = VertexArray[CurrentVertexIndex]->distance + 1;
                VertexArray[i]->previous = CurrentVertexIndex;
            }
        }
    }
}
```

```
CurrentVertexIndex =
i =
```

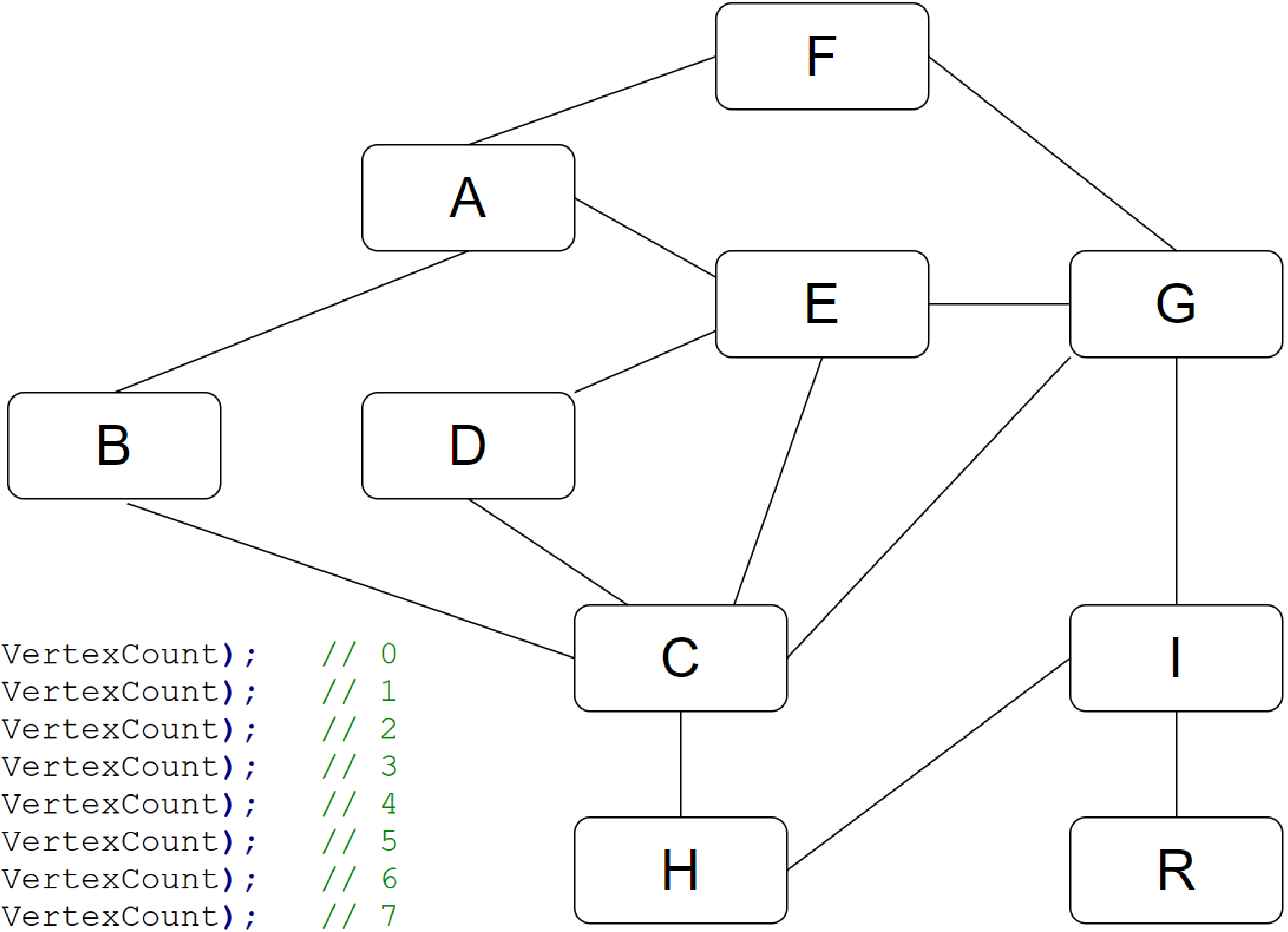
Breadth-first Search



```
enqueue(queue, &head, &tail, i);
queueItemCount++;
VertexArray[i]->visited = 1;
VertexArray[i]->distance = VertexArray[CurrentVertexIndex]->distance + 1;
VertexArray[i]->previous = CurrentVertexIndex;
```

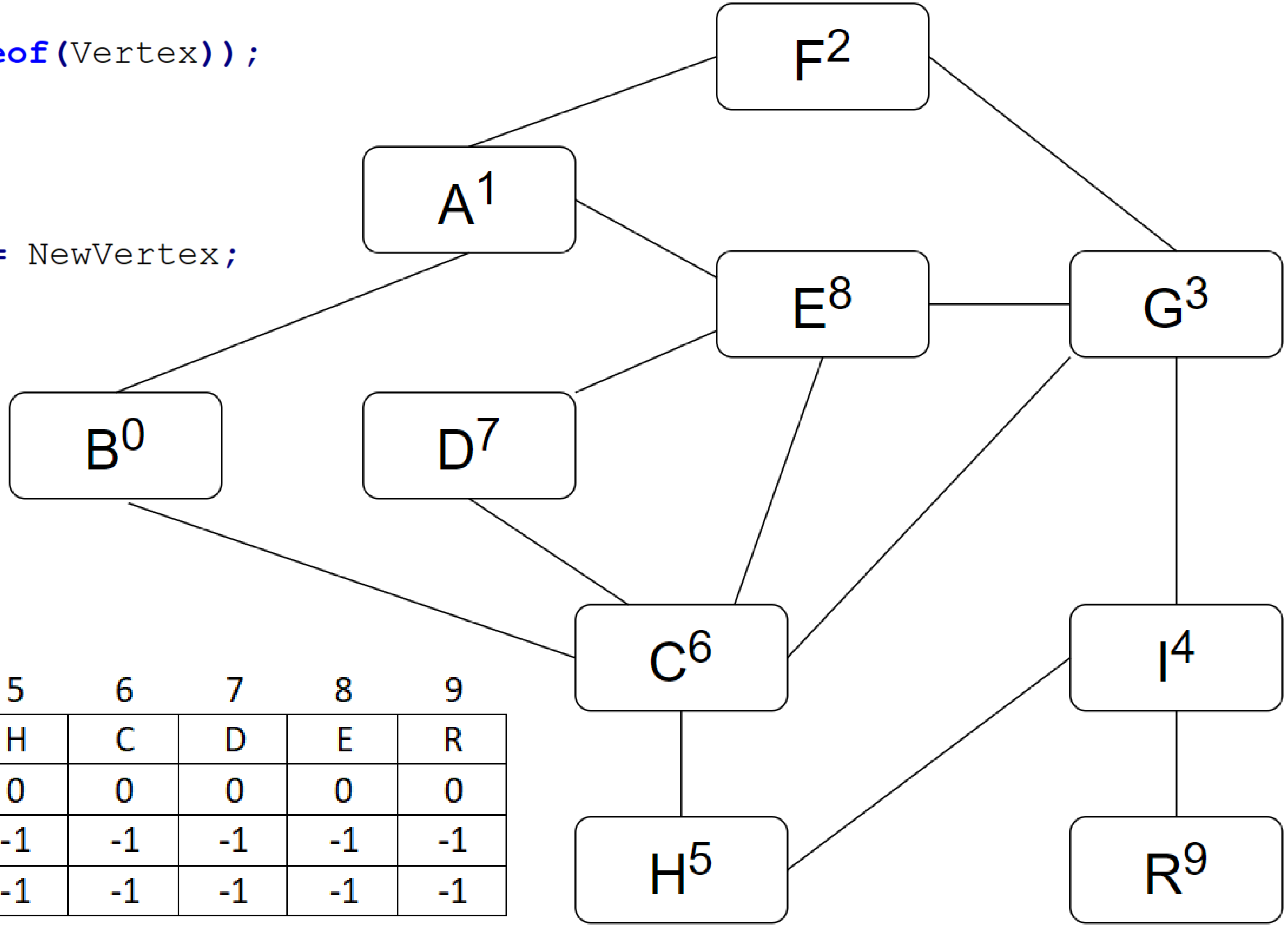
	0	1	2	3	4
Label	A	B	C	D	E
Visited	1	1	1	1	1
Distance	0	1	2	2	1
Previous	-1	0	1	4	0

```
addEdge(0, 1, AdjMatrix);
addEdge(1, 2, AdjMatrix);
addEdge(2, 3, AdjMatrix);
addEdge(3, 4, AdjMatrix);
addEdge(4, 5, AdjMatrix);
addEdge(5, 6, AdjMatrix);
addEdge(6, 0, AdjMatrix);
addEdge(8, 3, AdjMatrix);
addEdge(7, 8, AdjMatrix);
addEdge(3, 6, AdjMatrix);
addEdge(6, 7, AdjMatrix);
addEdge(6, 8, AdjMatrix);
addEdge(1, 8, AdjMatrix);
addEdge(4, 9, AdjMatrix);
```



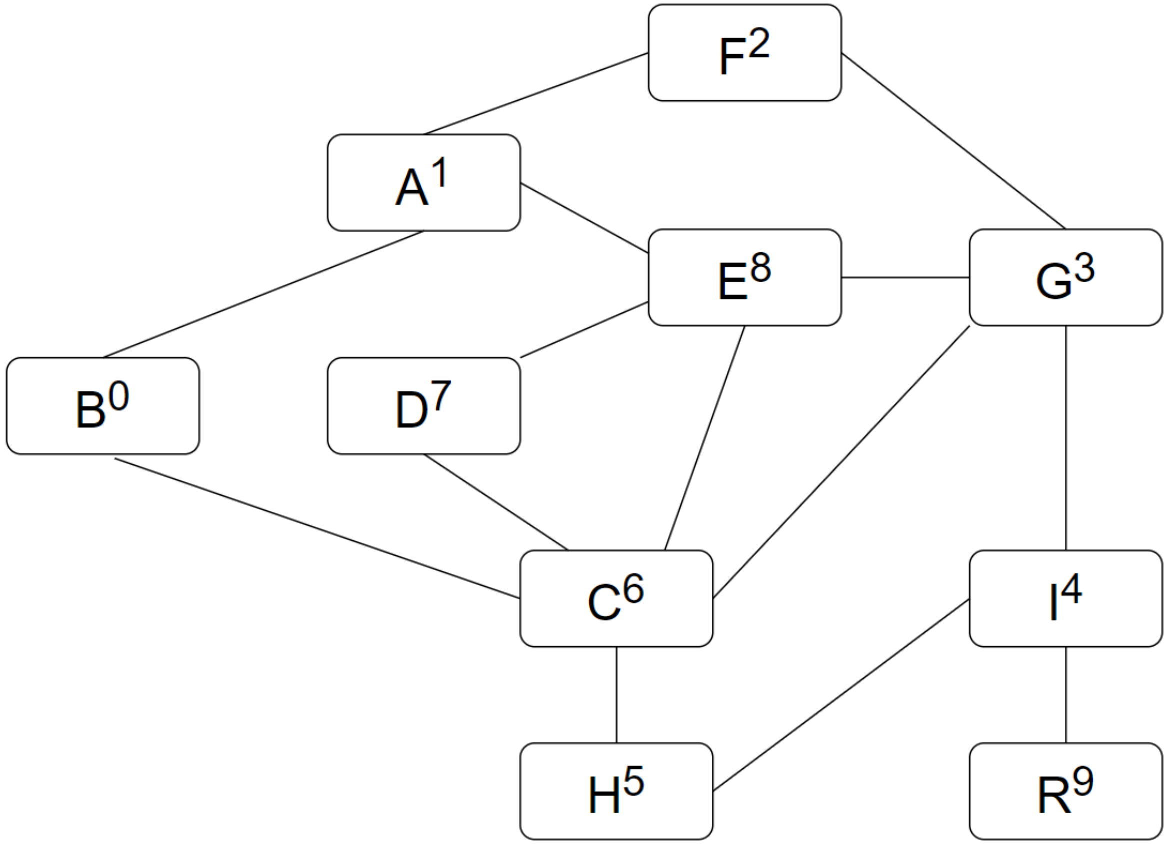
```
addVertex('B', VertexArray, &VertexCount);    // 0
addVertex('A', VertexArray, &VertexCount);    // 1
addVertex('F', VertexArray, &VertexCount);    // 2
addVertex('G', VertexArray, &VertexCount);    // 3
addVertex('I', VertexArray, &VertexCount);    // 4
addVertex('H', VertexArray, &VertexCount);    // 5
addVertex('C', VertexArray, &VertexCount);    // 6
addVertex('D', VertexArray, &VertexCount);    // 7
addVertex('E', VertexArray, &VertexCount);    // 8
addVertex('R', VertexArray, &VertexCount);    // 9
```

```
void addVertex(char label, Vertex *VertexArray[], int *VertexCount)
{
    Vertex *NewVertex = malloc(sizeof(Vertex));
    NewVertex->label = label;
    NewVertex->visited = 0;
    NewVertex->previous = -1;
    NewVertex->distance = -1;
    VertexArray[(*VertexCount)++] = NewVertex;
}
```



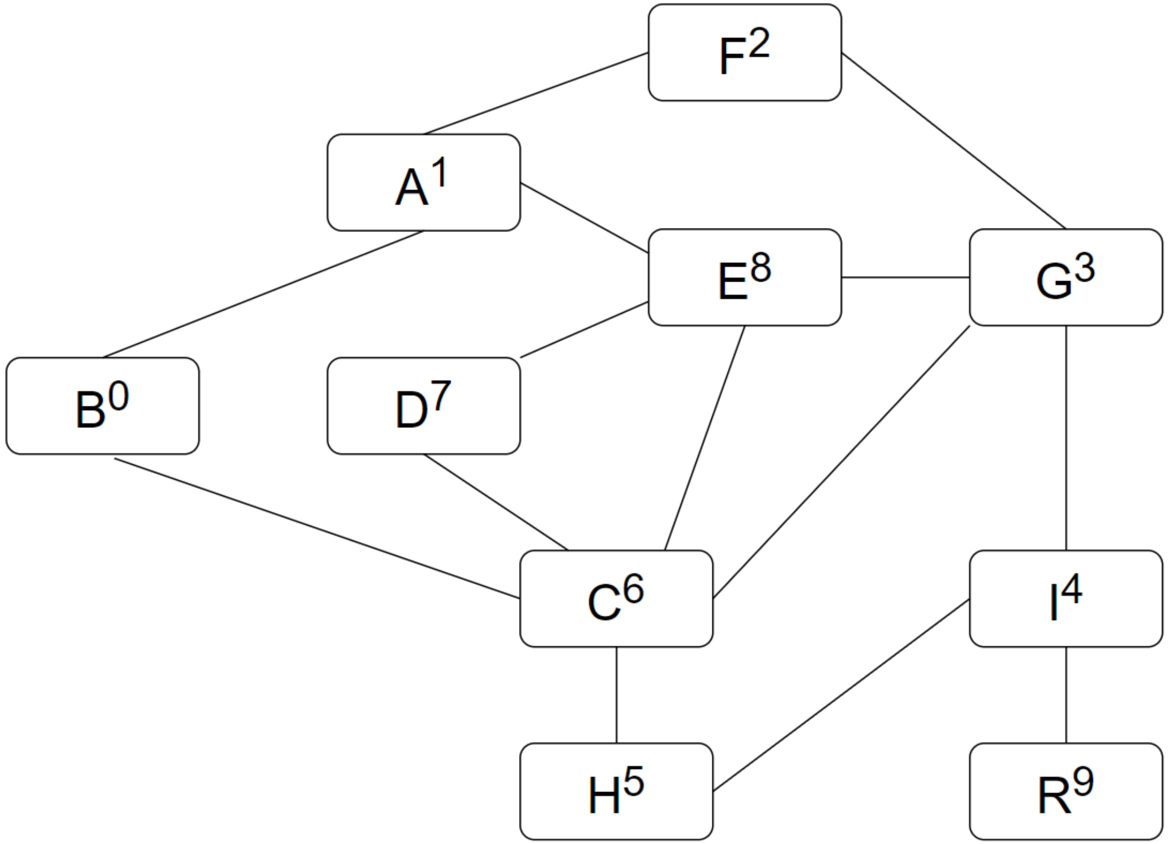
	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	0	0	0	0	0	0	0	0	0	0
Distance	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Previous	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	0	0	0	0	0	0	0	0	0	0
Distance	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Previous	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

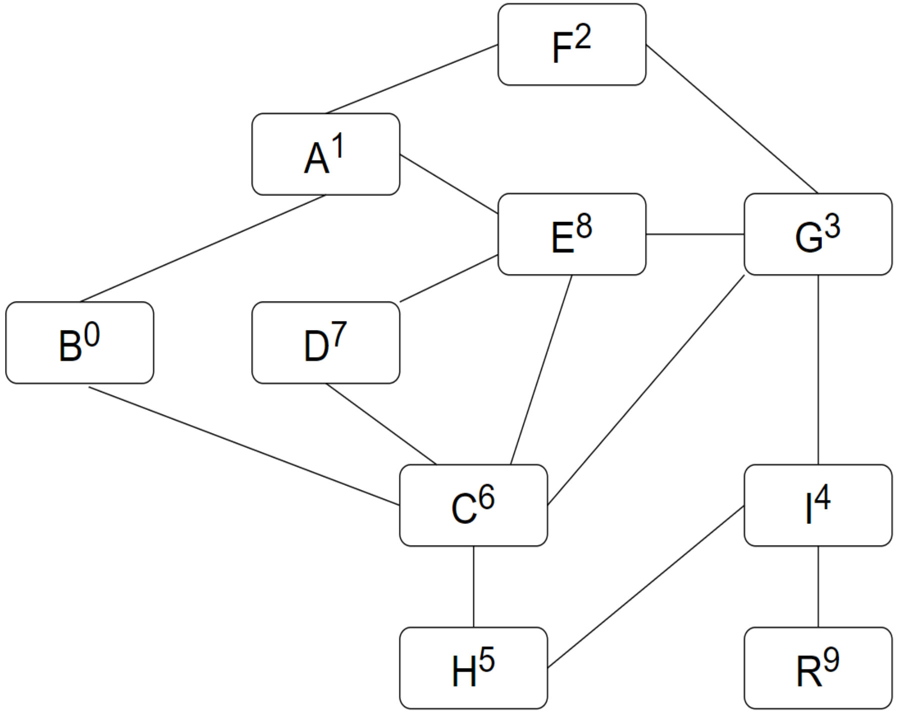


	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4

B, AC, FE GHD, I, R



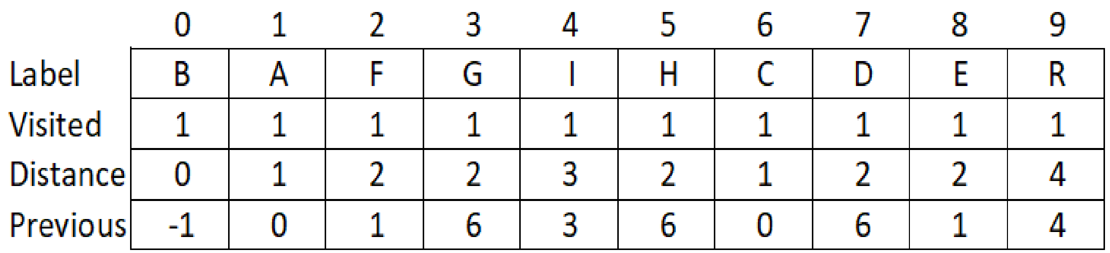
```
Source is B.  What is the destination? A
BA
Source is B.  What is the destination? C
BC
Source is B.  What is the destination? D
BCD
Source is B.  What is the destination? E
BAE
Source is B.  What is the destination? F
BAF
Source is B.  What is the destination? G
BCG
Source is B.  What is the destination? H
BCH
Source is B.  What is the destination? I
BCGI
Source is B.  What is the destination? R
BCGIR
```



	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4

B, AC, FE GHD, I, R

```
Source is B.  What is the destination? W
Destination W is not in graph

Source is B.  What is the destination? A
BA
```



	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4

```c
printf("Source is %c.  What is the destination? ", VertexArray[0]->label);
scanf("%c", &dest);
getchar();

destindex = 0;
while (destindex < VertexCount && dest != VertexArray[destindex]->label)
{
      destindex++;
}

if (destindex == VertexCount)
      printf("Destination %c is not in graph\n", dest);
else
{


}
```

	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4

```
char path[10] = {};

int pathindex = -1;

int destindex = -1;

int previndex = -1;
```

```
pathindex = VertexArray[destindex]->distance;
previndex = VertexArray[destindex]->previous;
path[pathindex] = VertexArray[destindex]->label;

while (pathindex > 0)
{
        pathindex--;
        path[pathindex] = VertexArray[previndex]->label;
        previndex = VertexArray[previndex]->previous;
}
printf("%s\n", path);
```

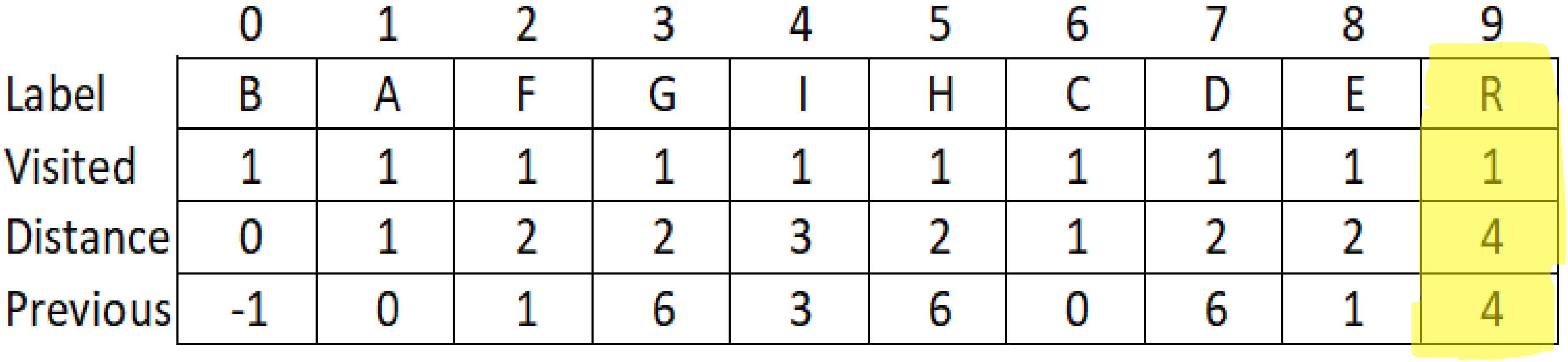
```
dest = 'A'
```

	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4

```
char path[10] = {};

int pathindex = -1;

int destindex = -1;

int previndex = -1;
```

```
pathindex = VertexArray[destindex]->distance;
previndex = VertexArray[destindex]->previous;
path[pathindex] = VertexArray[destindex]->label;

while (pathindex > 0)
{
      pathindex--;
      path[pathindex] = VertexArray[previndex]->label;
      previndex = VertexArray[previndex]->previous;
}
printf("%s\n", path);
```

```
dest = 'G'
destindex = 3
```



	0	1	2	3	4	5	6	7	8	9
Label	B	A	F	G	I	H	C	D	E	R
Visited	1	1	1	1	1	1	1	1	1	1
Distance	0	1	2	2	3	2	1	2	2	4
Previous	-1	0	1	6	3	6	0	6	1	4

```
char path[10] = {};

int pathindex = -1;

int destindex = -1;

int previndex = -1;
```

```
pathindex = VertexArray[destindex]->distance;
previndex = VertexArray[destindex]->previous;
path[pathindex] = VertexArray[destindex]->label;

while (pathindex > 0)
{
      pathindex--;
      path[pathindex] = VertexArray[previndex]->label;
      previndex = VertexArray[previndex]->previous;
}
printf("%s\n", path);
```

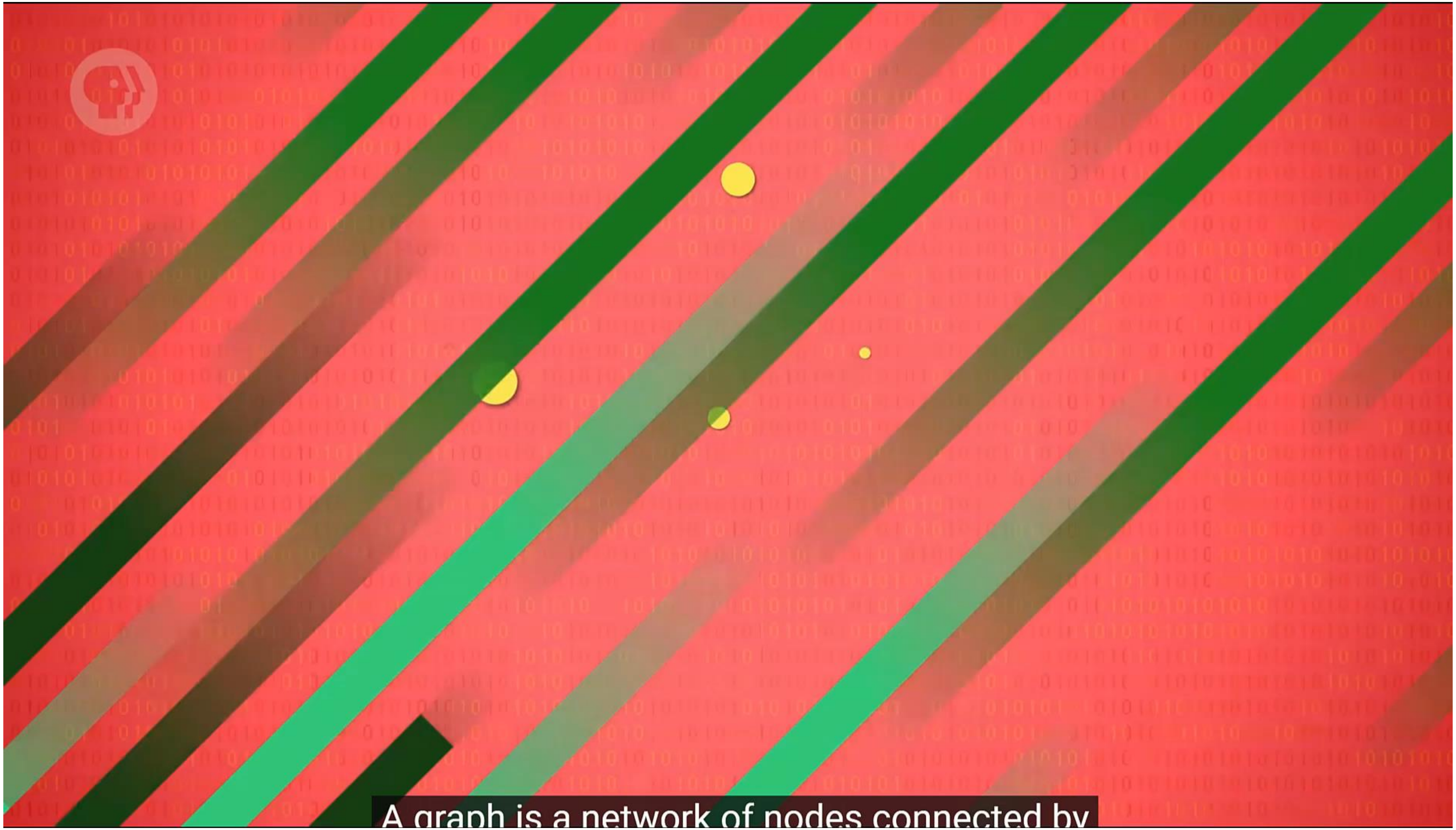
```
dest = 'R'
destindex = 9
```

Breadth-first Search

Using the Breadth-first Search technique to traverse a graph can give us the shortest path between two vertices when we keep track of the distance from the source vertex to every vertex and each vertex's previous vertex.

This shortest path is not unique, but it is the shortest.

This technique only works for unweighted graphs.



A graph is a network of nodes connected by

Intro to Algorithms: Crash Course Computer Science #13 https://www.youtube.com/watch?v=rL8X2mlNHPM

Dijkstra's Algorithm

Dijkstra's Shortest Path First Algorithm (SPF)

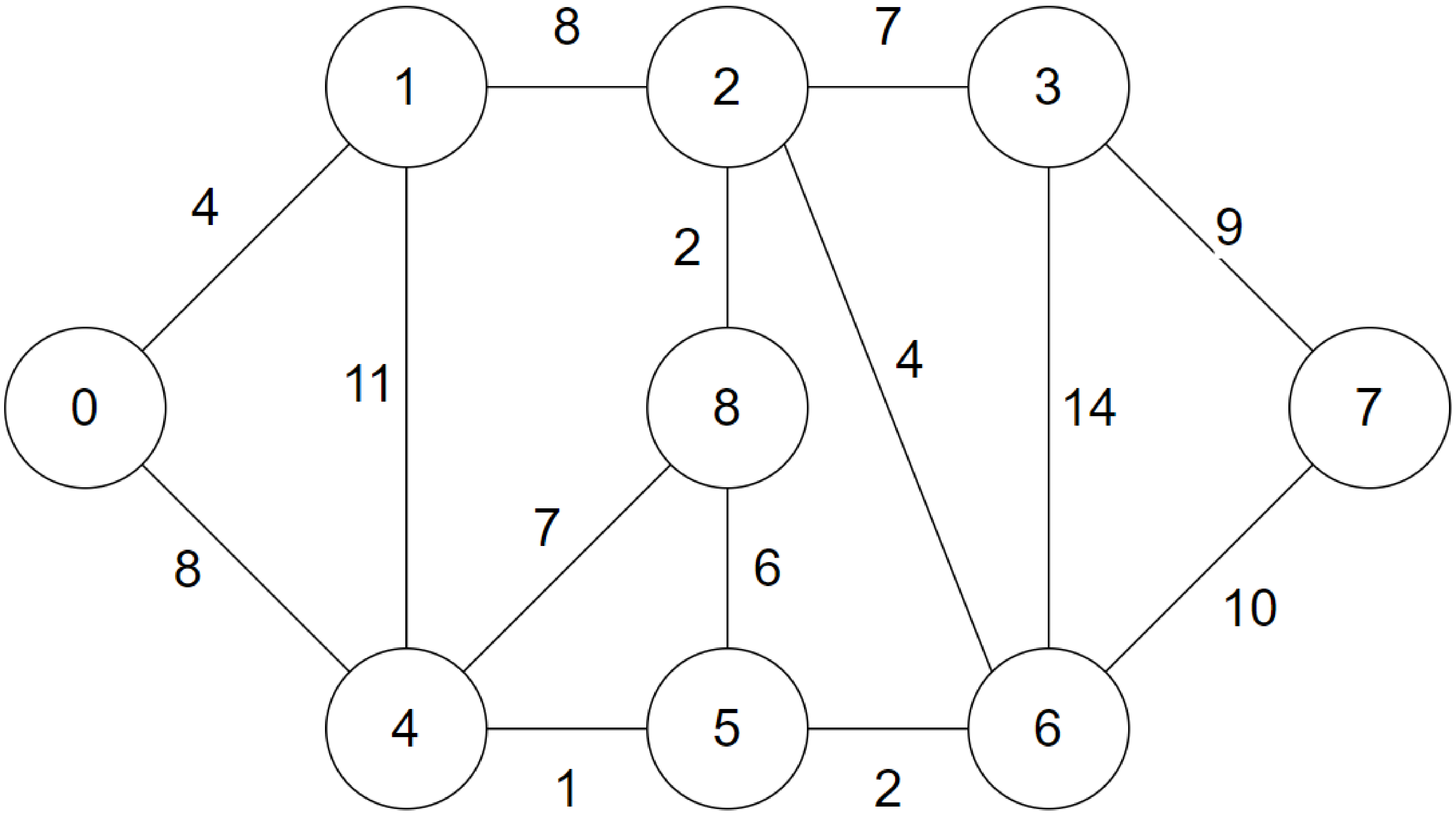
Works on a weighted graph.

Starts with an initial vertex and has a goal vertex.

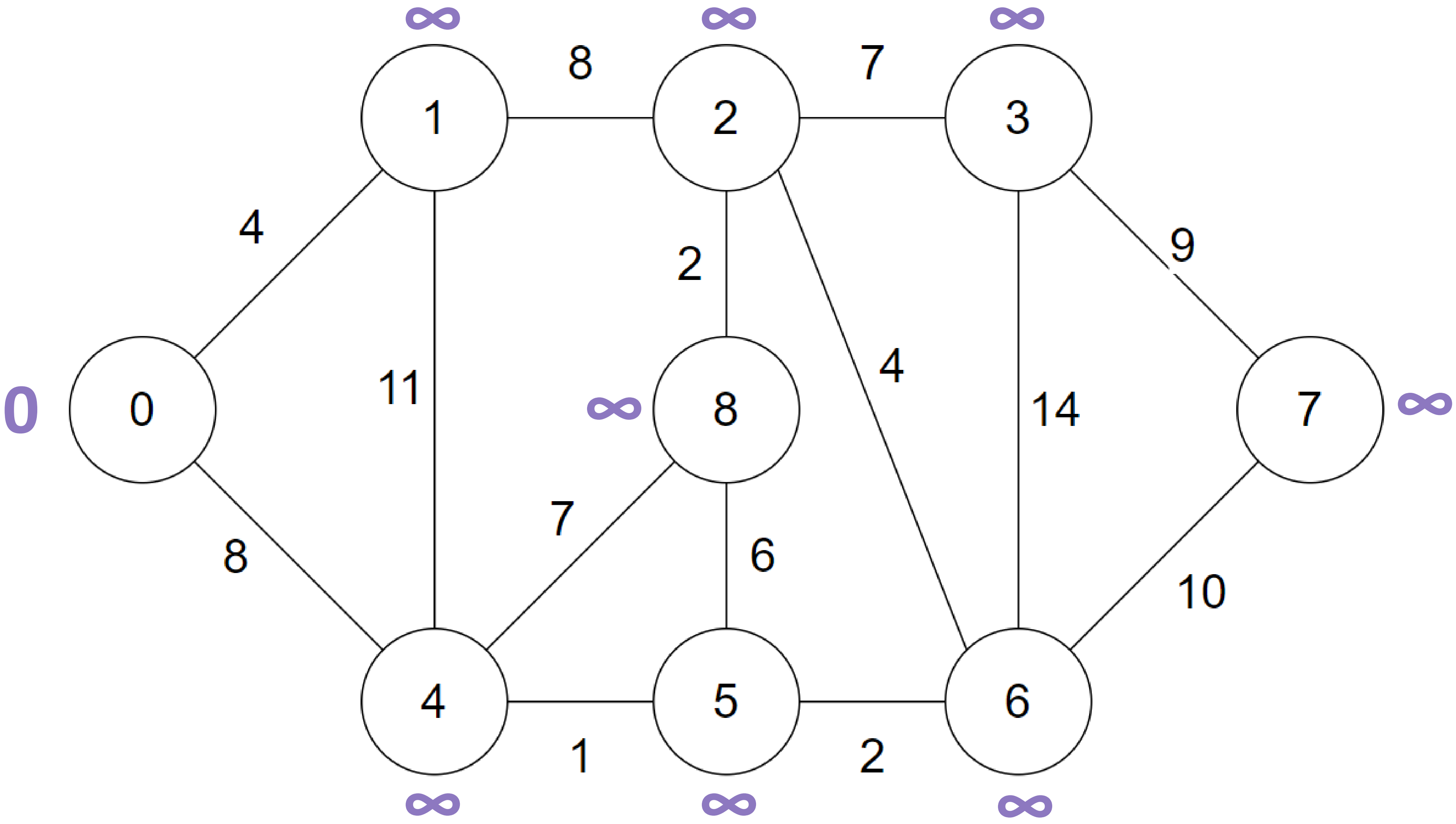
Finds the path with the lowest cost where the cost is based on the weights assigned to edges.

Path may not be unique.

Dijkstra's Algorithm



Dijkstra's Algorithm

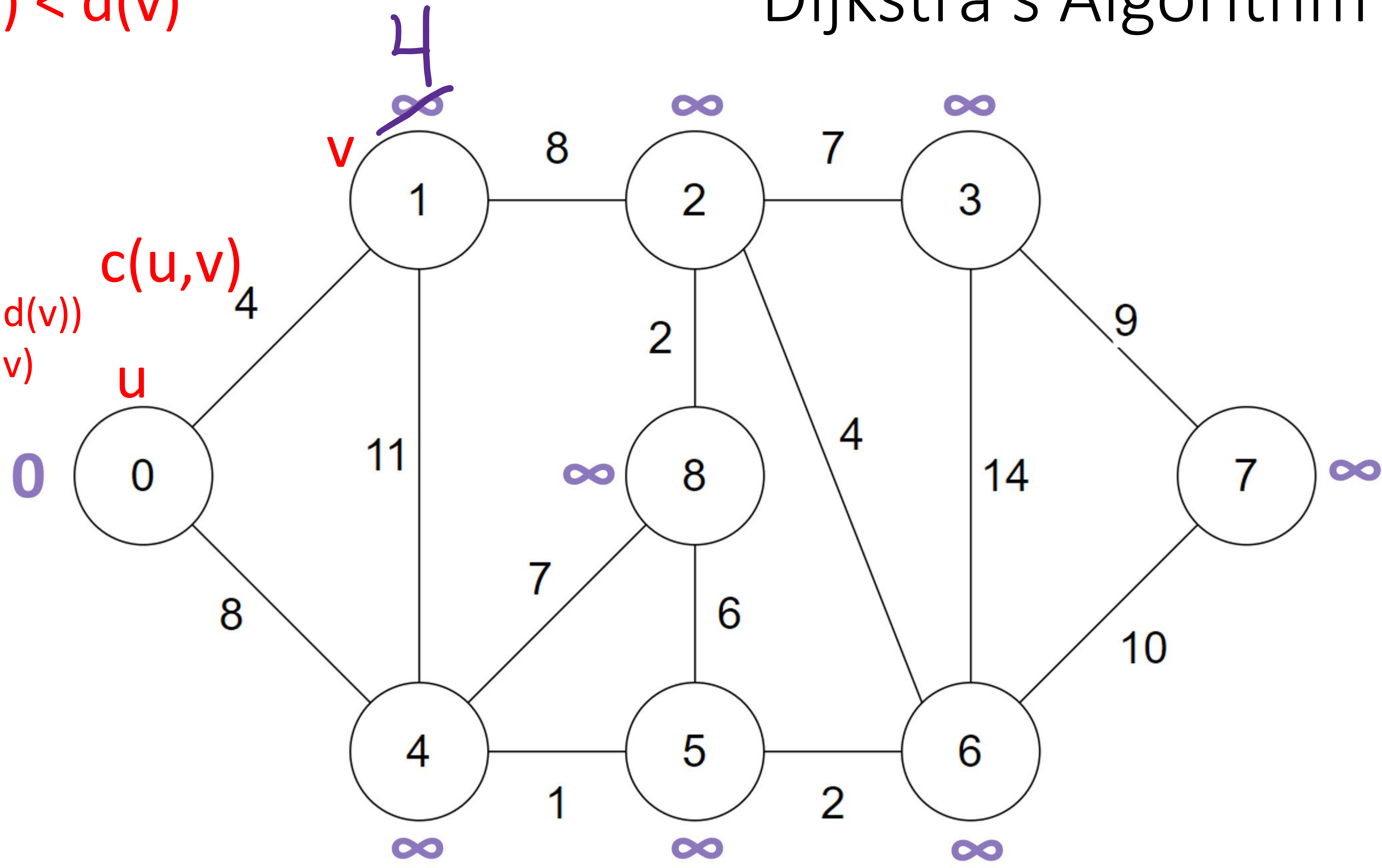


Dijkstra's Algorithm

$d(u) + c(u,v) < d(v)$

$0 + 4 < \infty$

```
if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)
```

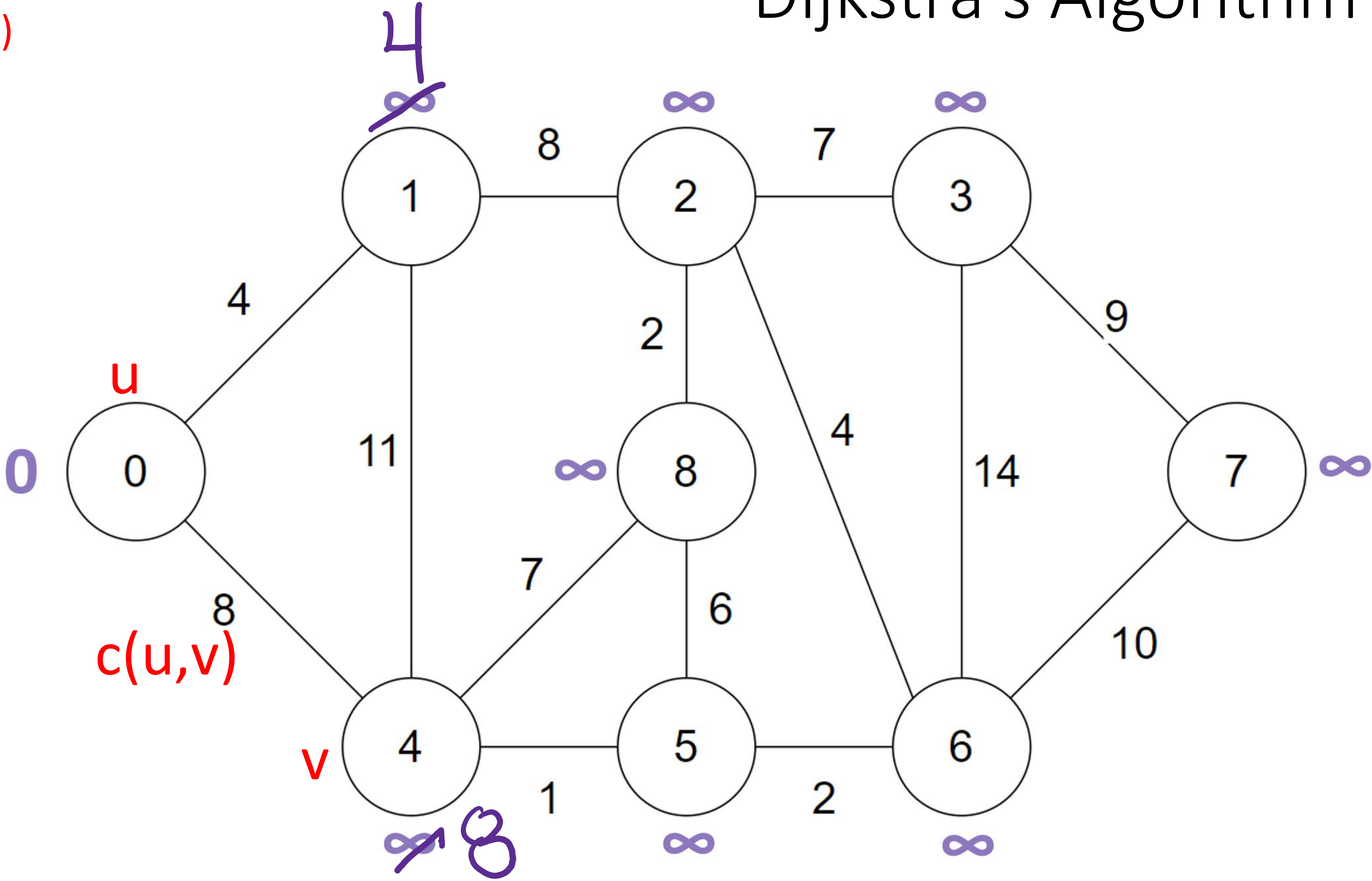


```
if (0 + 4) < ∞)
  d(v) = 0 + 4
  d(v) = 4
```

Dijkstra's Algorithm

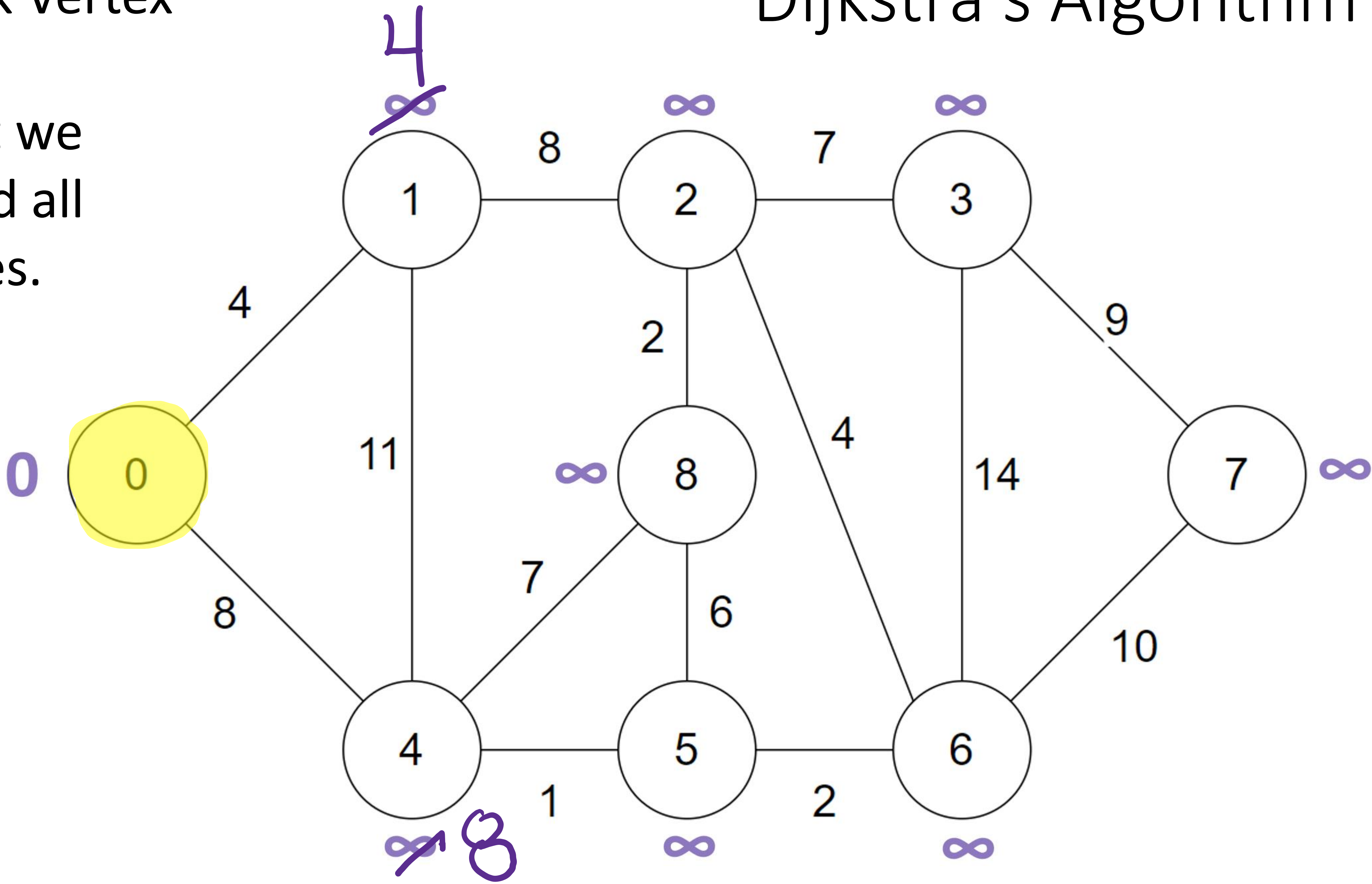
```
if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)
```

```
if (0 + 8) < ∞)
  d(v) = 0 + 8
  d(v) = 8
```



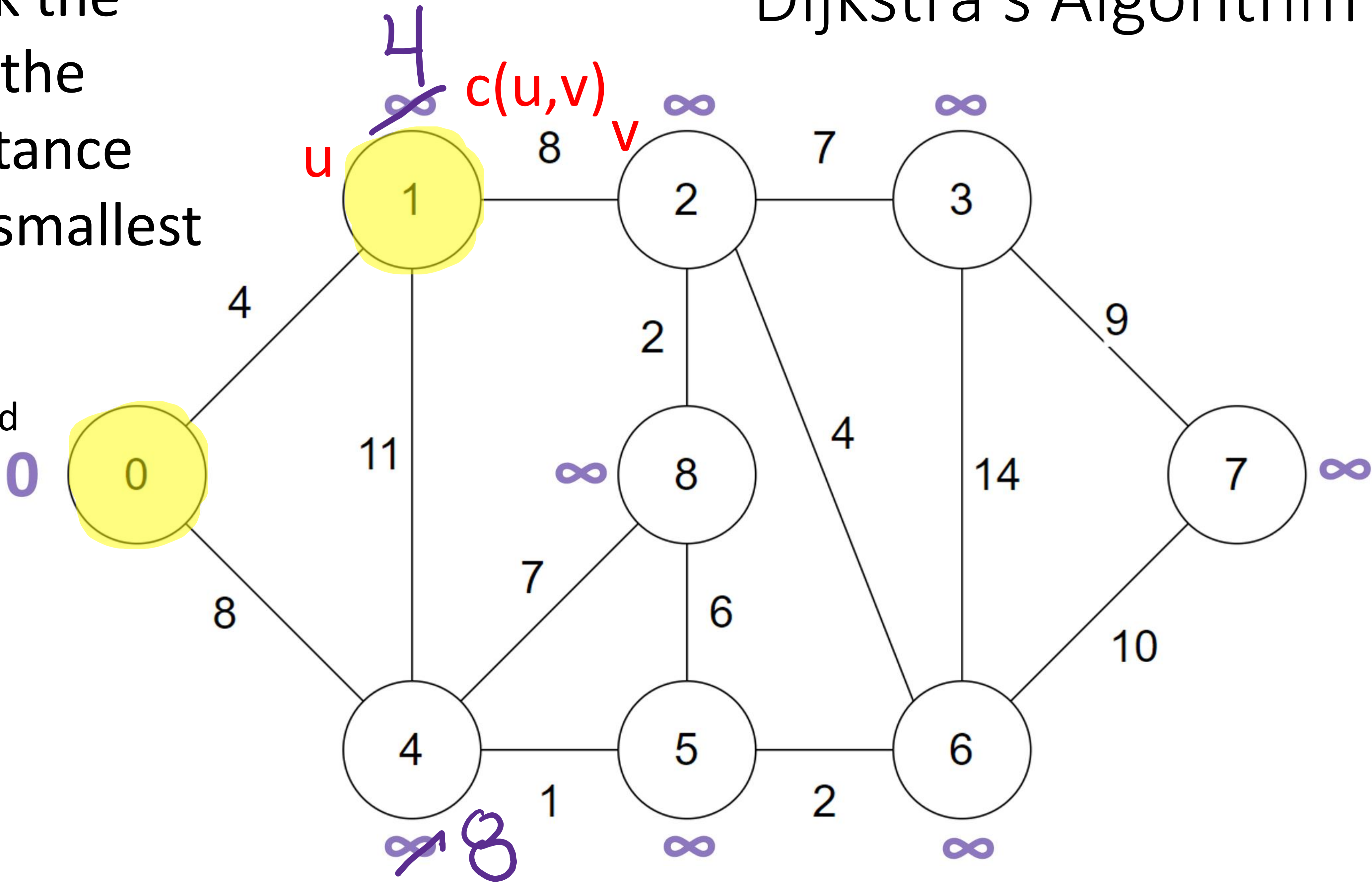
Dijkstra's Algorithm

Now we mark Vertex 0 as "visited" meaning that we have explored all incident edges.



Dijkstra's Algorithm

Now we pick the Vertex with the smallest distance value – the smallest d(u)



0 – already visited
1 – d(u) of 4
2 – d(u) of ∞
3 – d(u) of ∞
4 – d(u) of 8
5 – d(u) of ∞
6 – d(u) of ∞
7 – d(u) of ∞
8 – d(u) of ∞

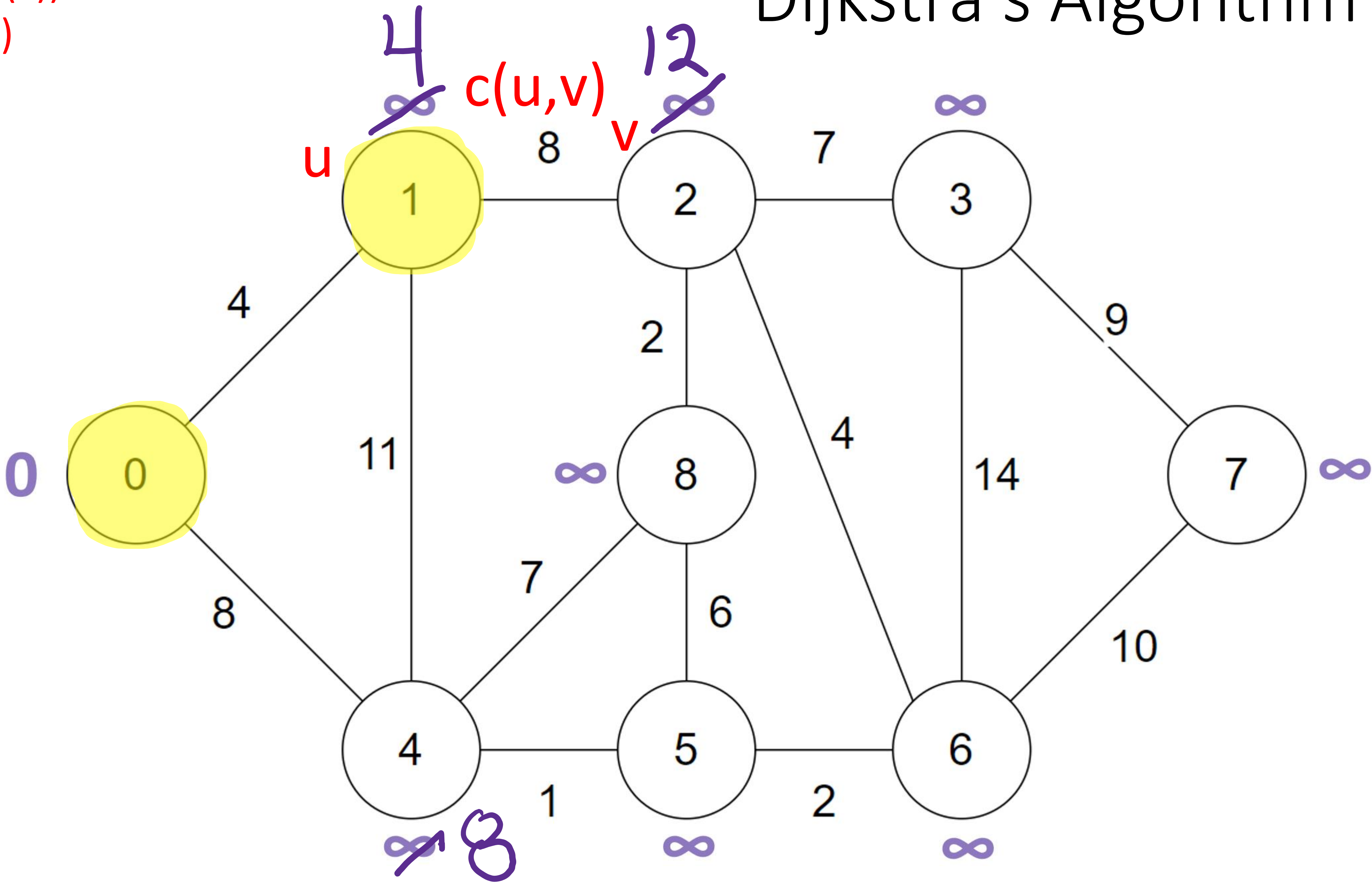
Dijkstra's Algorithm

```
if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)
```

if (4 + 8) < ∞)

d(v) = 4 + 8

d(v) = 12

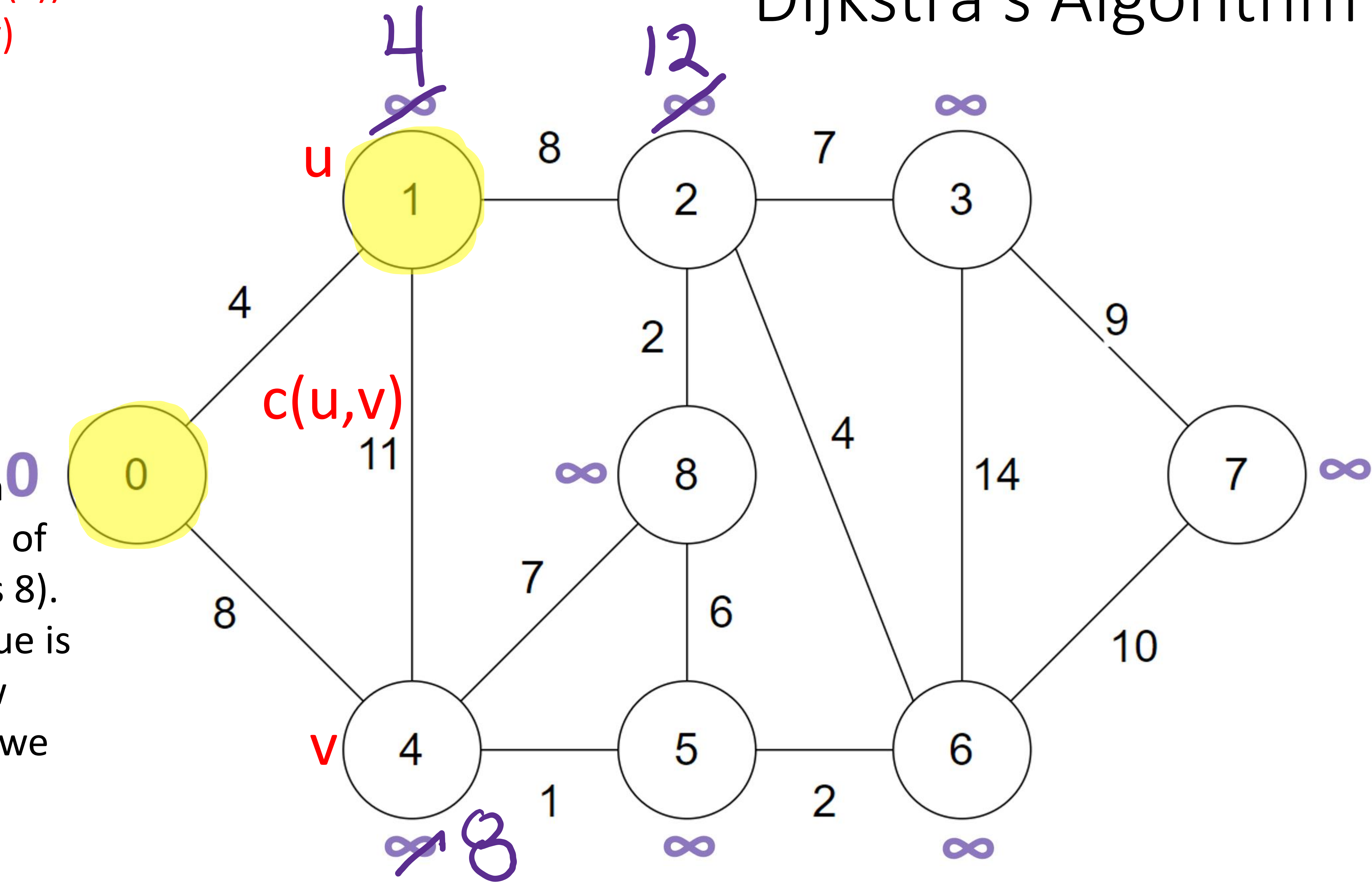


Dijkstra's Algorithm

if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)

if (4 + 11) < ∞)
  d(v) = 4 + 11
  d(v) = 15

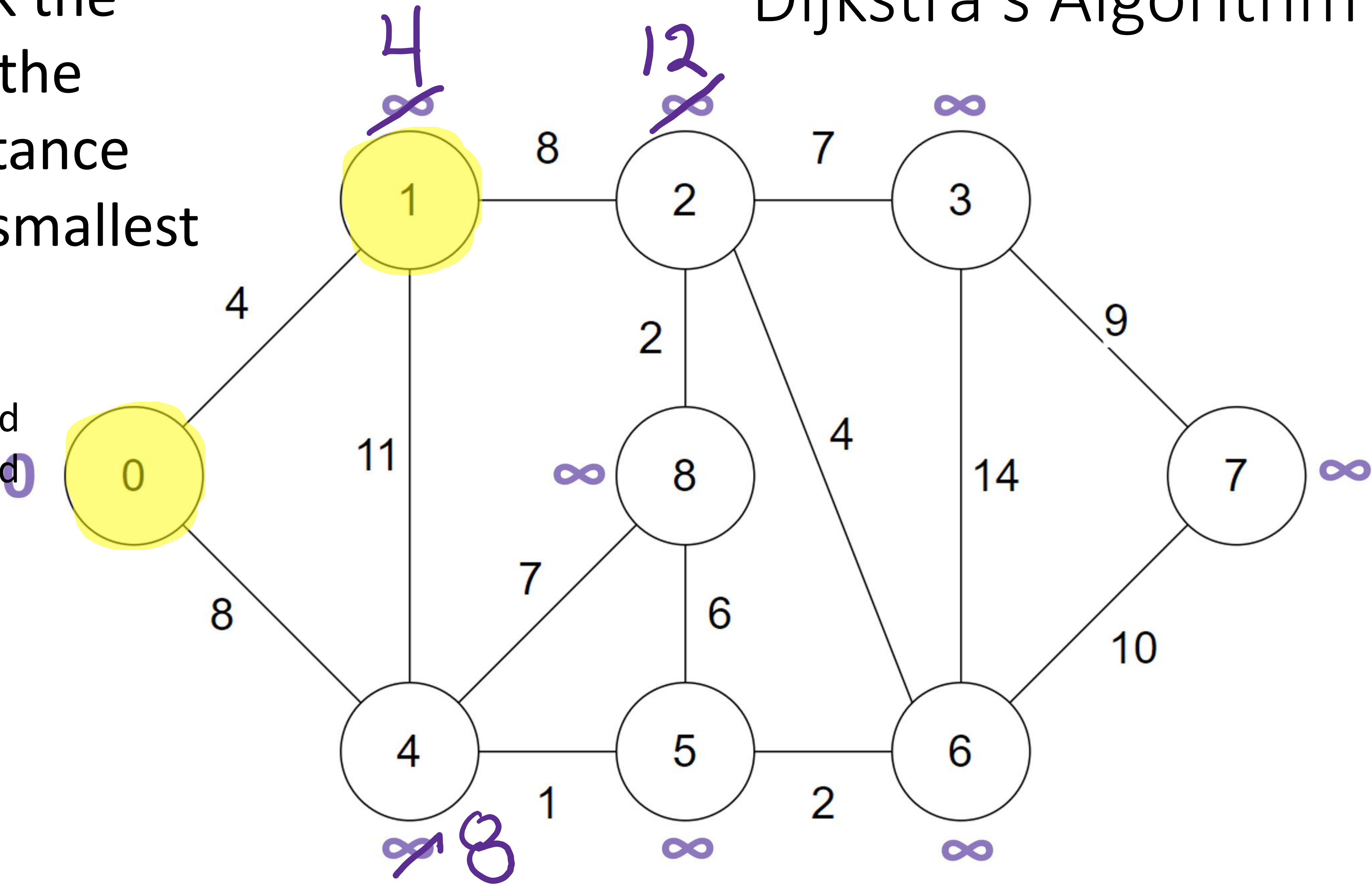
15 is greater than the existing value of Vertex 4 (which is 8). If the existing value is less than our new calculation, then we keep the original value.



Dijkstra's Algorithm

Now we pick the Vertex with the smallest distance value – the smallest d(u)

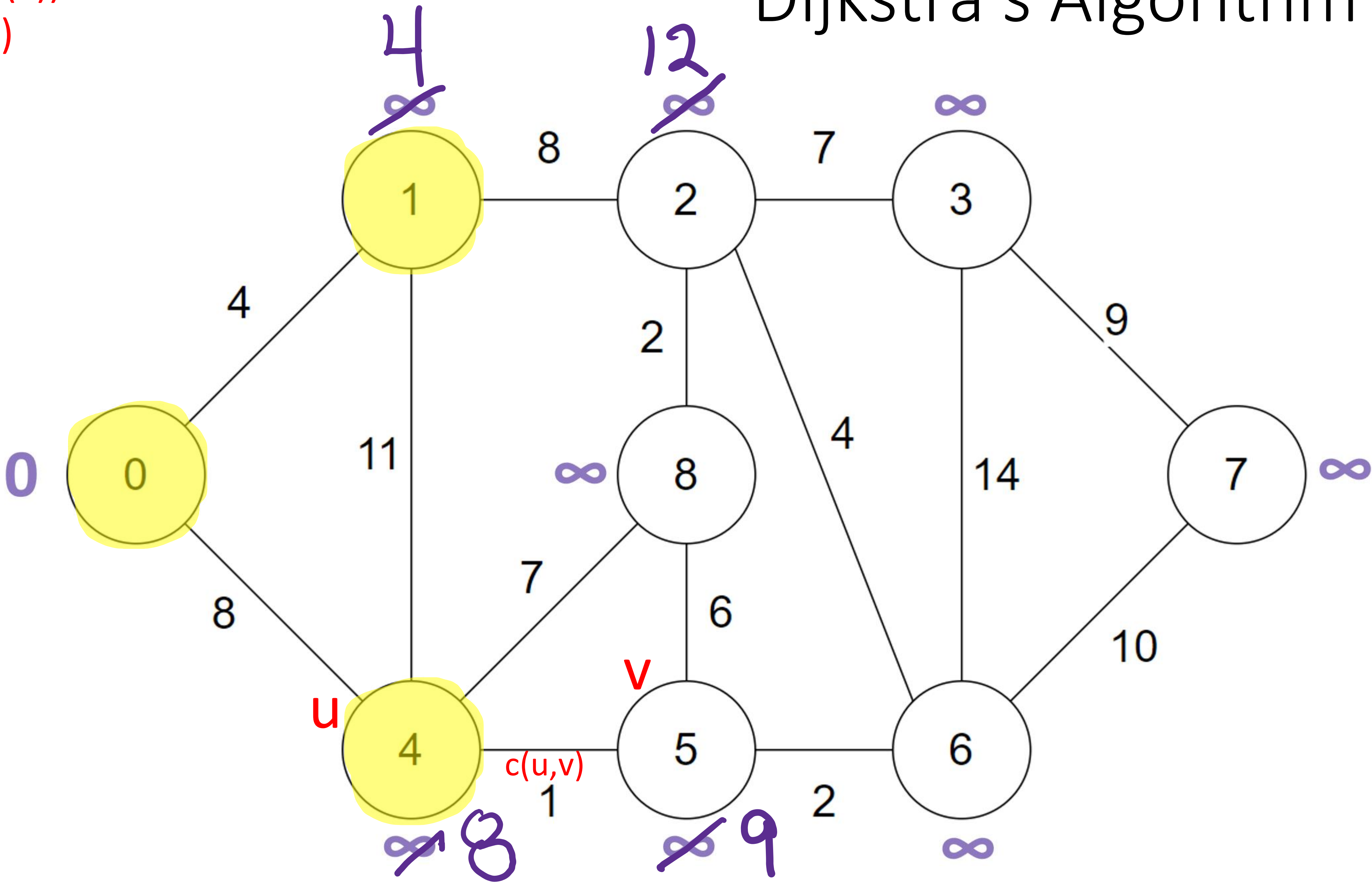
0 – already visited
1 – already visited
2 – d(u) of 12
3 – d(u) of ∞
4 – d(u) of 8
5 – d(u) of ∞
6 – d(u) of ∞
7 – d(u) of ∞
8 – d(u) of ∞



if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)

if (8 + 1) < ∞)
  d(v) = 8 + 1
  d(v) = 9

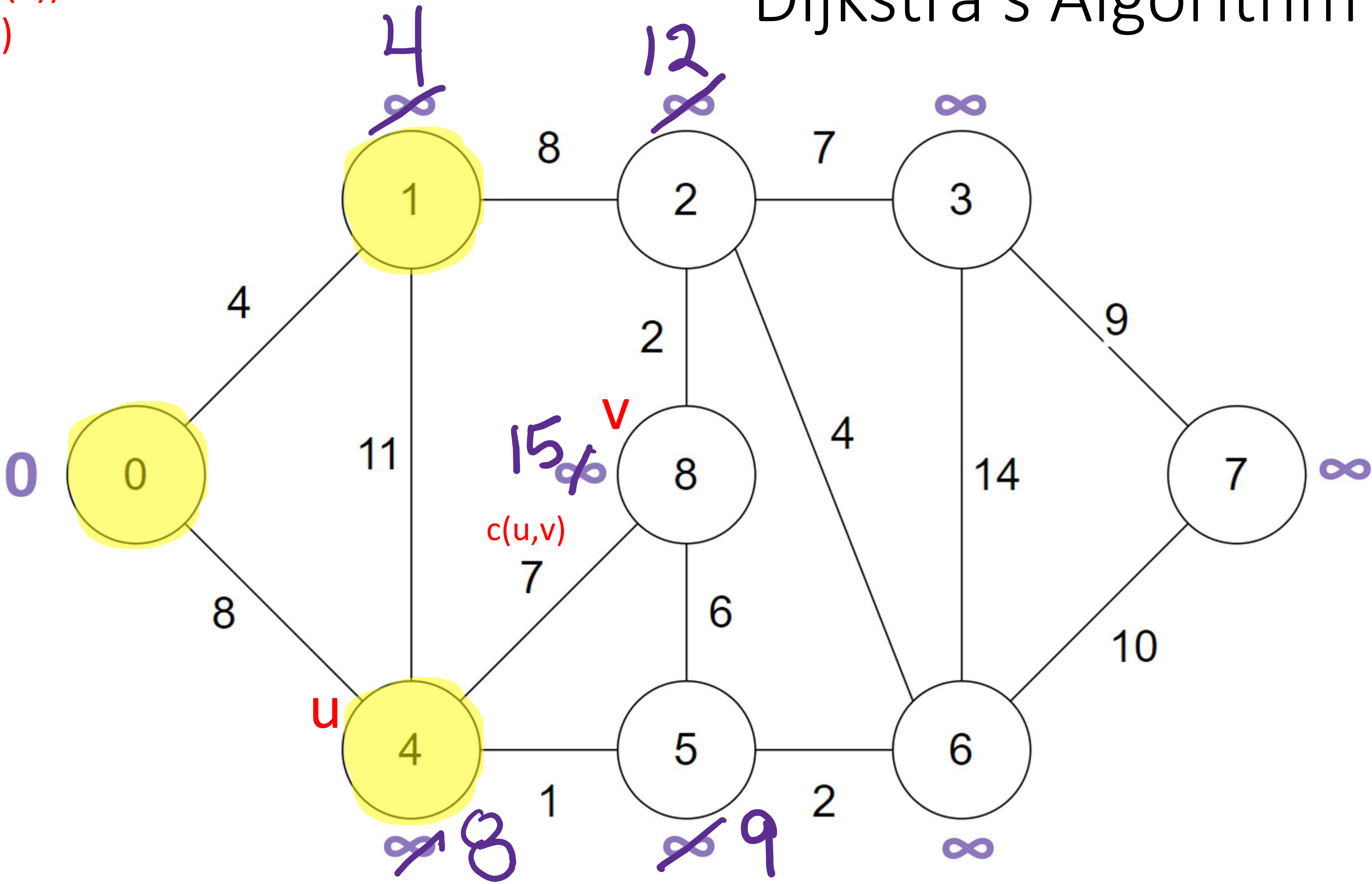
Dijkstra's Algorithm



Dijkstra's Algorithm

if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)

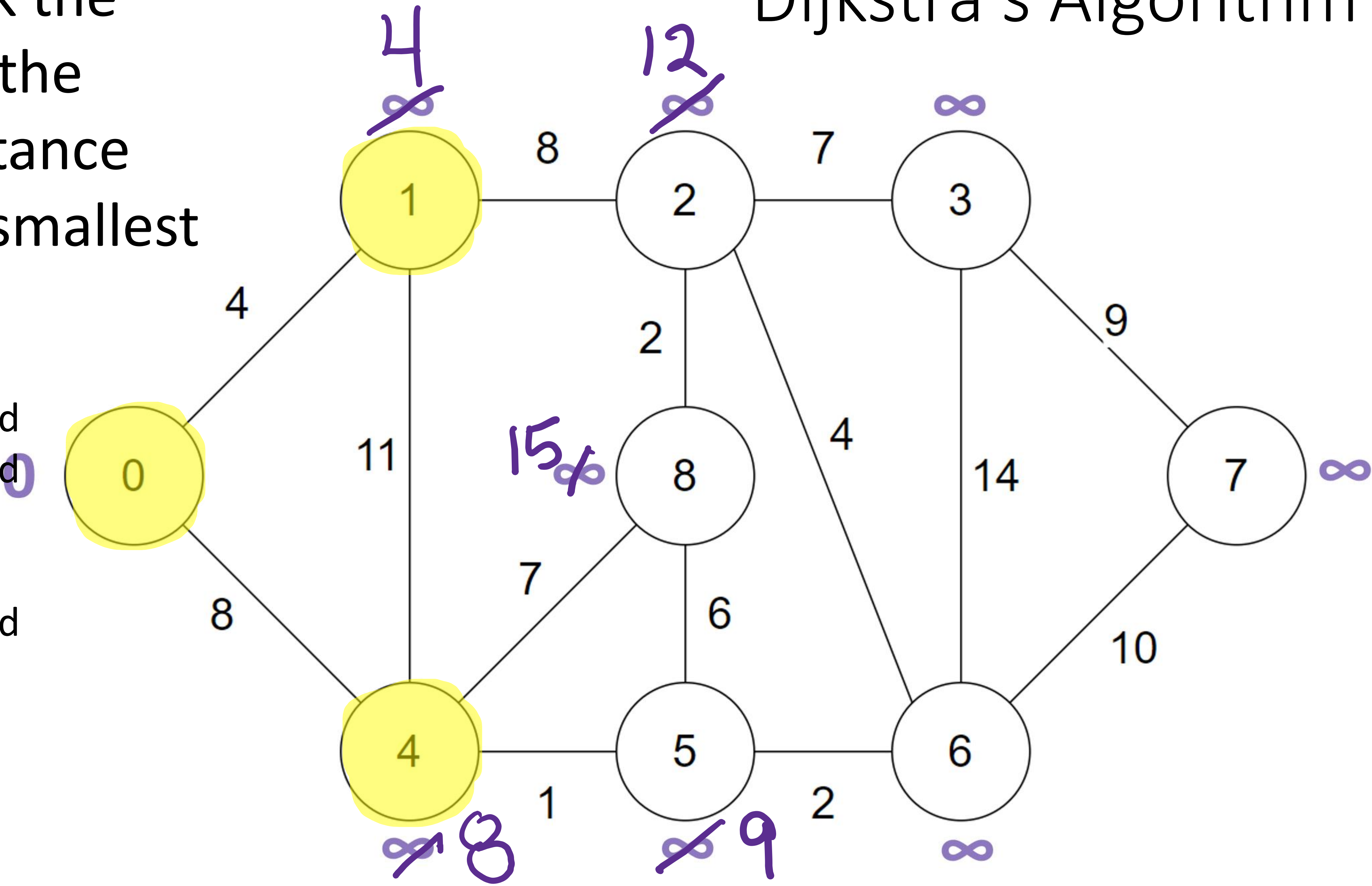
if (8 + 7) < ∞)
 d(v) = 8 + 7
 d(v) = 15



Dijkstra's Algorithm

Now we pick the Vertex with the smallest distance value – the smallest d(u)

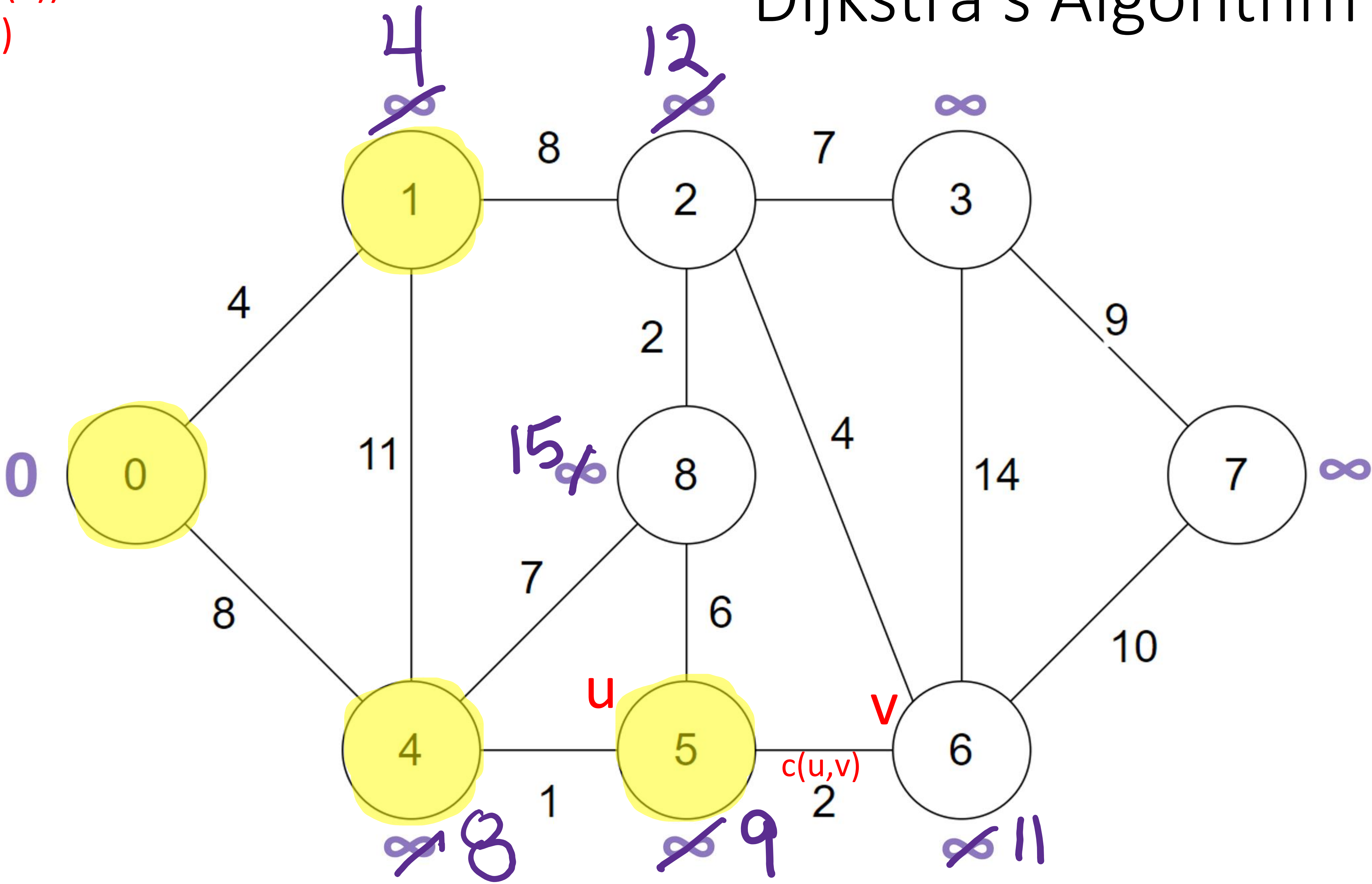
0 – already visited
1 – already visited
2 – d(u) of 12
3 – d(u) of ∞
4 – already visited
5 – d(u) of 9
6 – d(u) of ∞
7 – d(u) of ∞
8 – d(u) of 15



Dijkstra's Algorithm

if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)

if (9 + 2) < ∞)
  d(v) = 9 + 2
  d(v) = 11



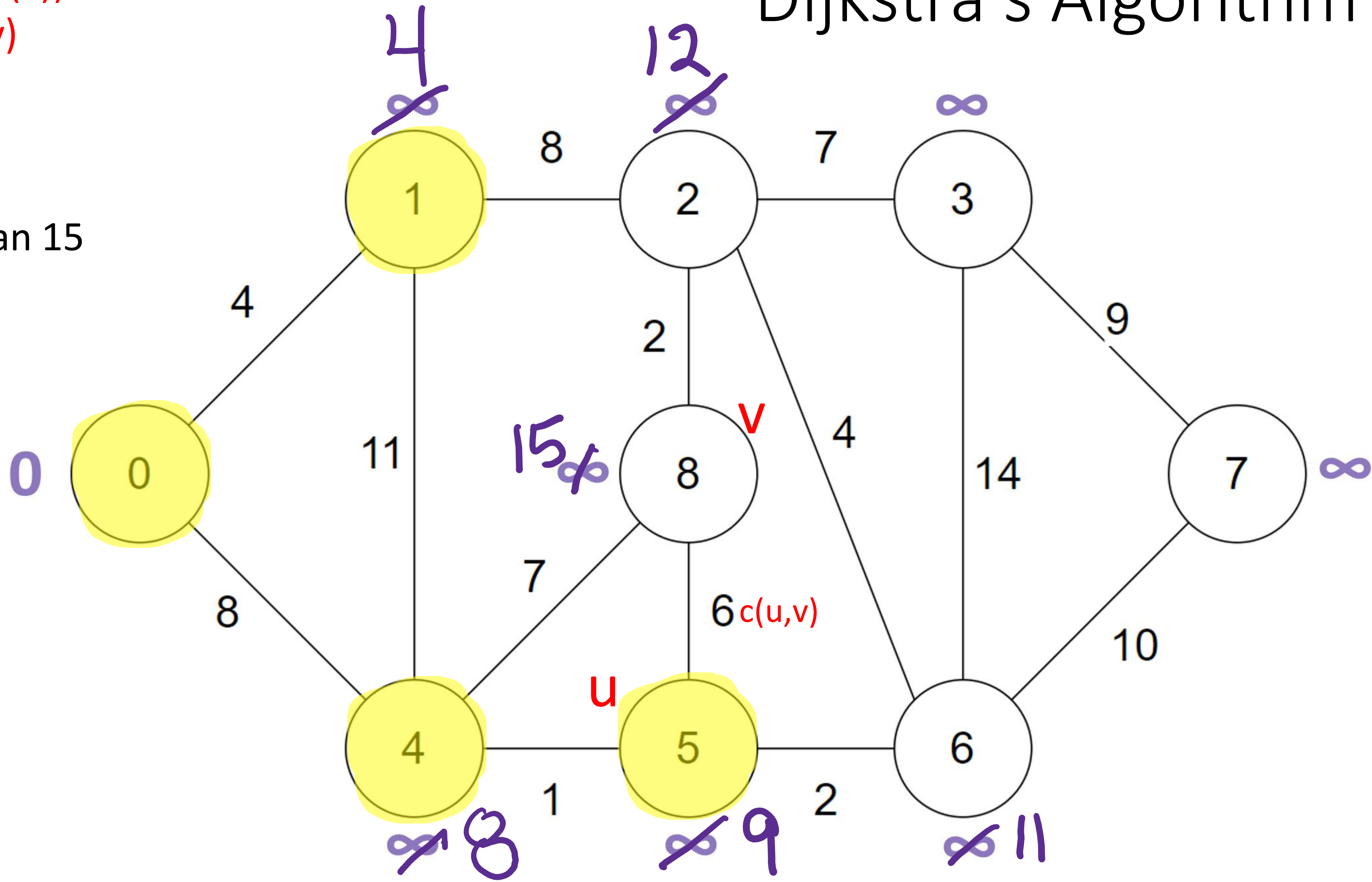
Dijkstra's Algorithm

if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)

if (9 + 6) < 15)

15 is not less than 15

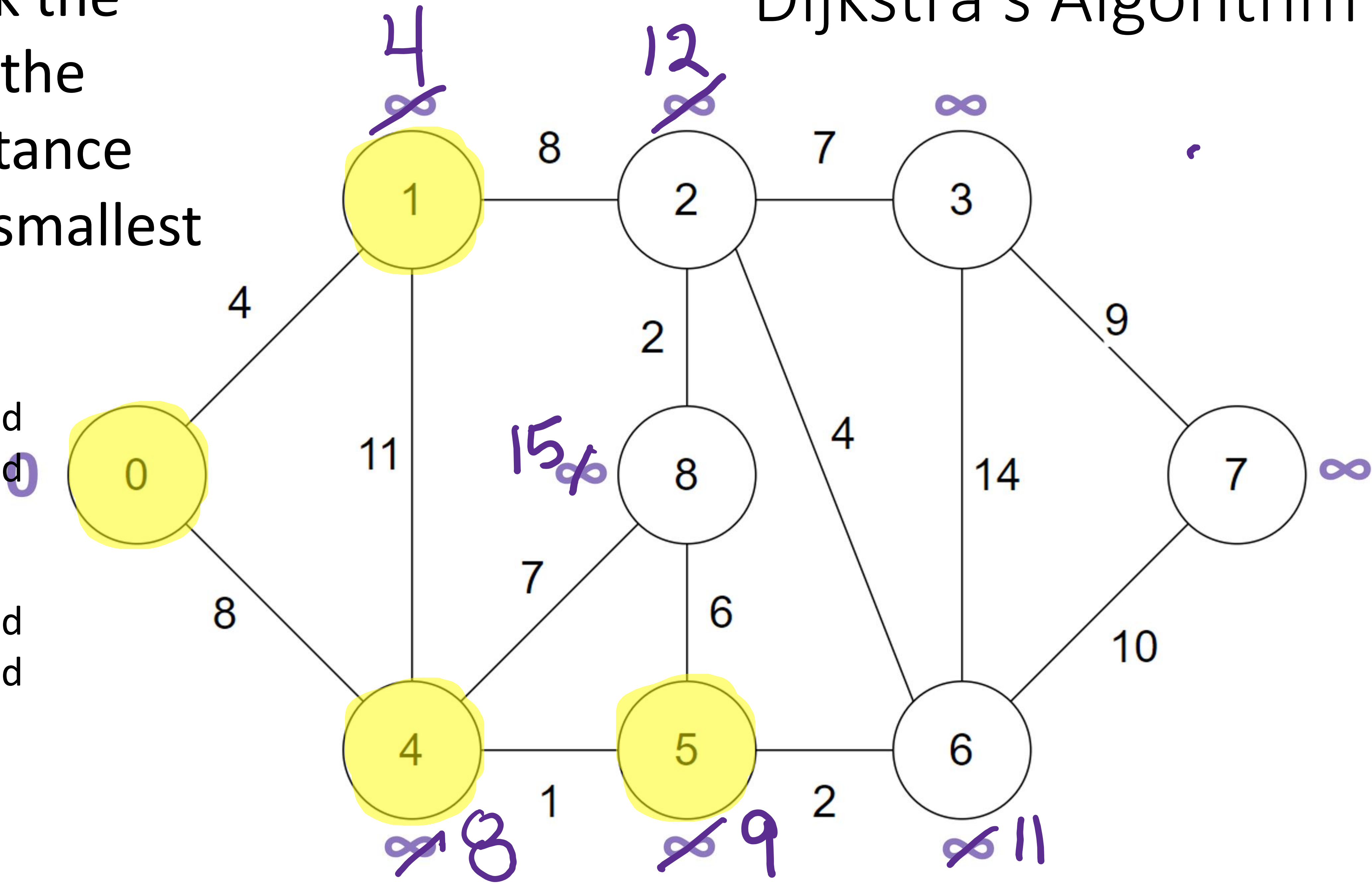
NO CHANGE



Dijkstra's Algorithm

Now we pick the Vertex with the smallest distance value – the smallest d(u)

0 – already visited
1 – already visited
2 – d(u) of 12
3 – d(u) of ∞
4 – already visited
5 – already visited
6 – d(u) of 11
7 – d(u) of ∞
8 – d(u) of 15



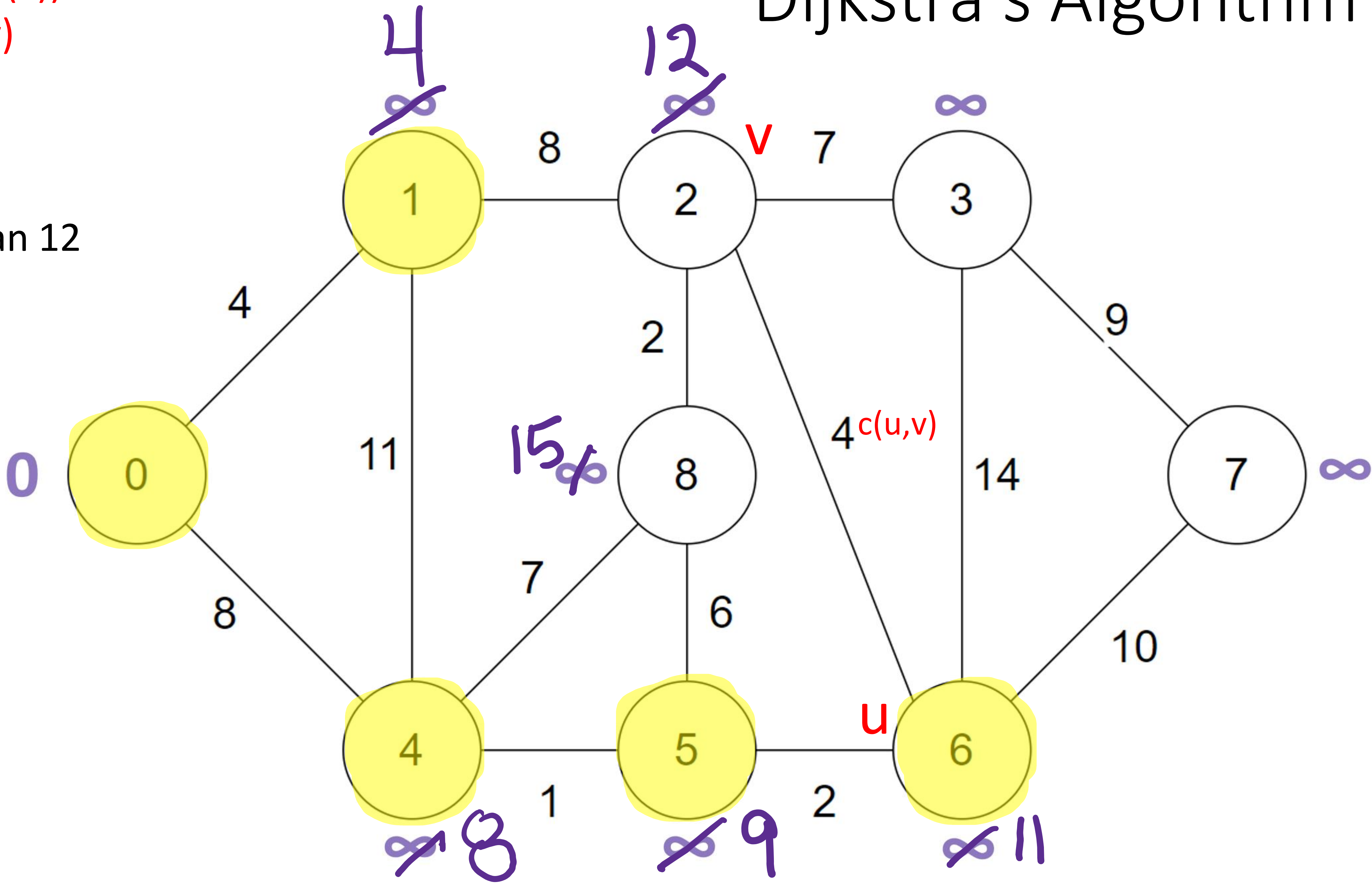
Dijkstra's Algorithm

if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)

if (11 + 4) < 12)

15 is not less than 12

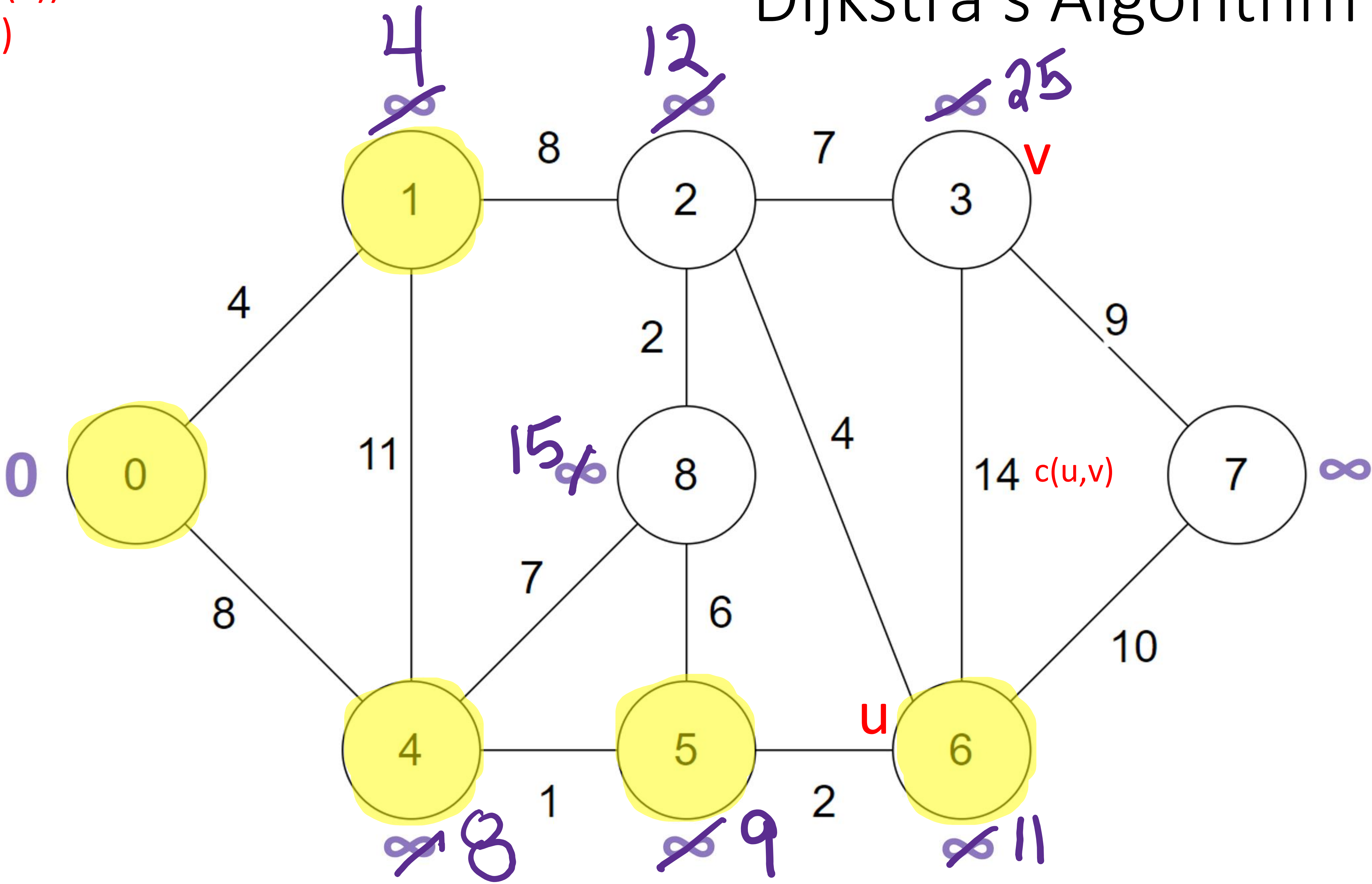
NO CHANGE



Dijkstra's Algorithm

if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)

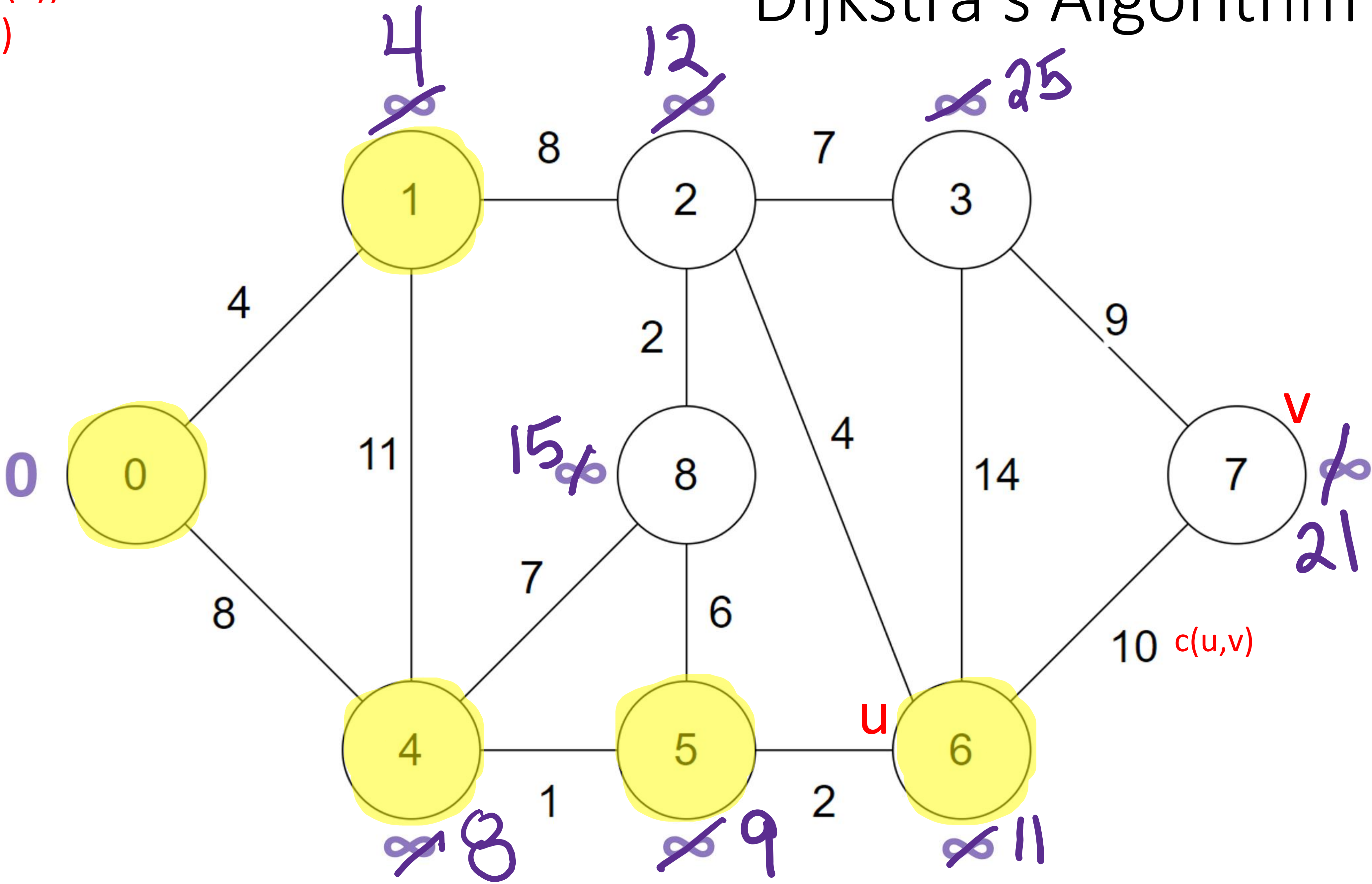
if (11 + 14) < ∞)
 d(v) = 11 + 14
 d(v) = 25



Dijkstra's Algorithm

if (d(u) + c(u,v) < d(v))
d(v) = d(u) + c(u,v)

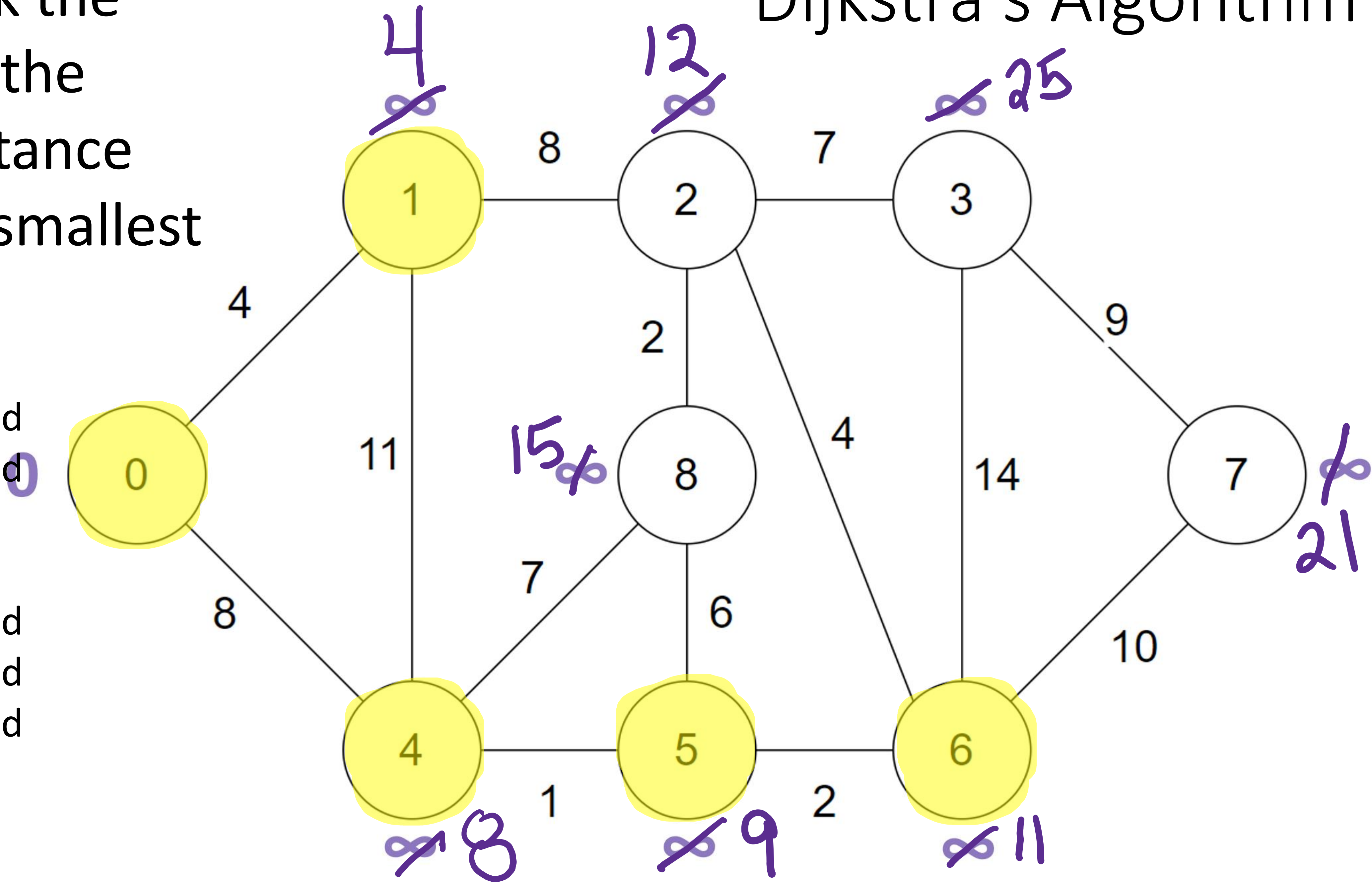
if (11 + 10) < ∞)
d(v) = 11 + 10
d(v) = 21



Dijkstra's Algorithm

Now we pick the Vertex with the smallest distance value – the smallest d(u)

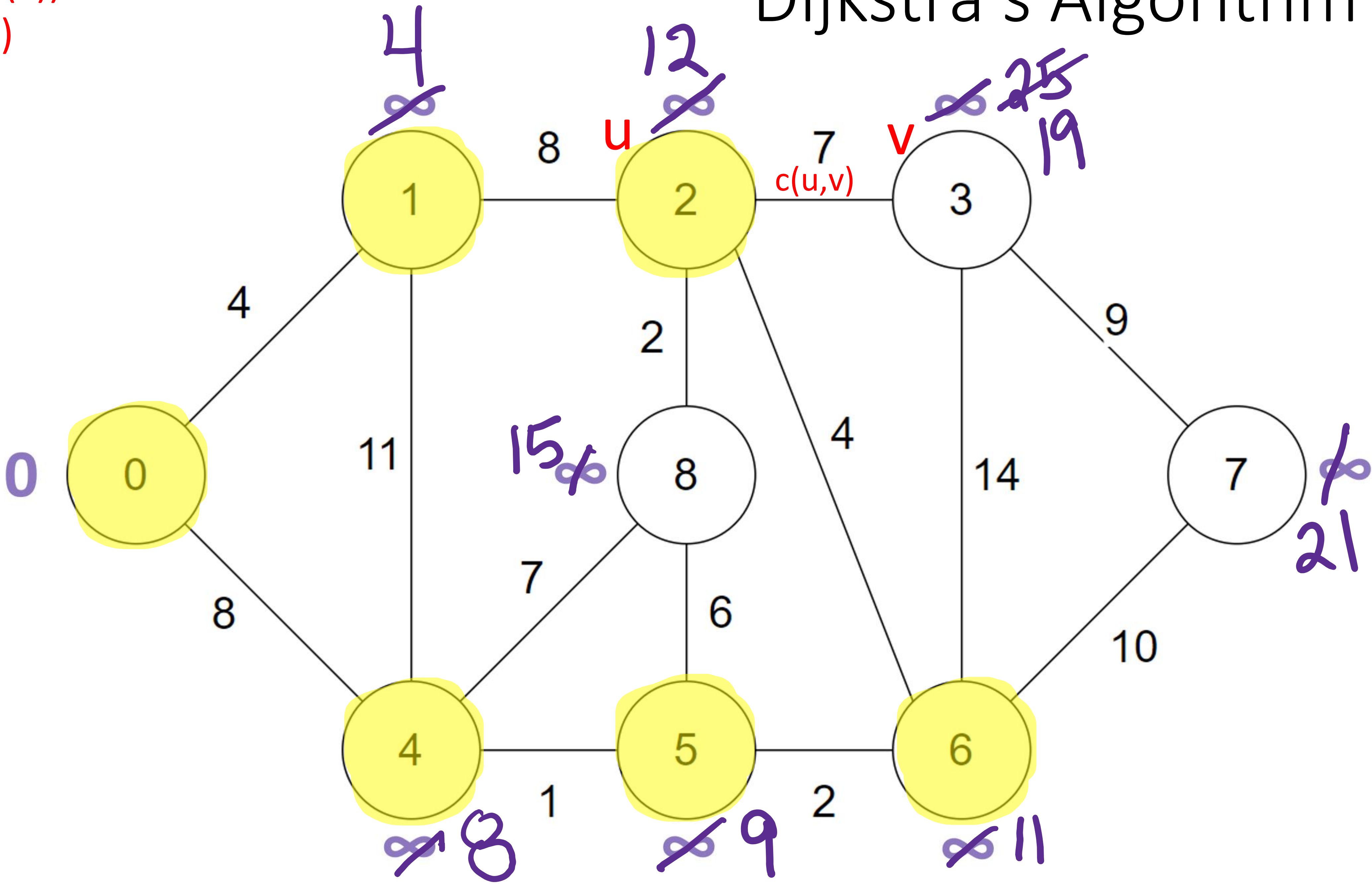
0 – already visited
1 – already visited
2 – d(u) of 12
3 – d(u) of 25
4 – already visited
5 – already visited
6 – already visited
7 – d(u) of 21
8 – d(u) of 15



Dijkstra's Algorithm

if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)

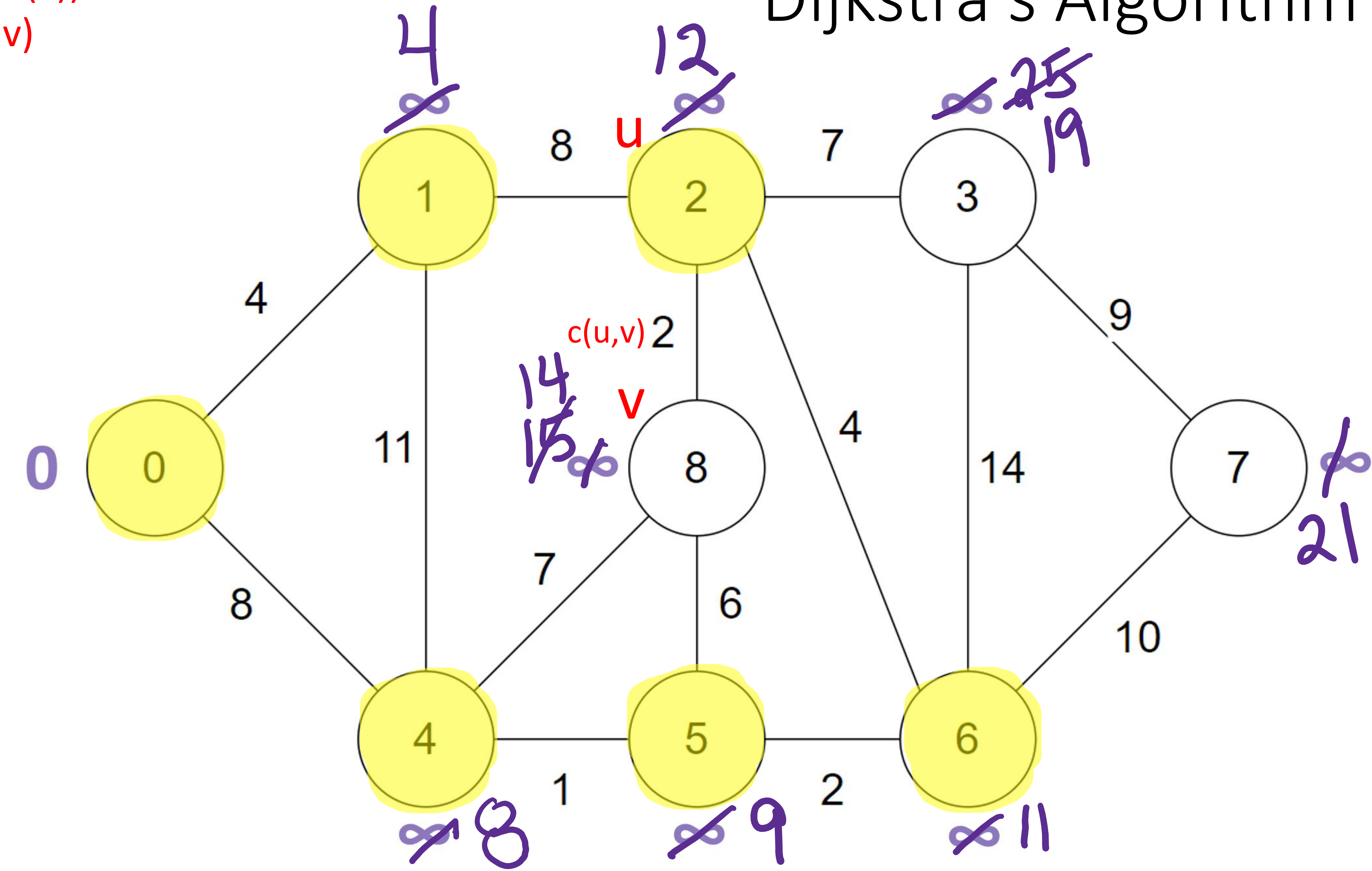
if (12 + 7) < 25)
 d(v) = 12 + 7
 d(v) = 19



Dijkstra's Algorithm

if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)

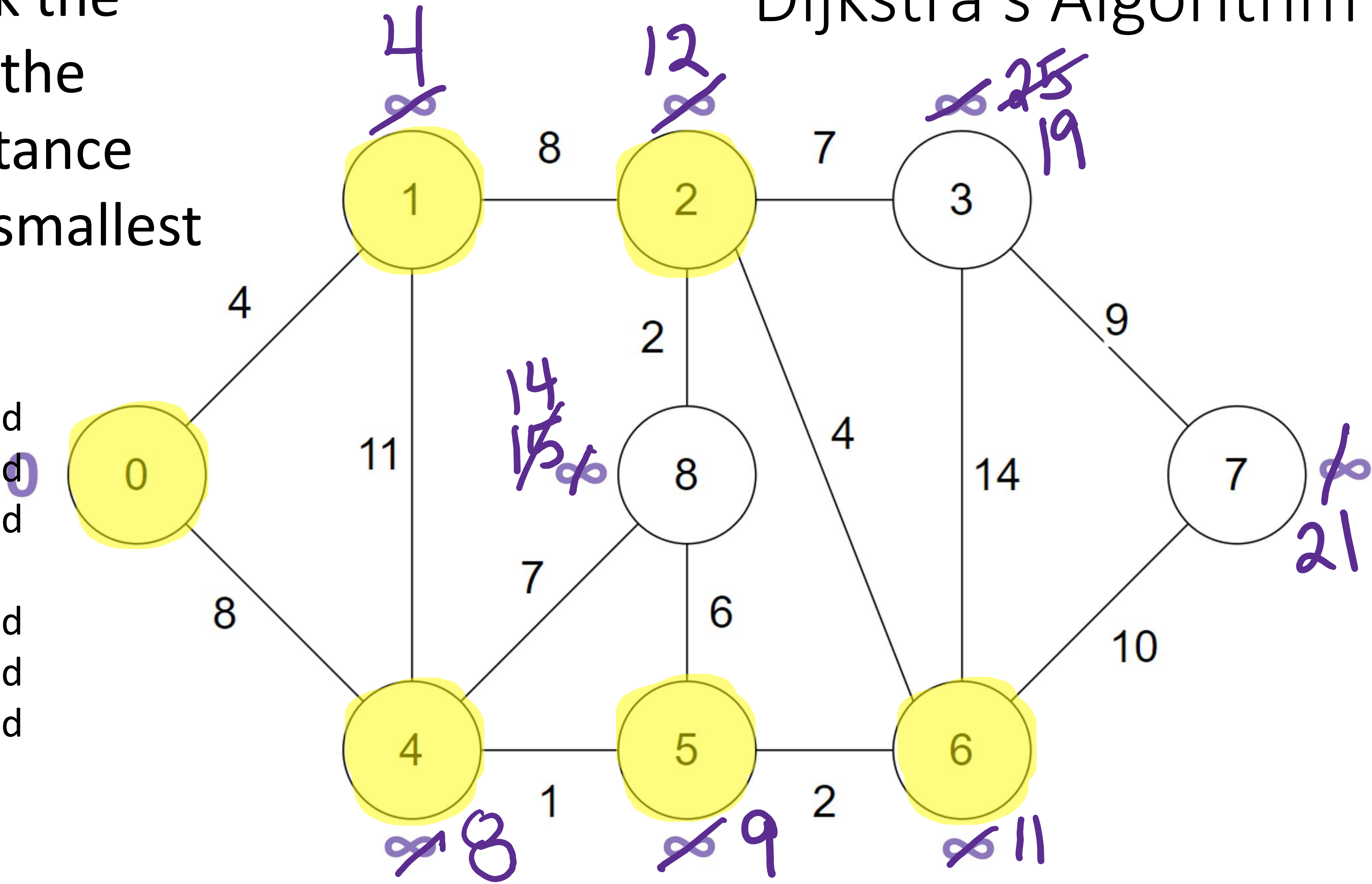
if (12 + 2) < 15)
 d(v) = 12 + 2
 d(v) = 14



Dijkstra's Algorithm

Now we pick the Vertex with the smallest distance value – the smallest d(u)

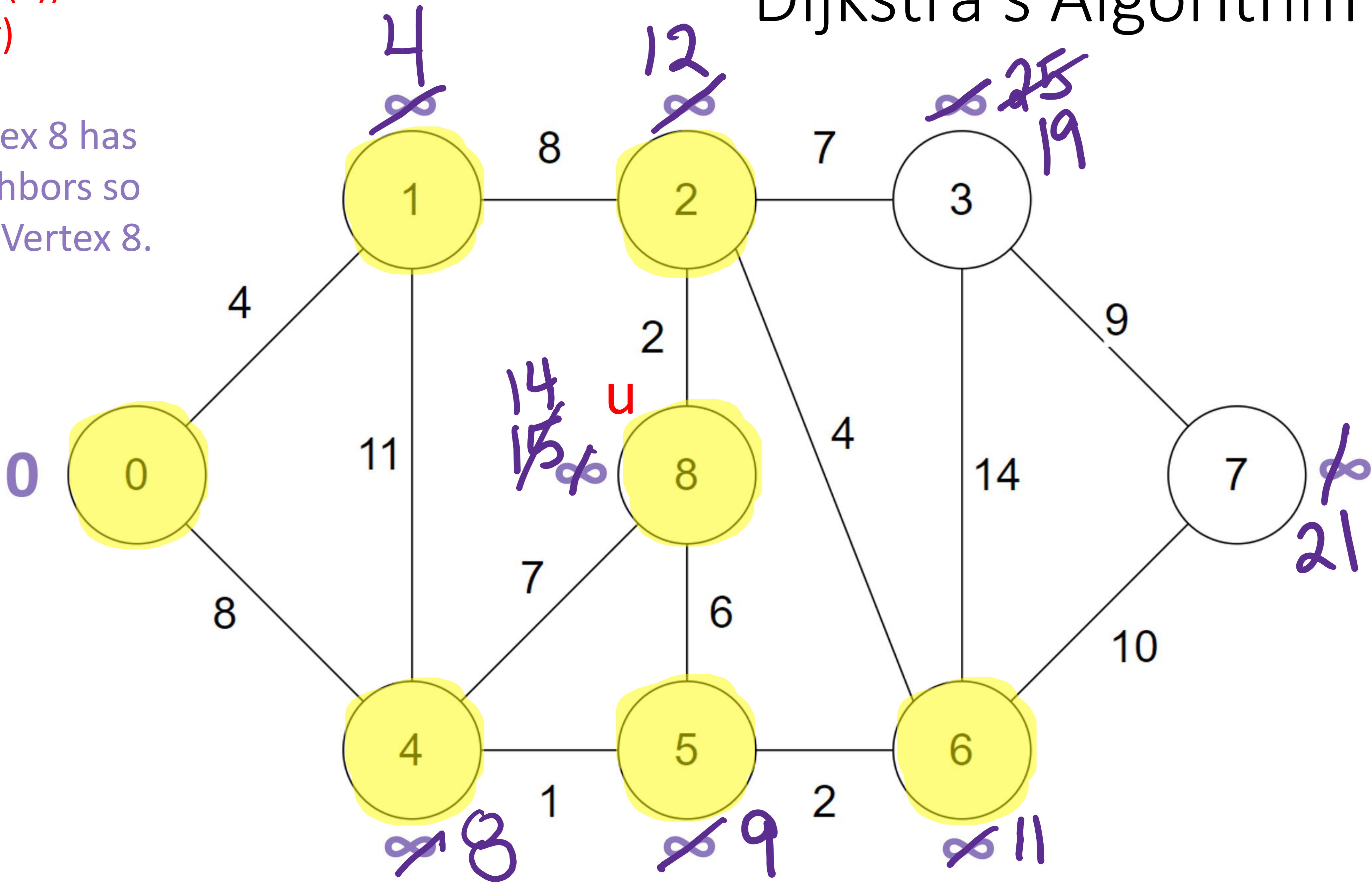
0 – already visited
1 – already visited
2 – already visited
3 – d(u) of 19
4 – already visited
5 – already visited
6 – already visited
7 – d(u) of 21
8 – d(u) of 14



Dijkstra's Algorithm

```
if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)
```

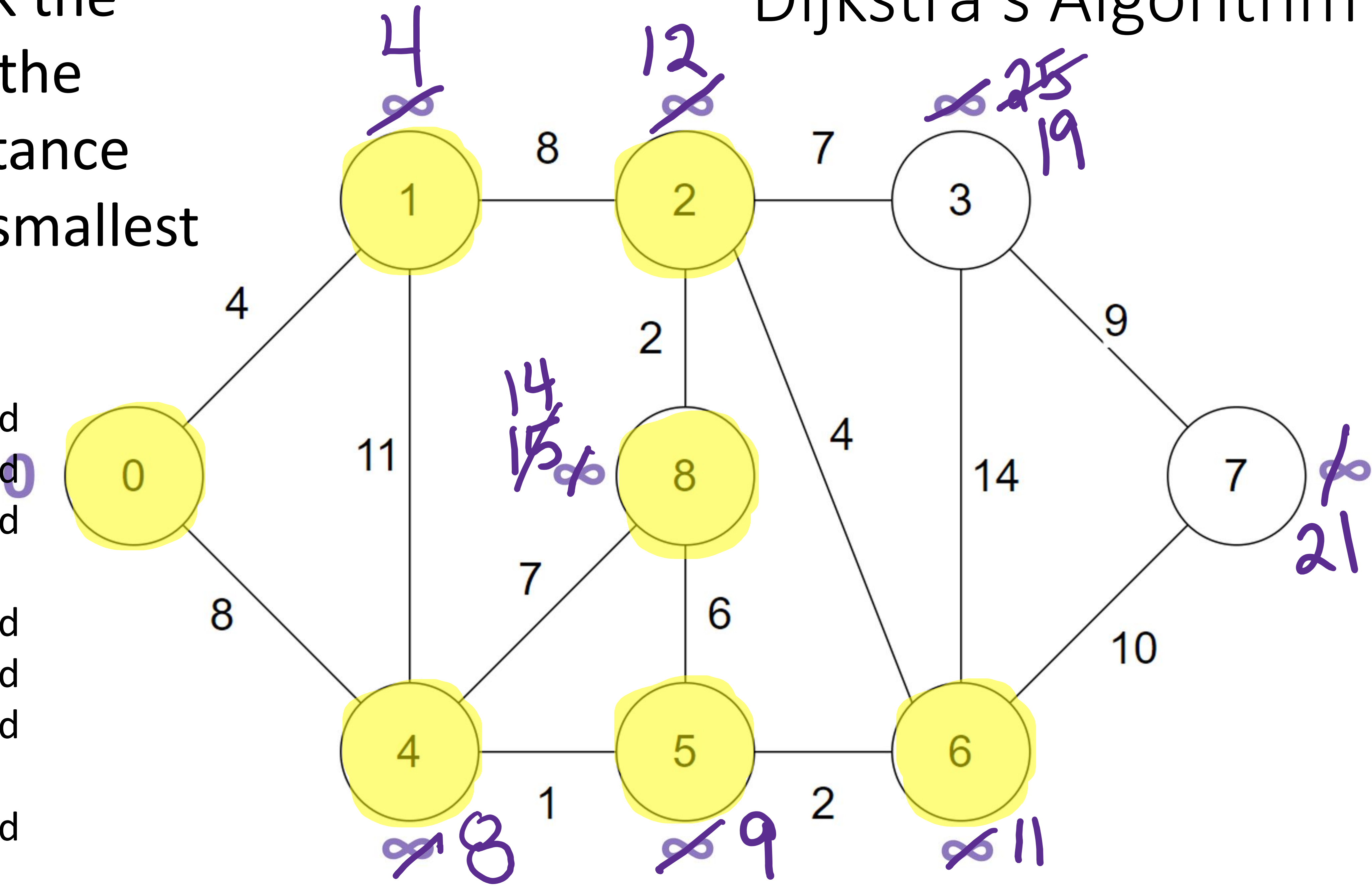
d(u) is 14 but Vertex 8 has NO unvisited neighbors so we are done with Vertex 8.



Dijkstra's Algorithm

Now we pick the Vertex with the smallest distance value – the smallest d(u)

0 – already visited
1 – already visited
2 – already visited
3 – d(u) of 19
4 – already visited
5 – already visited
6 – already visited
7 – d(u) of 21
8 – already visited



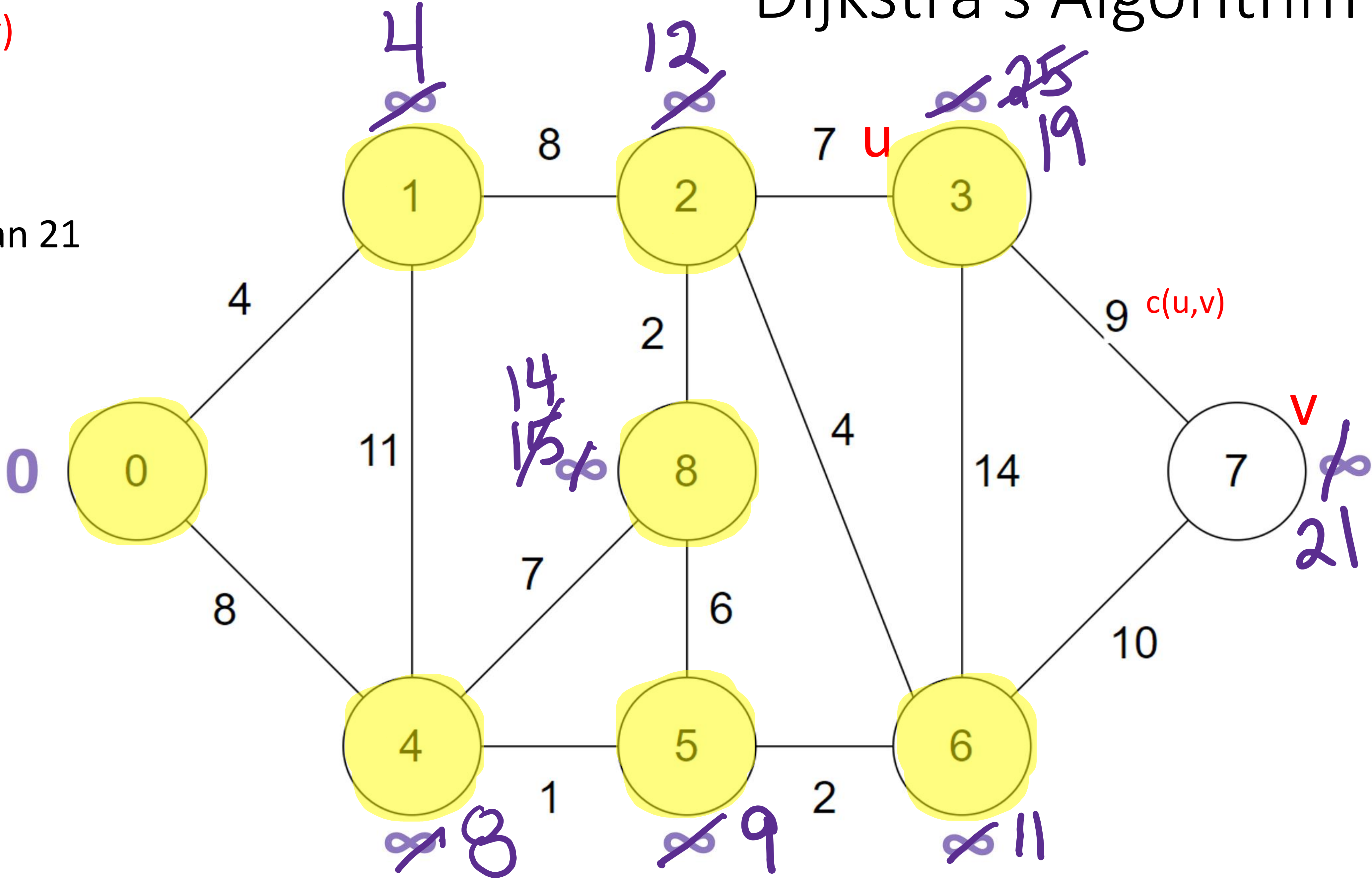
Dijkstra's Algorithm

if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)

if (19 + 9) < 21)

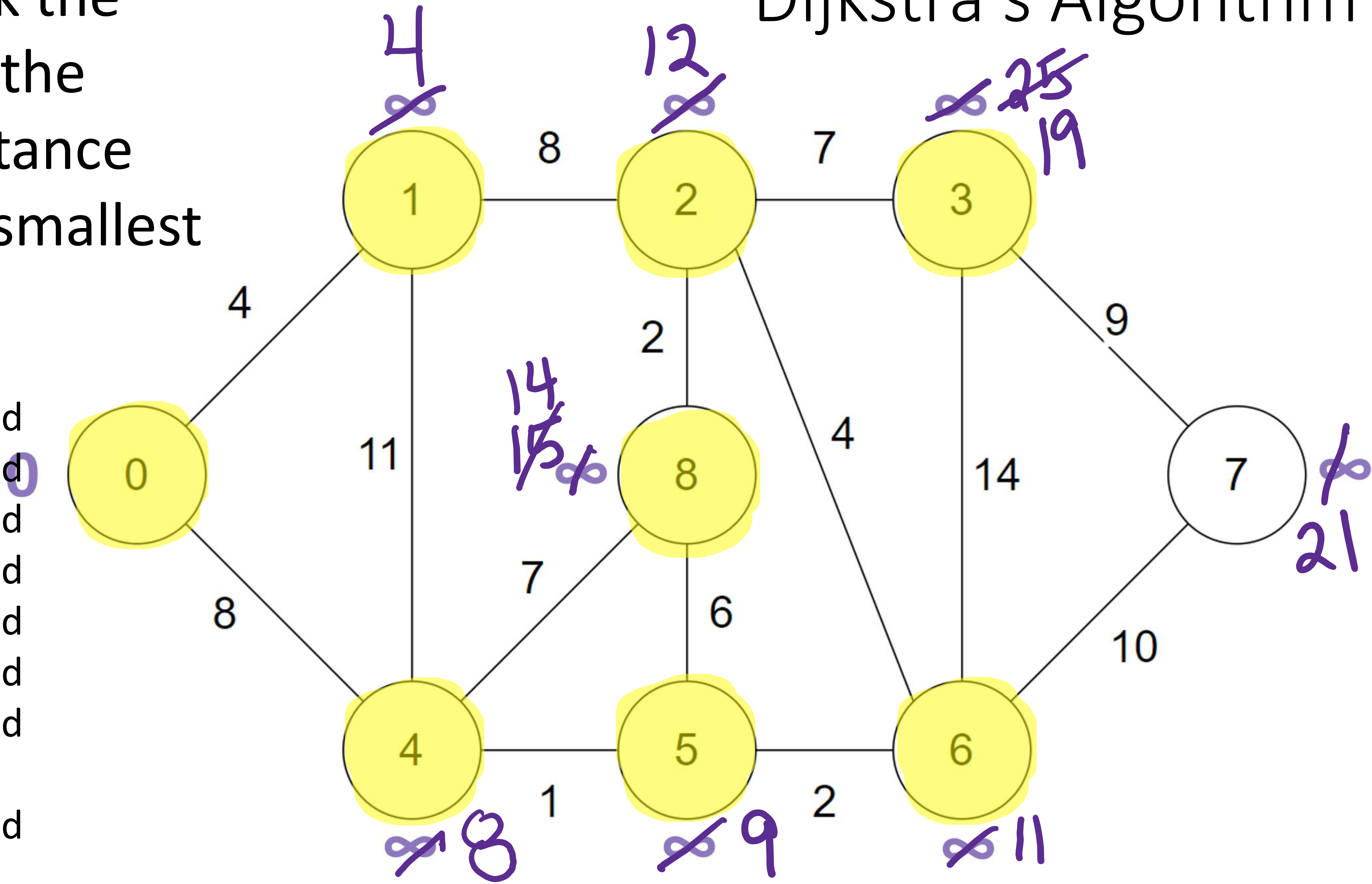
28 is not less than 21

NO CHANGE



Dijkstra's Algorithm

Now we pick the Vertex with the smallest distance value – the smallest d(u)

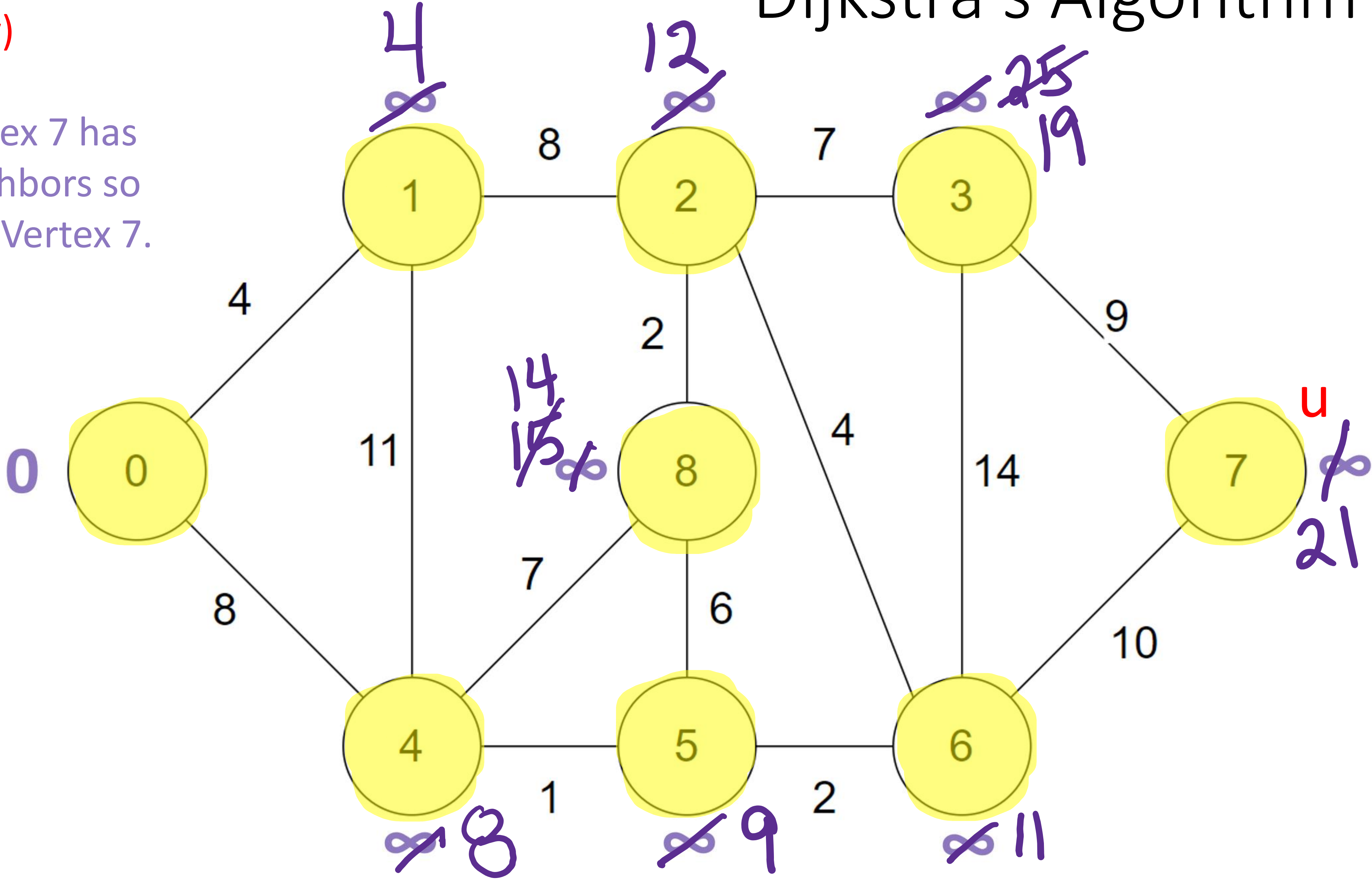


0 – already visited
1 – already visited
2 – already visited
3 – already visited
4 – already visited
5 – already visited
6 – already visited
7 – d(u) of 21
8 – already visited

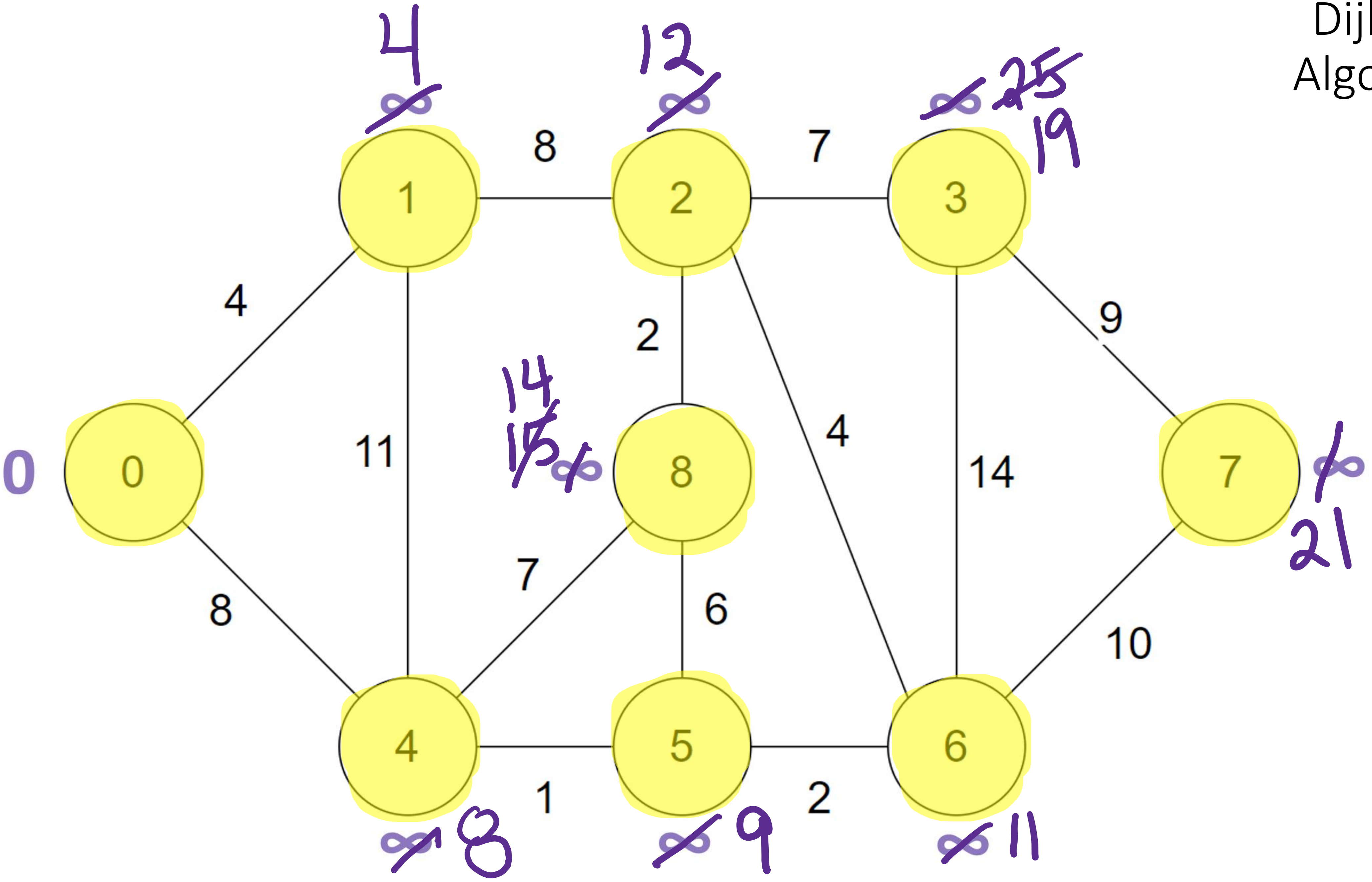
Dijkstra's Algorithm

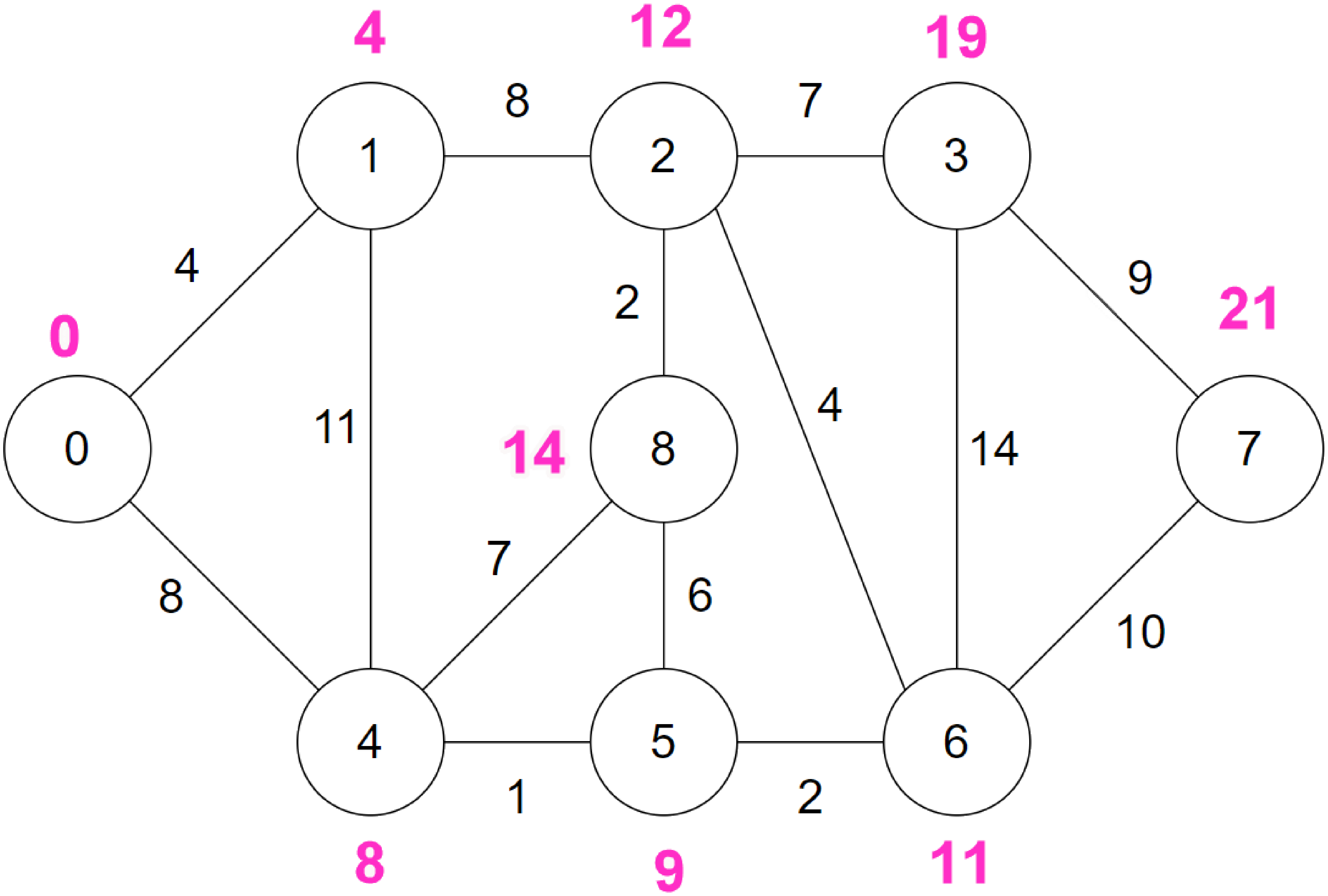
```
if (d(u) + c(u,v) < d(v))
 d(v) = d(u) + c(u,v)
```

d(u) is 21 but Vertex 7 has NO unvisited neighbors so we are done with Vertex 7.



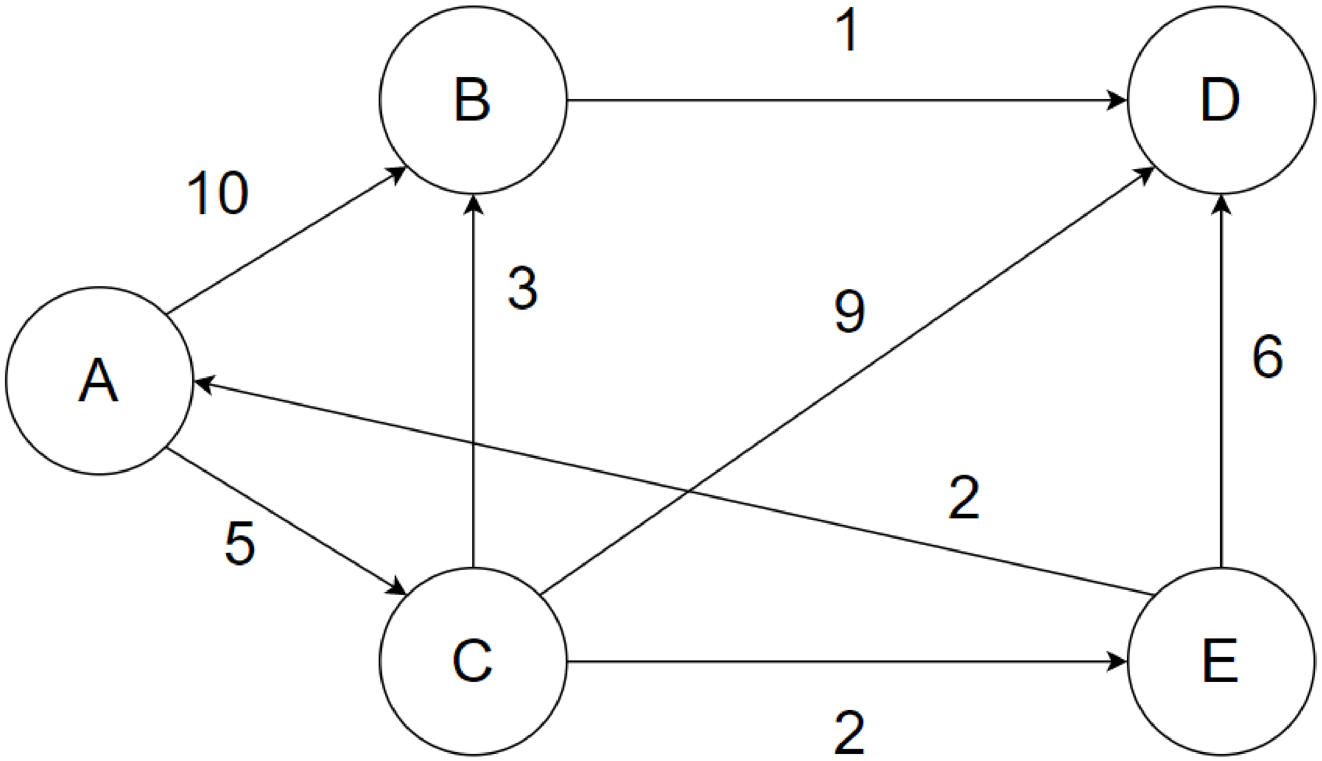
Dijkstra's Algorithm




4
12
19
8
7
1
2
3
4
0
9
21
11
2
4
14
0
8
14
7
8
7
6
10
4
1
5
2
6
8
9
11

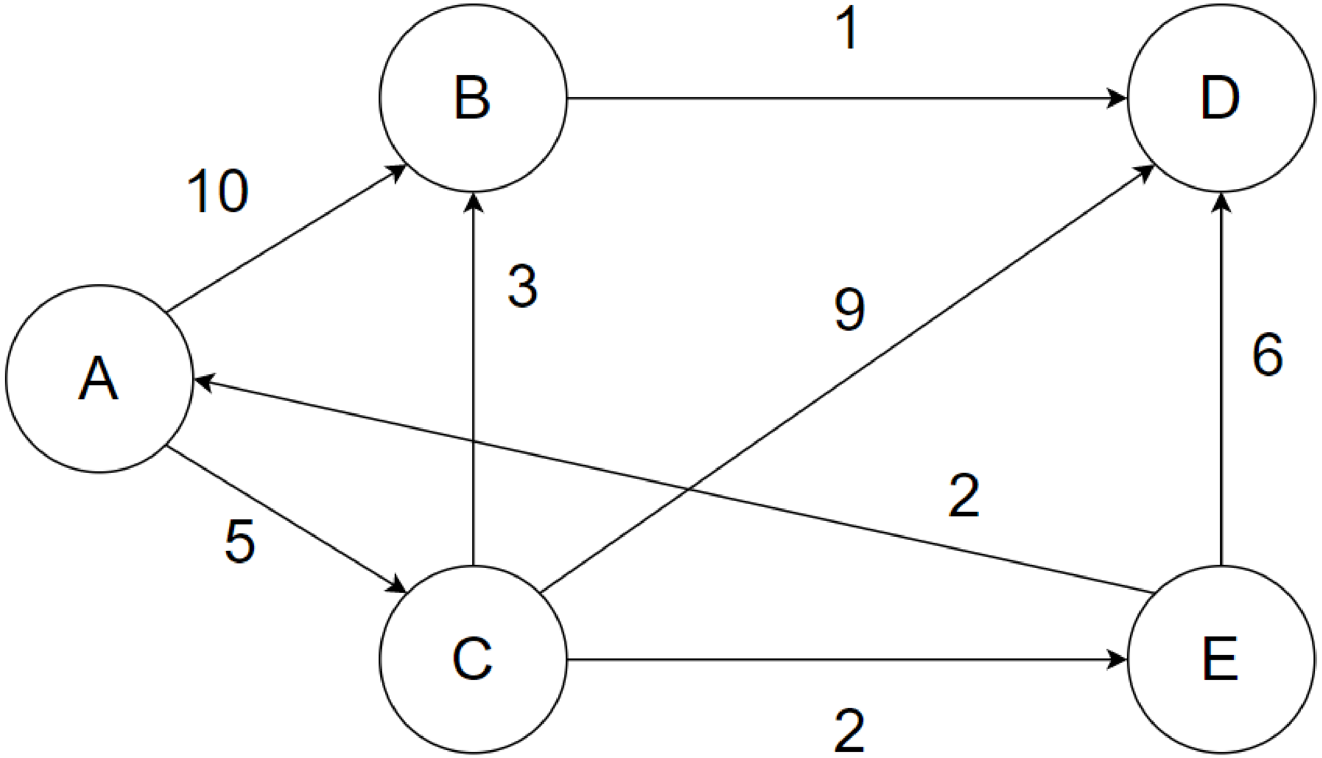
Source vertex MUST be listed first.

	A	B	C	D	E
	0	∞	∞	∞	∞



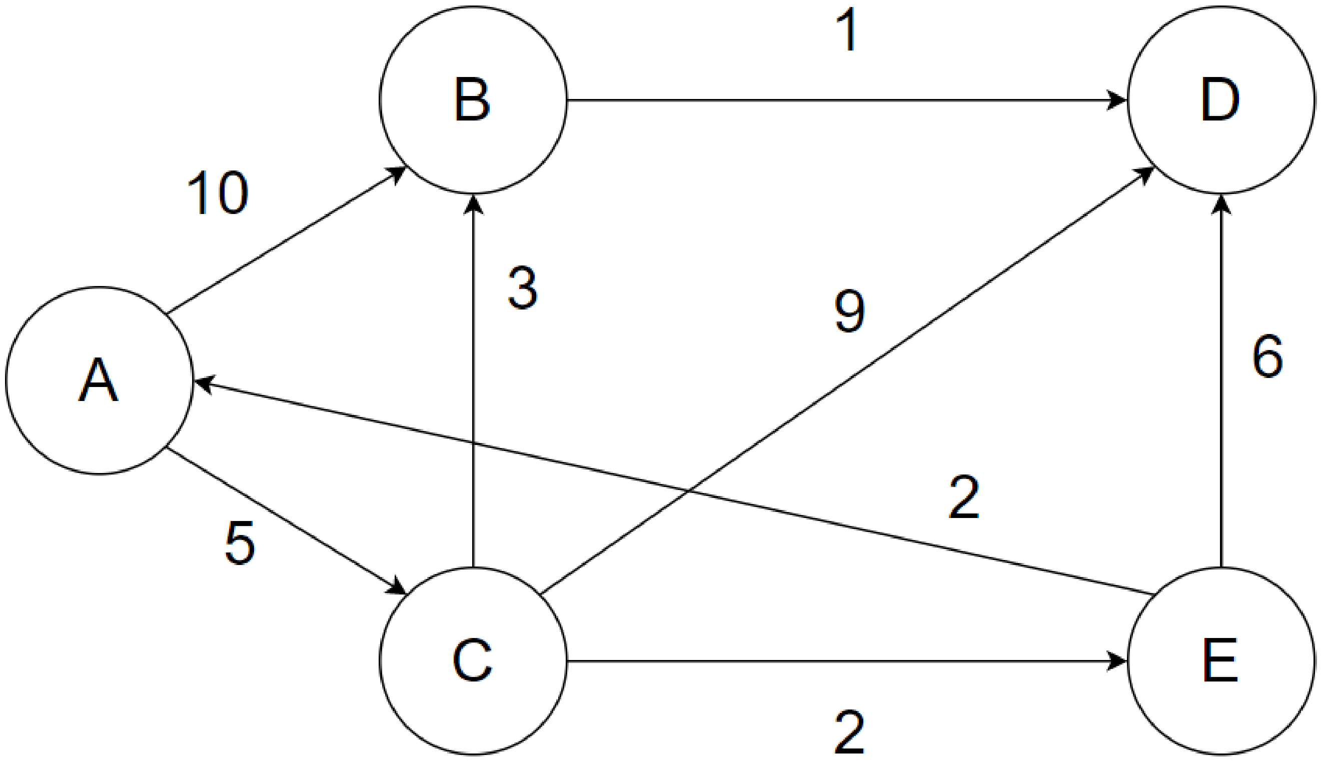
Which vertex has the smallest d(u)?

	A	B	C	D	E
A	0	∞	∞	∞	∞



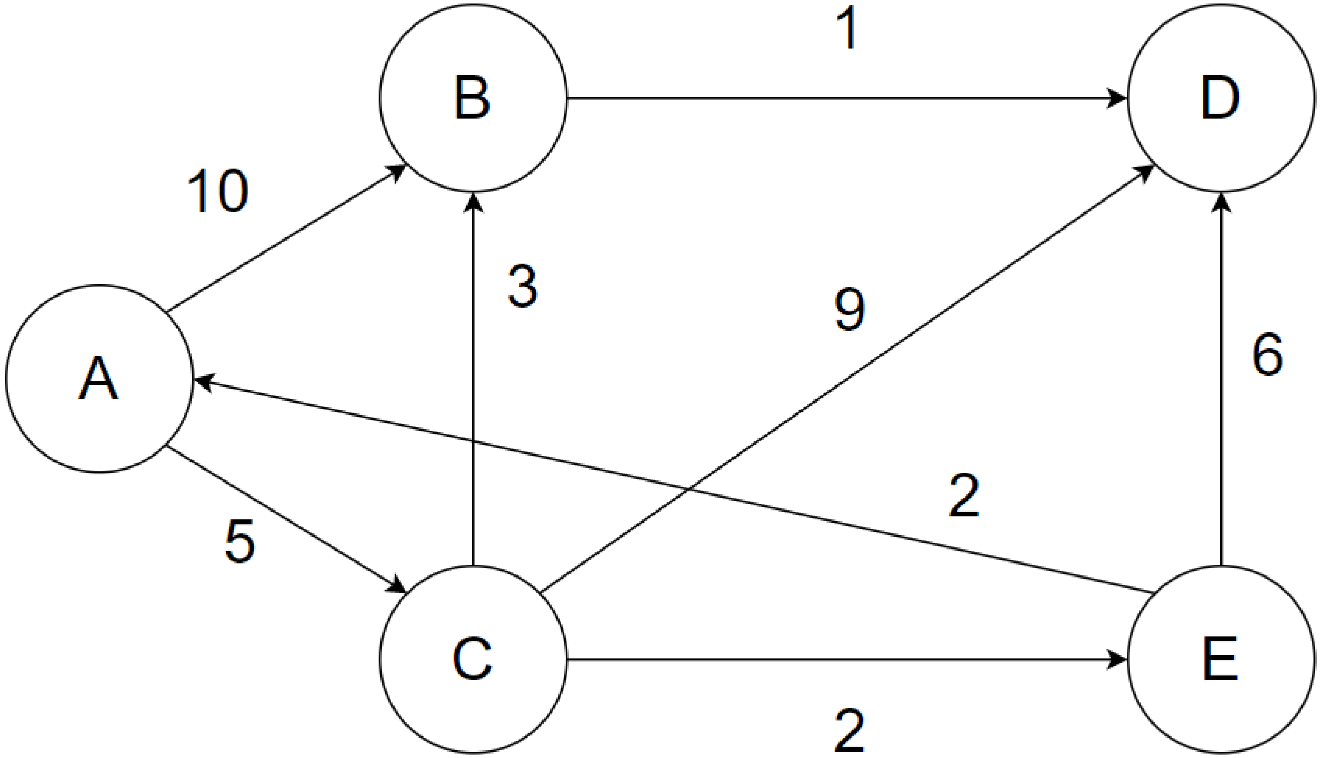
A -> B and A -> C

	A	B	C	D	E
A	0	∞	∞	∞	∞
		10	5	∞	∞



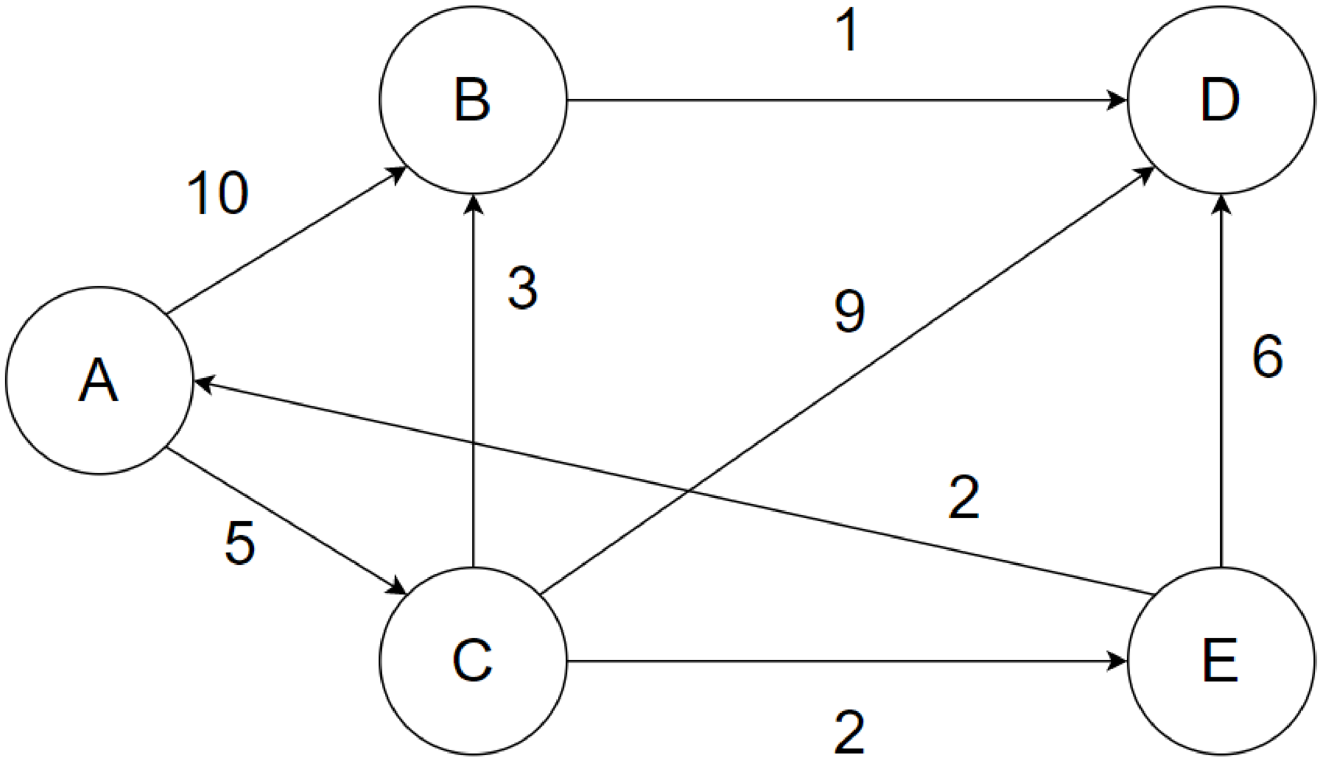
Which vertex has the smallest d(u)?

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞



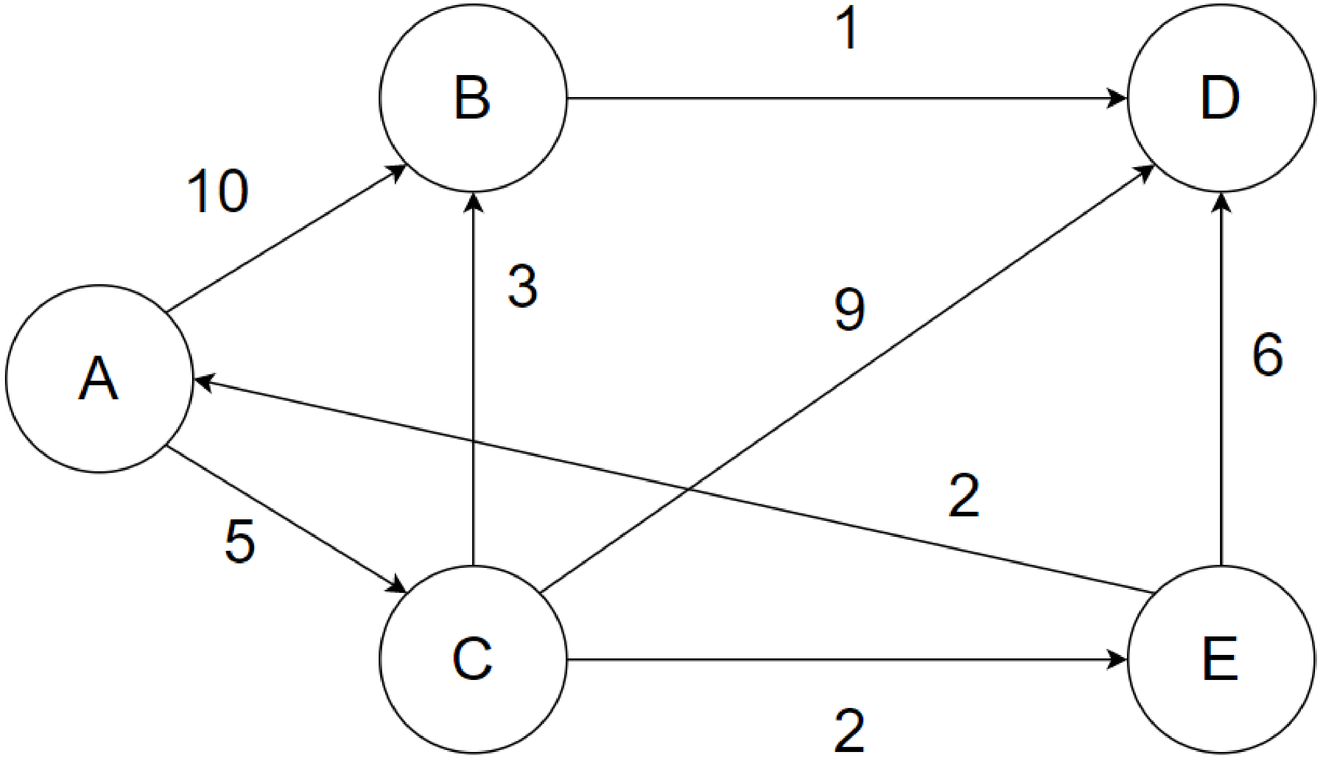
C -> B and C -> D and C->E

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
		8		14	7



Which vertex has the smallest d(u)?

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7

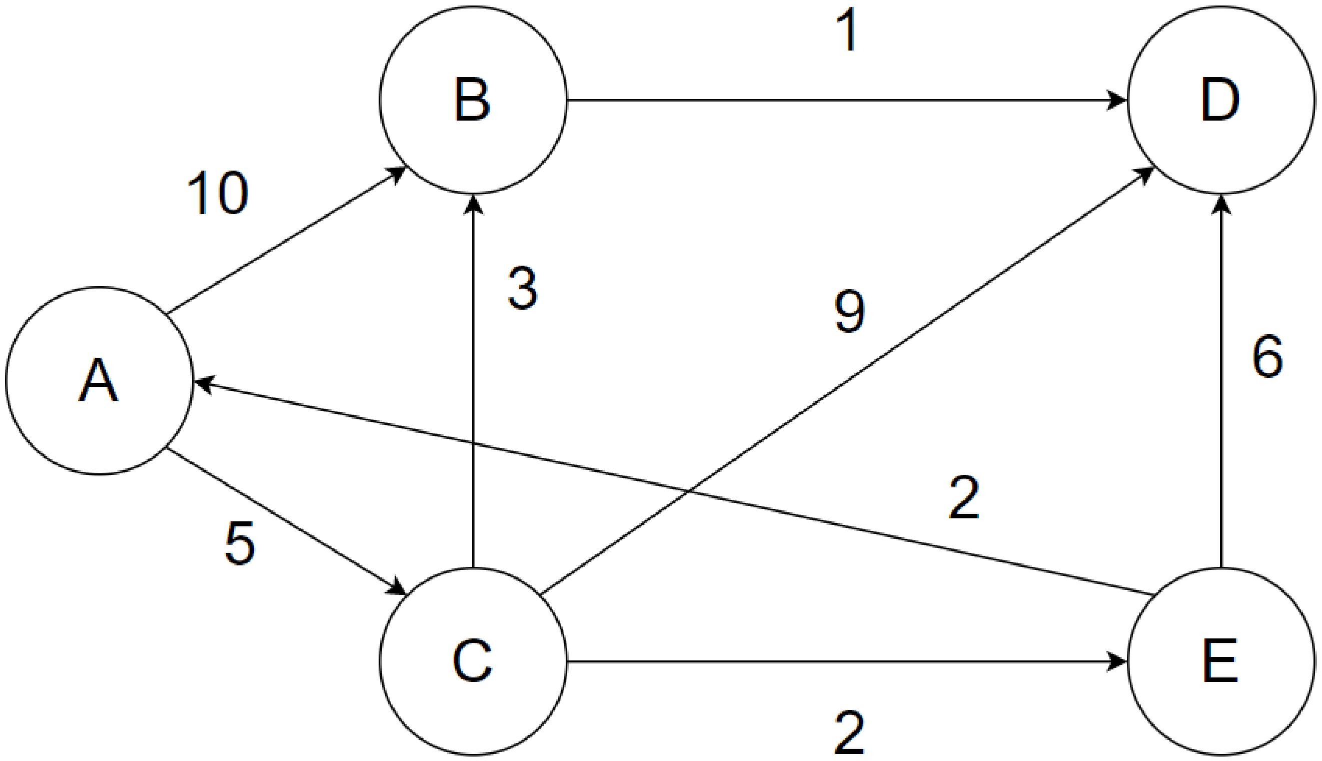


Dijkstra's Algorithm

E -> A and E -> D

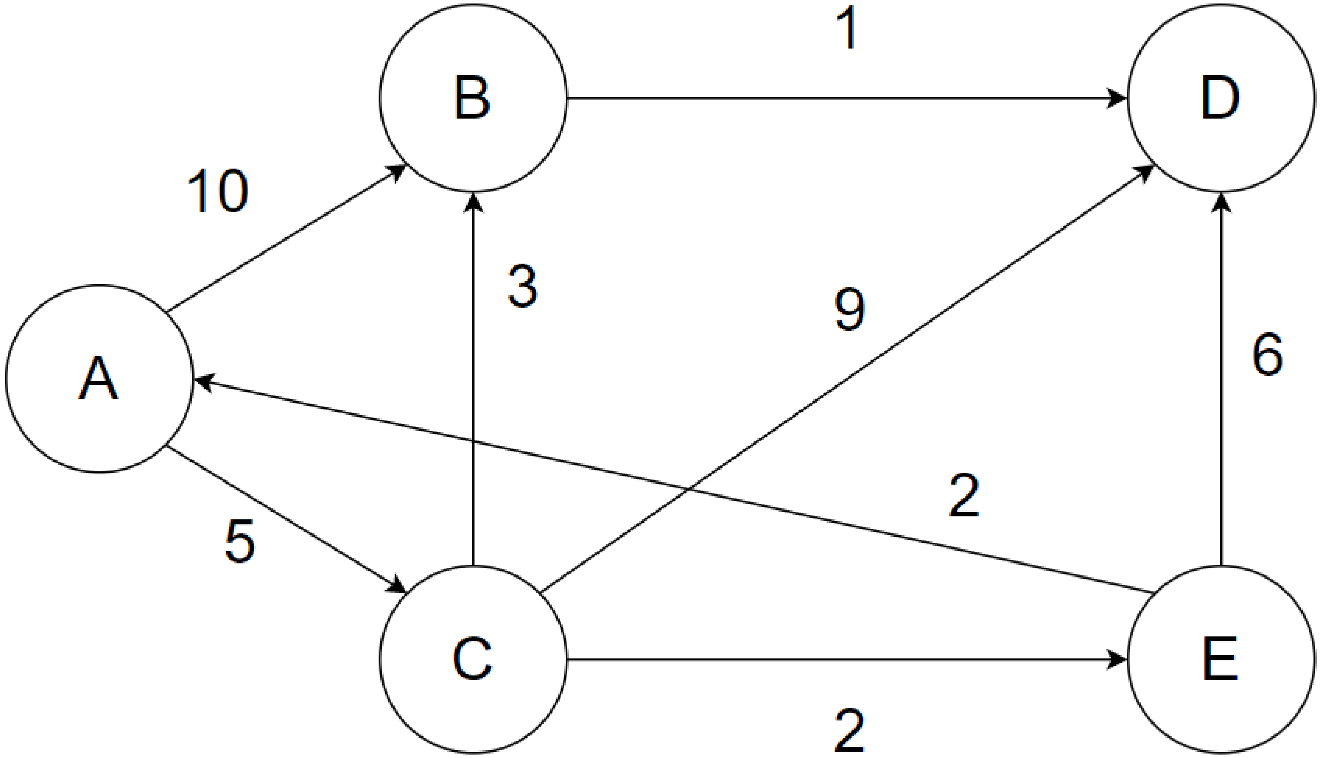
7+6

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
		8		13	



Which vertex has the smallest d(u)?

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	

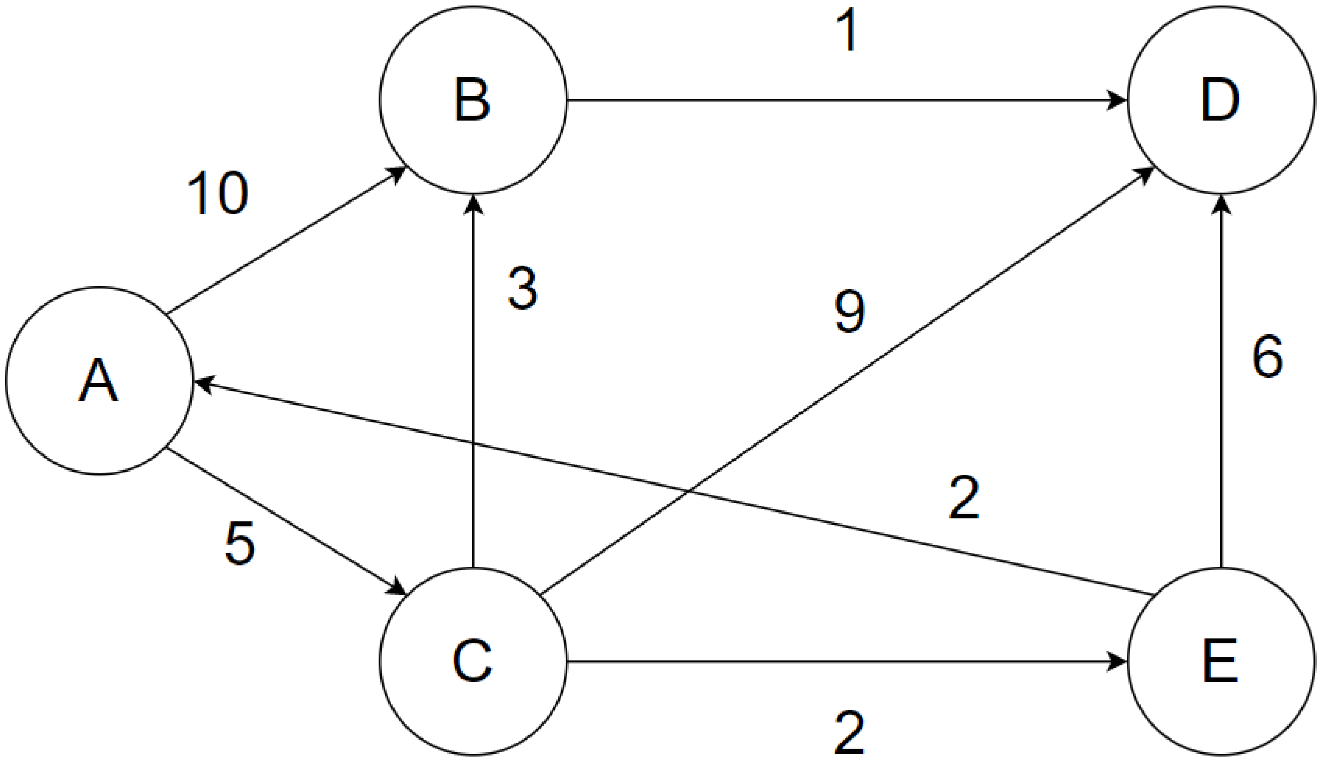


Dijkstra's Algorithm

B -> D

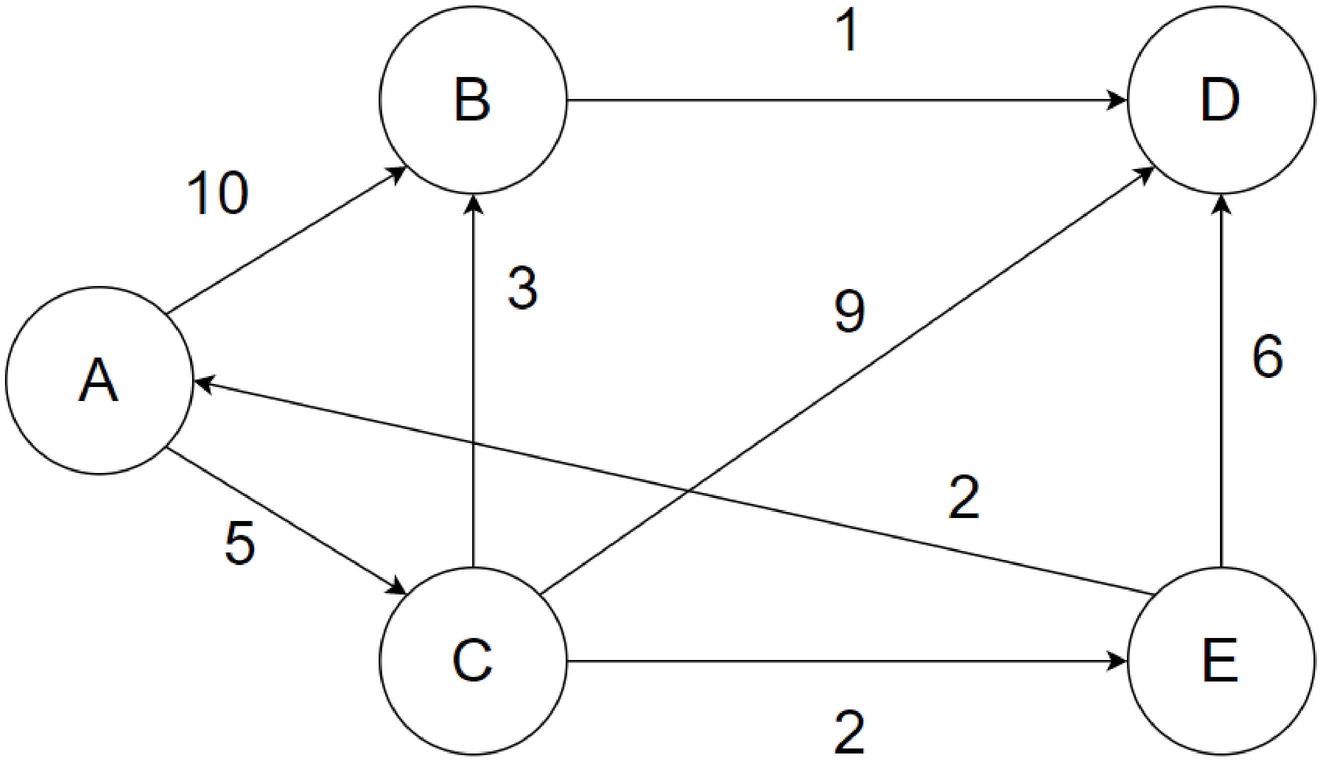
8+1

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	
D				9	



Which vertex has the smallest d(u)?

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	
D				9	

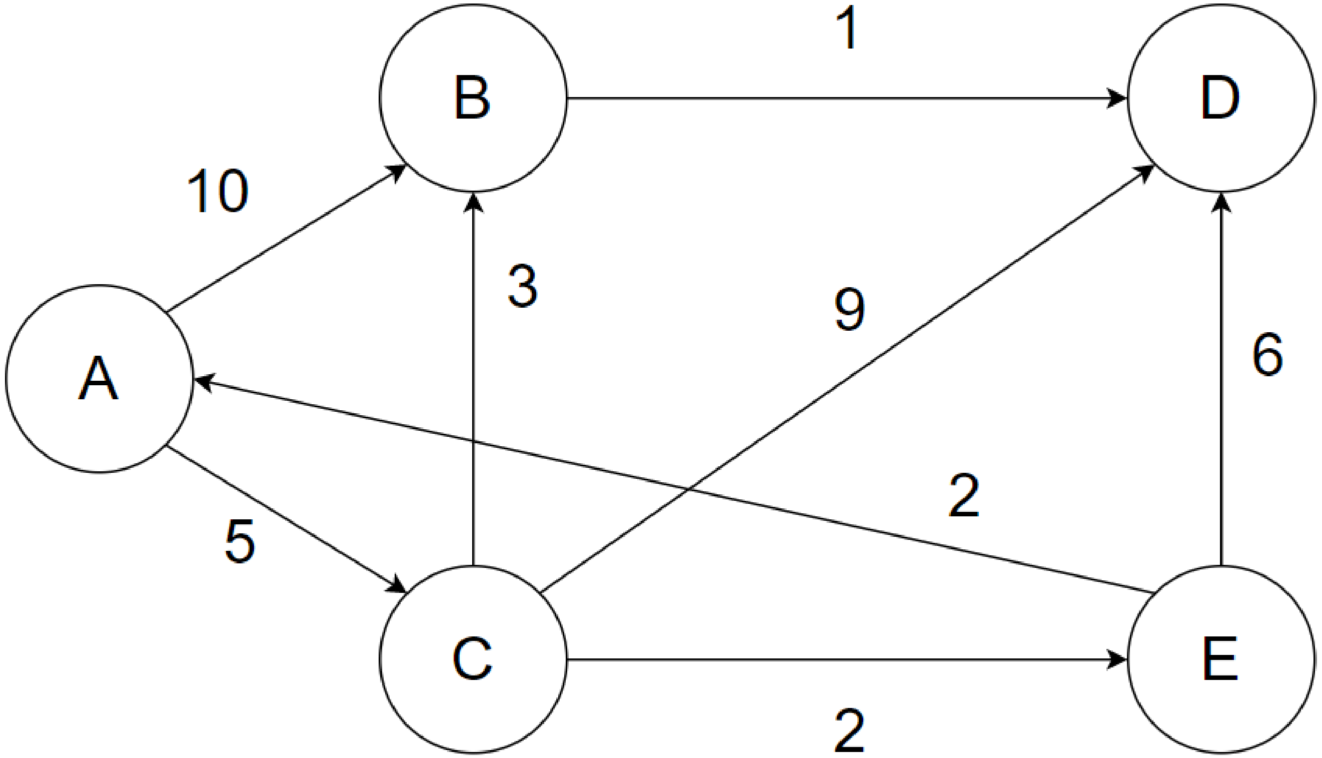


Dijkstra's Algorithm

What is the shortest distance from A to D?
9

What is the shortest distance from A to B?
8

	A	B	C	D	E
A	0	∞	∞	∞	∞
C		10	5	∞	∞
E		8		14	7
B		8		13	
D				9	

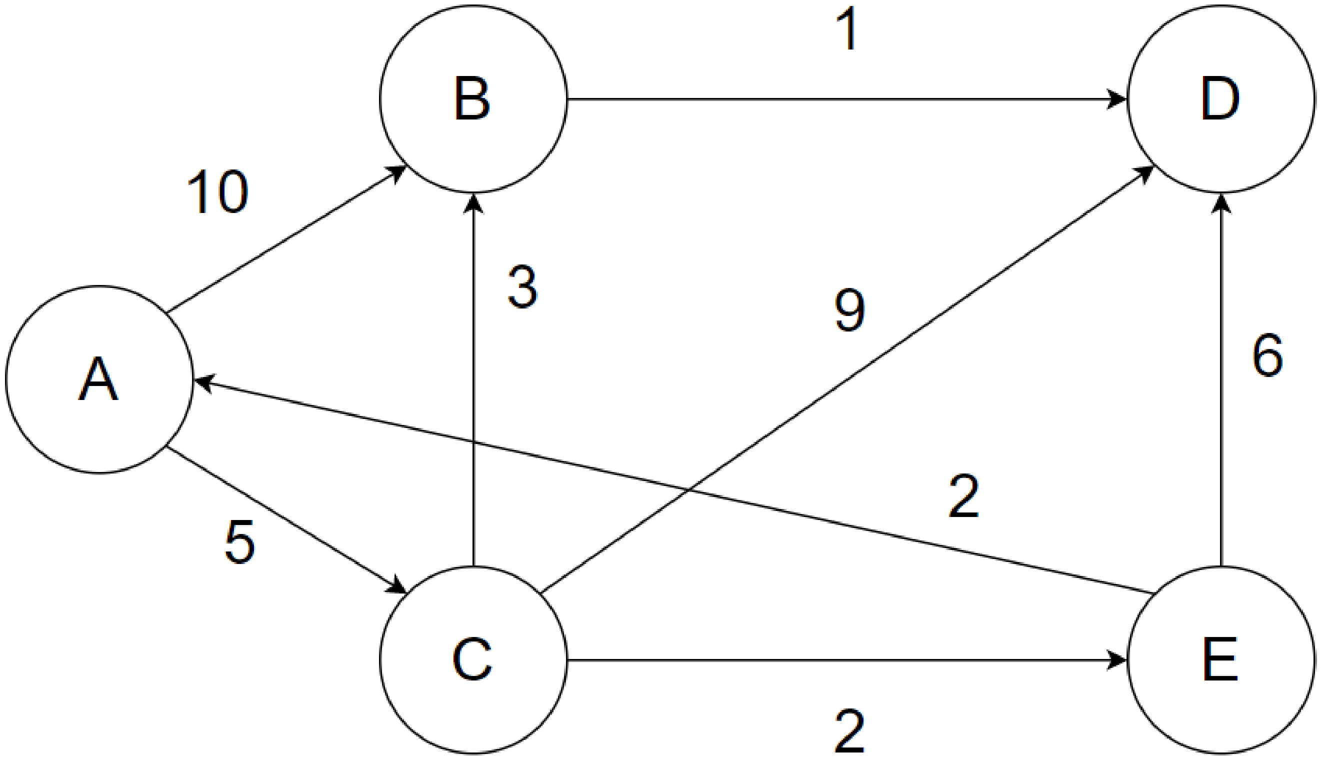


Dijkstra's Algorithm

What is the path from A to D?

ACBD

	A	B	C	D	E
A	0	∞	∞	← 4	∞
C		10	5	∞	∞
E		8	← 3	14	7
B		8		13	← 2
D				9	← 1

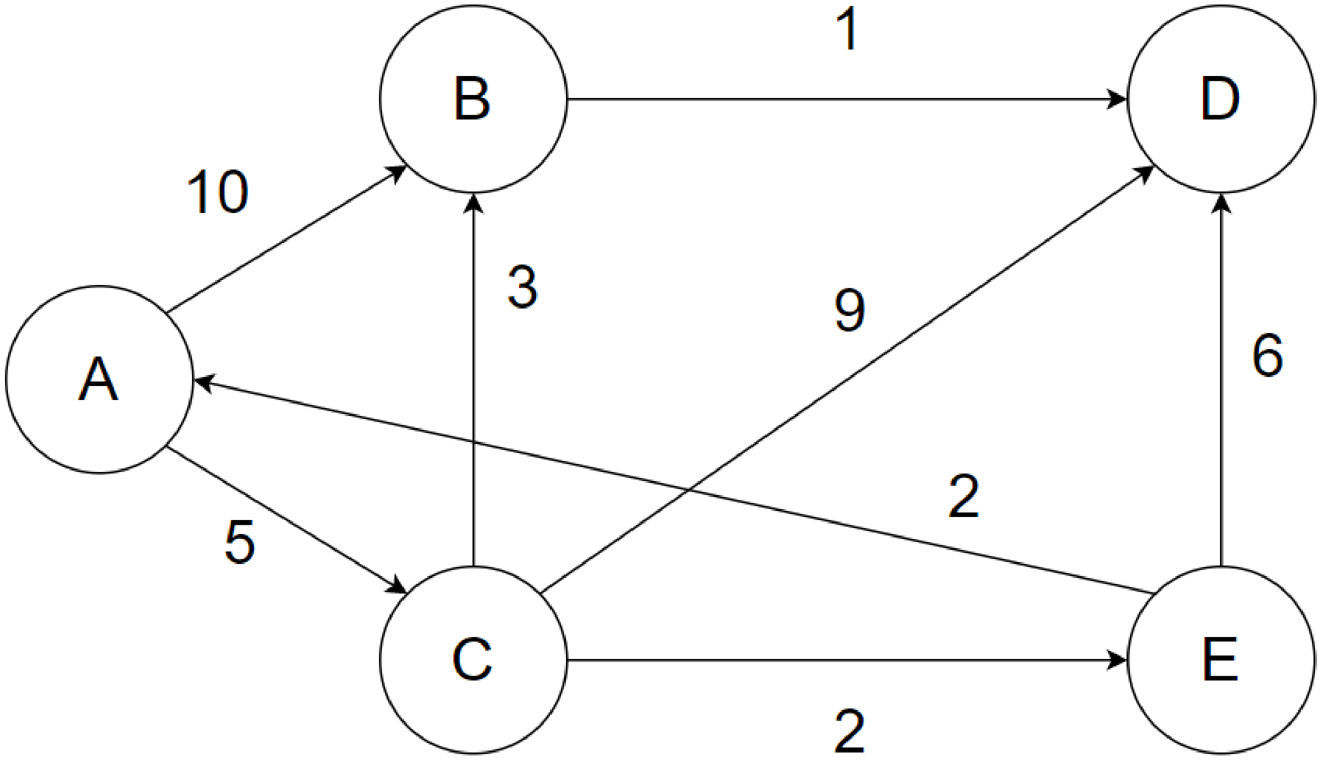


Dijkstra's Algorithm

What is the path from A to B?

ACB

	A	B	C	D	E
A	0	∞	∞	⬅ 3	∞
C		10	5	∞	∞
E		8	⬅ 2	14	7
B		8	⬅ 1	13	
D				9	


A
B
C
D
E
10
5
3
1
9
6
2
2